A DEPENDENTLY TYPED LANGUAGE WITH NONTERMINATION

Vilhelm Sjöberg

A DISSERTATION
in
Computer and Information Science
Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2015

Supervisor of Dissertation

Stephanie Weirich
Professor of Computer and Information Science

Graduate Group Chairperson

Lyle Ungar
Professor of Computer and Information Science

Dissertation Committee
Jean Gallier (Professor of CIS)
Benjamin C. Pierce (Professor of CIS; Committee Chair)
Aaron Stump (Professor of Computer Science, University of Iowa)
Steve Zdancewic (Professor of CIS)

Acknowledgments

The most thanks for this thesis go to Stephanie Weirich, who has been a fantastic research advisor and mentor. Stephanie always has time for her students—whenever I ran into technical difficulties she seemed genuinely happy to drop everything else to work together on the whiteboard (where her skills are very impressive). Her enthusiasm is contagious, and I always leave her office happy and full of energy. All in all I could not wish for a better phd advisor.

The work described in this thesis came out of the Trellys project, and I benefitted very much from co-operation with the rest of the Trellys team. Their contributions are described in more detail in Section 1.2. Here I would like to particularly thank two of them. Chris Casinghino was my closest collaborator at Penn. Both our research (on two different parts of the same programming language) was improved by having someone to bounce ideas with. Aaron Stump was a constant source of new ideas and insights. I would also like to thank him for inviting me to spend a very enjoyable summer visiting the University of Iowa.

The University of Pennsylvania is a great place to be a programming languages student. The Penn PL Club is a vibrant and tightly-knit place, the faculty (Benjamin Pierce and Steve Zdancewic) are very helpful to everyone in the group, and the students and postdocs always have interesting research projects to talk about. Special thanks to the plclub people who I shared my office with over the years—it was lots of fun chatting with you all the time!

When typesetting this document, two very helpful tools were `Ott` by Sewell et al. [115], and `pulp` by Daniel Wagner.[1]

This work was supported by the National Science Foundation (NSF grants 0910500 and 1319880).

[1] `https://github.com/dmwit/pulp`

ABSTRACT

A DEPENDENTLY TYPED LANGUAGE WITH NONTERMINATION

Vilhelm Sjöberg
Stephanie Weirich

We propose a full-spectrum dependently typed programming language, *Zombie*, which supports general recursion natively. The Zombie implementation is an elaborating typechecker. We prove type saftey for a large subset of the Zombie core language, including features such as computational irrelevance, CBV-reduction, and propositional equality with a heterogeneous, completely erased elimination form. Zombie does not automatically β-reduce expressions, but instead uses congruence closure for proof and type inference. We give a specification of a subset of the surface language via a bidirectional type system, which works "up-to-congruence," and an algorithm for elaborating expressions in this language to an explicitly typed core language. We prove that our elaboration algorithm is complete with respect to the source type system. Zombie also features an optional termination-checker, allowing nonterminating programs returning proofs as well as external proofs about programs.

Contents

List of Figures

Chapter 1

Introduction

Of all the ideas to come out of programming language research, my favorite is *dependent types*. Formally, dependency is a very simple feature: types are allowed to contain program expressions, and the return type of a function is allowed to mention the function arguments. Starting with a description of a type system for an ordinary functional programming language, dependent types can be added with just a handful of extra typing rules, much less formalism than is needed to describe e.g. a Java-style object system. Yet they add lots of extra power.

First, dependent types enable *type-level programming*. For example, C++ programmers may (ab)use the details of template instance selection in order to chose one of two types depending on a compile-time parameter, or encode lists of types by a stylized use of templated classes [6]. In a dependent language one can use ordinary if-expressions and the ordinary list datatype. One particularly impressive example of how dependent types allow more programs to be written is *type-generic programming*. Languages like Haskell provide special-purpose infrastructure and libraries to let programmers write generic recipes for how to, e.g., compare, pretty-print, serialize, or randomly generate values, and then automatically instantiate that recipe for arbitrary data types [68, 75]. In a dependently typed language, it turns out the same programs can be written without any special language support [138].

Second, dependent types enable *program verification*. While ordinary types express fairly coarse properties like "being a floating-point number" or "being an RSA key", dependent type systems add precision, e.g. "a number whose square is 4" or "a key which is used by Alice to certify text messages for declassification". In fact, the sky is the limit. There is a close connection between (terminating) programs and (constructive) proofs, the "propositions as types" principle [140], which lets us formulate any property expressible in logic as a dependent type, and any proof as a dependent program. Indeed, considered as a logic dependent types are more expressive than most other formulations (such as first-order logic), because propositions can be defined by type-level recursion.

In a traditional verification system such as ACL2 [72] the programmer uses one programming language to write functions, a different logic to write specifications, and yet another command language to drive the theorem prover. In a dependently typed system the same language is used for programming ordinary functions, for type-level programming, for writing specifications, and for proving programs correct. And the typing rules that enable this are beautifully elegant, just as one would expect given that the idea originated in mathematical logic.

Working software engineers may not primarily care about beauty or elegance when they select their tools. But dependent type systems have the potential to be useful for practical software development also. Among technologies for formally reasoning about software, type systems are by far the most widely adopted. Probably this is because type checking is relatively unobtrusive, while still providing benefits to the programmer (such as improved understanding of, and confidence in the software). We hope to move from simple type systems to dependent types without losing that advantage.

In order to make software verification more generally attractive, we believe that it is important to support what we call **lightweight verification** [117]. This term should be understand as the opposite of full functional verification: instead of spending a Herculean effort on proving that a piece of software is completely correct, the programmer identifies a few important properties and spends a limited development budget on proving those.

A programming language aimed at lightweight verification needs above all to not get in the way: a programmer who is uninterested in formal verification should be able to write programs in exactly the same way as in a normal mainstream language. (For the purposes of this thesis, we consider the mainstream to consist of functional programming languages like ML and Haskell. ☺) As the program evolves, one can add more precise specifications and proofs in places where more assurance is desired.

Dependent types seem particularly suited for this type of language. Because specifications are stated in types, one can add more precise specifications by adding more information to existing types, rather than having to switch to a separate system to manage pre- and postconditions.

Dependent types also offer a low barrier to entry. In a non-type-based verification system like ACL2, the programmer has to learn to write specifications and proofs as a completely separate activity. The effort to learn to use the tool and integrate it in the programming workflow is hard to justify unless one intends to use it heavily. Dependent types use a single language, so they should be easier both to learn and to dip one's toes into.

1.1 Zombie: Full-spectrum dependent types with nontermination

Dependently typed languages already support almost all features of functional languages like ML or Haskell. However, one big limitation is **termination-checking**. Functional programmers are used to being able to write general recursive programs without ceremony, but Coq and Agda[2] require every function definition to pass a (necessarily conservative) termination checker. The goal of this thesis is to lift this restriction by developing a dependently typed language which allows functions defined by general recursion. We call our language *Zombie*.

Further, we aim to allow nontermination while retaining the power of dependent types. While there are many examples of languages that combine nontermination with dependent or indexed types, most take care to ensure that nonterminating expressions can not occur inside types, either by making the type language completely separate from the expression language or by restricting dependent application to values or "pure" expressions. In Zombie, types and expressions are unified and types can be computed by functions. In other words, this is a **full-spectrum** dependent type system.

Full-spectrum dependency with nontermination is a less explored area of the design space. The most comparable languages are Cayenne [10], Cardelli's Type:Type language [29], Nuprl extended with partial types [36, 40], and $\Pi\Sigma$ [9]. However, there are still things to be learned here, and as we will see, Zombie is quite different from all of these languages when it comes to evaluation-order, equational theory, support for logical proofs, and reasoning principles for recursive functions.

One of the lessons from combining dependency and nontermination has been to tease apart two different roles that the restriction to normalizing programs served in previous designs: for *semantic* concerns (particularly erasability, which in turn is related to logical consistency and type safety) and to enable *type checking* (by making $\beta\eta$-convertibility decidable).

The Zombie prototype implementation is structured around a typed core language (similar to e.g. the GHC Haskell compiler). In that setting, the above concerns map neatly onto compiler stages. We work out the semantic issues by defining the core language and proving type safety. Then we study type checking and proof inference by defining a surface language and writing a type-checker/elaborator from the surface language to the core. This approach is helpful from a software engineering point of view (bugs in the elaborator can be detected when checking the generated core

[2] In this thesis we consider Coq and Agda as the prototypical examples of "mainstream" dependently typed programming languages, and assume that the reader is passingly familiar with at least one of them. For an introduction for beginners, see e.g. Norell and Chapman's Agda tutorial [99].

expressions), and also from a language design point of view (because it factors the design process into two smaller tasks).

The design work benefited from having an implemented typechecker—writing medium-sized programs in the language uncovered several interesting corner cases in the core semantics, and provided guidance for the design of the surface language. Being able to write programs is also a good minimum test of whether the proposed language is really practical or not. Chapter 2 gives some examples of the programming we have done in Zombie, while at the same time giving a tour of the novel features of the language.

1.1.1 Core language: nontermination, erasure, CBV, and equality

In Chapters 3 and 4 we formally define and study the core language.

Because of our unusual design goal, our core language includes several novel features. It is *call-by-value*, which is a particularly good fit for nonterminating dependent languages (Section 3.3). To enable efficient compilation it supports *computational irrelevance* (i.e. erasure reflected in the type system). Irrelevance requires some care in a language with nontermination, because we must be careful to only erase terminating expressions. And perhaps most interestingly, we adopt a *novel treatment of propositional equality*, which is computationally irrelevant (equality proofs do not need to be examined during computation) and "very heterogenous" (we can state *and* use equations between terms of different types).

In addition to these novel features, the Zombie implementation also supports the standard features needed for dependently typed programming and verification: function definitions by (terminating) structural recursion, inductively defined datatypes, type-level computation with large eliminations, a universe hierarchy Type_ℓ, etc.

This presents a methodological problem, because the full core language is too large to prove results about. The type-safety of the full language depends on correctness of the (optional) termination checking, and normalization proofs for realistic dependently typed languages are technically very complicated. The solution is to study calculi which include some but not all of the features of the full language.

In Chapter 3 we define such a calculus, which forms the centerpiece of this dissertation. It is a subset of the full Zombie core language—it keeps most of the novel features, but omits Zombie's sophisticated system for optional termination checking. Instead, we just assume that any expression may diverge, and in places where the full language requires a know-terminating expression, the calculus requires a syntactic value. At the same time we can simplify the language by omitting features, such as structural recursors and stratified universes, which are only used for ensure termination. Apart from the lack of the termination checking, this calculus is even slightly more expressive than the current version of the full language, because we collapse all the universe levels and assume Type : Type. (As we describe in Section 7.7.2, there are technical reasons that make Type : Type challenging to combine with our termination-checking rules.) Figure 1.1 shows the features of the full language and the formalized calculus.

In Chapter 3, we prove that this core calculus is type safe. As we describe in Section 1.1.3 below, we have also formally studied the typing rules for termination-checking So the formal metatheory reassures us that the various features of Zombie are sound. At the same time, the implementation lets us gather experience about how these ideas carry over into a full-scale programming language.

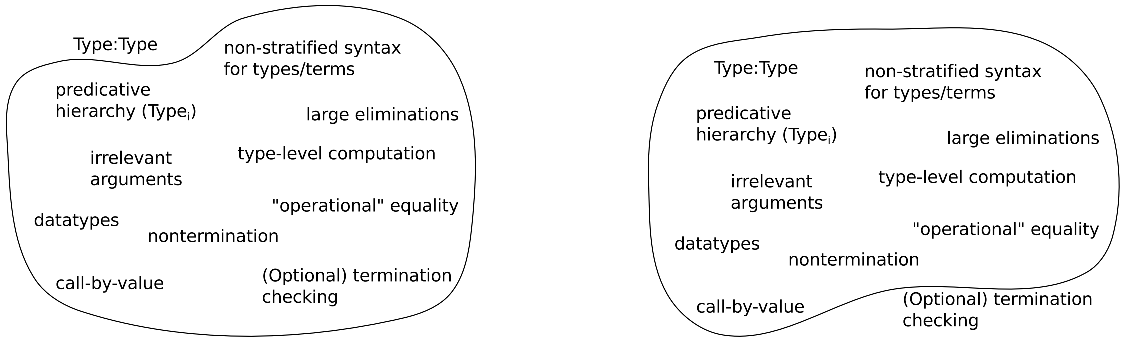

Figure 1.1: Features included in the full Zombie implementation (left), and in the core calculus defined in Chapter 3 (right).

The core calculus models Zombie as it is currently implemented, but this is only one point in a larger space of possible designs. In Chapter 4 we describe two ways in which the typing rules could be varied. First, there is a range of choices for internalizing the operational semantics in the propositional equality type. Second, we contrast a "value-dependent" application rule (which is commonly used by other effectful dependent languages) with the Zombie application rule (which is more expressive, but precludes certain side effects).

1.1.2 Surface language: congruence closure and unification

In Chapters 5 and 6, we turn to designing a programmer-friendly surface language which elaborates into the core.

The main innovation here is that the surface language is based around a novel new style of dependent programming **up to congruence closure**, as opposed to the "programming up to β-convertibility" provided by most existing languages. This lets us to handle nonterminating expressions gracefully, because the programmer is in control of which expressions are evaluated during typechecking.

Congruence closure is an important algorithm for automatic theorem proving (so far mainly used for untyped first-order logic), so even apart from nontermination it is interesting to integrate congruence closure into a dependent type system. In doing so we made novel contributions: we extend the definition of congruence closure to handle equations between equations, and we give a new algorithm for generating proofs of equations between *typed* expressions.

In addition to the implementation of the elaborator, the way we formulate the *specification* of the surface language is also interesting. Designing a language around an elaborator—an unavoidably complicated piece of software—raises the risk of making the language hard to understand. Programmers could find it difficult to predict what core term a given surface term will elaborate to, or they may have to think about the details of the elaboration algorithm in order to understand whether a program will successfully elaborate at all.

We avoid these problems using two strategies. First, the syntax of the surface and the core language differ only by *erasable annotations* and the operational semantics ignores these annotations. Therefore the semantics of an expression is apparent just from looking at the source; the elaborator only adds annotations that can not change its behavior. Second, we define a *declarative specification* of the surface language, and prove that the elaborator is complete for the specification. As a result, the programmer does not have to think about the concrete elaboration algorithm.

In order to keep the proof of completeness small, the declarative type system in Chapter 5 again only treats a subset of the full Zombie language. It elaborates into the type system from Chapter 5, but omits two features: the rules related to datatypes, and the fully general application rule (instead using value-dependent application). The resulting calculus is shown in Figure 1.2. These features are present in the full Zombie implementation, and as we describe in Section 5.6 the elaboration algorithm handles them well.

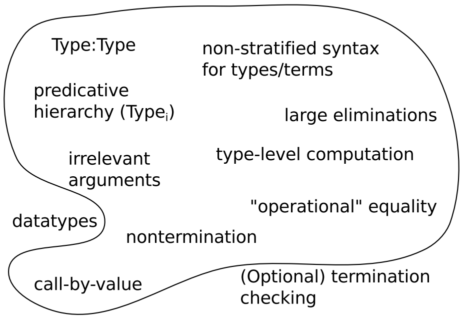

Figure 1.2: The fragment of the core calculus targeted by the surface type system in Chapter 5.

To get a pleasant programming experience the surface language must also support type inference by **unification**. This is the subject of Chapter 6. The material in this chapter is more preliminary than the rest of the thesis, because we do not yet have a good theory of how much type inference can be done when combining congruence closure and unification. However, we bracket the problem. On the one hand we present an undecidibility result, which says that even with a limited set of features we can not hope to infer *all* type arguments. On the other hand we describe the current implementation in Zombie, which uses a heuristic algorithm but performs very well on our suite of examples. We leave as future work the problem of the defining a declarative type system that explains exactly what arguments unification can succesfully infer.

1.1.3 Optional termination checking

Finally, Chapter 7 deals with termination checking in Zombie.

Zombie allows the programmer to identify certain expressions as terminating and have the typechecker verify that. This capability is important both to express precise specifications and to compile programs efficiently. Our interest in lightweight verification influeced the way we mix terminating and nonterminating code. Unlike most other languages, nonterminating functions in Zombie are first-class citizens, which can be written with no syntactic overhead and which reduce according the same operational semantics as other expressions.

In addition to this thesis, Chris Casinghino also wrote a thesis about the theory of the Zombie Core language [30], focusing specifically on the termination checking. His main theorem is that all expressions that are marked terminating do in fact normalize. To keep the proof tractable, he studies a core calculus that omits many of Zombie's features, e.g. general datatypes, erased arguments, large eliminations, and the collapsed syntax for types and terms.

Casinghino's thesis is the definite reference for how to handle Zombie's features in a normalization proof. In the first half of Chapter 7 we describe the same rules for termination checking, while comparing them to other dependently typed languages. However, the calculus in this chapter represents an even smaller subset of full Zombie, because we leave out features (such as polymorphism and type-level computation) which are standard but make normalization proofs difficult. Figure 1.3 compares these two calculi.

After this recapitulation of previously published work, the second half of Chapter 7 presents a new result. The way termination is handled in Zombie is inspired by type systems based on modal logic. We prove that one can design an equally expressive type system in the standard type-and-effect style. This result is useful for several reasons. First, thinking about nontermination as an effect is a common intutition in other dependent languages, so it is interesting to know how the expressivity of

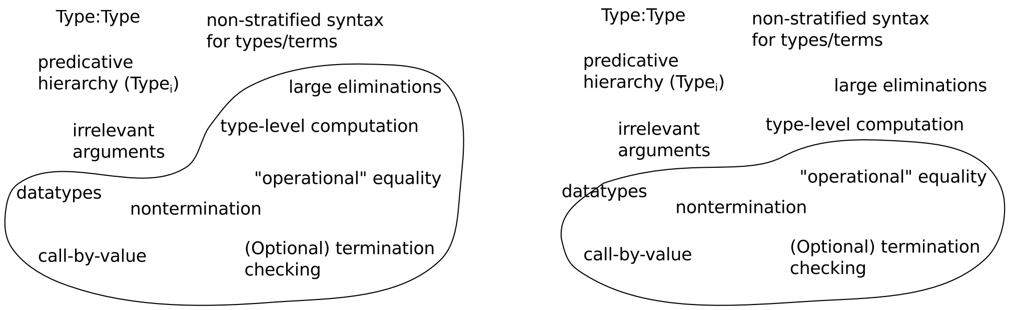

Figure 1.3: The calculi studied in Casinghino's thesis [30] (left) and in Chapter 7 (right).

Zombie compares (it is equivalent). More importantly, we argue that the effect-style formulation is a more suitable target for elaboration, and that it simplifies the metatheory. In particular, we give a new proof of normalization (Section 7.6), which is considerably simpler than the proof we previously published.

1.2 Who did what?

This research was carried out in the context of the Trellys project, an NSF-founded research project about language design for dependent types. The NSF grant lasted for 4 years plus an extension, and involved researchers from three universities:

University of Pennsylvania	University of Iowa	Portland State University
Stephanie Weirich	Aaron Stump	Tim Sheard
Chris Casinghino	Harley Eades	Ki Yung Ahn
Vilhelm Sjöberg	Peng (Frank) Fu	Nathan Collins
	Garrin Kimmell	

The closeness of collaboration varied of the course of the project. We began by getting into a conference room to think about what features a dependently typed language should have (nontermination, erasure), and how to combine them in a core language. That language eventually evolved into Zombie Core. In later years the collaboration became looser, with different people focusing on particular subproblems.

Some of the material in this thesis therefore owes credit to other team members. The type system for Zombie Core includes design ideas from the entire Trellys team. The particular subset of it that is formalized in Chapter 3 was first presented and proved type safe in a workshop paper [120] where I was the lead author. So the precise formulation of the typing rules of the core calculus are due to me, as is the type safety proof. At the same time, the Trellys team implemented a typechecker for the core language (mostly done by Garrin Kimmel) and wrote about ten examples in it (mostly done by Nathan Collins), e.g. natural number division and vector append. This gave us an idea of how many annotations the core language by itself requires. I then extended the typechecker to implement the Zombie surface language described in Chapter 5, and used that implementation to write the examples described in Chapter 2.

The most closely related Trellys research was done by Chris Casinghino, who also wrote a thesis about the theory of the Zombie Core language [30]. As described above, he focuses on the termination-checking rules. In Chapter 7, I also discuss termination in Zombie, so the material in that chapter overlaps with Casinghino's work. In particular, we first presented the type system in Section 7.3 in a paper (Casinghino, Sjöberg, and Weirich [31]) as an incremental step towards a normalization proof for the full core language. (The same system is also presented, as a warm-up excercise, in Chapter 4 of Casinghino's thesis.) The design of these typing rules were done by all three paper authors jointly, but Casinghino was mainly responsible for the termination proof. In Chapter 7 I define a new type system (Section 7.4) and prove a theorem relating it to the old one—these are new contributions due to me.

1.3 Prior publications

The material in this thesis draws on the following papers.

- Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In James Chapman and Paul Blain Levy, editors, *MSFP '12: Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *EPTCS*, pages 112–162. Open Publishing Association, 2012.

This paper is the basis for Chapter 3. It formalizes a subset of the Zombie core language and proves type safety.

- Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 275–286, 2010. doi: 10.1145/1706299.1706333.

- Vilhelm Sjöberg and Aaron Stump. Equality, quasi-implicit products, and large eliminations. In *ITRS 2010: Proceedings of the 5th workshop on Intersection*

Types and Related Systems, 2010. doi: 10.4204/EPTCS.45.7.

- Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *PLPV '12: Proceedings of the sixth workshop on Programming languages meets program verification*, 2012. doi: 10.1145/2103776.2103779.

These three papers also prove type safety for dependent languages with nontermination, but the languages are not as feature-rich as the one in Chapter 3. The two first can be seen as gradually working towards the full core language, while the third one applies similar proof techniques to study language interoperability.

For the purposes of this thesis, the interesting feature is that all four papers listed above use slightly different interpretations of propositional equality in terms of operational semantics, so together they span a gamut of ways to handle equality in a progress-and-preservation style type safety proof. We discuss the different options in Chapter 4.

- Vilhem Sjöberg and Stephanie Weirich. Programming up to congruence. In *POPL '15: 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015. doi: 10.1145/2676726.2676974.

This paper is the basis for Chapter 5.

- Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. Termination casts: A flexible approach to termination with general recursion. In *PAR '10: Proceedings of the Workshop on Partiality and Recursion in Interactive Theorem Provers*, 2010. doi: 10.4204/EPTCS.43.6.
- Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *POPL '14: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

These two papers propose different ways to include optional termination-checking into a core language; only the second one has a normalization proof. Chapter 7 proves that (slight variants of) the two calculi can type exactly the same programs, and also contains a simpler proof of normalization.

- Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *PLPV '12: Proceedings of the sixth workshop on Programming languages meets program verification*, 2012. doi: 10.1145/2103776.2103780.

This paper reports on Sep^3, another language to come out of the Trellys project. For the purposes of this thesis, the most interesting result is a new language construct called *termination-case*, which is used to reason about potentially nonterminating programs. It turns out that it would *not* be sound to add termination-case to Zombie, for reasons explained in Section 7.7.1.

Chapter 2

Zombie by example

At its most basic, Zombie is yet another functional programming language. The syntax is inspired by Haskell with some ML influences For example, the following code defines a datatype of binary search trees, and a function `member` which checks whether an element is present in the tree.

```
data Tree (a:Type) : Type where
  EmptyTree
  BranchTree of (t1 : Tree a) (x : a) (t2 : Tree a)

prog member : [a:Type] ⇒ Ord a ⇒ (x : a) → (t : Tree a) → Bool
member = λ[a] order x. rec member t =
  case t of
    EmptyTree → False
    BranchTree t1 y t2 → case (ordLt x y) of
                            True → member t1
                            False → case (ordLt y x) of
                                       True → member t2
                                       False → True
```

The syntax of Zombie datatype declarations looks a bit like ML. The datatype `Tree` contains two constructors, `EmptyTree` which takes no arguments, and `BranchTree` which takes three arguments named `t1`, `x`, and `t2`. Like in OCaml, datatype constructors need to be fully applied when they are used.

The function `member` illustrates two ways to form functions: by λ-expressions and by recursive functions `rec` $f\ x.a$. Because the first three arguments stay constant throughout the recursion we choose to bind those by λ-expressions, and then use a single-argument recursive function. The `rec`-expression binds a variable `member`, which is distinct from the top-level declaration.

The function member is polymorphic, because it takes a type a as an argument. The square brackets around a indicates that a is *computationally irrelevant*. That is, a is only used to typecheck the program, but is not used at runtime, so the compiler can erase it.

In order to reduce clutter, we mark the arguments a and ord as *inferable*, using the fat arrow ⇒. Whenever the function member is called later in the program, the typechecker will create two unification variables corresponding to these arguments, and try to infer what they should be using unification. (Unification will not be able determine the order parameter, so this example relies on a rather hacky second pass that fills in uninstantiated unification variables using any variable in context. We further discuss unification-based inference in Chapter 6.)

The keyword prog indicates the main contribution of this thesis: it tells the typechecker that member is a potentially nonterminating program, which can be written using general recursion.

A simple function like member is also easy to write in a way that lets the typechecker verify that the function is always terminating. We use keyword log to say that the definition does not use any features which can cause nontermination. (The word log alludes to the fact that terminating functions correspond to logical proofs under the "propositions as types" principle.)

```
log member’ : [a:Type] ⇒ Ord a ⇒ (x : a) → (t : Tree a) → Bool
member’ = λ[a] order x. ind member’ t =
  case t [t_eq] of
    EmptyTree → False
    BranchTree t1 y t2 → case (ordLt x y)  of
                            True → member’ t1 [ord t_eq]
                            False → case (ordLt  y x) of
                                       True → member’ t2 [ord t_eq]
                                       False → True
```

To make the typechecker accept the function as terminating we have to use a structurally recursive function ind $f\ x.a$ instead of a generally recursive function rec $f\ x.a$. The difference is that with ind every recursive call takes an extra argument, for example we write member’ t1 [ord t_eq] instead of member t1. The expression inside the brackets is a proof that t1 is a smaller value than t, so the recursion is decreasing.

In order to manufacture such a proof, we use the fact that t1 is an immediate subterm of t. And this in turn uses a novel feature of the Zombie case-expressions. Instead of case t of ... we write case t [t_eq] of .... This binds a variable t_eq in each of the braches, where the type of the variable records the result of the pattern match. For example, in the second branch of the case expression, the context contains the assumption

```
    t_eq : t  = BranchTree t1 y t2
```

The ord (which is a built-in construct in Zombie) converts such an equation into a proof that t1 is smaller than t. (The details of the typing rule are shown in Section 7.2.1.)

2.1 Dependently typed programming

The above code did not yet use dependent types. As an example of dependently typed programming, we can define two datatypes which encode the propositions "the element x occurs in the tree t", and "the tree t satisfies the binary-search-tree invariant".

```
data InTree (a:Type) (x : a ) (t : Tree a) : Type where
  InHere  of (t1 : Tree a)
             (t2 : Tree a)
             (t = BranchTree t1 x t2)
  InLeft  of (t1 : Tree a)
             (y : a )
             (t2 : Tree a)
             (InTree a x t1)
             (t = BranchTree t1 y t2)
  InRight of (t1 : Tree a)
             (y : a )
             (t2 : Tree a)
             (InTree a x t2)
             (t = BranchTree t1 y t2)

data IsBST (a:Type) (t : Tree a) (order : Ord a) : Type where
  IsBSTEmpty of (t = (EmptyTree : Tree a))
  IsBSTBranch of (t1 : Tree a) (x : a) (t2 : Tree a)
                 (t = BranchTree t1 x t2)
                 (IsBST a t1 order)
                 ((y : a) → InTree a y t1 → ordLt y x = True )
                 (IsBST a t2 order)
                 ((y : a) → InTree a y t2 → ordLt x y = True )
```

Unlike Tree, the datatypes InTree and IsBST are dependent, because in the constructors, the types of later constructor arguments mention ("depend on") the names of previous arguments. For example, the third argument of InHere is an equation which mention the first two arguments t1 and t2.

We can use these declarations to define a more strongly typed function `memberS`. Instead of just returning a boolean, we make it return `Maybe (InTree a x t)`. That is, we do not have to trust the function that an element is in the tree—it returns a proof that the element is there. This stronger type is enough to catch some simple bugs, e.g. if the programmer accidently swapped the arguments in one of the calls to `ordLt`, the last line of `memberS` would not typecheck.

```
prog memberS : [a:Type] ⇒ Ord a ⇒ (x : a) → (t : Tree a)
                → Maybe (InTree a x t)
memberS = λ [a] order x. rec memberS t =
  case t of
    EmptyTree → Nothing
    BranchTree t1 y t2 →
      case (ordLt x y) of
         True → case (memberS t1) of
                   Just p  → Just (InLeft t1 y t2 p _)
                   Nothing → Nothing
         False → case (ordLt  y x) of
                    True → case (memberS t2) of
                              Just p  → Just (InRight t1 y t2 p _)
                              Nothing → Nothing
                    False →
                      let _ = ordLtAntiSymm [a] order x y _ _ in
                        Just (InHere t1 t2 _)
```

The implementation is mostly unsurprising, except that the constructors of `InTree` require proofs of equations. In particular, the last two lines refer to a lemma that states that $x \not< y \wedge y \not< x \implies x = y$. The line `let _ = ordLtAntiSymm ... in ...` introduces a new equation of type `x=y` into the context, and the underscore tells the typechecker to pick an arbitrary fresh name for it. To produce the last argument for `InHere` we then use a feature of Zombie that lets us write just an underscore for a proof of an equation, as long as the equation is provable using assumptions in the context.

The function `memberS` is an example of *internal* verification. That is, we encode the specification of the function (that it is correct about membership claims) into the type of the function itself. Instead of a separate proof about a function, we have a function which returns a proof.

The ability to verify programs in the internal style is one of the big advantages of using dependent types for program verification, instead of other logical frameworks such as first-order or higher-order logics. To write a separate proof about the correctness of `member` we would use structural induction over the tree it takes as input. That induction follows exactly the same pattern as the recursion over the tree that the function itself carries out. By incorporating the correctness property into the function type itself we avoid such duplication—the “induction hypothesis” `p` is conveniently available after each recursive call. What enables this is the dependent type

```
(x : a) → (t : Tree a) → Maybe (InTree a x t)
```

where the return type of the function mentions the arguments `x` and `t`.

2.2 External proofs about programs

However, the internal verification style also has drawbacks. In particular, it does not scale well to more ambitious verification efforts that aim to prove full correctness.

For example, the above property alone does not characterize `member`; we also want to know the converse, that if `member` returns `False` and the tree satisfies the search-tree invariant, then the element really is not present in the tree. For more complicated functions, full correctness may include even more theorems. In general, it becomes clumsy to require the programmer to think of all correctness properties ahead of time, bake them into the function type, and mix proofs of all of them into the function definition. In such cases, it is better to use *external* reasoning. That is, we first write the simply-typed function `member`, and later prove separate theorems about the function.

The feature that enables external proofs in dependent languages is that the compiler can evaluate expressions during typechecking. In Zombie, this is done by the equality constructor `join`. The expression `join` is a proof of an expression $a = b$ if both a and b reduce to some common expression c. For example, we can prove that looking up 5 in a tree representing the set $\{2, 4, 5, 7\}$ returns `True`:

```
log example :
    (member 5
           (BranchTree (BranchTree EmptyTree 2 EmptyTree)
                        4
                        (BranchTree (BranchTree EmptyTree 5 EmptyTree)
                                    7
                                    EmptyTree)))
    = True
example = join
```

In order to check this expression, the typechecker will reduce the two sides of the equation (`(member 5 ...)` and `True`), and note that they both evalute to the same thing (`True`).

Note that example is classified as log. The definition is already a value (join), so it certainly terminates. On the other hand, the type of example involves member, which was defined using general recursion. This illustrates an important property of Zombie, which we call *freedom of speech*: although proofs cannot themselves use general recursion, they are allowed to *refer* to arbitrary programmatic expressions. By contrast, in so-called value-dependent languages this theorem cannot even be stated, because non-value expressions are not allowed in types. (We discuss this further in Section 4.2.)

While we know that join will not diverge at runtime, we also have to worry about the typechecker diverging when checking it. In Zombie, the typechecker will reduce the two sides until they reach a value, or at most for a certain number of steps (by default 1000), and signal a type error if they do not rearch a common reduct by then. The programmer can override this to give more fuel for particular equations by writing e.g. join 5000 (reduce for up to 5000 steps). Because Zombie types may involve nontermination, a general design principle is that the programmer should always be in control over when expressions get reduced during typechecking, e.g. by using join. Unlike other dependently typed languages, the Zombie typechecker will not *automatically* reduce expressions that appear in types.

Most interesting theorems about programs are proved by induction. For example, if we were writing an informal pen-and-paper proof about the property of member that we proved internally above, we might begin as follows.

> **Claim**: For all x and t, if member x t = True, then x is in the tree t.
>
> **Proof**: By induction on the structure of t. Consider the case when t = (BranchTree t1 y t2). By the definition of member, we have
>
> ```
>   member x (BranchTree t1 y t2)
> = case (ordLt x y)  of
>     True → member x t1
>     False → case (ordLt y x)  of
>                True → member x t2
>                False → True
> ```
>
> Now there are two cases, depending on whether (ordLt x y) is true or false. If it is true, then we know
>
> ```
>   case True of
>     True → member x t1
>     False → case (ordLt y x)  of
>                True → member x t2
>                False → True
> = member x t1
> ```

> so by transitivity we have `member x t1 = member x t = True`. By the induction hypothesis for `t1` we know that `x` is in `t1`, so therefore it is also in `t`. (Several more cases of the proof omitted.)

Under "propositions as types", inductive proofs correspond to structurally recursive functions. We can express the above proof as a Zombie function using `ind` and `join` as follows:

```
log member_In1 : (a:Type) → (order : Ord a) → (x : a) → (t : Tree a)
                  → member x t = True  →  InTree a x t
member_In1 = λa order x. ind member_In1 t = λ isMember.
  case t [t_eq] of
    BranchTree t1 y t2 →
      let _ = (pjoin : member x (BranchTree t1 y t2)
                  = ((case (ordLt x y)  of
                        True → member x t1
                        False → case (ordLt y x)  of
                                  True → member x t2
                                  False → True) : Bool)) in
      case (ordLt x y)  of
         True →
           let _ = (pjoin  : ((case True of
                                 True → member x t1
                                 False → case (ordLt y x) of
                                           True → member x t2
                                           False → True) : Bool)
                         = member x t1) in
           let IH = (member_In1 t1 [ord t_eq] _) in
               InLeft t1 y t2 IH _
         False →  -- ... several lines omitted
    EmptyTree → -- ... several lines omitted
```

The function follows exactly the same plan as the informal proof; induction corresponds to `ind`, case analysis to `case`, and appeals to the definition of `member` to `pjoin`. (The difference between `join` and `pjoin` is that the latter asks the typechecker to reduce using parallel reduction instead of plain CBV reduction. We discuss this further in Section 4.1.3.) In order to deal with equational reasoning we use the same idiom as in `memberS`, by first introducing equations into the context with fresh names (`let _ = ...`), and then writing underscores for proofs. In particular, the recursive call to `member_In1` requires a proof that `member x t1 = True`, and the underscore builds the proof using transitivity.

The Zombie term looks a bit daunting, particularly the big `pjoin`-expressions. To some extent this is an unavoidable problem when combining reasoning using equations with reasoning by reduction. In particular, it would not do to reduce the expression `member x (BranchTree t1 y t2)` *too* much, by starting to step into the implementation of `ordLt` (which projects out a comparison function from the record `order` and applies it), because then the hypothesis `(ordLt x y) = True` would no longer apply. A proof of the same property in Coq or Agda would also need care from the programmer to simplify expressions just enough to make them match the known equations, although a Coq programmer would use tactics to build the proof term rather than write it manually.

The Zombie implementation does however include a feature that can help here. The keywords `smartjoin` and `unfold` ask the typechecker to use *reduction modulo congruence*, i.e. to reduce expressions in a "smart" way using equation assumptions in the context. This is implemented entirely in the elaborator, and the generated core term will use multiple `let`s and `join`s similar to the program above. (We describe the details in Section 5.6.3.) The implementation of this feature is one of the least polished parts of Zombie, and we have not studied its theory, but when it works it is very helpful indeed. For example, the proof of `member_In1` can be written much more compactly, as follows.

```
log member_In1 : (a:Type) → (order : Ord a) → (x : a) → (t : Tree a)
                  → member x t = True  →  InTree a x t
member_In1 = λa order x. ind member_In1 t = λ isMember.
  case t [t_eq] of
    BranchTree t1 y t2 →
      case (ordLt x y)  of
        True →
          let _ = (smartjoin : member x t = member x t1) in
          let IH = (member_In1 t1 [ord t_eq] _) in
              InLeft t1 y t2 IH _
        False →
          case (ordLt y x) of
            True →
              let _ = (smartjoin : member x t = member x t2) in
              let IH = (member_In1 t2 [ord t_eq] _) in
                InRight t1 y t2 IH _
            False →
              let _ = ordLtAntiSymm [a] order x y _ _ in
                InHere t1 t2  _
    EmptyTree → unfold (member x t) in
                   contra (_ : True = False)
```

We can also write a similar proof of the converse property, this time by structural recursion on the witness of `InTree a x t`.

```
log member_In2 : (a:Type) → (order : Ord a) → (x : a) → (t : Tree a)
               → IsBST a t order → (InTree a x t)→ member x t = True
```

We show only the type and omit the proof itself (which can be found in the Zombie test suite).

2.3 Programming up to congruence

As we have seen in the above examples, the proof obligations of programming in Zombie are somewhat different from Agda or Coq. On one hand, in order to deal with nontermination, equations that are true by reduction must be introduced by explicit `join`-expressions, wheras in other languages some such proofs can be omitted because types are considered up to $\beta\eta$-convertibility. But on the other hand Zombie is able to automatically construct easy proofs of equations using equality assumptions in the context. This is not just used in places where the programmer explicitly writes an underscore; it is a definitional equality, so the type system treats every type modulo the equations in the context.

There is a precise definition of which proofs are considered "easy". The definitional equality is the *congruence closure* of the equations in the context, i.e. the equations which are provable by symmetry, reflexivity or transitivity, or by injectivity of constructors, or by rewriting a subexpression of a larger expression. (A formal definition is shown in Figure 5.2). Congruence closure is a standard notion in automatic theorem proving for first-order logic, although we adapted it to fit with dependent types (Chapter 5).

So Zombie and conventional languages like Agda differ about what annotations are needed, but neither system provides a strictly stronger equational theory than the other. As an illustration of the difference, we implement first-order unification in both Agda and Zombie (Figures 2.1 and 2.2), taking care to make the two programs correspond as closely as we can. For this example, the term language is the simplest possible, consisting only of binary trees constructed by `branch` and `leaf` and possibly containing unification variables, `var`, represented as natural numbers. We also use a type `Substitution` of substitutions, which are built by the functions `singleton` and `compose`, and applied to terms by `ap`. Finally, we need some lemmas about substitutions. The types are as follows (the definitions are elided):

```
data Term : Type where
  leaf
  branch of (t1 : Term) (t2 : Term)
  var of (x : Nat)

Substitution : Type
```

```
empty : Substitution
singleton : Nat → Term → Substitution
ap : Substitution → Term → Term
compose : Substitution → Substitution → Substitution

apCompose : (s1 s2 : Substitution) → (t : Term) →
                    ap (compose s1 s2) t = ap s1 (ap s2 t)
-- determining whether a variable appears in a term
isin : (x: Nat) → (t : Term) → Dec (In x t)
varSingleton : (x : Nat) → (t : Term) → t = ap (singleton x t) (var x)
singletonNotIn : (t : Term) → (x: Nat) → (s : Term)
               → (((In x t)@log) → Void) → ap (singleton x s) t = t
```

First-order unification is an interesting opportunity for lightweight verification. Proving that `unify` terminates is difficult because the termination metric involves not just the structure of the terms but also the number of unassigned unification variables. (For example, see McBride [83].) To save development effort, a programmer may elect to prove only a partial correctness property: *if* the function terminates then the substitution it returns is a unifier.

In other words, if the `unify` function returns, it either says that the terms do not match, or produces a substitution `s` and a proof that `s` unifies them. We write the data structure in Zombie as follows (the Agda version is similar):

```
data Unify (t1 : Term) (t2 : Term) : Type where
  nomatch
  match of (s : Substitution) (pf : ap s t1 = ap s t2)
```

With the scene thus set, we can compare the Agda and Zombie implementations to see the effect of programming up-to-congruence instead of up-to-β. For example, in the case when the function successfully unifies `(branch t11 t12)` and `(branch t21 t22)` and returns the unifier `(compose s' s)`, it needs to construct a proof of equality. Both the Zombie and the Agda version use the lemma `apCompose` here. However, they differ in what parts of the proof can be left implicit. The Agda definitional equality automatically includes the equation

```
ap s' (ap s (branch t11 t12)
  ≡ branch (ap s' (ap s t11)) (ap s' (ap s t12))
```

which the Zombie programmer has to explicitly request using an unfold statment. On the other hand, Zombie can automatically conclude that two `branch`-terms are equal if their subterms are, and to use symmetry and transivitity, while the Agda programmer invokes lemmas fom the Agda standard library:

```
cong₂ : ∀ {A B C : Set } (f : A → B → C) {x y u v}
```

```
        → x ≡ y → u ≡ v → f x u ≡ f y v
sym : ∀ {A : Set} {x y : A} → x ≡ y → y ≡ x
trans : ∀ {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
```

2.3.1 Smart case

Congruence closure is directly helpful to infer proofs of equations. Having it available also has a subtler effect on the language design, because it allows us to make Zombie's type rules for dependent pattern matching both simple and powerful. As we saw above, in a `case`-expression

```
case a of
   d1 x y ⇒ body1
   d2 x y ⇒ body2
```

when checking `body1` the typing context is extended with an equation `a = d1 x y`. This idea is known as *smart case* [7], and lets the typing rule exactly express what branch was taken.

The smart-case rule is attractive because Zombie will use the equation automatically. Languages like Coq and Agda do not extend the context with an equation, but if the above expression is checked against some type A, the typechecker will see whether A contains a as a subexpression. If so, it will check `body1` against a different type A' where all occurences of a have been replaced with $d_1\ x\ y$. In simple cases, this can typecheck the same examples as Zombie's rule, without the programmer having to write an explicit proof using an equality assumption.

However, in more complicated examples the replacement-based rule can be clumsy. For example, A may contain multiple occurences of a and we only want to replace some of them. Or a may not occur in A, but will show up if we later reduce a subexpresion of A. Agda deals with this by the "inspect on steroids" trick, using a datatype constructor `[_]` to make some parts of the type opaque to the typechecker. A fun puzzle[3] which illustrates how this can sometimes be tricky, is to define a the operation `snoc` which appends an element to the end of a list, and then try to prove that it is injective using only pattern matching.

Figure 2.3 compares (parts of) of the proofs in Zombie and Agda. The Zombie proof `snoc_inv` is quite pedestrian: when both lists are nonempty, the proof argument can be used to derive that `x = y` (using the injectivity of `Cons`), and the recursive call shows that `xs' = ys'`. Congruence closure both puts these together in a proof of `Cons x xs' = Cons y ys'` and supplies the necessary proof for the recursive call. The pure pattern matching Agda solution `snoc-inv` is not long, but it is not at all

[3] Posed by Eric Mertens on `#agda`.

```
{-# NO_TERMINATION_CHECK #-}
unify : (t1 t2 : Term) → Unify t1 t2
unify leaf leaf = match empty refl
unify leaf (branch t2 t3) = nomatch
unify (branch t1 t2) leaf = nomatch
unify (branch t11 t12) (branch t21 t22)
      with unify t11 t21
...    | nomatch = nomatch
...    | match s p with unify (ap s t12) (ap s t22)
...               | nomatch = nomatch
...               | match s' q
  = match (compose s' s)
       (trans (apCompose (branch t11 t12))
         (trans (cong₂ (λ t1 t2 →
                    branch (ap s' t1) t2) p q)
            (sym (apCompose (branch t21 t22)))))
unify t1 (var x) with (x is∈ t1)
...              | no q
  = match (singleton x t1)
       (trans (singleton-∉ t x t q)
          (varSingleton x t))
...              | yes p with t
...                      | var y
  = match empty (cong var (sym (invvar p)))
...                      | _
  = nomatch
unify (var x) t2 with unify t2 (var x)
...              | nomatch   = nomatch
...              | match s p = match s (sym p)
```

Figure 2.1: First-order unification in Agda

```
prog unify : (t1 t2 : Term) → Unify t1 t2
rec unify t1 = λ t2 . case t1, t2 of
    leaf, leaf → match empty _
    leaf, branch _ _ → nomatch
    branch _ _, leaf → nomatch
    branch t11 t12, branch t21 t22 →
      case (unify t11 t21) of
        nomatch → nomatch
        match s p → case (unify (ap s t12) (ap s t22)) of
            nomatch → nomatch
            match s' _ →
              unfold (ap s' (ap s t1)) in
              unfold (ap s' (ap s t2)) in
              let _ = apCompose s' s t1 in
              let _ = apCompose s' s t2 in
                 match (compose s' s) _
    _ , var x → case (isin x t1) of
        no q  →
           let _ = varSingleton x t1 in
           let _ = singletonNotIn t1 x t1 q in
             match (singleton x t1) _
        yes _ → case t1 of
           var y → let [_] = invvar x y p in
                    match empty _
           _     →
                      nomatch
    var x, _ →  case (unify t2 (var x)) of
        nomatch → nomatch
        match s p → match s _
```

Figure 2.2: First-order unification in Zombie

straightforward, and requires advanced knowledge of Agda idioms. Alternatively, the reasoning used in the Zombie example is also available in Agda, as in the definition of `snoc-inv'`. However, this version requires the use of helper functions to prove that cons is injective and congruent.

```
-- Solution in Zombie
log snoc_inv : (xs ys : List A) → (z : A)
               → ((snoc xs z) = (snoc ys z)) → xs = ys
ind snoc_inv xs = λ ys z pf.  case xs [xeq], ys of
      Cons x xs' , Cons y ys' →
         let _ = (smartjoin : (snoc xs z) = Cons x (snoc xs' z)) in
         let _ = (smartjoin : (snoc ys z) = Cons y (snoc ys' z)) in
         let _ = snoc_inv xs' [ord xeq] ys' z _ in
           _
      ...

-- Solution in Agda using pattern matching
snoc-inv : ∀ xs ys z → (snoc xs z ≡ snoc ys z) → xs ≡ ys
snoc-inv (x :: xs') (y :: ys') z pf
  with (snoc xs' z) | (snoc ys' z)
    | inspect (snoc xs') z | inspect (snoc ys') z
snoc-inv (.y :: xs') (y :: ys') z refl
 | .s2 | s2 | [ p ] | [ q ] with (snoc-inv xs' ys' z (trans p(sym q)))
snoc-inv (.y :: .ys') (y :: ys') z refl
 | .s2 | s2 | [ p ] | [ q ] | refl = refl
...

-- Alternative Agda solution based on congruence and injectivity
cons-inj1 : ∀ {x xs y ys} → ((x :: xs) ≡ y :: ys) → x ≡ y
cons-inj1 refl = refl

cons-inj2 : ∀ {x xs y ys} → x :: xs ≡ y :: ys → xs ≡ ys
cons-inj2 refl = refl

snoc-inv' : ∀ xs ys z → (snoc xs z ≡ snoc ys z) → xs ≡ ys
snoc-inv' (x :: xs') (y :: ys') z pf =
  cong₂ _::_ (cons-inj1 pf) (snoc-inv' xs' ys' z (cons-inj2 pf))
...
```

Figure 2.3: Pattern matching can be tricky in Agda

2.4 Lightweight verification: a DPLL SAT-solver

In the unification example above, we noted that proving that a program terminates might not be the best use of the programmer's time. An application which shows this even clearer is a SAT-solver for propositional logic. More and more programs rely on SAT-solvers, and the solvers themselves are complicated pieces of software, so constructing a formal proof that they return the correct answer is valuable. On the other hand, showing that the solver always terminate is both very subtle (to prove that a clause-learning solver terminates, one must prove that the clauses it learns eventually rule out all possible assignments), and uninteresting in practice (typically it will run out of memory long before it terminates).

Most modern SAT-solvers use the clause-learning algorithm pioneered in zChaff [148]. A realistic solver is too big to be written as part of this thesis, but we can illustrate the interplay between termination-checked and nonterminating code by verifying a simple Davis-Putnam-Logemann-Loveland (DPLL) solver. The DPLL algorithm is an immediate ancestor of the clause-learning algorithm, so it is a good illustration of how to structure such proofs.

Recall that a boolean formula is a conjunction of *clauses*, each clause is a disjunction of *literals*, and a literal is either a plain or a negated variable. The SAT problem is to either find an assigment from variables to booleans that makes the formula true, or prove that there exists no such assignment.

In our implementation we represent variables as bounded natural numbers (`Fin`), literals as pairs (`Times`) of a variable and a boolean, and assignments as vectors of boolean. Formulas are represented as lists of lists of literals, and the `interp` function evaluates a formula under a given assignment.[4] We take advantage of dependent types by indexing the formula by the number of variables, so the typechecker can check that the vector access is in bounds.

```
log Formula : (n:Nat) → Type
Formula = λn. List (List (Times (Fin n) Bool))

log interp_lit : [n:Nat] ⇒ Vector Bool n
                  → (Times (Fin n) Bool @log) → Bool
interp_lit [n] assign lit = case lit of
                              Prod i b → bool_eq b (lookup i assign)

log interp_clause : [n:Nat] ⇒ Vector Bool n →
                      (List (Times (Fin n) Bool) @log) → Bool
```

[4] For reasons explained in Section 7.7.3, the functions `interp_lit` and `interp_clause` have a spurious `@log` in their types. This qualifier makes no difference semantically here, but is needed due to a limitation of type inference.

```
interp_clause [n] env clause = any (interp_lit env) clause

log interp : [n:Nat] ⇒ Vector Bool n → Formula n → Bool
interp = λ [n] env  .
  unfold (Formula n) in all (interp_clause env)
```

While searching for a solution, the solver manipulates *partial assigments*, which specify the values of only some of the variables. In our program we represent these as `Vector (Maybe Bool) n`. It easy to treat a partial assignment as an assigment, by picking some arbitrary value (`True`, say) for the missing variables. In order to state the invariants of the algorithm, we say an assignment ϕ *extends* a partial assignment ψ if it agrees whereever ψ is defined.

```
log extend : [n:Nat] ⇒ Vector (Maybe Bool) n → Vector Bool n
extend = λ [n] xs . vmap (maybe True) [n] xs

log Extends : (n:Nat) ⇒ Vector (Maybe Bool) n → Vector Bool n → Type
Extends n psi phi =
  (i : Fin n) → (b:Bool)
   → (lookup i psi = (Just b))
   → (lookup i phi = b)
```

Now we can state the specification of the solver. When given an argument `partial` it should return either an assignment together with a proof that the formula evaluates to true (`Sat n formula`), or a proof that there is no way to extend `partial` into a satisfying assignment (`Unsat n formula partial`). At the beginning of the run we will ask for a solution extending the empty assignment, i.e. for any solution at all.

```
data Sat (n:Nat) (formula : Formula n) : Type where
  SAT    of (partial : Vector (Maybe Bool) n)
             [_ : interp (extend partial) formula = True ]

data Unsat (n:Nat) (formula : Formula n)
           (phi : Vector (Maybe Bool) n) : Type where
  UNSAT  of [_ : (phi' : Vector Bool n)
                    → Extends phi phi'
                    → (interp phi' formula = False )]
```

So how do we find the solution? Davis, Putnam, Logemann, and Loveland [42, 43] proposed two rules to use:

Unit propagation If there is some clause which is currently unsatisfied and which only contains one unassigned variable, set that variable to make the clause true. This rule corresponds to inference using implications. For example, an implication $(p \wedge q \implies r)$ is encoded as the clause $(\neg p \vee \neg q \vee r)$. So if at some point we learn that p and q are true, then r is the only remaining unassigned variable in the clause and we can set r to true as well.

Branch Otherwise, pick some currently unassigned variable and guess its value. If the search later fails, we backtrack and try the other value instead.

(The original presentation also included a third rule, **pure literal elimination**. However, this is expensive to implement, so modern solvers usually omit it, and we do not include it in our program.)

We use two helper functions that implement these rules, which each take a formula and a partial assignment as inputs. The function `setunits` finds all unit clauses in the formula, and returns a new assignment with the corresponding variables set. The function `partial_interp` either notes that the formula is alread satisfied, or that all variables are assigned and it is unsatisfied, or it picks some currently unassigned variable. Its return type is `Or (Sat n f) (Or (Unsat n f partial) (Fin n))`.

With these two helper function, we can implement the solver itself:

```
prog dpll : [n:Nat]
            ⇒ (formula : Formula n)
            → (partial : Vector (Maybe Bool) n)
            → (Or (Sat n formula) (Unsat n formula partial))
rec dpll [n] = λ formula partial .
 let upartial = setunits formula partial in
   case (partial_interp upartial formula) [s_eq] of
     InL sat  → InL sat
     InR (InL unsat) → InR (unsat_units formula partial unsat)
     InR (InR i) →
            case (dpll formula
                    (set i (Just True) upartial)) of
              InL sat → InL sat
              InR unsat1   →
                 case (dpll formula
                        (set i (Just False) upartial)) of
                   InL sat → InL sat
                   InR unsat2 →
                     InR (unsat_units formula partial
                           (unsat_branch i formula upartial
                                         unsat1 unsat2))
```

Most of the function directly implements the backtracking search. Because the function is in `prog` this can be done using general recursion, just as in any other programming language. The extra work to verify correctness shows up at the end of the function, in the call to `InR`. This is when the search failed and we must return a proof that the formula is unsatisfiable. From the recursive calls we have available two proofs `unsat1` and `unsat2`, stating that there are no solutions after unit-propagating and guessing either `True` or `False` for the variable `i`. So to complete the proof we need two *correctness lemmas* about unsatisfiablility:

```
log unsat_units : [n:Nat] ⇒ (f : Formula n)
                  → (p : Vector (Maybe Bool) n)
                  → Unsat n f (setunits f p)
                  → Unsat n f p

log unsat_branch : [n:Nat] ⇒ (i : Fin n) → (f : Formula n)
                  → (phi : Vector (Maybe Bool) n)
                  → Unsat n f (set i (Just True) phi)
                  → Unsat n f (set i (Just False) phi)
                  → Unsat n f phi
```

A real SAT solver would be more complicated in various ways. It needs to use more efficient data structures in memory. It needs to pick the branch variable intelligently (`partial_interp` just picks the first unassigned variable), and use a better heuristic of how far to backtrack. It should use the efficient "two watched variables" scheme to find unit clauses (`setunits` scans through all the clauses each time). But note that none of these affect the correctness statement that we are proving, so they can all be done in `prog`. Finally, it should use clause learning to record more information when it backtracks. That would require an additional soundness lemma stating that the new clause is implied by the existing ones.

The function `dpll` illustrates many of the ideas that we discussed earlier in this chapter. First, it is an example of a program mostly defined by general recursion, without worrying about termination.

Second, it uses both internal and external reasoning. For the functions `dpll` and `partial_interp` it is very convenient to use the internal style (which is well suited to these kind of partial correctness statements for potentially nonterminating functions). But we also found that some lemmas, like `unsat_units`, are more natural to state and prove separately from the function they talk about.

Finally, `dpll` makes crucial use of *erasure*. The implementation of the function chains together many applications of the soundness lemmas to eventually build a proof of unsatisfiability. It would be very slow if those functions had to be actually invoked at runtime. But the proof arguments of `SAT` and `UNSAT` are marked erasable (square brackets). So at runtime, the `Unsat` type is isomorphic to a unit type, and the return type of `dpll`, `(Or (Sat n formula) (Unsat n formula partial))` is isomorphic to `Maybe (Vector (Maybe Bool) n)`. The reasoning about correctness is done completely statically, during the type checking.

2.5 Other Zombie examples

Combined, all the files in the Zombie test suite contain a little over 8000 lines of code. However, that includes a lot of duplication and boring unit tests that exercise the different rules of the type checker. The interesting programs are as follows.

Name	*LOC*	*Description*
`Prelude`, `List`, `Logic`, `Maybe`, `Fin`, `Product`, `Vectors`, `Arithmetic`, `ArithmeticDiv`	860	Standard datatypes like natural numbers and lists, and lemmas about them.
`NatElimination`	110	Written by Nathan Collins. Examples of lexicographic recursion (Section 7.2.1).
`Admiss2`	104	Smith's Paradox (Section 7.7.1).
`LessThanNat`, `LessThanNat_LT`, `LessThanNat_lt_eq_True`, `StrongNat`	1152	Written by Nathan Collins. Lemmas about the less-than relation on natural numbers, and a proof that course-of-values induction is derivable from ordinary induction.
`Paper`	396	Examples associated with our previous paper [31]. Includes an example of course-of-values induction using Zombie's built-in structural order type (Section 7.2.1).
`RLE`	112	An implementation of data compression by run-length encoding, and proof of correctness.
`Sort`	480	Written by Chris Casinghino. An implementation of merge sort, and an external proof that it terminates.
`BSTprog`	408	Lookup and insertion into binary search trees (Section 2.2 above).
`Unify2`	310	First-order unification (Section 2.3 above).
`Snoc`	46	The example in Section 2.3.1 above.
`UnsatUnit`	371	DPLL-style SAT solver (Section 2.4 above).
Total	4349	

The overall contribution of this thesis is to present one new point in the design-space of dependently typed languages, and evaluate how well it works. One kind of evaluation is what we will carry out in the following chapters: formalize various subsets of the language, and prove that it satisfies properties such as type safety, completeness of elaboration, and logical consistency. These results form a good sanity check by establishing that the design does not suffer from certain particularly bad flaws. However, they do not show that a language is useable for programming—the only way to do that is to actually try it and see what it feels like.

Of course, this presents a methodological problem, because developing realistically large programs takes a lot of time and resources, all the more so when using a prototype implementation with poor tooling. The example programs presented in this chapter could be called "program sketches". We hope to show that Zombie is usable for a wide-variety of tasks, but for each of these, we tackle a simplified version of the problem which is solvable in a few hundred lines. The small examples means that we can not make any claims about productivity or programming in the large, but the fact that the programs are possible to write at all is still informative. In fact, originally not all of them were—having a set of examples to refer to was invaluable when developing Zombie, and uncovered lots of design problems which were not obvious from just looking at the typing rules.

Chapter 3

Core language

Zombie is implemented as an elaborator from the surface language that the programmer write, into a core language. Designing a language this way helps protect against errors in the implementation, because the generated core language terms can be separately type checked, and mistakes in the elaborator will usually produce an ill-typed result. It also helps the language designer, because the design can be divided into two parts. The core language needs enough features to express all desired programs, but because it will not be manually written by the programmer it is okay for it to be very verbose. Conversely, as long as the core language is type safe and logically consistent, there is no need to worry about the soundness of the surface language.

Accordingly, this chapter has two goals. In Sections 3.1 to 3.8 we describe the what the core language looks like, with particular emphasis on the combination of features that sets Zombie apart from other dependently typed languages. Then in Section 3.9 we describe our proof that the language is type safe (we will come back to the issue of logical consistency in Chapter 7).

Programs in the core language are quite verbose, particularly when involving proofs of equations. In Chapter 5 we will discuss how to create a surface language which can automatically infer some such proofs.

Formalizing a subset of the language as a calculus In order to keep the type safety proof tractable, we do not formalize the full Zombie Core language as implemented. Instead, the calculus we define in this chapter has been simplified by omitting all typing rules to do with termination checking. Whenever the type checker in the full implementation requires an expression to be known-terminating, the type system defined in this chapter requires the expression to be a syntactic value. (This is the simplest possible scheme to check termination.) At the same time we simplify the rest of the language by removing features that are only used for termination checking. The full language contains a keyword `ind` defining structurally recursive functions, but in this core calculus we only need to treat (possibly nonterminating) recursive functions rec. Similarly, the full language contains an infinite hierarchy of universes $\mathsf{Type}_0, \mathsf{Type}_1, \ldots$, but because we do not worry about diverging expressions we can do even better, and collapse all these with the powerful $\mathsf{Type} : \mathsf{Type}$ axiom. Figure 3.1 (repeated from the Chapter 1) shows this schematically.

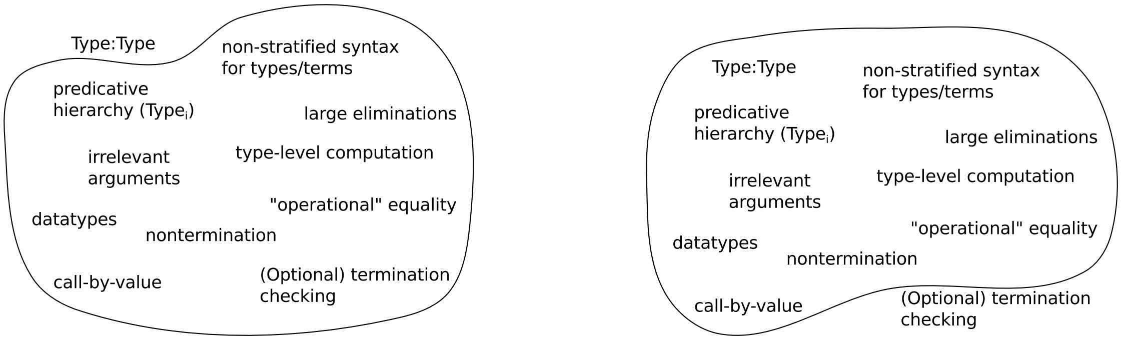

Figure 3.1: Features included in the full Zombie implementation (left), and in the core calculus defined in this chapter (right).

Relying only on value restrictions for termination checking is too simple to handle all the programs we want to write (see Section 3.8.3 and Section 7.1). In Chapter 7 we will go back and study Zombie's rules for termination checking, but in the context of a different smaller calculus. Meanwhile, the results in this chapter gives confidence that the rest of the typing rules are sound.

Novel features in Zombie The core calculus defined in this chapter combines full-spectrum dependent types with nontermination, so it is inconsistent as a logic but very expressive as a programming language. In this respect it is comparable to languages like Cayenne [10] or $\Pi\Sigma$ [9]. However, Zombie has a number of features which sets it apart from other dependently typed language, which we will describe in this chapter.

- Our operational semantics is call-by-value. As we describe in Section 3.3 this is a good choice when the language includes nontermination.
- To clarify the relation between surface language expressions and the core expressions they elaborate into, we distinguish certain subexpressions as *erasable annotations* (Section 3.2). A core language expression is just a surface expression with additional annotations.
- The idea of annotations also fits nicely with supporting *computationally irrelevant arguments*, i.e. function arguments that are used to state the types but do not affect runtime behavior. Similar to previous work on dependent languages [15, 91] we use the type system to track which arguments are irrelevant,

and use that information to aid compile-time reasoning. However, unlike previous languages we combine irrelevant arguments with nontermination, which reveals some new issues (Section 3.8).

- Propositional equality in this language is a primitive, and we treat the elimination form for equality (type casts) as computationally irrelevant (Section 3.6). Taking equality as a primitive instead of defining it as a datatype simplifies our typing rules for datatypes (Section 3.7), since the machinery for casting types does not need to be baked into the case-rule.
- Our treatment of equality is unusual in that it is completely "operational", i.e. based on the reduction behavior of expressions with little reference to their types. This has some drawbacks, but it means that we can easily support "very heterogeneous" type casts and n-ary type casts (Section 3.6).

Our proof of type safety (Section 3.9) is also a contribution. We use a fairly standard Wright-Felleisen progress/preservation proof extended with one extra lemma to deal with equality types. This is much simpler than previous work on dependent types with nontermination, which used denotational semantics or logical interpretations.

3.1 Syntax

The syntax of the core calculus appears in Figure 3.2. Terms, types and the sort Type are collapsed into one syntactic category, as in pure type systems [14]. By convention, we use lowercase metavariables a, b for expressions that are terms and uppercase metavariables A, B for expressions that are types. Some of the expressions are standard: the sort of types Type, variables, the usual dependent function type, recursive function definition, function application, datatype constructors, data constructors, and case-expressions. The two final lines of the figure deal with irrelevance and propositional equality; these will be explained in Sections 3.8 and 3.6. In this calculus all functions are defined as expressions $\mathsf{rec}\, f\ x.a$, but we write $\lambda x.a$ as syntactic sugar when x is not free in a.

Figure 3.2 also defines which expressions are values. Because our operational semantics (Section 3.3) is call-by-value, this grammar controls when one is allowed to reduce a β-redex. Also, several typing rules require that subexpressions be values in order to ensure that that they terminate—this restriction ensures type safety despite logical inconsistency. Notice that variables are classed as values; this is sound in a CBV language.

In this syntax, we use overline notation to denote a sequence of syntactic elements. For example, $\overline{a_i}$ is a list of expressions. Several expression forms bind expression variables, including arrow types and recursive functions. Each branch of a case expression binds

$$
\begin{array}{rcl}
x, y, f, g, h & \in & \text{variables} \\
D & \in & \text{datatypes, including } \mathsf{Nat} \\
d & \in & \text{constructors, including } 0 \text{ and } \mathsf{S} \\
i, j, \ell & \in & \text{natural numbers}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{expressions} & a, b, c, A, B, C & ::= & \mathsf{Type} \mid x \\
 & & \mid & (x\!:\!A) \to B \mid \mathsf{rec}\, f_A\ x.a \mid a\ b \\
 & & \mid & D\ \overline{A_i} \mid d_A\ \overline{a_i} \mid \mathsf{case}\ a_y\ \mathsf{of}\ \{\overline{d_j\ \Delta_j \Rightarrow b_j}^{j \in 1..k}\} \\
 & & \mid & \bullet(x\!:\!A) \to B \mid \mathsf{rec}\, f_A\ \bullet_x.a \mid a\ \bullet_b \\
 & & \mid & a = b \mid \mathsf{join}_\Sigma \mid a_{\triangleright b}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{strategies} & \Sigma & ::= & \ldots \\
\text{telescopes} & \Delta & ::= & \cdot \mid (x : A)\Delta \mid \bullet_{x:A}\ \Delta \\
\text{expression lists} & \overline{a_i} & ::= & \cdot \mid a\ \overline{a_i} \mid \bullet_a\ \overline{a_i}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{values} & v & ::= & \mathsf{Type} \mid x \\
 & & \mid & (x\!:\!A) \to B \mid \mathsf{rec}\, f_A\ x.a \\
 & & \mid & D\ \overline{A_i} \mid d_A\ \overline{v_i} \\
 & & \mid & \bullet(x\!:\!A) \to B \mid \mathsf{rec}\, f_A\ \bullet_x.a \\
 & & \mid & a = b \mid \mathsf{join}_\Sigma \mid v_{\triangleright b}
\end{array}
$$

Figure 3.2: Syntax

$$
\begin{array}{lcl}
|\mathsf{Type}| & = & \mathsf{Type} \\
|x| & = & x \\
|D\ \overline{A_i}| & = & D\ |\overline{A_i}| \\
|d_A\ \overline{a_i}| & = & d\ |\overline{a_i}| \\
|\mathsf{rec}\ f_A\ x.a| & = & \mathsf{rec}\ f\ x.|a| \\
|\mathsf{rec}\ f_A\ \bullet_x.a| & = & \mathsf{rec}\ f\ \bullet.|a| \\
|a\ b| & = & |a|\ |b| \\
|a\ \ \bullet_b\ | & = & |a|\ \bullet \\
|(x\!:\!A) \to\ B| & = & (x\!:\!|A|) \to\ |B| \\
|\bullet\ (x\!:\!A) \to\ B| & = & \bullet(x\!:\!|A|) \to\ |B| \\
|\mathsf{case}\ a_y\ \mathsf{of}\ \{pats\}| & = & \mathsf{case}\ |a|\ \mathsf{of}\ \{|pats|\} \\
|a = b| & = & |a| = |b| \\
|\mathsf{join}_\Sigma| & = & \mathsf{join} \\
|a_{\rhd b}| & = & |a|
\end{array}
$$

Figure 3.3: The erasure function $|\cdot|$

the given *telescope* of variables Δ in the body of that branch. Within the telescope, the variables are also bound in subsequent typing annotations. Both expression lists and telescopes can contain a mix of computationally relevant and irrelevant items, where the latter are shown as subscripted bullets ($\bullet$). We use the notation $\{a/x\}\,B$ to denote the capture-avoiding substitution of a for x in B and the notation $\mathsf{FV}\,(a)$ to calculate the set of free variables appearing in an expression.

The typing judgement is written $\Gamma \vdash a : A$. The typing contexts Γ are lists containing variable declarations and datatype declarations

$$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \mathsf{data}\ D\,\Delta\ \mathsf{where}\ \{\ \overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1..j}\ \}$$

3.2 Annotations and erasure

The syntactic elements appearing in subscripts are called *annotations* and are optional elements. When all annotations are present, the core typing judgement $\Gamma \vdash a : A$ is syntax-directed and type checking is trivially decidable. Given a context and an expression, there is a simple algorithm to determine the type of that expression (if any). We call an expression that includes all annotations an *annotated* expression.

Annotations are needed for type-checking, but are erased during compilation and require no run-time representation. Since this is a dependent language we want to reflect this fact in the type system: if two expressions are identical except in their annotations, they should be treated as equivalent when comparing types. We formalize this using the meta-operator $|a|$ (Figure 3.3), which removes all annotations from expressions. (This includes erasing binders, e.g. $|\mathsf{rec}\ f_A\ \bullet_x.a| = \mathsf{rec}\ f\ \bullet.|a|$, so strictly speaking the operation is only well defined for well-typed expressions. The type system has a check that $x \notin \mathsf{FV}(|a|)$.) Expressions that contain no typing annotations are called *erased*. The core typing judgement is also used to specify when erased expressions type check by non-deterministically guessing the annotations. Specifically, we define $\Gamma \vdash^{\exists} a : A$ if there exists some a' such that $|a'| = a$ and $\Gamma \vdash a' : A$ However, this type system is undecidable. There is no algorithm to decide whether an erased expression should type check.

Thus figure 3.2 can be seen as defining two languages at once (in the implementation the annotated and the erased languages are defined as two separate datatypes). This distinction between annotated and erased expressions follows ICC* [15] and EPTS [91]. It is also similar to the design of Nuprl [35], where our erased expressions are similar to Nuprl expressions, and annotated expressions are similar to Nuprl typing derivations.

In Chapter 5 we go on to define a source language which retains decidable type checking but allows the removal of *some* of these annotations. Zombie is implemented as an elaborator from that surface language into the fully annotated core language. Defining the syntax with optional annotations in this way presents a novel point of view: the source language (what the programmer writes), the core language (what the elaborator targets), and the erased language (what the compiler backend and runtime system deals with) are essentially the same language, just with different amounts of annotation.

Elaborating compilers for simply-typed languages like Haskell do not need an organizing principle like this—there the core language is just an implementation concern, and it does not matter that Haskell Core expressions can look quite different from the source program. But in a dependent language the programmer needs to reason about the reduction behavior of elaborated expressions. As a simple example, if a Coq programmer writes a case-expression with deep patterns, the compiler expands it into several nested case expression—and the expanded version will be displayed in all proof obligations. More advanced Coq features, like `Function` and `Program`, result in more transformations of the source code, and some Coq programmers recommend avoiding them in order to have more detailed control over the generated expressions [32]. Agda and Idris make still more radical changes, because their target language is a simple version of Type Theory where only top-level function definitions can do pattern matching [25]. Because termination checking and definitional equality operate on that elaborated version, seemingly small changes in the source code can cause the program to no longer type check.

In this formalized calculus the connection is perfectly clear, because the source program and its elaboration are always identical up to erasure. The Zombie implemen-

$\boxed{a \leadsto_{\mathsf{cbv}} b}$

$$\frac{}{(\mathsf{rec}\, f\ x.a)\ v \leadsto_{\mathsf{cbv}} \{v/x\}\,\{\mathsf{rec}\, f\ x.a/f\}\ a}\textsc{SCappbeta}$$

$$\frac{}{(\mathsf{rec}\, f\ \bullet.a)\ \bullet \leadsto_{\mathsf{cbv}} \{\mathsf{rec}\, f\ \bullet.a/f\}\ a}\textsc{SCiappbeta}$$

$$\frac{}{\mathsf{case}\ (d_i\,\overline{v_i})\ \mathsf{of}\ \{\overline{d_j\,\overline{x_{ij}} \Rightarrow a_j}^{j\in 1..k}\} \leadsto_{\mathsf{cbv}} \{\overline{v_i}/\overline{x_{ii}}\}\ a_i}\textsc{SCcasebeta}$$

$$\frac{a \leadsto_{\mathsf{cbv}} b}{\mathcal{E}[a] \leadsto_{\mathsf{cbv}} \mathcal{E}[b]}\textsc{SCctx}$$

Evaluation contexts

$$\mathcal{E} ::= [] \mid \mathcal{E}\ a \mid v\,\mathcal{E} \mid \mathcal{E}\ \ \bullet\ \mid \mathsf{case}\,\mathcal{E}\,\mathsf{of}\,\{d_j\,\overline{x_{ij}} \Rightarrow a_j\} \mid d\,\overline{v_i}\,\mathcal{E}\,\overline{a_i}$$

Figure 3.4: Operational semantics

tation departs from this ideal in a few places (in particular, complex pattern matches are expanded in the same way as in Coq), but we try to resist that temptation.

3.3 Operational semantics

Strongly normalizing languages can be agnostic about evaluation order, since any order gives the same answer (e.g. Coq can extract to both ML and Haskell). With nontermination we have to choose, and we pick call-by-value. This choice is different from previous full-spectrum dependent languages, which are generally call-by-name.

The rules for the evaluation relation $\leadsto_{\mathsf{cbv}}$ are shown in figure 3.4. The rules are entirely standard except the rule for irrelevant applications (SCiappbeta), which will become clearer in Section 3.8. Note that the step relation is defined for *erased* expressions, so it is independent of typing. In other words, we take a Curry-style approach to semantics, where the evaluation of raw lambda expressions is the gold standard and we design a type system around it. Occasionally we will abuse notation and write $a \leadsto_{\mathsf{cbv}} b$ meaning $|a| \leadsto_{\mathsf{cbv}} |b|$ when a and b contains annotations. Later, when designing the surface language, we will also define a relation $a \leadsto_{\mathsf{a}} b$ which reduces annotated expressions without first erasing them (Section 5.6.3).

Why choose CBV? In addition to the usual reasons (easier to optimize for speed; a simple cost model for the programmer), there is a particularly nice fit between nonterminating dependent languages and CBV evaluation, because the strictness of evaluation partially compensates for the fact that all types are inhabited.

For example, consider integer division. Suppose the standard library provides a function

```
div : Nat → Nat → Nat
```

which performs truncating division and throws an error if the divisor is zero. If we are concerned about runtime errors, we might want to be more careful. One way to proceed is to define a wrapper around `div`, which requires a proof of div's precondition that the denominator be non-zero:

```
safediv : Nat → (y:Nat) → (p: isZero y = false) → Nat
safediv = λx:Nat.λy:Nat.λp:(isZero y = false).div x y
```

Programs written using `safediv` are guaranteed to not divide by zero, even though our language is inconsistent as a logic. This works because λ-abstractions are strict in their argument, so if we provide an infinite loop as the proof then the entire expression `safediv 1 0 (loop())` diverges and never reaches the division.

With `safediv` strictness is "only" a matter of expressivity. But when type casts are involved, strictness is required for type safety. For example, if a purported proof of $\mathsf{Bool} = \mathsf{Nat}$ were not evaluated strictly, we could use an infinite loop as a proof and try to add two booleans. (This affects all languages with nontermination. For example, even though Haskell is generally lazy it evaluates equality proofs strictly, see Section 8.1.7.)

While strict λ-abstractions give preconditions, strict data constructors can be used to express postconditions. For example, we might define a datatype characterizing what it means for a string (represented as a list of characters) to match a regular expression:[5]

```
data Match : String → Regexp → Type where
  MChar : (x:Char) → Match (x::nil) (RCh x)
  MStar0 : (r:Regexp) → Match (nil) (RStar r)
  MStar1 : (r:Regexp) → (s s':String) →
   Match s r → Match s' (RStar r) → Match (s ++ s') (RStar r)
  ...
```

and then define a regexp matching function to return a proof of the match

```
match : (s:String) → (r:Regexp) → Maybe (Match s r)
```

Such a type can be read as a partial correctness assertion: we have no guarantee that the function will terminate, but if it does and says that there was a match, then there really was. Even though we are working in an inconsistent logic, if the function

[5] For familiarity, this datatype declaration is written using Coq-style indexed datatypes. In real Zombie, the programmer would write it in "parameters only" style (Section 3.7)

$\boxed{\Gamma \vdash a : A}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}}\,\textsc{Ttype} \qquad \frac{\begin{array}{c}\vdash \Gamma \\ x : A \in \Gamma\end{array}}{\Gamma \vdash x : A}\,\textsc{Tvar} \qquad \frac{\begin{array}{l}\Gamma \vdash A : \mathsf{Type} \\ \Gamma, x : A \vdash B : \mathsf{Type}\end{array}}{\Gamma \vdash (x\!:\!A) \to\ B : \mathsf{Type}}\,\textsc{Tpi}$$

$$\frac{\begin{array}{l}\Gamma \vdash (x\!:\!A_1) \to\ A_2 : \mathsf{Type} \\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash a : A_2\end{array}}{\Gamma \vdash \mathsf{rec}\, f_{(x{:}A_1)\to\ A_2}\ x.a : (x\!:\!A_1) \to\ A_2}\,\textsc{Trec} \qquad \frac{\begin{array}{l}\Gamma \vdash a : (x\!:\!A) \to\ B \\ \Gamma \vdash b : A \\ \Gamma \vdash \{b/x\}\, B : \mathsf{Type}\end{array}}{\Gamma \vdash a\ b : \{b/x\}\, B}\,\textsc{Tapp}$$

Figure 3.5: Typing: Basics

returns at all we know that the constructors of `Match` were not given bogus looping terms.

3.4 Basic typing rules

Having defined the syntax and operational semantics of the language, we go on to define a type system for it.

Figure 3.5 shows the basic typing rules for variables and functions. The language uses collapsed syntax and uses the type system to identify which expressions are types: A is a well-formed type if $\Gamma \vdash A : \mathsf{Type}$. The rule Ttype ensures that Type itself is typeable. In a language with general recursion it makes sense to use $\mathsf{Type} : \mathsf{Type}$ rather than a full predicative hierarchy. Having $\mathsf{Type} : \mathsf{Type}$ makes the system logically inconsistent, but because of general recursion it already is.

Function types are kind-checked by the rule Tpi. We use the notation $A \to B$ as syntactic sugar for $(x\!:\!A) \to\ B$ when x is not free in B.

(Recursive) functions are introduced using expressions $\mathsf{rec}\, f\ x.a$, with the typing rule Trec. Rec-expressions are values, and applications step by the rule $(\mathsf{rec}\, f\ x.a)\ v \leadsto_{\mathsf{cbv}} \{v/x\}\, \{\mathsf{rec}\ f\ x.a/f\}\, a$. For non-recursive functions (when f does not occur in a) we also write $\lambda x.a$ as syntactic sugar for $\mathsf{rec}\ f\ x.a$. Functions are eliminated by the application rule Tapp.

The application rule is different from pure dependently typed languages, because it has an extra kinding premise $\Gamma \vdash \{b/x\}\, B : \mathsf{Type}$. In pure languages, the type $\{b/x\}\, B$ would always be well-formed, so the kinding premise would be superfluous. In this calculus, the best we can say is that the arrow type is well-formed. However, the derivation of $\Gamma \vdash (x : A) \to\ B : \mathsf{Type}$ may rely on the fact that x is a value. Specifically, the derivation may involve CBV-reductions, and substituting a non-value

b for x may block a β-reduction that used to have x as an argument. Intuitively this makes some sense: under CBV-semantics, a is really called on the *value* of b, so the type B should be able to assume that x is an effect-free value. In practice, this is a very mild restriction. While it is possible to construct examples where it triggers (see Section 4.2), those examples are very artificial, and when writing real programs we never encountered it.

3.5 Operational semantics at typechecking time

The relation $a \rightsquigarrow_{\mathsf{cbv}} b$ specifies how compiled programs evaluate, but we are not done yet. We also need to consider how the typechecker reduces expressions at typechecking time, which is internalized in the type system by the equality type $a = b$.

The reduction relation used by the typechecker does not have to be exactly the same as the evaluation relation. Since the typechecker deals with open expressions, it is beneficial to allow more reductions. In our core calculus the rule is specified in terms of a *parallel reduction* relation $a \rightsquigarrow_{\mathsf{p}} b$, shown in figure 3.6. The difference is that $\rightsquigarrow_{\mathsf{p}}$ allows reduction under binders, e.g. $(\lambda x.1 + 1) \rightsquigarrow_{\mathsf{p}} (\lambda x.2)$ even though $(\lambda x.1 + 1)$ is already a CBV value.

The $\rightsquigarrow_{\mathsf{p}}$ relation respects CBV contextual equivalence. The typechecker can only carry out a β-reduction of an application or case-expression if the argument or scrutinee is a value. Note however, that values include variables. Treating variables as values is crucial when reasoning about open terms. For example, to typecheck the usual append function we want $\mathsf{Vec}\ \mathsf{Nat}\ (0 + x)$ and $\mathsf{Vec}\ \mathsf{Nat}\ x$ to be equal types, but if x was considered a non-value the expression $0 + x$ would be stuck.

The reduction relation is used by the introduction rule for equality, which says that two expressions can be considered equal if their erasures reduce to a common expression c. Formulating suitable annotations for this rule is slightly subtle. In previous work [120] we proved type safety when the type system included the following rule:

$$\frac{|a| \rightsquigarrow_{\mathsf{p}}^{*} c \qquad |b| \rightsquigarrow_{\mathsf{p}}^{*} c \qquad \Gamma \vdash a = b : \mathsf{Type}}{\Gamma \vdash \mathsf{join} : a = b}\ \textsc{Tjoin}$$

This rule is underspecified in two ways. First, the relation $\rightsquigarrow_{\mathsf{p}}$ is itself nondeterministic: it is allowed, but not required, to reduce expressions under binders. Second, we need to know how many steps to reduce the two expressions to reach the common reduct c. Because of nontermination, it is undecidable whether two arbitrary expressions a and b are joinable in this sense. So phrasing the type-safety theorem in this way is very strong, because it tells us that any choice of reduction strategy is OK. But to use this rule in practice requires additional annotations, which in general have to by supplied by the programmer.

$$\frac{}{a \leadsto_{\mathsf{p}} a}\textsc{SPrefl} \qquad \frac{a \leadsto_{\mathsf{p}} a'}{(\mathsf{rec}\, f\ x.a) \leadsto_{\mathsf{p}} (\mathsf{rec}\, f\ x.a')}\textsc{SPrec}$$

$$\frac{\begin{array}{c} A \leadsto_{\mathsf{p}} A' \\ B \leadsto_{\mathsf{p}} B' \end{array}}{(x\!:\!A) \to\ B \leadsto_{\mathsf{p}} (x\!:\!A') \to\ B'}\textsc{SPpi} \qquad \frac{\begin{array}{c} A \leadsto_{\mathsf{p}} A' \\ B \leadsto_{\mathsf{p}} B' \end{array}}{\bullet(x\!:\!A) \to\ B \leadsto_{\mathsf{p}} \bullet(x\!:\!A') \to\ B'}\textsc{SPipi}$$

$$\frac{\begin{array}{c} a \leadsto_{\mathsf{p}} a' \\ b \leadsto_{\mathsf{p}} b' \end{array}}{a = b \leadsto_{\mathsf{p}} a' = b'}\textsc{SPeq} \qquad \frac{\begin{array}{c} a \leadsto_{\mathsf{p}} a' \\ b \leadsto_{\mathsf{p}} b' \end{array}}{a\ b \leadsto_{\mathsf{p}} a'\ b'}\textsc{SPapp}$$

$$\frac{\begin{array}{c} a \leadsto_{\mathsf{p}} a' \\ v \leadsto_{\mathsf{p}} v' \end{array}}{(\mathsf{rec}\, f\ x.a)\ v \leadsto_{\mathsf{p}} (\{v'/x\}\,\{\mathsf{rec}\ f\ x.a'/f\}\ a')}\textsc{SPappbeta}$$

$$\frac{a \leadsto_{\mathsf{p}} a'}{(\mathsf{rec}\, f\ \bullet.a)\ \bullet \leadsto_{\mathsf{p}} (\{\mathsf{rec}\ f\ \bullet.a'/f\}\ a')}\textsc{SPiappbeta}$$

$$\frac{\forall i.\quad A_i \leadsto_{\mathsf{p}} A'_i}{D\ \overline{A_i} \leadsto_{\mathsf{p}} D\ \overline{A_i}'}\textsc{SPtcon} \qquad \frac{\forall i.\quad a_i \leadsto_{\mathsf{p}} a'_i}{d\,\overline{a_i} \leadsto_{\mathsf{p}} d\,\overline{a_i}}\textsc{SPdcon}$$

$$\frac{\begin{array}{c} a \leadsto_{\mathsf{p}} a' \\ \forall j.\quad a_j \leadsto_{\mathsf{p}} a'_j \end{array}}{\mathsf{case}\ a\ \mathsf{of}\ \{\overline{d_j\,\overline{x_{ij}} \Rightarrow\ a_j}^{j\in 1..k}\} \leadsto_{\mathsf{p}} \mathsf{case}\ a'\ \mathsf{of}\ \{\overline{d_j\,\overline{x_{ij}} \Rightarrow\ a'_j}^{j\in 1..k}\}}\textsc{SPcase}$$

$$\frac{\begin{array}{c} \forall i.\quad v_i \leadsto_{\mathsf{p}} v'_i \\ a_i \leadsto_{\mathsf{p}} a'_i \end{array}}{\mathsf{case}\ (d_i\,\overline{v_i})\ \mathsf{of}\ \{\overline{d_j\,\overline{x_{ij}} \Rightarrow\ a_j}^{j\in 1..k}\} \leadsto_{\mathsf{p}} \{\overline{v_i}'/\overline{x_{i}}_i\}\ a'_i}\textsc{SPcasebeta}$$

Figure 3.6: Parallel reduction

In the implementation, the type checker for the core language offers two particular reduction strategies. The first is standard CBV-evaluation. The second is a particular determinized version of parallel reduction, which works as follows. First, reducing β-redex takes precedence over reducing its subexpressions, and the implementation does try to reduce both the redex and its subexpressions in the same step. For example, when the implementation encounters an application expression $(a\ b)$ it tries the two following rules in order. Both rules are subsets of the full nondeterministic $\leadsto_{\mathsf{p}}$-relation.

$$\frac{}{(\mathsf{rec}\, f\ x.a)\ v \leadsto_{\mathsf{p}} \{v/x\}\,\{\mathsf{rec}\, f\ x.a/f\}\ a}\ \textsc{beta} \qquad \frac{\begin{array}{c} a \leadsto_{\mathsf{p}} a' \\ b \leadsto_{\mathsf{p}} b' \end{array}}{a\ b \leadsto_{\mathsf{p}} a'\ b'}\ \textsc{congruence}$$

Second, in order to avoid unfolding expressions indefinitely (see Section 4.1.3), the implementation "turns off" reduction of recursive functions when going under binders. For example, the rule for reducing the body of a rec-expression is

$$\frac{a \leadsto_{\mathsf{p}}^{-} a'}{(\mathsf{rec}\, f\ x.a) \leadsto_{\mathsf{p}} (\mathsf{rec}\, f\ x.a')}\ \textsc{congruence}$$

where $\leadsto_{\mathsf{p}}^{-}$ denotes the same determinized reduction relation except that the β-rule only triggers for non-recursive functions $\lambda x.a$, but not properly recursive functions $\mathsf{rec}\, f\ x.a$.

With these choices, the reduction path of an expression is completely determined by the number of steps to take, so the core term is annotated to specify reduction strategy and step-count. This is shown as rules TJOINC and TJOINP in figure 3.7. We abuse notation by using the same symbol $\leadsto_{\mathsf{p}}$ for both the nondeterministic relation and the determinized version in the implementation.

In the annotated syntax, the term constructor for these typing rules is written join_{Σ}, where Σ is the "strategy" for proving the equality (i.e. in TJOINC/P the Σ specifies evaluation strategy and the step counts). The annotation Σ is erasable, so the type-checker will consider two proofs of equations to be equal regardless of the details of how they were proven.

3.6 Reasoning about equality

The previous section discussed the introduction form for equality (join). Next we consider how equations can be eliminated. The typing rules are shown in Figure 3.7. The rules are different in several ways from other dependently typed languages. First, the equational theory is very "operational", i.e. two expressions are considered equal

$\boxed{\Gamma \vdash a : A}$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a = b : \mathsf{Type}}\textsc{Teq} \qquad \frac{\begin{array}{l}\Gamma \vdash a : A \quad \Gamma \vdash v : A = B \\ \Gamma \vdash B : \mathsf{Type}\end{array}}{\Gamma \vdash a_{\triangleright v} : B}\textsc{Tcast}$$

$$\frac{\begin{array}{l}|a| \leadsto^{i}_{\mathsf{cbv}} c \quad |b| \leadsto^{j}_{\mathsf{cbv}} c \\ \Gamma \vdash a = b : \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} i\, j : a = b} : a = b}\textsc{TjoinC} \qquad \frac{\begin{array}{l}|a| \leadsto^{i}_{\mathsf{p}} c \quad |b| \leadsto^{j}_{\mathsf{p}} c \\ \Gamma \vdash a = b : \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{p}} i\, j : a = b} : a = b}\textsc{TjoinP}$$

$$\frac{\begin{array}{l}\Gamma \vdash B : \mathsf{Type} \\ \forall k.\ \ \Gamma \vdash v_k : a_k = b_k \\ |B| = |(\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c)|\end{array}}{\Gamma \vdash \mathsf{join}_{\{\sim v_1/x_1\} \ldots \{\sim v_j/x_j\} c : B} : B}\textsc{Tjsubst}$$

$$\frac{\Gamma \vdash v : ((x : A_1) \to\ B_1) = ((x : A_2) \to\ B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\ v} : A_1 = A_2}\textsc{TjInjDom}$$

$$\frac{\begin{array}{l}\Gamma \vdash v_1 : ((x : A) \to\ B_1) = ((x : A) \to\ B_2) \\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash \mathsf{join}_{\mathsf{injrng}\ v_1\ v} : \{v/x\}\, B_1 = \{v/x\}\, B_2}\textsc{TjInjRng}$$

$$\frac{\Gamma \vdash v_1 : (\bullet(x : A_1) \to\ B_1) = (\bullet(x : A_2) \to\ B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\ v_1} : A_1 = A_2}\textsc{TjiInjDom}$$

$$\frac{\begin{array}{l}\Gamma \vdash v_1 : (\bullet(x : A) \to\ B_1) = (\bullet(x : A) \to\ B_2) \\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash \mathsf{join}_{\mathsf{injrng}\ v_1\ v} : \{v/x\}\, B_1 = \{v/x\}\, B_2}\textsc{TjiInjRng}$$

$$\frac{\Gamma \vdash v_1 : D\ \overline{A_i} = D\ \overline{A_i}'}{\Gamma \vdash \mathsf{join}_{\mathsf{ntheq}_k\ v_1} : A_k = A'_k}\textsc{TjInjTcon}$$

$$\frac{\Gamma \vdash v_1 : (A_1 = A_2) = (B_1 = B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injeq}\ i\ v_1} : A_i = B_i}\textsc{TjInjEq}$$

Figure 3.7: Typing: Equality

based on whether they evaluate to the same thing, with the types of the expressions playing little role. Second, using an equation in a type cast is considered a computationally irrelevant operation.

The formation rule TEQ states $a = b$ is a well-formed type whenever a and b are two well-typed expressions. There is no requirement that they have the same type—that is to say, our equality type is heterogeneous. Allowing heterogeneous equations is useful because developments using indexed types often involve expressions whose types are provably but not definitionally equal. (As a simple example, consider stating that depedently typed functions respect equality, $x = y \to f\ x = f\ y$, where $f\ x : A\ x$ and $f\ y : A\ y$.) Coq and Agda include a heterogeneous equality type in their standard library. In Zombie heterogeneous equality is even more useful, because the Zombie definitional equality is weaker (doesn't include $\beta\eta$-convertibility).

The equality elimination rule is TCAST. Given a proof of an equation $A = B$ we can change the type of an expression from A to B. Since our equality is heterogeneous, we need to check that the rhs of the equation is in fact a type, which is done by the premise $\Gamma \vdash B : \mathsf{Type}$. In the core calculus we require the proof to be a value in order to rule out type safety problems from diverging "proofs". In the full Zombie language the proof is allowed to be a nonvalue as long as it is known to terminate (Chapter 7).

The proof term v in a type cast is treated as an erasable annotation, so the typechecker will consider equalities like $a = a_{\triangleright v}$ to be trivially true. This corresponds to the fact that casting between two provably equal types has no operational significance. The fact that type casts are erasable is similar to Nuprl, but different from Coq and Agda—in Coq and Agda the computational behavior of a type cast is to first evaluate the proof until it reaches $\mathsf{refl\_eq}$, the only constructor of the equality type, and then step by $a_{\triangleright \mathsf{refl\_eq}} \leadsto a$.

The reason Coq and Agda uses this computationally-relevant semantics is to ensure strong normalization for open terms. If casts are irrelevant, then in a context containing the assumption $h : (\mathsf{Nat} \to \mathsf{Nat}) = \mathsf{Nat}$ it is possible to type the term $(\lambda x_{\mathsf{Nat}}.x\ x)\ (\lambda x_{\mathsf{Nat}}.x\ x)_{\triangleright h}$ which loops since evaluation does not get stuck on the assumption h. Of course, in our language expressions are not normalizing in the first place, so we enjoy the more powerful reduction relation and accept that reduction may not terminate. (Section 8.2 discusses these tradeoffs in more detail.)

Another interesting feature of the TCAST rule is that the two sides of the used equality are allowed to have different types. By contrast, the conventional treatment of heterogeneous equality is JMeq [84], which allows the programmer to *state* any equality, but only *use* it if both sides have (definitionally) the same type. The lack of such a restriction in this rule means less work for the programmer when using equations. (But there is corresponding drawback, described in Section 3.6.1 below.)

The rest of the figure specifies additional introduction rules for equality, which produce new equations from old ones. Equality proofs do not carry any information at runtime, so all the introduction rules use the same term constructor join, but with different (erasable) annotations.

The rule TJSUBST states that two expressions are equal if their subexpressions are equal, or in other words that equality is a congruence. The simplest use of the rule is to change a single subexpression, using a proof v. The use of the proof is marked with a tilde; for example, if $\Gamma \vdash v : y = 0$ and $\Gamma \vdash a : \mathsf{Vec\ Nat}\ y$, then $\mathsf{join}_{\mathsf{Vec\,Nat}\,(\sim v)}$ has type $\mathsf{Vec\ Nat}\ y = \mathsf{Vec\ Nat}\ 0$.

Unlike other dependent languages, Zombie also allows eliminating several different equality proofs in one use of the rule: for example if $\Gamma \vdash v_1 : x = 0$ and $\Gamma \vdash v_2 : y = 0$, then we can use both proofs at once in the expression $\mathsf{join}_{\mathsf{Vec\,Nat}\,(\sim v_1 + \sim v_2)} : \mathsf{Vec\ Nat}\ (x + y) = \mathsf{Vec\ Nat}\ (0 + 0)$. Having an n-ary congruence rule available is helpful for the programmer (Section 3.6.3) and when implementing the elaborator (Section 8.3.1).

The tilde is part of the concrete syntax of the annotation. In other words, the Σ used with TJSUBST can be an arbitrary expression A, except additionally it can contain subexpressions of the form $\sim v$ in places where the syntax of ordinary expressions would allow a variable x. When stating the typing rule we express this using substitution, so in the expression $\mathsf{join}_{\{\sim v/x\}A}$, the tilde is part of the syntax of the core calculus, but the $\{b/x\}$ part denotes the meta-operation substitution. If the full annotation has the form $\{\sim v_1/x_1\} \ldots \{\sim v_j/x_j\}A$ and $\Gamma \vdash v_k : a_k = b_k$, one can think of the expression A as the "template" for the proved equality, e.g. the left-hand side of the proved equality is obtained by substituting the left-hand sides a_k into the appropriate places in A.

In addition to the template and the proofs, the subst strategy also includes an annotation B which states the equation to be proven. The typechecker verifies that the annotation matches the equation produced by substituting into the template up to erasure. Making B an explicit annotation serves two purposes. First, it provides a way for the elaborator to specify what kinding derivation should be used to check $B : \mathsf{Type}$, rather than leaving it to core checker to construct it. Second and more importantly, this allows the rule to exploit computational irrelevance, because the two types are only compared up to erasure.

Finally, the rules TJINJDOM, TJINJEQ, TJINJRNG, TJIINJDOM, TJIINJRNG, and TJINJDTCON state that datatype constructors, the equality type constructor, and arrow type constructors are injective. The rule for arrow domains is exactly what one would expect: if $(x : A) \to B = (x : A') \to B$, then $A = A'$. The rule for arrow codomains needs to account for the bound variable x, so it states that the codomains are equal when any value v_2 is substituted in. Making type constructors injective is unconventional for a dependent language. On the one hand, it is validated by our interpretation of equality in terms of operational semantics. On the other hand, it is incompatible with e.g. Homotopy Type Theory, which proves $\mathsf{Nat} \to \mathsf{Void} = \mathsf{Bool} \to \mathsf{Void}$ (because semantically, both sides are empty). In our language we need injectivity to prove type preservation, because type casts do not block reduction (see Sections 3.9.4 and 5.6.3). Haskell Core includes type-constructor injectivity for the same reason. It also turns out to be helpful to have injectivity available when designing the elaborator for the surface language (Section 5.4).

3.6.1 Lack of functional extensionality

One of the unique things about Zombie's treatment of equality is that heterogenous equations can be used even if the two sides have different types. This freedom has a downside: we are unable to support certain type-directed equality rules. In particular, adding *functional extensionality* would be unsound. Extensionality implies $(\lambda x_{\mathsf{Void}}.1) = (\lambda x_{\mathsf{Void}}.0)$ since the two functions agree on all arguments (vacuously). But our annotation-ignoring equality shows $(\lambda x_{\mathsf{Void}}.1) = (\lambda x_{\mathsf{Nat}}.1)$, so by transitivity we would get $(\lambda x_{\mathsf{Nat}}.1) = (\lambda x_{\mathsf{Nat}}.0)$, and from there $1 = 0$. This also entails a restriction on rewriting under binders: even if $a = b$ is provable in the context $\Gamma, x : A$ we can not in general conclude $\lambda x.a = \lambda x.b$ in the context Γ, because bringing the variable x into scope requires a use of extensionality. (On the other hand, if x is not free in a and b, then the equation is provable using TJSUBST.)

How limiting is this? In the example programs we have written so far (Chapter 2) we did not encounter any situations where we would have wanted to use functional extensionality, even if it had been available. This may be simply because we have not written enough programs to hit upon an example where it is needed, but it may also be an effect of the type of theorems we are proving. Zombie is primarily intended as a language for software verification, and in that context it is fairly natural to avoid the use of extensionality by rephrasing a theorem $f = g$ to instead say $\forall x. f\ x = g\ x$. After all, a user can never directly observe the equality of two programs, only compare their input-output behavior. Extensionality is much more frequently used in purely mathematical developments, where theorems often directly talk about functions being equal to each other.

While we cannot support functional extensionality in general, it could be that there are some weaker instances which could be added. Since the above counter-example involved an empty function domain, one could add a precondition to the extensionality axiom which requires the domain-type to be inhabited. Also, one can wonder about instances of the η-rule for functions, which states that $f = \lambda x. f\ x$ whenever f can be given a function type (although this rule must be restricted in a CBV language). We are not aware of any ways to derive inconsistency using either of these axioms. However, we have not pursued any formal proof that they are consistent, because doing so would make the metatheory of the language more difficult. When studying the current language we can interpret equality as simply joinability under reduction

(Section 3.9.5). In order to justify extensionality-like axioms, one has to use a more complicated interpretation (typically some form of logical relation).

3.6.2 Proof irrelevance for equations

In intensional type theories (ITTs) such as Coq and Agda, it is interesting to ask if all equality proofs are provably equal. That is to say, given $p : a = b$ and $q : a = b$, is it always the case that $p = q$? It turns out that this claim is not provable in Coq, but it is consistent to assume it as an axiom. (There are several equivalent formulations of the axiom, such as Streicher's Axiom K [125] or "uniqueness of identity proofs".)

Having this principle available in ITT is helpful for several reasons. First, as mentioned above, those languages can prove $a_{\triangleright \mathsf{join}} = a$, but not generally $a_{\triangleright b} = a$, so being able to replace an arbitrary proof b with the equality constructor join allows reduction to make progress. Also, this principle turns out to be needed to show injectivity of type constructors. And McBride [82] showed that it can be used to elaborate dependent pattern-matching statements into nested uses of induction over datatypes. On the other hand, the axiom is only sound if there really only is one way to create equality proofs. In particular, Homotopy Type Theory [135] extends type theory with an a new constructor for equality proofs, the "univalence axiom" $\mathsf{ua}\ f$, with the reduction rule $a_{\triangleright (\mathsf{ua}\ f)} \leadsto f(a)$. This extension is incompatible with assuming that all equality proofs are equal, since we would have $f(a) = a_{\triangleright (\mathsf{ua}\ f)} = a_{\triangleright (\mathsf{ua}\ g)} = g(a)$ for unrelated f and g.

Zombie is halfway towards proof-irrelevant equations. Presumably nothing would go wrong if we added an Axiom K-like rule to the type system, but in the current system it is not possible to prove that two arbitrary equality proofs are equal. However, the use-cases that motivate Axiom K are already satisfied, since computationally irrelevant type casts (Tcast), injective datatype constructors (Tjinjtcon), and dependent pattern matching (Tcase, below) are included as primitives.

Because type casts are computationally irrelevant, all equality proofs can be "canonicalized" to just join. That is, whenever we can prove an equation by *some* proof v, we can prove the same equation with join:

$$\dfrac{\dfrac{}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} 0 0 : a = a} : a = a}\ \textsc{Tjoinc} \qquad \dfrac{\Gamma \vdash v : a = b}{\Gamma \vdash \mathsf{join}_{a = \sim v} : (a = a) = (a = b)}\ \textsc{Tjsubst}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} 0 0 : a = a_{\triangleright \mathsf{join}_{a=\sim v}}} : a = b}\ \textsc{Tcast}$$

This means that the programmer never needs to worry about whether a function argument of equality type should be relevant or irrelevant, since an irrelevant assumption can be used to create a join-expression which can be used in relevant positions.

3.6.3 Carefree equality reasoning

The fact that our equality is not compatible with functional extensionality or the univalence axiom is certainly a limitation, and in e.g. a system aimed at formalizing advanced mathematics this would probably not be a good choice. On the other hand, the rules address some frustrations that come up in everyday programming in Coq.

First, Coq programmers view type casts with suspicion. Particularly when writing functions that takes indexed datatypes as input, it can be tricky to make the types agree. (For an example, see `zip` in Section 3.7 below.) Often the most obvious solution would be to insert a type cast. However, such a cast would interfere with reduction of the function, so the preferred solution is instead to try to use partial function applications and extra λ-expressions (the "convoy pattern" [32]) to finagle the Coq pattern matching rule into producing the right type. Because type casts in Zombie are irrelevant we avoid the need for this.

Second, we allow a single use of TSUBST to eliminate more than one proof in order to make sure that equality is a congruence. For example, suppose that we have the context

$$f : A \to C, \quad g : B \to C, \quad a : A, \quad b : B, \quad p : f = g, \quad q : a = b$$

and we want to prove $f\ a = g\ b$. In our system the proof is just $\mathsf{join}_{\sim p\ \sim q}$. However, if we could only use one equation at a time, we would have to use a chain of reasoning such as $f\ a = f\ b = g\ b$, where the intermediate term is not well-typed. The usual way this problem comes up in practice is when reasoning about indexed datatypes such as vectors; for example consider proving $\mathsf{Cons}\ m\ x\ xs = \mathsf{Cons}\ n\ x\ ys$, where m and n are propositionally but not definitionally equal. It is still possible to prove this kind of equations through judicious use of type casts and abstraction (see Chlipala [32] Chapter 9, and Bertot and Castéran [20] Section 8.2.7), but it is more tricky.

In this author's experience, reasoning about equality is by far the most subtle and confusing part of programming in Coq. (Programming in Agda is not quite as bad, since the typechecker works hard behind the scenes to unify type indices when pattern matching.) With our set of rules for equality, equality works more like in first-order logic: equal expressions can be used interchangeably, as long as the types still work out. Since our ultimate goal is to make it easier for programmers to transition from functional to dependent programming, we try to make reasoning about equality as simple as we can.

$\boxed{\Gamma \vdash a : A}$

$$\frac{\begin{array}{l}\mathsf{data}\ D\,\Delta^{+}\ \mathsf{where}\,\{\,\overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1..j}\,\} \in \Gamma\\ \Gamma \vdash \overline{A_i} : \Delta^{+}\end{array}}{\Gamma \vdash D\ \overline{A_i} : \mathsf{Type}}\ \textsc{Ttcon} \qquad \frac{\begin{array}{l}\mathsf{data}\ D\,\Delta^{+} \in \Gamma\\ \Gamma \vdash \overline{A_i} : \Delta^{+}\end{array}}{\Gamma \vdash D\ \overline{A_i} : \mathsf{Type}}\ \textsc{Tabstcon}$$

$$\frac{\begin{array}{l}\mathsf{data}\ D\,\Delta^{+}\ \mathsf{where}\,\{\,\overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1..j}\,\} \in \Gamma\\ \Gamma \vdash \overline{A_i} : \Delta^{+}\\ \Gamma \vdash \overline{a_i} : \{\overline{A_i}/\Delta^{+}\}\,\Delta_i\end{array}}{\Gamma \vdash d_{k\,D\,\overline{A_i}}\ \overline{a_i} : D\ \overline{A_i}}\ \textsc{Tdcon}$$

$$\frac{\begin{array}{l}\Gamma \vdash b : D\ \overline{B_i}\\ \Gamma \vdash A : \mathsf{Type}\\ \mathsf{data}\ D\,\Delta^{+}\ \mathsf{where}\,\{\,\overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1..j}\,\} \in \Gamma\\ \forall i.\ \ \Gamma, \{\overline{B_i}/\Delta^{+}\}\,\Delta_i, y : b = d_i\,\Delta_i \vdash a_i : A\\ \forall i.\ \ y\ \notin \mathsf{FV}\,(|a_i|)\\ \forall i.\ \ \mathsf{dom}^{-}(\Delta_i)\ \#\ \mathsf{FV}\,(|a_i|)\end{array}}{\Gamma \vdash \mathsf{case}\ b_y\ \mathsf{of}\ \{\overline{d_i\,\Delta_i \Rightarrow a_i}^{\,i\in 1..j}\} : A}\ \textsc{Tcase}$$

Figure 3.8: Typing: Datatypes

3.7 Datatypes

Our core language includes datatypes $D\ \overline{A_i}$, which are introduced by data constructors $d_{D\ \overline{A_i}}\ \overline{a_i}$ and eliminated by case-expressions. The treatment of datatypes is mostly standard, but the way the typing rule for case handles type dependency is interesting.

Datatype declarations have the form

$$\mathsf{data}\ D\,\Delta\ \mathsf{where}\ \{\ \overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i\in 1..j}\ \}$$

Once a declaration datatype is in the context, the rule TTCON allows it to be used as a well-formed type.

To reduce clutter we write the rules using *telescope notation.* Metavariables Δ range over lists of relevance-annotated variable declarations like $(x : A)\ \bullet_{y:B}\ (z : C)$, also known as telescopes, while overlined metavariables $\overline{a_i}$ range over lists of terms like $a\ \bullet_b\ c$. Metavariables Δ^+ range over telescopes that have only relevant declarations. Depending on where in an expression they occur, telescope metavariables stand in for either declarations or lists of variables, according to the following scheme:

$$\begin{array}{lcl}
a_1\,\Delta & \equiv & a_1\ x\ \bullet_y\ z \\
\{\overline{a_i}/\Delta\}\,a_1 & \equiv & \{a/x\}\,\{b/y\}\,\{c/z\}\,a_1 \\
\Gamma, \Delta & \equiv & \Gamma, x : A, y : B, z : C \\
\Gamma \vdash \overline{a_i} : \Delta & \equiv & \Gamma \vdash a : A \quad \wedge \quad \Gamma \vdash b : \{a/x\}\,B \quad \wedge \quad \Gamma \vdash c : \{b/y\}\,\{a/x\}\,C
\end{array}$$

Datatypes have *parameters but not indices.* That is, a datatype has a list of parameters Δ, and the return type of each of its data constructor d_i is $D\,\Delta$ (the type constructor D applied to the same list of variables). Cases where one would want non-uniform indexes, e.g. the usual type Vec of length-indexed lists, can instead be handled by a combination of parameters and equality proofs:

```
data Vec (a:Type) (n:Nat) where
  Nil  of (p:n=0)
  Cons of (m:Nat)  (p:n=Succ m) (x:a) (xs:Vec a m)
```

The simplicity of parameters-only makes it attractive for a core language (e.g. this is how Haskell GADTs are elaborated into GHC Core [131]). One could elaborate the usual parameters-and-indexes formulation into this format. However, we have found that with some support from the typechecker (CREFL in Section 5.3) it is quite easy to manage the extra equality proofs explicitly even in the surface language.

The introduction rule TDCON checks data constructors by looking them up in the datatype declaration. For simplicity, data constructors are fully-applied (like in OCaml) rather than implicitly curried (like in Haskell). In other words, d by itself is not a well-formed expression; it must be applied to a list of type parameters and arguments $d_{D\,\overline{A_i}}\ \overline{a_i}$.

Finally, datatypes are eliminated by case-expressions in the rule TCASE. The programmer must write one branch $d_i\,\Delta_i \Rightarrow a_i$ for each constructor of the datatype D. The branch only introduces pattern variables for the constructor arguments, as the parameters are fixed throughout the case. The parameters are used to refine the context that the match is checked in: if the scrutinee has $\Gamma \vdash b : D\ \overline{B_i}$, then for each branch we check

$$\Gamma, \{\overline{B_i}/\Delta^+\}\,\Delta_i, y : b = d_i\,\Delta_i \vdash a_i : A$$

The context also includes an equality assumption. The case expression (case a_y of $\ldots$) binds the variable y, which can occur in the match branches. In the concrete syntax we write (`case b [y] of ...`). The variable y must only occur in computationally irrelevant positions in the branches, which is enforced by the premise $y \notin \mathsf{FV}\,(|a_i|)$.

From a language-design perspective, this equality assumption is the most interesting part of the rule—sometimes this rule is called *smart case* [7]. Having the equation y available resolves a notoriously delicate issue in dependent languages, which is how to make the new information from a case statically available. For example, consider a function which does a match on a length-indexed list:

```
zip : (A:Type) → (B:Type) → (n:nat)
      → (u : Vec A n) → (v : Vec A n) → Vec (A*B) n
zip A B n u v =
  case v [y] of
     Nil  p        ⇒ ... e1 ...
     Cons m p x xs ⇒ ... e2 ...
```

If the program execution reaches e_1, we have learned two new things: the expression v is equal to `Nil`, and the type index n is equal to 0. In our system, this information is exposed very straightforwardly, since the context has been extended with the two equations $y : v = \mathsf{Nil}$ and $p : n = 0$. This design is feasible because type casts are irrelevant, so uses of y and p will not interfere with the reduction behavior of `zip`, and because the surface language uses the equations automatically (Chapter 5).

Existing languages have less convenient case rules. In Coq one would typically write the datatype in a non-uniform-indices style, and by default no equation y is added. Then in the above example, e_2 would be checked in a context

```
xs : Vec A m
v : Vec A n
```

with no relation between m and n, so there is no way to complete the function. The programmer can manually specify that the branches should be checked at a different type (calculated as a function of the indexes and the scrutinee), but even simple functions such as `zip` can require some care to write.

By contrast, Agda makes information about type indices available, but at the cost of a quite complicated typing rule. In each branch the type checker computes the most general unifier of the indices appearing in the scrutinee type and constructor type, and then applies that substitution to the context that the branch body is checked in. For example, in the branch e_2 the type of v would be `Vec A (suc m)` rather than `Vec A (suc n)`. This makes programming easier, but the specification of the typing rule is complex since it involves unification. Indeed, pattern matching in Agda and Idris is a rather heavy operation which is only available in top-level functions; expression-level pattern matching has to be elaborated into new anonymous top-level functions, which can sometimes be confusing for the programmer. The use of unification can also get in the way: for example if we instead had a variable of type `v : vec A (f n)`, for some function f, then Agda would not allow a case expression on `v` at all because it could not guarantee a unique solution to the unification problem `f n = suc m`. Our language simply adds that equation to the context, instead of having to first solve for `n`.

Also, we saw in Section 2.3.1 that even in cases when no unification is done, the rule used by Coq and Agda can be clumsy because the typechecker needs to guess how to refine the return type.

Sometimes, adding a propositional equation to the context is exactly what the programmer wants, for example in writing an explicit proof term which reasons by cases. (For example, consider the branch on `(ordLt x y)` in the proof in Section 2.2.) In these circumstances, Coq and Agda both have established idioms (the `case` tactic, and the `inspect` pattern) which produce essentially our typing rule as a derived rule. Because equality proofs in these languages are not erased, however, indiscriminate use of these idioms can interfere with the reduction behavior of a term, so they are not a general solution.

3.8 Computational Irrelevance

Previously we saw that since the equality proof in a cast $a_{\triangleright b}$ has no computational role, it can be treated as an erasable annotation. The same idea can be extended to function arguments. To support this, in addition to the normal function type, we also include *computationally irrelevant* function types $\bullet(x\!:\!A) \to B$, which are inhabited by irrelevant functions $\mathsf{rec}\, f_A\ \bullet_x.b$ and eliminated by irrelevant applications $a\ \bullet_b$.

In a dependent language many arguments are only used for type checking, but do not affect the runtime behavior of the program—consider for instance type arguments to polymorphic functions (e.g. $\mathsf{map} : \bullet(A\ B : \mathsf{Type}) \to (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B)$, indices to dependent datatypes (e.g. $\mathsf{append} : \bullet(n\ m : \mathsf{Nat}) \to \mathsf{Vec}\ n \to \mathsf{Vec}\ m \to \mathsf{Vec}\ (n+m))$, and preconditions to functions (e.g. $\mathsf{safediv} : (x\ y : \mathsf{Int}) \to \bullet(y \neq$

$\boxed{\Gamma \vdash a : A}$

$$\frac{\begin{array}{l}\Gamma \vdash A : \mathsf{Type} \\ \Gamma, x : A \vdash B : \mathsf{Type}\end{array}}{\Gamma \vdash \bullet(x\!:\!A) \to\ B : \mathsf{Type}}\;\textsc{TiPi} \qquad \frac{\begin{array}{l}\Gamma \vdash \bullet(x\!:\!A_1) \to\ A_2 : \mathsf{Type} \\ \Gamma, f : \bullet(x\!:\!A_1) \to\ A_2, x : A_1 \vdash a : A_2 \\ x\ \notin \mathsf{FV}\,(|a|)\end{array}}{\Gamma \vdash \mathsf{rec}\ f_{(xA_1)\to A_2}\ \bullet_x.a : (x\!:\!A_1) \to\ A_2}\;\textsc{TiRec}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : \bullet(x\!:\!A) \to\ B \\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash a\ \bullet_v : \{v/x\}\,B}\;\textsc{TiApp}$$

Figure 3.9: Typing: Irrelevant function arguments

0) → Int). If the programmer marks these arguments as irrelevant, the typechecker will ignore them when comparing expressions for equality, making it easier to write proofs. For example, if both p and q have type $y \neq 0$, the typechecker will treat $\mathsf{safediv}\ x\ y\ \bullet_p$ and $\mathsf{safediv}\ x\ y\ \bullet_q$ as equal without looking at the last argument.

One good example which shows how irrelevant arguments make it easier to write proofs is reasoning about appending vectors. Suppose we write a function to append vectors belonging to the `Vec` type above.[6]

```
app : (n1 n2 : Nat) → (a : Type) → Vec a n1 → Vec a n2 → Vec a (n1+n2)
app n1 n2 a xs ys =
  case xs of
    (Nil _) → ys
    (Cons n x xs) → Cons a (n+n2) _ x (app n n2 a xs ys)
```

Having defined this operation, we might wish to prove that the append operation is associative. This amounts to defining a recursive function of type

```
  app-assoc : (n1 n2 n3:Nat) →
              (v1 : Vec a n1) → (v2 : Vec a n2) → (v3 : Vec a n3) →
              (app a n1 (n2+n3) v1 (app a n2 n3 v2 v3))
                =  (app a (n1+n2) n3 (app a n1 n2 v1 v2) v3)
```

If we proceed by pattern-matching on `v1`, then when `v1 = cons n x v` we have to show, after reducing the RHS, that

[6] The underscores represent the equality proofs about length. The shown function definition is not valid Zombie code; in order to make it typecheck one also needs two lines introducing join-proofs for the equations $0 + n = n$ and $(\mathsf{Suc}\ n) + m = \mathsf{Suc}\ (n + m)$. See `Append2.trellys` in the Zombie testcases.

```
    (Cons (n + (n2 + n3)) _
          x
          (app a n (n2 + n3) v (app a n2 n3 v2 v3)))
  = (Cons ((n + n2) + n3) _
          x
          (app a (n + n2) n3 (app a n n2 v v2) v3))
```

By a recursive call/induction hypothesis, we have that the tails of the vectors are equal, so we are *almost* done, except we also need to show

```
n + (n2 + n3) = (n + n2) + n3
```

which requires a separate lemma about associativity of addition. In other words, when reasoning about indexed data, we are also forced to reason about their indices. In this case it is particularly frustrating because these indices are completely determined by the shape of the data—a Sufficiently Smart Compiler would not even need to keep them around at runtime [26].

The solution is to change the definition of `Vec` to make the length argument `n` to Cons an irrelevant argument. For good measure, we can make the equation `p` irrelevant also. In the concrete syntax we use square brackets for the irrelevant function type constructor and application (`[x:A]` $\rightarrow$ `B` and `a [b]`), while the LaTeX-typeset version uses a bullet ($\bullet(x : A) \rightarrow B$ and $a \bullet_b$).

```
    data Vec (a:Type) (n:Nat) where
      Nil  of [p:n=0]
      Cons of [m:Nat]  [p:n=Succ m] (x:a) (xs:Vec a m)
```

Irrelevant constructor arguments are not represented in memory at run-time, and equations between irrelevant arguments are trivially true since our T_JOIN rule is stated using erasure. With this definition of `Vec`, the proof of `app-assoc` can be concluded without needing any lemmas about natural numbers.

3.8.1 Computationally irrelevant functions and applications

The typing rules for irrelevant functions are shown in Figure 3.9. These rules are closely inspired by previous languages, in particular ICC* [15] and EPTS [91]. The introduction rule for irrelevant functions, TIREC, is similar to the rule for normal functions, but with the additional restriction that the bound variable must not remain in the erasure of the body b. This restriction means that x can only appear in irrelevant positions in b, such as type annotations and proofs for type casts. On the other hand, x is available at type-checking time, so it can occur freely in the type B.

The application rule TIAPP is almost identical to the computationally relevant application rule. However, there is one difference: the argument is required to be a value. This is done to ensure that the argument terminates. While it is fine to allow `safediv 1 0 (loop())` (which is stuck in a loop and never reaches the division), we really must forbid `safediv 1 0 [loop()]` (which reduces in one step to `div 1 0`).

In the full Zombie implementation, the implicit application rule is more permissive, allowing any implicit argument which is known to terminate. The value-restricted version used in this core calculus is a useful approximation for which we can develop the metatheory without getting bogged down in termination proofs.

In addition to the `safediv` example, there is another reason to require termination checking of irrelevant arguments. If we want propositional equality to approximate CBV contextual equivalence, we should not substitute nonterminating expressions for variables. This is discussed more in Section 4.1.5.

3.8.2 Computationally irrelevant datatype arguments

In addition to functions, data constructors also support irrelevant arguments. They are typically used for data constructor fields which serve as type indexes or for fields containing proofs, as we saw with `Vec` above.

Whether a datatype parameter is irrelevant or not depends on where it occurs. This was first worked out by Mishra-Linger in his thesis [90]. Parameters to data constructors are always irrelevant, since they are completely fixed by the types. So in an expression like $d_{D\,\overline{A_i}}\ \overline{a_i}$, the $D\ \overline{A_i}$ part is erasable.

On the other hand, it never makes sense for parameters to data*type* constructors to be irrelevant. For example, if the parameters to `Vec` were made irrelevant, the join rule would validate $\Gamma \vdash \mathsf{join} : (\mathsf{Vec}\ \bullet_{\mathsf{Nat}}\ \bullet_1) = (\mathsf{Vec}\ \bullet_{\mathsf{Bool}}\ \bullet_2)$, which would defeat the point of having the parameters in the first place. So in the typing rule, the telescope of parameters is a Δ^+, i.e. a telescope with no irrelevant variables.

Finally, arguments to data constructors d_i can be marked as relevant or not. Marking them as irrelevant leads to a corresponding restriction for case expressions: since the argument has no run-time representation it may only be used in irrelevant positions. For example, in a case branch

```
    Cons [m:Nat] [p:n=Succ m] (x:a) (xs:Vec a m) ⇒ ...body...
```

the code in the body can use `x` without restrictions but can only use `m` in irrelevant positions such as type annotations and type casts. (In the typing rule this is the premise $\mathsf{dom}^-(\Delta_i)\ \#\ \mathsf{FV}\,(|a_i|)$; the notation $\mathsf{dom}^-(\Delta_i)$ means the variables in the telescope Δ_i that are marked irrelevant.) With the original `Vec` type we could write a constant-time length function by projecting out `m`, but that is not possible if `m` is made irrelevant.

The annotation $D\ \overline{A_i}$ in a data constructor application $d_{D\ \overline{A_i}}\ \overline{a_i}$ is erased, just as the argument in an implicit application $a\ \bullet_v$. Despite this, in this calculus we do not require the type parameters to be values. Intuitively, this is sound because type parameters are only compared for equality, not used as typing assumptions. Formally, this is because when proving type preservation for reduction of case-expressions we only use the substitution lemma for the constructor arguments, not for the type parameters. The full Zombie language as implemented is actually a little more restrictive. In order to form the type $D\ \overline{A_i}$, all the subcomponents A_i are required to check in the termination-checked fragment (c.f. Section 7.7.2). So in full Zombie the treatment of type parameters agree with the treatment of irrelevant arguments. In future work it would be useful to see if type parameters can be handled more freely even in a termination-checked language.

3.8.3 The value restriction is too restrictive

The value restriction on irrelevant arguments, as adopted in this core language, is a severe limitation on the practical use of irrelevant arguments. For example, one of the cases of the `app` function returns `Cons [a] [n+n2] x (app n n2 a xs ys)`, and `n+n2` is not a syntactic value. As noted above, this is permitted because `Cons` is a data constructor rather than a general function. However, if `Cons` had been defined as a function (even just an η-expansion of the constructor), we would have to evaluate the expression to a value at runtime:

```
app : (n1 n2 : Nat) → (a : Type) → Vec a n1 → Vec a n2 → Vec a (n1+n2)
app n1 n2 a xs ys =
  case xs of
    Nil → ys
    (Cons n x xs) → let m = n+n2 in
                             Cons a [m] x (app n n2 a xs ys)
```

(In order to make the above program typecheck we would also have to insert a type cast, to change the return type from `Vec a m` to `Vec a (n1+n2)`.) Evaluating type parameters at runtime is very unsatisfying. First, if our natural number type uses unary representation (as is common, to make it easier to reason about numbers at compile time), then the above change makes the running time of `append` quadratic instead of linear. Second, because we do the addition at runtime, the arguments `n1` and `n2` to `app` itself can not be made irrelevant.

Of course, in general we do want to use irrelevant arguments with functions as well as constructors. For example, just as `app` is defined in terms of `Cons`, we may later want to define other functions such as `concat` in terms of `app`, and we want the length arguments of `app` to be irrelevant to ease reasoning about `concat`.

In full Zombie this is not a problem. It is easy to prove that addition in terminating, so the expression $n + n_2$ is accepted as a type index as-is. Value-restricted irrelevance already has uses. For example, except for type-level computation all types are values, so we could compile ML into this core language erasing all types. But in general, it is not enough for dependently type programming.

3.9 Metatheory

We prove type safety for the core calculus via standard preservation and progress theorems. We previously published a detailed proof [120], so we refer the reader to the appendix of that paper for details, and in this chapter we only highlight the most interesting parts of the proof. (In fact, in the few years since the publication of that paper we have slightly updated the typing rules, as described in Section 3.9.1. However, none of the changes significantly affects the proof.)

One of the appealing results of this research is that the proof is not very difficult. Most prior work on full-spectrum dependent languages either did not prove type safety at all (e.g. Cayenne, ΠΣ, Idris, Agda), or proved much more ambitious results which also included logical consistency (e.g. Coq, Nuprl). While consistency proofs are often notoriously hard, if we are only interested in type safety a mostly standard Wright-Felleisen[144]-style proof suffices, even though our calculus includes advanced features like nontermination, type-level computation, and large eliminations. The sequence of lemmas goes as follows.

1. Structural properties (weakening and substitution).
2. Preservation.
3. Equational soundness: if $\cdot \vdash v : A = B$, then there exists C such that $A \leadsto_{\mathsf{p}}^{*} C$ and $B \leadsto_{\mathsf{p}}^{*} C$.
4. Progress.

Only step 3 would be unfamiliar to a programming languages researcher who has not previously worked with dependent types, and even there the interpretation of equalities is simply (some would say crudely) based on operational semantics. This treatment of equations is probably the most interesting part of the proof. In later chapters we will see that the same interpretation also works in a normalization proof (Section 7.6; also Casinghino [30] Section 5.4.2), and that it can be varied to accommodate some more powerful equational reasoning (Section 4.1).

3.9.1 The language

In a previously published paper [120] we described essentially the same calculus as the one in this chapter. In the present version we have made three improvements:

- The presentation of the syntax is re-structured to emphasize the role of annotations (but both presentations define essentially the same syntax).
- Instead of a single rule TCONV which can eliminate multiple equality proofs, we have refactored it as two rules TCAST and TJSUBST. This makes it easier to reduce annotated expressions (Section 5.6.3).
- The paper had three different expression types to define functions: relevant lambdas, irrelevant lambdas, and one-argument recursors which can be combined with either type of lambda-expression. However, this formulation does not fit with adding structurally recursive functions (Section 7.3.1), so in this thesis we fuse them into two expression types. (That is, the expression $\mathsf{rec}\, f\ x.a$ in this chapter corresponds to $\mathsf{rec}\, f.\lambda\, x.a$ in the paper).

Also, the paper featured uncatchable exceptions abort, which have been omitted from this chapter for simplicity. (They are however included in full Zombie, and mentioned in Section 4.1.4.)

To make this section less confusing when compared to the previous parts of the chapter, the discussion below "cheats" by changing the statement of canonical forms to the current syntax, and by talking about TCAST instead of TCONV. This is different from the actual published proof. It still discusses λ- rather than rec-expressions, because λs already illustrate all the interesting cases.

3.9.2 Annotated and unannotated type systems

While we have defined the typing judgement in terms of a system with optional annotations, in the preservation proof it is inconvenient to take into consideration that expressions may have annotations embedded inside them. So the first step of the proof is to develop a second set of typing rules which characterizes when a completely erased term is well-typed (according to the nondeterministic $\Gamma \vdash^{\exists} a : A$ judgement).

We use metavariables m, n, M, N for completely erased expressions, and the metavariable H for contexts Γ containing only completely erased types.

We then define a second set of typing rules for a different judgement, which we will punningly write $H \vdash m : M$. In practice, the unannotated rules simply mirror the annotated rules, except that all the terms in them have gone through erasure. As an example, compare the annotated and unannotated versions of the IAPP rule:

$$\frac{\begin{array}{l}\Gamma \vdash a : \bullet(x\!:\!A) \to\ B \\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash a \bullet_v : \{v/x\}\, B}\textsc{Tiapp} \qquad \frac{\begin{array}{l}H \vdash m : \bullet(x\!:\!M) \to\ N \\ H \vdash u : M\end{array}}{H \vdash m \bullet : \{u/x\}\, N}\textsc{ETiapp}$$

By design, the erasure into the unannotated system preserves well-typedness.

***Lemma* 1** (Annotation soundness)**.** If $\Gamma \vdash a : A$ then $|\Gamma| \vdash |a| : |A|$ (where the second judgement is in the annotated system).

Since our operational semantics is defined for unannotated terms, the preservation and progress theorems will be also stated in terms of the unannotated system.

3.9.3 Properties of parallel reduction

The key fact about propositional equality is that in an empty context it coincides with joinability under parallel reduction. We will need some basic lemmas about parallel reduction throughout the proof. These are familiar from, e.g., the metatheory of pure type systems [14], except that the usual substitution lemma is replaced with two special cases because we work with CBV reduction.

***Lemma* 2** (Substitution of $\rightsquigarrow_{\mathsf{p}}$)**.** If $N \rightsquigarrow_{\mathsf{p}} N'$, then $\{N/x\}\, M \rightsquigarrow_{\mathsf{p}} \{N'/x\}\, M$.

***Lemma* 3** (Substitution into $\rightsquigarrow_{\mathsf{p}}$)**.** If $u \rightsquigarrow_{\mathsf{p}} u'$ and $m \rightsquigarrow_{\mathsf{p}} m'$, then $\{u/x\}\, m \rightsquigarrow_{\mathsf{p}} \{u'/x\}\, m'$.

***Definition* 4** (Joinability)**.** We write $m_1 \curlyvee m_2$ if there exists some n such that $m_1 \rightsquigarrow_{\mathsf{p}}^{*} n$ and $m_2 \rightsquigarrow_{\mathsf{p}}^{*} n$.

***Lemma* 5** (Confluence of $\rightsquigarrow_{\mathsf{p}}$)**.** If $m \rightsquigarrow_{\mathsf{p}}^{*} m_1$ and $m \rightsquigarrow_{\mathsf{p}}^{*} m_2$, then $m_1 \curlyvee m_2$

Using the above lemmas we can prove the important properties of joinability.

***Lemma* 6** (Properties of $\curlyvee$)**.**

1. $\curlyvee$ is an equivalence relation.
2. If $m_1 \rightsquigarrow_{\mathsf{cbv}} m_2$, then $m_1 \curlyvee m_2$.
3. If $m_1 \curlyvee m_2$ then $\{m_1/x\}\, m \curlyvee \{m_2/x\}\, m$.
4. If $m_1 \curlyvee m_2$ and $\{m_1/x\}\, m \curlyvee \{m_1/x\}\, m'$, then $\{m_2/x\}\, m \curlyvee \{m_2/x\}\, m'$.

Proof. For property (1), symmetry and reflexivity are immediate from the definition of joinability. Transitivity uses the fact that $\rightsquigarrow_{\mathsf{p}}$ is confluent (lemma 5).

Property (2) is true because the relation $\rightsquigarrow_{\mathsf{p}}$ is a superset of $\rightsquigarrow_{\mathsf{cbv}}$.

Property (3) follows by lemma 2.

For property (4), note that we have

$$\begin{array}{ll} \{m_2/x\}\, m \curlyvee \{m_1/x\}\, m & \text{by property (3) and symmetry,} \\ \{m_1/x\}\, m \curlyvee \{m_1/x\}\, m' & \text{by assumption,} \\ \{m_1/x\}\, m' \curlyvee \{m_2/x\}\, m' & \text{by property (3),} \end{array}$$

then conclude by transitivity. □

3.9.4 Preservation

For the preservation proof we need the usual structural properties: weakening and substitution. Weakening is standard, but substitution is restricted to substituting values u into the judgement, not arbitrary terms. This is because our equality is CBV, so substituting a non-value could block reductions and cause types to no longer be equal.

***Lemma* 7** (Substitution)**.** If $H_1, x_1 : M_1, H_2 \vdash m : M$ and $H_1 \vdash u_1 : M_1$, then $H_1, \{u_1/x_1\}\, H_2 \vdash \{u_1/x_1\}\, m : \{u_1/x_1\}\, M$.

Preservation also needs inversion lemmas for λs, irrelevant λs, rec, and data constructors. They are similar, and we show the one for λ-abstractions as an example.

***Lemma* 8** (Inversion for λs)**.** If $H \vdash \lambda x.n : M$, then there exists m_1, M_1, N_1, such that $H \vdash m_1 : M = ((x\!:\!M_1) \to N_1)$ and $H, x : M_1 \vdash n : N_1$.

Notice that this is weaker statement than in a language with computationally relevant type casts. For example, in a pure type system we would have that M is β-convertible to the type $(x\!:\!M_1) \to N_1$, not just provably equal to it. But in our language, if the context contained the equality $(\mathsf{Nat} \to \mathsf{Nat}) = \mathsf{Nat}$, then we could show $H \vdash \lambda x.x : \mathsf{Nat}$ using a (completely erased) cast. As we will see, we need to add extra injectivity rules to the type system to compensate.

Now we are ready to prove preservation. For type safety we are only interested in preservation for $\leadsto_{\mathsf{cbv}}$, but it is convenient to generalize the theorem to $\leadsto_{\mathsf{p}}$.

***Theorem* 9** (Preservation)**.**

If $H \vdash m : M$ and $m \leadsto_{\mathsf{p}} m'$, then $H \vdash m' : M$.

The proof is mostly straightforward by an induction on the typing derivation. There are some wrinkles, all of which can be seen by considering some cases for applications.

The typing rule looks like

$$\frac{\begin{array}{l} H \vdash m : (x\!:\!M) \to\ N \\ H \vdash n : M \\ H \vdash \{n/x\}\,N : \mathsf{Type} \end{array}}{H \vdash m\ n : \{n/x\}\,N}\ \textsc{ETapp}$$

First consider the case when $m\ n$ steps by congruence, $m\ n \rightsquigarrow_{\mathsf{p}} m\ n'$. Directly by IH we get that $H \vdash n' : M$, but because of our CBV-style application rule we also need to establish $H \vdash \{n'/x\}\,N : \mathsf{Type}$. By substitution of $\rightsquigarrow_{\mathsf{p}}$ we know that $\{n/x\}\,N \rightsquigarrow_{\mathsf{p}} \{n'/x\}\,N$, so this also follows by IH (this is why we generalize the theorem to $\rightsquigarrow_{\mathsf{p}}$).

This showed $H \vdash m\ n' : \{n'/x\}\,N$, but we needed $H \vdash m\ n' : \{n/x\}\,N$. Since $\{n/x\}\,N \rightsquigarrow_{\mathsf{p}} \{n'/x\}\,N$ we have $H \vdash \mathsf{join} : \{n'/x\}\,N = \{n/x\}\,N$, and we can conclude using the conv rule. This illustrates how fully erased type casts generalize the β-equivalence rule familiar from pure type systems.

Second, consider the case when an application steps by β-reduction, $(\lambda x.m_0)\ u \rightsquigarrow_{\mathsf{p}} \{u/x\}\,m_0$, and we need to show $H \vdash \{u/x\}\,m_0 : \{u/x\}\,N$. The inversion lemma for $\lambda x.m_0$ gives $H, x : M_1 \vdash m_0 : N_1$ for some $H \vdash \mathsf{join} : ((x\!:\!M) \to\ N) = ((x\!:\!M_1) \to N_1)$. Now we need to convert the type of u to $H \vdash u : M_1$, so that we can apply substitution and get $H \vdash \{u/x\}\,m_0 : \{u/x\}\,N_1$, and finally convert back to $\{u/x\}\,N$. To do this we need to decompose the equality proof from the inversion lemma into proofs of $M = M_1$ and $\{u/x\}\,N_1 = \{u/x\}\,N$. We run into the same issue in the cases for irrelevant application and pattern matching on datatypes. So we add a set of injectivity rules to our type system to make these cases go through (figure 3.7).

For the preservation proof we only need injectivity rules for arrow types and datatypes, but we also include a rule for equations (TJINJEQ). This rule is also justified by our interpretation of equality as joinability, and it is useful to have it available in the surface language (see Section 5.4).

3.9.5 Progress

As is common in languages with dependent pattern matching, when proving progress we have to worry about "bad" equations. Specifically, this shows up in the canonical forms lemma. We want to say that if a closed value has a function type, then it is actually a function. However, what if we had a proof of $\mathsf{Nat} = (\mathsf{Nat} \to \mathsf{Nat})$? To rule that out, we start by proving a lemma characterizing when two expressions can be propositionally equal. From now on, H_{D} denotes a context which is empty except that it may contain datatype declarations.

***Lemma* 10** (Soundness of equality)**.** If $H_{\mathsf{D}} \vdash u : M$ and $M \curlyveedownarrow (m_1 = n_1)$, then $m_1 \curlyveedownarrow n_1$.

Proof. By induction on $H_{\mathsf{D}} \vdash u : M$. To rule out rules like λ-abstraction, we need to know that it is never the case that $(x : M) \to N \curlyveedownarrow (m_1 = n_1)$, which follows because $\leadsto_{\mathsf{p}}$ preserves the top-level constructor of a term. To handle the injectivity rules, we need to know that $\curlyveedownarrow$ is injective in the sense that $(x : M_1) \to N_1 \curlyveedownarrow (x : M_2) \to N_2$ implies $M_1 \curlyveedownarrow M_2$; again this follows by reasoning about $\leadsto_{\mathsf{p}}$.

Finally, consider the type cast rule.

$$\frac{\begin{array}{l} H_{\mathsf{D}} \vdash u_1 : M = N \\ H_{\mathsf{D}} \vdash u : M \\ H_{\mathsf{D}} \vdash N : \mathsf{Type} \end{array}}{H_{\mathsf{D}} \vdash u : N}\ \textsc{ETcast}$$

We have as an assumption that $N \curlyveedownarrow (m_1 = n_1)$, and the result would follow from the IH for u if we knew that $M \curlyveedownarrow (m_1 = n_1)$. But by the IH for u_1 we know that $N \curlyveedownarrow M$, so this follows by transitivity of $\curlyveedownarrow$. □

The soundness lemma is the place in the proof where we benefit from working with a simplified subset of the full Zombie core language. As can be seen above, it is slightly subtle, because we need the induction hypothesis itself in order to deal with the case for ETcast. If we wanted to show type safety for full Zombie, we would need a similar lemma which talks about expressions in the termination-checked language instead of about values.

With the soundness lemma in hand, canonical forms and progress follow straightforwardly.

***Lemma* 11** (Canonical forms)**.** Suppose $H_{\mathsf{D}} \vdash u : M$.

1. If $M \curlyveedownarrow (x : M_1) \to M_2$, then u is $\mathsf{rec}\, f\ x.m$.
2. If $M \curlyveedownarrow \bullet(x : M_1) \to M_2$, then u is $\mathsf{rec}\, f\ \bullet.m$
3. If $M \curlyveedownarrow D\ \overline{M_i}$ then u is $d\, \overline{u_i}$, where $\mathsf{data}\, D\, \Delta^{+}\, \mathsf{where}\, \{\, \overline{d_i\ \mathsf{of}\ \Delta_i}^{\,i \in 1..j}\, \} \in H_{\mathsf{D}}$ and d is one of the d_i.

***Theorem* 12** (Progress)**.** If $H_{\mathsf{D}} \vdash m : M$, then either m is a value or $m \leadsto_{\mathsf{cbv}} m'$ for some m'.

Chapter 4

Variations on the core language

The previous chapter formalized a subset of the Zombie core language as it is actually implemented. This is the outcome of several years of tinkering. Along the way, there are points where the language designer could reasonably choose one of several alternatives, and indeed in previous publications we have gone back and forth on some of these.

This chapter describes two such choices:

- The notion of reduction used by the typechecker can be varied a lot, from just CBV evaluation, to any approximation of contextual equivalence, or even to allowing reductions which change the termination behavior of an expression. (Sections 4.1.1–4.1.4).
- The presence of nontermination means that the application rule must be restricted. Zombie adopts the most generous application rule to date, which allows any expressions as arguments. One alternative, adopted by many other dependent languages with nontermination, is to only allow a *value-dependent* rule. That is less expressive, but it opens the door for adding more effects. (Section 4.2).

We also use the value-dependent application rule when formalizing a subset of the surface language in Chapter 5—the application rule as stated in that chapter is a restricted version of the rule as implemented in Zombie. This choice was made mainly to make it easier to compare the surface language calculus against other dependent languages when we published it as a separate paper (other language often use the value-dependent version), but it also turned out to make one key lemma easier to prove (Section 5.6.1).

4.1 Interpretations of propositional equality

As we saw in Section 3.9, the most interesting part of the type safety proof is how deal with propositional equality. In that proof we interpret equality as joinability under parallel reduction ($\rightsquigarrow_{\mathsf{p}}$). So even though at runtime terms evaluate according to $\rightsquigarrow_{\mathsf{cbv}}$, it is safe to state the equality introduction form TJOIN in terms of the relation $\rightsquigarrow_{\mathsf{p}}$ which equates more terms. The fact that the compile-time reduction relation does not have to be exactly the same as the run-time evaluation relation is curious. Just how different can we make them?

The core language described in Chapter 3 is the latest iteration of a series of similar calculi to which we have gradually added more features [71, 102, 119]. Each of these used a slightly different reduction relation for the equality introduction rule, so together they show much this choice can be varied while still making the type safety proof work.

On the other hand, the three preliminary versions had more limited features than the system in Chapter 3, so their type safety proofs do not directly carry over. The important difference is that in the three preliminary versions, types and terms are in separate syntactic categories, and propositional equality equates terms only.

4.1.1 Abstracting the equivalence relation

In previous work [71] we sought to find out how much the interpretation of equality can be varied by defining small calculus which we called $\lambda^{\cong}$, where the type system is parameterized by an abstract equivalence relation $a \cong b$. We then axiomatized a list of properties of $\cong$ that are sufficient to prove type safety. The list includes some structural properties about reasoning in a context of assumptions (which the $\cong$ relation takes as input), and also the following:

1. If $a \rightsquigarrow_{\mathsf{cbv}} b$, then $a \cong b$.
2. $a \cong b$ is an equivalence relation.
3. If $a \cong b$ then $\{v/x\}a \cong \{v/x\}b$.
4. In an empty assumption context, it is not the case that $d\ v \cong d'\ v'$ for two distinct data constructors d and d'.
5. If $d\ v \cong d\ v'$, then $v \cong v'$.

Each of these properties are motivated by particular steps in the preservation and progress proofs. When an application steps $a\ b \rightsquigarrow_{\mathsf{cbv}} a\ b'$ its type changes from $\{b/x\}\,B$ to $\{b'/x\}\,B$, so we need (1) to know that those two types are equal. To handle multiple steps we need transitivity (2). When a beta redex steps, $(\lambda x.a)\ v \rightsquigarrow_{\mathsf{cbv}}$

$\{v/x\}\, a$, we need (3) to preserve the type derivation of a. We need discrimination (4) to deal with types defined by type-level computation: we have types like $\mathsf{if}\ b\ \mathsf{then\ Nat\ else\ Bool}$, so we need to know that $\mathsf{true} \cong \mathsf{false}$ is not provable, or all types would be equal and progress would fail. Property (5) is an artifact of how the $\lambda^{\cong}$ type system is set up; in a stronger calculus (like the system defined in Chapter 3) it is provable using the other typing rules.

The $\lambda^{\cong}$ type system includes equations between terms, but not between types. In order to adapt it to a system with $\mathsf{Type} : \mathsf{Type}$ one would have to extend the discrimination property (4) to talk about type constructors as well as data constructors, to prove the canonical forms lemma (Section 3.9.5).

4.1.2 Contextual equivalence

This generalized treatment of equivalence justifies systems where the propositional equality identifies more terms. In particular, we proved that *contextual equivalence* satisfies the above list of properties. Two expressions a and b are said to be contextually equivalent if, for any evaluation context $\mathcal{E}$ and any closing substitution σ, the expression $\mathcal{E}[\sigma\ a]$ terminates iff $\mathcal{E}[\sigma\ b]$ terminates.[7] This the gold standard when reasoning about programs.

As a simple application, we see that it would be safe to use a more generous β-reduction rule. As we explained in Section 3.3, in Zombie the reduction rule only triggers when the argument to a function is a value. But since contextual equivalence is substitutive, it would also be safe to allow it to trigger whenever the argument is known to be terminating:

$$\frac{\exists v. b \leadsto^{*}_{\mathsf{cbv}} v}{(\lambda x.a)\ b = \{b/x\}\, a}$$

In a language with a termination checker, one can conservatively approximate the condition that b is a pure, terminating expression. We have studied one calculus where reduction rule is formulated in exactly this way, introducing an auxiliary judgement **val** b meaning that b is provably terminating [74].

This picture, where the equality type is seen as a decidable approximation of contextual equivalence, provides helpful intuition. The classic typing rules of e.g. pure type systems, where equivalence is hardcoded as β-convertibility, can lead the imagination in the wrong direction because they seem to suggest that the essence of dependency is to evaluate program expressions at compile-time. But that view does not provide guidance for adding dependent types to effectful programs—if your type contains a

[7] To be precise, Mason and Talcott [81] call this relation *CIU-equivalence* and prove it is equivalent to the usual definition of contextual equivalence. And while $\lambda^{\cong}$ has only term-level equations, for a system with $\mathsf{Type} : \mathsf{Type}$ one would presumably need to expand the notion of observation to also discriminate expressions headed by different type constructors, not just different termination.

`printf` expression, should the typechecker print a message? In this picture the answer is clear: the type system should fix some notion of observable effects (which may or may not include IO), and reason about which programs would be equivalent up to those observations *if they were executed*, but no actual effects should happen during type-checking.

However, one must admit that in the current framework the added generality of using contextual equivalence instead of β-convertibility is somewhat illusory. The problem is that the way we have set up the system, we only consider the equivalence of erased terms, and pay no attention to their types. Accordingly, the version of contextual equivalence that we consider is untyped contextual equivalence: two terms are equal if they behave the same in any context, not just well-typed contexts. It is a classic theorem [81] that untyped contextual equivalence contains β-convertibility, but it does not contain *much* more than that. For instance $\lambda x_{\mathsf{Nat}}.x$ and $\lambda x_{\mathsf{Nat}}.x + 0$ are not equivalent, since they behave differently when applied to a constant true (one of them crashes and one of them does not). As noted in Section 3.6.1, the inability to do this kind of type-based reasoning is a limitation of the treatment of equality in Zombie also.

4.1.3 Evaluation only

Conversely, one can ask what's the *least* ambitious we can make the equivalence relation. By looking at the list of axioms, we see that any such relation must at least contain joinability under $\leadsto_{\mathsf{cbv}}$. That is to say, if there exists c such that $a \leadsto^*_{\mathsf{cbv}} c$ and $b \leadsto^*_{\mathsf{cbv}} c$, then $a \cong b$.

One can indeed design the type system using $\leadsto_{\mathsf{cbv}}$ instead of $\leadsto_{\mathsf{p}}$, and we have fully worked out one calculus which makes this choice [102]. The formal development is mostly unsurprising, but there is one interesting interaction with the application rule. In the proof of type preservation, we need to know that when an application steps $a\ b \leadsto_{\mathsf{cbv}} a\ b'$ the result type $\{b'/x\}\,B$ is still well-kinded. In Section 3.9 the type system used parallel reduction and the preservation lemma was stated in terms of $\leadsto_{\mathsf{p}}$, so that followed by induction hypothesis. But in the system using $\leadsto_{\mathsf{cbv}}$ it is not the case that $\{b/x\}\,B \leadsto_{\mathsf{cbv}} \{b'/x\}\,B$. Instead we generalize the statement of the preservation lemma:

$$\text{If } \Gamma \vdash \{a/x\}\,a_0 : A \text{ and } a \leadsto_{\mathsf{cbv}} a', \text{ then } \Gamma \vdash \{a'/x\}\,a_0 : A.$$

Even when the type system is defined in terms of $\leadsto_{\mathsf{cbv}}$, the $\leadsto_{\mathsf{p}}$ relation is still useful in the type safety proof in order to state an intermediate lemma. This is because we need to show that the equivalence relation does not equate distinct constructors (property 4 in Section 4.1.1). In our development [102] we do this in exactly the same ways as in Section 3.9.5, by showing that term equivalence is contained in $\leadsto_{\mathsf{p}}$-joinability. By contrast, it is not contained in $\leadsto_{\mathsf{cbv}}$-joinability, because $\leadsto_{\mathsf{cbv}}$ is not closed under substitution (c.f. lemma 2 in Section 3.9.3).

This type safety proof show that it is possible to base a type system around just evaluation without going under binders. One attractive aspect of such a system is that it is very easy for the programmer to understand how the evaluation will proceed. If the language also provides a rule for substitution of equal terms (i.e. given $a = b$, conclude $\{a/x\}\, c = \{b/x\}\, c$) it turns out that this is sufficiently expressive to typecheck many simple programs. For example, if we define natural number addition `plus` by structural recursion, then CBV-evaluation is enough to prove all the properties we want.

However, when we consider slightly bigger examples where functions call other functions, we begin to need reduction under binders. For example, consider a function to look up an item in a binary search tree (c.f. Section 2.2).

```
member : (x : a) → (t : Tree a) → Bool
member = λx. rec mem t =
  case t of
    EmptyTree → False
    BranchTree t1 y t2 →
        case (compare x y) of
          LT → mem t1
          GT → mem t2
          EQ → True
```

In order to reason about this function, one wants to prove the recursion equation

```
(member x (BranchTree t1 y t2))
= (case (compare x y) of
           LT → member x t1
           GT → member x t2
           EQ → True)
```

However, under CBV reduction the left-hand-side of the equation instead reduces to

```
case (compare x y) of
           LT → (rec mem ...) t1
           GT → (rec mem ...) t2
           EQ → True
```

Bridging the gap between `member x t1` and `(rec mem ... ) t1` requires a β-reduction, but CBV-evaluation alone will not reach this subexpression because evaluation is stuck on `compare x y`.

Part of the problem is that the function `member` is written with a 1-argument `rec`-expression, where the parameter x is held constant outside the recursion. We could also implement it with a 2-argument function `rec mem x t = ... .` Then the left-hand side gets closer:

```
case (compare x y) of
           LT → (rec mem ...) x t1
           GT → (rec mem ...) x t2
           EQ → True
```

That still leaves the question of whether `member` (a variable bound by a top-level definition) and the function expression `(rec mem ...)` are considered equal. In the calculus in Chapter 3 we did not say anything about how definitions are handled, but in the Zombie implementation there is a reduction rule which replaces a defined name by its definition. (Coq calls this a δ-reduction). So even with a 2-argument rec, Zombie does not consider the two expressions convertable under CBV-evaluation, because the δ-reduction does not take place.

In full Zombie there is an even more common third variation on this problem. In order to support provably terminating programs we include a type of recursive function expressions, ind $f\ x.a$, which enforces structural recursion. However, the reduction rule for these (Section 7.3.1) introduces an extra λ-abstraction, which again prevents the equation from holding.

For each of these, one could imagine a workaround (e.g., extend the definitional equality in Chapter 5 to include top-level definitions, and phrase the reduction rule for ind-expressions in terms of hereditary substitution to eliminate any redexes that arise from substituting a λ-expression). But in general, these problems seem to suggest that having only CBV-evaluation at compile time is too brittle, and that in a practical language one wants something stronger.

In pure normalizing languages, the obvious choice is full β-reduction, which is simple to implement and provably the "best" reduction relation. But in a language which includes nontermination the choice is not so clear. Aggressively reducing under binders will diverge even for functions which are in fact total. For example, the usual definition of plus can be unfolded arbitrarily much,

```
plus x y
= case x of
    Zero → y
    Succ x' → Succ (plus x' y)
= case x of
    Zero → y
    Succ x' → Succ (case x' of
                         Zero → y
                         Succ x'' → Succ (plus x'' y))
= ...
```

This issue is known from strongly normalizing languages, and requires some restriction on when to unfold recursive calls. But with nontermination the situation is worse still, since reduction is no longer convergent. This means that there is no simple choice of which subexpression should be chosen for reduction, and it seems that checking β-convertibility has to rely on heuristics or an expensive exhaustive search.

For now, Zombie provides two possible strategies, simple CBV evaluation and a version of parallel reduction (described in more detail in Section 3.5). The surface language (Chapter 5) mentions evaluation explicitly rather than hiding it in an automatic conversion rule, partly because there is no one-size-fits-all algorithm.

4.1.4 Unrestricted β-reduction

The sharp-eyed reader may have noticed that the list of axioms in Section 4.1.1 permits equivalence relations which identify *more* terms than contextual equivalence. For example, we will not get in trouble if we make the typechecker treat an infinite loop as equal to the number 42. (Of course, we can not make the same loop equal to the number 7 at the same time, or by transitivity we would get the contradiction $42 = 7$).

Such an arbitrary identification would be silly. But there does exists a useful and tempting way to coarsen the equivalence relation: allow beta reductions with non-value arguments. That is, make $(\lambda x.a)\ b = \{b/x\}\,a$ unconditionally provable. We have developed a calculus which takes this approach [119], and proved that it still enjoys type safety and logical consistency.

One argument against this idea is that the meaning of equality becomes unclear for the programmer. Previously, proving $a = b$ meant that a and b could always be used interchangeably, whereas now it means—what exactly? Looking back at the example in Section 3.3 this variation would let us prove `safediv 3 0 (loop()) = div 3 0`, which is at least surprising. On the other hand, even ordinary contextual equivalence will ignore some important distinctions between different programs, such as time or space cost.

Whether unrestricted β-reduction is type-safe or not depends on exactly what other features are included in the language. For example, in one version of the core calculus [120] we included the uncatchable error expression abort, which can be given any well-formed type and propagates past any evaluation context by the rule $\mathcal{E}[\mathsf{abort}] \leadsto_{\mathsf{cbv}} \mathsf{abort}$. This is a standard treatment of errors. However, note that this evaluation rule conflicts with unrestricted β-reduction, e.g. $(\lambda x.3)\ \mathsf{abort}$ evaluates to abort under CBV but to 3 under CBN. We can not have both equations $((\lambda x.3)\ \mathsf{abort}) = \mathsf{abort}$ and $((\lambda x.3)\ \mathsf{abort}) = 3$ at the same time, since by transitivity all terms would be equal. So in a language with abort the unrestricted β-rule would not be type-safe.

The unrestricted β-rule may be philosophically doubtful, but there is an eminently practical reason to adopt it. It turns out to be extremely useful when writing proofs about potentially nonterminating programs. As an example, suppose we have defined natural number addition as recursive function, and want to prove associativity `(x+y)+z = x+(y+z)`. But using induction on `x`, we get stuck already in the base case, where we need to prove `0+(y+z) = (0+y)+z` for some arbitrary variables `y` and `z`. In Coq or Agda that equation would hold just by definitional equivalence. In our language, we can get a bit on the way by reducing `(0+y)+z` to `y+z`. However, the left hand side of the equation is harder to simplify. It is a function call where the argument `(y+z)` is a non-value, so we are not allowed to β-reduce it.

If the `+` function is known to be terminating (because we wrote it in a sub-language for which we have a termination checker), then this equation is provable. One can either use the generalized β-rule described in section 4.1.2, or in Zombie one can get the same effect by adding a let-expression naming the subexpression `y+z` (see Section 5.6.3). But the whole point of this research is to abolish the requirement for termination-checking, and as this example illustrated, in such a setting the CBV-respecting β-rule is rather weak.

Of course, addition *is* associative with respect to the ordinary notion of contextual equivalence. So rather than bringing out the sledgehammer of unrestricted β-reduction, we would ideally like to strengthen the equational reasoning enough to make this and similar theorems provable, without making it strong enough to prove false statements. We discuss some possible directions for this in Section 7.7.1.

4.1.5 Erasure as a form of unrestricted β-reduction

There is a connection between the interpretation of equality and the reduction rule for irrelevant applications. In full Zombie, any terminating expression b may be marked as irrelevant, and the reduction rule proves $\mathsf{join} : (\lambda^{\bullet}_{x:A}.a)\ \bullet_b = \{b/x\}\ a$. This involves substituting a non-value b, even though Zombie is a CBV language. However, as long as b terminates there is no cause for concern; as mentioned in Section 4.1.2, we could consistently assume this equation even for computationally relevant applications because it is included in CBV contextual equivalence.

Obviously we need *some* restriction on what expressions are allowed as irrelevant arguments, because if a function uses an erased variable as the proof of a type cast we had better know that there exists a corresponding proof term (c.f. the `safediv` example in Section 3.8). But one may wonder if it would be possible to be even more permissive than Zombie. In the `safediv` example the erased argument belonged to the uninhabited type $0 \neq 0$. Perhaps it would be safe to erase any argument, terminating or not, as long as it belonged to a type inhabited by some value?

However, doing so would involve substituting nonterminating expressions for variables. This is similar to the unrestricted β-reduction discussed in section 4.1.4, and at least as soon as one adds a control effect like abort it leads to inconsistency. In particular, the following example shows that we can get in trouble by erasing an abort of type Nat (a type which is very inhabited). First, since the reduction relation treats variables as values, $(\lambda x.\mathsf{Bool})\ x \leadsto_{\mathsf{p}} \mathsf{Bool}$. So we have $\mathsf{join} : ((\lambda x_{\mathsf{Nat}}.\mathsf{Bool})\ x = (\lambda x_{\mathsf{Nat}}.\mathsf{Nat})\ x) = (\mathsf{Bool} = \mathsf{Nat})$. Then the following term typechecks:

$$\lambda^{\bullet}_{x:\mathsf{Nat}}.\lambda h_{((\lambda x_{\mathsf{Nat}}.\mathsf{Bool})\ x)=((\lambda x_{\mathsf{Nat}}.\mathsf{Nat})\ x)}.h_{\triangleright\mathsf{join}}$$

$$: \bullet(x:\mathsf{Nat}) \to\ (h:(\lambda x_{\mathsf{Nat}}.\mathsf{Bool})\ x = (\lambda x_{\mathsf{Nat}}.\mathsf{Nat})\ x) \to\ (\mathsf{Bool} = \mathsf{Nat})$$

On the other hand, by our reduction rule for error terms $(\lambda x.\mathsf{Bool})\ \mathsf{abort} \leadsto_{\mathsf{p}} \mathsf{abort}$, so

$$\mathsf{join} : ((\lambda x_{\mathsf{Nat}}.\mathsf{Bool})\ \mathsf{abort}_{\mathsf{Nat}}) = ((\lambda x_{\mathsf{Nat}}.\mathsf{Nat})\ \mathsf{abort}_{\mathsf{Nat}}).$$

So if we allowed $\mathsf{abort}_{\mathsf{Nat}}$ to be given as an irrelevant argument, then we could write a terminating proof of $\mathsf{Bool} = \mathsf{Nat}$. Note that all the equality proofs involved are just join, so this example does not depend on type casts being computationally irrelevant.

4.2 General versus value-dependent application

As discussed in Section 3.4, our application rule contains a premise $\Gamma \vdash \{b/x\}\, B : \mathsf{Type}$ which is not needed pure languages like Coq and Agda, but is needed to account for the various value-restrictions in our typing rules:

$$\frac{\begin{array}{l}\Gamma \vdash a : (x\!:\!A) \to\ B \\ \Gamma \vdash b : A \\ \Gamma \vdash \{b/x\}\, B : \mathsf{Type}\end{array}}{\Gamma \vdash a\ b : \{b/x\}\, B}\textsc{Tapp}$$

In our experience, this restriction essentially never rules out interesting programs. It is possible to construct examples where it triggers, but they are quite artificial. There are essentially two sources of value restrictions in the language. First, the β-reduction rule requires function arguments to be values, so if `f : Nat → Nat`, then we have e.g.

```
join : (((λ y.0) : Nat→Nat) x) = 0
```

but not

```
join : (((λ y.0) : Nat→Nat) (f 3)) = 0
```

So if the context contained a function `g`

```
g : (x : Nat) →  (join : (((λ y.0) : Nat→Nat) x) = 0)=(join : 0 = 0)
```

then the type of g is well-formed, but an application like g (f 3) would be disallowed because the resulting type would not be well-formed.

There is also a value restriction on the type-cast rule, $a_{\triangleright v}$. So similarly, we could construct a typing context where the type of g requires a type cast to be wellkinded:

```
data T (n : Nat) : Type where
  mkT

S : T 0  → Type
n : Nat
g : (t: T n) → (n = 0) → S t
f : Unit → (n = 0)
```

Here the typechecker has to implicitly insert a cast (using the method described in Chapter 5) to make the return type S t well-formed. Then an application like g mkT (f()) is disllowed, because the expression f() is not known to terminate, so it cannot be used as the proof of the cast. The full Zombie language includes a more general system for termination checking (Chapter 7), which leads to analogous value-restrictions.

Both these examples also illustrate why the well-formedness restriction is rarely a problem in practice: the value restrictions have to do with what expressions count as valid proofs, and while it is common for dependent programs to involve proof-*terms*, it is less common to need nontrivial proofs just to state a type.

4.2.1 Value-dependent application

Several other languages with effects, such as Aura [70] and F* [133], adopt a more restricted typing rule for applications. They allow any application when the applied function has a simple type, but in cases when there is an actual dependency (i.e. the applied function has type $(x\!:\!A) \to\ B$ and x occurs in B), the argument must be a syntactic value:

$$\frac{\Gamma \vdash a : A \to B \qquad \Gamma \vdash b : A}{\Gamma \vdash a\ b : B}\,\textsc{TsimplApp} \qquad\qquad \frac{\Gamma \vdash a : (x\!:\!A) \to\ B \qquad \Gamma \vdash v : A}{\Gamma \vdash a\ v : \{v/x\}\,B}\,\textsc{TvalApp}$$

The value-restricted rule simplifies the metatheory, because there is no need to consider types which contain non-value expressions.

Zombie subsumes both these rules. Because the language satisfies regularity, $\Gamma \vdash a : (x : A) \to\ B$ implies $\Gamma, x : A \vdash B : \mathsf{Type}$. So given the premises of the two rules above we can satisfy the additional well-formedness premise of our application rule using strengthening (to get $\Gamma \vdash B : \mathsf{Type}$) or substitution of values (to get $\Gamma \vdash \{v/x\}\, B : \mathsf{Type}$).

Zombie's more expressive rule is easy to implement: instead of checking that the argument is a value, the typechecker makes an extra call to verify that the return type is well-kinded. Because function types are small this is quick, and if the check fails the error message is as easy to understand for the programmer as any other type error.

One may then ask how much extra expressivity the more general rule provides. The intuition from simply-typed languages may be that it does not matter at all, since we can always rewrite `a b` as `let x = b in a x`. Type systems for effects and monads often enforce such let-sequencing just to simplify the presentation.

However, in a dependent system let-expansion is not always so benign, since it changes the type of the application. We saw in Section 3.8.3 that it is too restrictive to limit computationally irrelevant function arguments to syntactic values when they are used as type indices. Indeed, F* offers the possibility to index types by "propositional formulas", which are syntactically separate from ordinary expressions and have no value restriction.

The value-restricted application rule also limits the ways one can write proof terms. It fits quite well with nonterminating programs written in the "internal" verification style, where the type of the function says that the returned value has the desired property. For example in the function `match` from Section 3.3,

```
match : (s:String) → (r:Regexp) → Maybe (Match s r)
```

the type constructor `Match` is applied to only variables.

Being able to apply functions to non-values becomes more interesting if we extend the language with a termination-checked fragment and start writing external proofs. For example, a programmer may want to first prove a lemma about addition

```
log plus_zero : (n:Nat) → plus n 0 = n
```

and then instantiate the lemma to prove a theorem about a particular expression in the logical fragment.

```
plus_zero (f x) : plus (f x) 0 = (f x).
```

This calls the function with a non-value expression, and creates a return type containing non-values.

4.2.2 Effects beyond nontermination

The generous application rule is implemented in Zombie, and seems very suitable for CBV-languages where the only effect is nontermination/error. On the other hand, it is not clear if the intuitions behind it will extend to systems with a richer class of effects. In particular, trying to add ML-style references to Zombie while keeping these type rules runs into at least two problems.

First, without memory effects it is the case that if $a \leadsto_{\mathsf{cbv}} a'$ then a and a' are contextually equivalent. With memory effects this is not true, and the best we can say (as in Mason and Talcott [81]) is that the pairs (S, a) and (S', a') are equivalent, where S is a syntactic representation of the store. This creates problems with our style of preservation proof, since we want to say that when an application $a\ b : \{b/x\}\,A$ steps to $a\ b' : \{b'/x\}\,B$, then the type is preserved because b and b' are equivalent. For example, in a store where the location `l` maps to `3`, we might expect to type

```
(λ (y:Nat) (q: !l = y) . q) (!l) join : !l = !l
```

But this steps to

```
(λ (y:Nat) (q: !l = y) . q) 3 join
```

which is probably not typeable—while `!l = !l` is justified by contextual equivalence, `!l = 3` is not.

Second, in the presence of memory effects, a type is not necessarily "well-behaved" just because it is well-kinded. For example, if `x` is a variable of type `Unit`, then it seems reasonable to be able to prove `(l:=2;x;!l)=(l:=2;();!l)`, since variables should range over values and so have no effects. That would allow writing a function

```
f : (x:Unit) → ((l:=2;x;!l)=(l:=2;();!l)).
```

On the other hand, the function call `f (l:=3)` should be disallowed, since the resulting type `(l:=2;l:=3;!l)=(l:=2;();!l)` is bad. However, although this type is plainly false it is well-kinded, so the premise $\Gamma \vdash \{b/x\}\,B : \mathsf{Type}$ does not rule it out.

The big advantage of restricting the application rule to value-dependency is that it can accommodate memory effects like this. For example F* supports ML-style references. Because F* prohibits non-values from appearing anywhere inside a type, in particular location references or assignments can never occur in a type. The condition one really needs is that the expressions appearing in types do not have any read or write memory effects, so it is safe to allow slightly more than just syntactic values. For example, Deputy [34] allows dependency on "pure expressions".

Thus, by being more restrictive about what kind of proofs the programmer can write, these languages can be more liberal about what effects they support. Of course, the restriction to pure expressions also precludes stating any interesting specifications about memory effects. To support that requires additional machinery in the programming language.

Chapter 5

Programming up to congruence

The Zombie core language provides a solid foundation—as the work in Chapter 3 and in Casinghino's dissertation [30] shows, it is type safe, logically consistent, and simple enough to be checked by a small trusted kernel. However, it is not feasible to write programs directly in the core language, because the terms get cluttered with *type annotations* and *type casts*. In this chapter we turn to the other half of the language design: crafting a programmer-friendly surface language which elaborates into the core.

We reduce the required number of type annotations by using bidirectional type checking, which is a standard technique. However, inferring type casts is more interesting. Most dependently-typed languages are able to omit many proofs of type equality because they define types to be equal when they are (at least) β-convertible, but in Zombie this is awkward because of nontermination. To check whether two types are β-equivalent the type checker has to evaluate expressions inside them, which becomes problematic if expressions may diverge—what if the type checker gets stuck in an infinite loop? The best we can say is that type checking terminates if the type-expression being checked terminates [5]. Existing languages fix an arbitrary global cut off for how many steps of evaluation the typechecker is willing to do (Cayenne [10]), or only reduce expressions that have passed a conservative termination test (Idris [27]).

Zombie, somewhat radically, omits automatic β-conversion completely. Instead, β-equality is available only through explicit conversion. Because Zombie does not include automatic β-conversion, it provides an opportunity to explore an alternative definition of equivalence in a surface language design.

Congruence closure is a basic operation in automatic theorem provers for first-order logic (particularly SMT solvers, such as Z3 [44]). Given some context Γ which contains assumptions of the form $a = b$, the congruence closure of Γ is the set of equations which are deducible by reflexivity, symmetry, transitivity, and changing subterms. Figure 5.1 specifies the congruence closure of a given context.

$$\frac{a = b \in \Gamma}{\Gamma \vdash a = b} \qquad \frac{}{\Gamma \vdash a = a} \qquad \frac{\Gamma \vdash b = a}{\Gamma \vdash a = b}$$

$$\frac{\Gamma \vdash a = c \quad \Gamma \vdash c = b}{\Gamma \vdash a = b} \qquad \frac{\Gamma \vdash a = a' \quad \Gamma \vdash b = b'}{\Gamma \vdash a\ b = a'\ b'}$$

Figure 5.1: The "classic" congruence closure relation for untyped first-order logic terms

Although efficient algorithms for congruence closure are well-known [49, 96, 118] this reasoning principle has seen little use in dependently-typed programming languages. This is not for lack of opportunity: programmers in dependently typed languages write lots of equality proofs. However, the adaption of this first-order technique to the higher-order logics of dependently-typed languages is not straightforward. The combination of congruence closure and full β-reduction makes the equality relation undecidable (Section 8.3.2). As a result, most dependently-typed languages take the conservative approach of only incorporating congruence closure as a meta-operation, such as Coq's `congruence` tactic. While this tactic can assist with the creation of equality proofs, such proofs must still be explicitly eliminated. Proposals to use equations from the context automatically [7, 124, 126] have done so *in addition* to β-reduction, which makes it hard to characterize exactly which programs will typecheck, and also leaves open the question of how expressive congruence closure is in isolation.

We define the surface language in this chapter to be fully "up to congruence", i.e. types which are equated by congruence closure can always be used interchangeably, and then show how the elaborator can implement this type system. Specifically, this involves the following contributions:

- We define a typed version of the congruence closure relation that is compatible with our core language, including features (erasure, injectivity, and generalized assumption) suitable for a dependent type system (Section 5.2).
- We specify the surface language using a bidirectional type system that uses this congruence closure relation as its *definition* of type equality (Section 5.3).
- We define an elaboration algorithm of the surface language to the core language (Section 5.4) based on a novel algorithm for typed congruence closure (Section 5.5). We prove that our elaboration algorithm is complete for the surface language and produces well-typed core language expressions. Our typed congruence closure algorithm both decides whether two terms are in the relation and also produces core language equality proofs.
- The full Zombie implementation extends the ideas in the chapter to also cover datatypes, pattern matching, the full application rule, etc. Congruence closure works well in this setting; in particular, it significantly simplifies the typing

rules for case-expressions (Section 5.6).

- The Zombie implementation also provides a facility for reducing expressions modulo equations in the context, which makes it easier to write external proofs (Section 5.6.3).

In order to make the proofs smaller, we only treat a subset of the full Zombie language. Although it elaborates into the core language from Chapter 3, the type system for the surface language in this chapter omits all rules dealing with datatypes, and uses the value-restricted restricted form of the application rule (which we described in Section 4.2). The actual Zombie implementation of course also handles datatypes and general application, as we describe in Section 5.6. The full proofs are given in Appendix A, so in this chapter only the statements of the lemmas are quoted.

5.1 Type annotations and type casts

In order to get an idea of what parts of core language terms can be inferred by our system, consider as an example this simple proof in Agda, which shows that zero is a right identity for addition.

```
npluszero : (n : Nat) → n + 0 ≡ n
npluszero zero    = refl
npluszero (suc m) = cong suc (npluszero m)
```

The proof follows by induction on natural numbers. In the base case, `refl` is a proof of `0 = 0`. In the next line, `cong` translates a proof of `m + 0 ≡ m` (from the recursive call) to a proof of `suc(m + 0) ≡ suc m`.

This proof relies on the fact that Agda's propositional equality relation (≡) is reflexive and a congruence relation. The former property holds by definition, but the latter must be explicitly shown. In other words, the proof relies on the following lemma:

```
cong : ∀ {A B} {m n : A}
       → (f : A → B) → m ≡ n → f m ≡ f n
cong f refl = refl
```

Now compare this proof to a similar result in Zombie. The same reasoning is present: the proof follows via natural number induction, using the reduction behavior of addition in both cases.

```
npluszero : (n : Nat) → (n + 0 = n)
npluszero n =
  case n [eq] of
   Zero → (join : 0 + 0 = 0)
   Suc m → let _ = npluszero m in
             (join : (Suc m) + 0 = Suc (m + 0))
```

Because Zombie does not provide automatic β-equivalence, reduction must be made explicit above. The term `join` explicitly introduces an equality based on reduction. However, in the successor case, the Zombie type checker is able to infer exactly how the equalities should be put together.

For comparison, the corresponding Zombie core language term includes a number of explicit type casts:

```
npluszero : (n : Nat) → (n + 0 = n)
npluszero (n : Nat) =
  case n [eq] of
   Zero → join [↝ 0 + 0 = 0]
              ▷ join [~eq + 0 = ~eq]
   Suc m →
     let ih = npluszero m in
      join [↝ (Suc m) + 0 = Suc (m + 0)]
        ▷ join [(Suc m) + 0 = Suc ~ih]
        ▷ join [~eq + 0 = ~eq]
```

Comparing the core language version against the surface language version, we see several improvements. First, the core term required a type annotation `(n : Nat)` on the argument to the recursive function. In the surface language that can be omitted, because bidirectional typechecking propagates the same information from the top-level type declaration.

More interestingly, several type casts $(a \triangleright b)$ in the core term can be omitted from the surface version. In core Zombie, the equality proof b can be formed in two ways, either via β-reduction (e.g. the proof `join [↝ 0 + 0 = 0]` above), or by congruence (e.g. the proof `join [~eq + 0 =  ~eq]` above, which proves $((0+0) = 0) = ((n+0) = n)$). In the above example, both kinds of reasoning are used. In the successor branch, we can express the reasoning in words as follows. We have $(\mathsf{Suc}\ m)+0 = \mathsf{Suc}\ (m+0)$ (by reduction, using the definition of plus). This is the same as $(\mathsf{Suc}\ m)+0 = \mathsf{Suc}\ m$ (by the IH, which states $m+0 = m$). And this in turn is the same as $n+0 = n$ (using hypothesis $eq : n = \mathsf{Suc}\ m$, which comes from the typing rule for pattern matching). The surface language term can omit the latter two casts because they were proved by congruence, which makes the term less cluttered.

On the other hand, the surface language still requires `join` expressions with explicit annotations. The annotations specify what terms should be reduced. For example, without the annotation in the first branch, the typechecker would have tried to reduce `join n n`, which gets stuck on a pattern match on `n`.

In this example, we just want to reduce the expression as much as possible. The Zombie typechecker includes some additional support to reduce expressions while taking equations in the context into account, with the syntax `unfold/smartjoin`

(Section 5.6.3). Using that feature, we can write a more streamlined version of the surface language term:

```
npluszero : (n : Nat) → (n + 0 = n)
npluszero n =
 case n [eq_n] of
    Zero → smartjoin
    Succ m → let _ = npluszero m in
                 smartjoin
```

The Zombie surface language is “the dual” of intensional type theories like Coq and Agda: while they automatically use equalities that follow from β-reductions but do not automatically use assumptions from the context, Zombie uses assumptions but does not automatically reduce expressions.

In the case of `npluszero`, the proof in Zombie ended up slightly longer than a similar proof in Coq or Agda would. In our experience, this is quite typical: most equational proofs tend to make more heavy use of β-reduction than of congruence reasoning, so writing them Zombie a little more clumsy than in Coq or Agda. However, one should keep in mind that Zombie is solving a harder problem, because programs are not restricted to be strongly normalizing. The payoff is that functional programs that do not make heavy use of equational reasoning, but which are written using general recursion, require less ceremony.

5.2 Congruence closure

The driving idea behind our surface language is that the programmer should never have to explicitly write a type cast $a_{\triangleright v}$ if the proof v can be inferred by congruence closure. In this section we exactly specify which proofs can be inferred, by defining the typed congruence closure relation $\Gamma \vDash a = b$ shown in Figure 5.2.

Like the usual congruence closure relation for first-order terms (Figure 5.1), the rules in Figure 5.2 specify that this relation is reflexive, symmetric and transitive. It also includes rules for using assumptions in the context and congruence by changing subterms. However, we make a few changes:

First, we add typing premises (in TCCRefl and TCCErasure) to make sure that the relation only equates well-typed and fully-annotated core language terms. In other words,

$$\text{If } \Gamma \vDash a = b, \text{ then } \Gamma \vdash a : A \text{ and } \Gamma \vdash b : B.$$

Next, we adapt the congruence rule so that it corresponds to the TJsubst rule of the core language. In particular, the rule TCCcongruence includes an explicit erasure

$\boxed{\Gamma \vDash a = b}$

$$\frac{\Gamma \vdash a : A}{\Gamma \vDash a = a}\,\text{TCC}\textsc{Refl} \qquad \frac{\Gamma \vDash a = b}{\Gamma \vDash b = a}\,\text{TCC}\textsc{Sym} \qquad \frac{\Gamma \vDash a = b \quad \Gamma \vDash b = c}{\Gamma \vDash a = c}\,\text{TCC}\textsc{Trans}$$

$$\frac{\begin{array}{l} |a| = |b| \\ \Gamma \vdash a : A \quad \Gamma \vdash b : B \end{array}}{\Gamma \vDash a = b}\,\text{TCC}\textsc{Erasure} \qquad \frac{\begin{array}{l} x : A \in \Gamma \\ \Gamma \vDash A = (a = b) \end{array}}{\Gamma \vDash a = b}\,\text{TCC}\textsc{Assumption}$$

$$\frac{\begin{array}{l} \Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k \\ |A = B| = |\{a_1/x_1\} \ldots \{a_j/x_j\}\ c = \{b_1/x_1\} \ldots \{b_j/x_j\}\ c| \end{array}}{\Gamma \vDash A = B}\,\text{TCC}\textsc{congruence}$$

$$\frac{\Gamma \vDash (a_1 = a_2) = (b_1 = b_2)}{\Gamma \vDash a_k = b_k}\,\text{TCC}\textsc{InjEq}$$

$$\frac{\Gamma \vDash ((x\!:\!A_1) \to\ B_1) = ((x\!:\!A_2) \to\ B_2)}{\Gamma \vDash A_1 = A_2}\,\text{TCC}\textsc{InjDom}$$

$$\frac{\Gamma \vDash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vDash B_1 = B_2}\,\text{TCC}\textsc{InjRng}$$

$$\frac{\Gamma \vDash (\bullet(x\!:\!A_1) \to\ B_1) = (\bullet(x\!:\!A_2) \to\ B_2)}{\Gamma \vDash A_1 = A_2}\,\text{TCC}\textsc{IInjDom}$$

$$\frac{\Gamma \vDash (\bullet A_1 \to\ B_1) = (\bullet A_2 \to\ B_2)}{\Gamma \vDash B_1 = B_2}\,\text{TCC}\textsc{IInjRng}$$

Figure 5.2: Typed congruence closure relation

step so that the two sides of the equality can differ in their erasable portions.

Furthermore, we extend the relation in several ways.[8] The rule TCCErasure makes sure that the programmer can ignore all annotations when reasoning about programs. Because irrelevant arguments $a\ \bullet_b$ are considered to be annotations, this also means that our surface language automatically makes use of computational irrelevance, at no extra implementation cost.

Also, we reason up to injectivity of datatype constructors (in rules TCCInjDom, TCCInjRng, and TCCInjEq). As mentioned in Section 3.6 these rules are valid in

[8] Systems based around congruence closure often strengthen their automatic theorem prover in some way, e.g. Nieuwenhuis and Oliveras [97] add reasoning about natural number equations, and the Coq `congruence` tactic automatically uses injectivity of data constructors [39].

the core language, and we will see in Section 5.4 that there is good reason to make the congruence closure algorithm use them automatically. Note that we restrict the rules TCCInjrng and TCCIinjrng so that they apply only to nondependent function types; we explain this restriction in Section 5.4.

Finally, the rule TCCAssumption is a bit stronger than the classic rule from first order logic. In the first-order logic setting, this rule is defined as just the closure over equations in the context:

$$\frac{x : a = b \in \Gamma}{\Gamma \vDash a = b}$$

However, in a dependently typed language, we can have equations between equations. In this setting, the classic rule does not respect CC-equivalence of contexts. For example, it would prove the first of the following two problem instances, but not the second.

$$x\!:\!\mathsf{Nat}, y\!:\!\mathsf{Nat}, a\!:\!\mathsf{Type}, h_1\!:\!(x = y) = a,\, h_2\!:\!x = y \vDash x = y$$

$$x\!:\!\mathsf{Nat}, y\!:\!\mathsf{Nat}, a\!:\!\mathsf{Type}, h_1\!:\!(x = y) = a,\, h_2\!:\!a \qquad \vDash x = y$$

Therefore we replace the rule with the stronger version shown in the figure.

We were led to these strengthened rules by theoretical considerations when trying to show that our elaboration algorithm was complete with respect to the declarative specification (see Section 5.4). Once we implemented the current set of rules, we found that they were useful in practice as well as in theory, because they improved the elaboration of some examples in our test suite.

The stronger assumption rule is useful in situations where type-level computation produces equality types, for example when using custom induction principles. Say we want to prove a theorem $\forall n.f(n) = g(n)$ by first proving that course-of-values induction holds for any predicate $P : \mathsf{Nat} \to \mathsf{Type}$, and then instantiating the induction lemma with $P := (\lambda n.f(n) = g(n))$. Then in the step case after calling the induction hypothesis on some number m, the context will contain $H : P(m)$, and by β-reduction we know that $P(m) = (f(m) = g(m))$. In that situation, the extended assumption rule says that H should be used when constructing the congruence closure of the context, even if the programmer does not apply an explicit type cast to H, which accords with intuition.

5.3 Surface language

Next, we give a precise specification of the surface language, which shows how type inference can use congruence closure to infer casts of the form $a_{\triangleright v}$. This process involves determining both the location of such casts and the proof of equality v.

Figures 5.3 and 5.4 define a *bidirectional type system* for a partially annotated lan-

$\boxed{\Gamma \vdash a \Rightarrow A}$

$$\frac{\vdash \Gamma \Leftarrow}{\Gamma \vdash \mathsf{Type} \Rightarrow \mathsf{Type}}\textsc{Itype} \qquad \frac{\vdash \Gamma \Leftarrow \quad x : A \in \Gamma \quad \Gamma \vdash A \Leftarrow \mathsf{Type}}{\Gamma \vdash x \Rightarrow A}\textsc{Ivar}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma, x : A \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash (x\!:\!A) \to\ B \Rightarrow \mathsf{Type}}\textsc{Ipi} \qquad \frac{\begin{array}{l}\Gamma \vdash a \Rightarrow (x\!:\!A) \to\ B\\ \Gamma \vdash v \Leftarrow A\\ \Gamma \vDash^{\exists}\ \mathsf{injrng}\,(x\!:\!A) \to\ B\,\mathsf{for}\ v\\ \Gamma \vdash \{v_A/x\}\,B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ v \Rightarrow \{v_A/x\}\,B}\textsc{Idapp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow A \to B \quad \Gamma \vdash b \Leftarrow A\\ \Gamma \vdash B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ b \Rightarrow B}\textsc{Iapp} \qquad \frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma, x : A \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash \bullet(x\!:\!A) \to\ B \Rightarrow \mathsf{Type}}\textsc{Iipi}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow \bullet(x\!:\!A) \to\ B\\ \Gamma \vdash v \Leftarrow A\\ \Gamma \vDash^{\exists}\ \mathsf{injrng}\ \bullet\,(x\!:\!A) \to\ B\,\mathsf{for}\ v\\ \Gamma \vdash \{v_A/x\}\,B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ \bullet_v \Rightarrow \{v_A/x\}\,B}\textsc{Iidapp}$$

$\boxed{\Gamma \vdash a \Leftarrow A}$

$$\frac{\begin{array}{l}\Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash a \Leftarrow A_2\\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vDash^{\exists}\ \mathsf{injrng}\,(x\!:\!A_1) \to\ A_2\,\mathsf{for}\ x\\ \Gamma, f : (x\!:\!A_1) \to\ A_2 \vdash (x\!:\!A_1) \to\ A_2 \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{rec}\, f\ x.a \Leftarrow (x\!:\!A_1) \to\ A_2}\textsc{Crec}$$

$$\frac{\begin{array}{l}\Gamma, f : \bullet(x\!:\!A_1) \to\ A_2, x : A_1 \vdash a \Leftarrow A_2\\ x\ \notin \mathsf{FV}\,(|a|)\\ \Gamma, f : \bullet(x\!:\!A_1) \to\ A_2, x : A_1 \vDash^{\exists}\ \mathsf{injrng}\ \bullet\,(x\!:\!A_1) \to\ A_2\,\mathsf{for}\ x\\ \Gamma, f : \bullet(x\!:\!A_1) \to\ A_2 \vdash \bullet(x\!:\!A_1) \to\ A_2 \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{rec}\, f\ \bullet.a \Leftarrow \bullet(x\!:\!A_1) \to\ A_2}\textsc{Cirec}$$

Figure 5.3: Surface language typing: functions and variables

$$\boxed{\Gamma \vdash a \Rightarrow A} \qquad\qquad \boxed{\Gamma \vdash a \Leftarrow A}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow A\\ \Gamma \vDash^{\exists} A = B \quad \Gamma \vdash B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a \Rightarrow B}\textsc{Icast} \qquad \frac{\begin{array}{l}\Gamma \vdash a \Leftarrow A\\ \Gamma \vDash^{\exists} A = B \quad \Gamma \vdash B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a \Leftarrow B}\textsc{Ccast}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma \vdash a \Leftarrow A}{\Gamma \vdash a_A \Rightarrow A}\textsc{Iannot} \qquad \frac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash a \Leftarrow A}\textsc{Cinf}$$

$$\frac{\begin{array}{c}\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type}\\ |a_1| \leadsto^{i}_{\mathsf{cbv}} b \quad |a_2| \leadsto^{j}_{\mathsf{cbv}} b\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\textsc{Ijoinc} \qquad \frac{\Gamma \vDash^{\exists} a = b}{\Gamma \vdash \mathsf{join} \Leftarrow a = b}\textsc{Crefl}$$

$$\frac{\begin{array}{c}\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type}\\ |a_1| \leadsto^{i}_{\mathsf{p}} b \quad |a_2| \leadsto^{j}_{\mathsf{p}} b\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{p}} i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\textsc{Ijoinp}$$

$$\boxed{\Gamma \vDash^{\exists} a = b}$$

$$\frac{\Gamma' \vDash a' = b' \quad |a| = |a'| \quad |b| = |b'| \quad |\Gamma| = |\Gamma'|}{\Gamma \vDash^{\exists} a = b}\textsc{Eeq}$$

$$\boxed{\Gamma \vDash^{\exists} \mathsf{injrng}\, A\, \mathsf{for}\, v}$$

$$\frac{\begin{array}{l}\forall A' B'.\ \Gamma \vDash^{\exists} ((x\!:\!A) \to B) = ((x\!:\!A') \to B')\\ \qquad\qquad \text{implies } \Gamma \vDash^{\exists} \{v/x\}\, B = \{v/x\}\, B'\end{array}}{\Gamma \vDash^{\exists} \mathsf{injrng}\, (x\!:\!A) \to B\, \mathsf{for}\, v}\textsc{EIRipi}$$

$$\frac{\begin{array}{l}\forall A' B'.\ \Gamma \vDash^{\exists} (\bullet(x\!:\!A) \to B) = (\bullet(x\!:\!A') \to B')\\ \qquad\qquad \text{implies } \Gamma \vDash^{\exists} \{v/x\}\, B = \{v/x\}\, B'\end{array}}{\Gamma \vDash^{\exists} \mathsf{injrng}\, (x\!:\!A) \to B\, \mathsf{for}\, v}\textsc{EIRipi}$$

Figure 5.4: Surface language typing: equality

guage. This type system is defined by two (mutually defined) judgements: *type synthesis*, written $\Gamma \vdash a \Rightarrow A$, and *type checking*, written $\Gamma \vdash a \Leftarrow A$. Here Γ and a are always inputs, but A is an output of the synthesizing judgement and an input of the checking judgement.

A few of the rules have side conditions of the form $\Gamma \vDash^{\exists} \mathsf{injrng}\, A\, \mathsf{for}\, v$, which is a separate judgement. Roughly, it states that the set of equations in the context Γ is closed under injectivity of function arrows, at least with respect to the expressions A and v. This condition is virtually always satisfied in practical programs, but it is needed to ensure completeness of our type checking and elaboration algorithm. We will leave the details of it mysterious for now and come back to it in when we discuss elaboration in Section 5.4.

Bidirectional systems are a standard form of local type inference. In such systems, the programmer must provide types for top-level definitions, and those definitions are then *checked* against the ascribed types. As a result, some type annotations can be omitted, e.g. in a definition like

```
foo : Nat → Nat → Nat
foo = λ x y . 2*x + y
```

there is no need for type annotations on the bound variables x and y, since the function is checked against a known top-level type.

Most rules of this type system are standard for bidirectional systems [109], including the rules for inferring the types of variables (IVar), the well-formedness of types (Ieq, Itype, and Ipi), non-dependent application (Iapp), and the mode switching rules Cinf and Iannot. Any term that has enough annotations to synthesize a type A also checks against that type (Cinf). Conversely, some terms (e.g. functions) require a known type to check against, and so if the surrounding context does not specify one, the programmer must add a type annotation (Iannot).

The rules Icast and Ccast in Figure 5.4 specify that checking and inference work "up-to-congruence." At any point in the typing derivation, the system can replace the inferred or checked type with something congruent. The notation $\Gamma \vDash^{\exists} A = B$ lifts the congruence closure judgement from Section 5.2 to the partially annotated surface language. These two rules contain kinding premises to maintain well-formedness of types.

The rule for checking functions (Crec) is almost identical to the corresponding rule in the core language, with just two changes. First, the programmer can omit the types A_1, and A_2, because in a bidirectional system they can be deduced from the type the expression is checked against. Second, the new premise injrng slightly restricts the use of this rule (as we will explain in Section 5.4).

Equations that are provable via congruence closure are available via the checking rule, CREFL. In this case the proof term is just join, written as an underscore in the concrete syntax. Because this is a checking rule, the equation to be proved does not have to be written down directly if it can be inferred from the context. We used this feature extensively in the examples in Chapter 2.

The rule IJOINP proves equations using the operational semantics. We saw this rule used in the `npluszero` example, written `join : 0 + 0 = 0` in the concrete syntax. Note that the programmer must explicitly write down the terms that should be reduced. The rule IJOINP is a synthesizing rather than checking rule in order to ensure that the typing rules are effectively implementable. Although the type system works "up to congruence" the operational semantics do not. So the expression itself needs to contain enough information to tell the typechecker which member of the equivalence class should be reduced—it cannot get this information from the checking context. (In practice, having to explicitly write this annotation can be annoying. The Zombie implementation includes a feature `smartjoin` which can help—see Section 5.6.3).

It is also interesting to note the rules that do *not* appear in Figures 5.3 and 5.4. For example, there is no rule or surface syntax corresponding to TCAST, because this feature can be written as a user-level function. Similarly, the rather involved machinery for rewriting subterms and erased terms (rule TJSUBST) can be entirely omitted, since it is subsumed by the congruence closure relation. The programmer only needs to introduce the equations into the context and they will be used automatically.

Finally we note that the surface language does not satisfy some of the usual properties of type systems. In particular, it does not satisfy a substitution lemma because that property fails for the congruence closure relation. (We might expect that $\Gamma, x : C \vDash a = b$ and $\Gamma \vdash v : C$ would imply $\Gamma \vDash \{v/x\}\, a = \{v/x\}\, b$. But this fails if C is an equation and the proof v makes use of the operational semantics.) It lacks a general weakening lemma, because the the injrng relation is not closed under arbitrary weakenings—if you add an equation to the context, you also need to add all the relevant consequences of injectivity. (The consequences are provable, but the programmer has to prove and add them manually.) And it does not satisfy a strengthening lemma, because even variables that do not occur in a term may be implicitly used as assumptions of congruence proofs.

The situations where weakening and substitution fail are rare (we have never encountered one when writing example programs in Zombie) and there are straightforward workarounds for programmers. Furthermore, these properties do hold for fully annotated expressions, so there are no restrictions on the output of elaboration. However, the typing rules for the declarative system must be formulated to avoid these issues, which requires some extra premises. The rule IVAR requires $\Gamma \vdash A \Leftarrow \mathsf{Type}$ (proving this from $\vdash \Gamma \Leftarrow$ would need weakening); IAPP requires $\Gamma \vdash B \Leftarrow \mathsf{Type}$ (proving this from $\Gamma \vdash A \to B : \mathsf{Type}$ would need strengthening); and CREC requires $\Gamma, f : (x : A_1) \to A_2 \vdash (x : A_1) \to A_2 \Leftarrow \mathsf{Type}$ (proving this from

$\Gamma \vdash (x\!:\!A_1) \to A_2 \Leftarrow \mathsf{Type}$ would need weakening).

5.4 Elaboration

We implement the declarative system using an elaborating typechecker, which translates a surface language expression (if it is well-formed according to the bidirectional rules) to an expression in the core language.

We formalize the algorithm that the elaborator uses as two inductively defined judgements, written $\Gamma' \mapsto a \Rightarrow a' : A'$ (Γ' and a are inputs) and $\Gamma' \mapsto a \Leftarrow A' \leadsto a'$ (Γ', a, and A' are inputs). The variables with primes (Γ', a' and A') are fully annotated expressions in the core language, while a is the surface language term being elaborated. The elaborator deals with each top-level definition in the program separately, and the context Γ' is an input containing the types of the previously elaborated definitions. The complete set of rules is shown in Figures 5.5 and 5.6.

The job of the elaborator is to insert enough annotations in the term to create a well-typed core expression. It should not otherwise change the term. Stated more formally,

***Theorem* 13** (Elaboration soundness)**.**

1. If $\Gamma \mapsto a \Rightarrow a' : A$, then $\Gamma \vdash a' : A$ and $|a| = |a'|$.
2. If $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma \mapsto a \Leftarrow A \leadsto a'$, then $\Gamma \vdash a' : A$ and $|a| = |a'|$.

Furthermore, the elaborator should accept those terms specified by the declarative system. If the type system of Section 5.3 accepts a program, then the elaborator succeeds (and produces an equivalent type in inference mode).

***Theorem* 14** (Elaboration completeness)**.**

1. If $\Gamma \vdash a \Rightarrow A$ and $\mapsto \Gamma \leadsto \Gamma'$ and $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \leadsto A'$, then $\Gamma' \mapsto a \Rightarrow a' : A''$ and $\Gamma' \vDash A' = A''$
2. If $\Gamma \vdash a \Leftarrow A$ and $\mapsto \Gamma \leadsto \Gamma'$ and $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \leadsto A'$, then $\Gamma' \mapsto a \Leftarrow A' \leadsto a'$.

Designing the elaboration rules follows the standard pattern of turning a declarative specification into an algorithm: remove all rules that are not syntax directed (in this case Icast and Ccast), and generalize the premises of the remaining rules to create a syntax-directed system that accepts the same terms. At the same time, the uses of congruence closure relation $\Gamma \vDash a = b$ must be replaced by appropriate calls to the congruence closure algorithm. We specify this algorithm using the following (partial) functions:

$$\boxed{\Gamma \vdash a \Rightarrow a' : A}$$

$$\frac{}{\Gamma \vdash \mathsf{Type} \Rightarrow \mathsf{Type} : \mathsf{Type}}\,\textsc{EIType} \qquad \frac{x : A \in \Gamma \quad \Gamma \vdash A \Leftarrow \mathsf{Type} \leadsto A_0}{\Gamma \vdash x \Rightarrow x : A}\,\textsc{EIVar}$$

$$\frac{\begin{array}{l}\Gamma \vdash A \Leftarrow \mathsf{Type} \leadsto A' \\ \Gamma, x : A' \vdash B \Leftarrow \mathsf{Type} \leadsto B'\end{array}}{\Gamma \vdash (x{:}A) \to\ B \Rightarrow (x{:}A') \to\ B' : \mathsf{Type}}\,\textsc{EIPi} \qquad \frac{\begin{array}{l}\Gamma \vdash a \Rightarrow a' : A_1 \\ \Gamma \vdash A_1 =^? (x : A) \to B \leadsto v_1 \\ \Gamma \vdash v \Leftarrow A \leadsto v' \\ \Gamma \vDash \mathsf{injrng}\,(x{:}A) \to\ B\,\mathsf{for}\,v'\end{array}}{\Gamma \vdash a\ v \Rightarrow {a'}_{\triangleright v_1}\ v' : \{v'/x\}\,B}\,\textsc{EIDApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow a' : A_1 \\ \Gamma \vdash A_1 =^? A \to B \leadsto v_1 \\ \Gamma \vdash b \Leftarrow A \leadsto b'\end{array}}{\Gamma \vdash a\ b \Rightarrow {a'}_{\triangleright v_1}\ b' : B}\,\textsc{EIApp} \qquad \frac{\begin{array}{l}\Gamma \vdash A \Leftarrow \mathsf{Type} \leadsto A' \\ \Gamma, x : A' \vdash B \Leftarrow \mathsf{Type} \leadsto B'\end{array}}{\Gamma \vdash \bullet(x{:}A) \to\ B \Rightarrow \bullet(x{:}A') \to\ B' : \mathsf{Type}}\,\textsc{EIIPi}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow a' : A_1 \\ \Gamma \vdash A_1 =^? [x : A] \to B \leadsto v_1 \\ \Gamma \vdash v \Leftarrow A \leadsto v' \\ \Gamma \vDash \mathsf{injrng}\ \bullet\,(x{:}A) \to\ B\,\mathsf{for}\,v'\end{array}}{\Gamma \vdash a\ \bullet_v \Rightarrow {a'}_{\triangleright v_1}\ \bullet_{v'} : \{v'/x\}\,B}\,\textsc{EIIDApp}$$

$$\boxed{\Gamma \vdash a \Leftarrow A \leadsto a'}$$

$$\frac{\begin{array}{l}\Gamma \vdash A =^? (x : A_1) \to A_2 \leadsto v_1 \\ \Gamma, f : (x{:}A_1) \to\ A_2, x : A_1 \vdash a \Leftarrow A_2 \leadsto a' \\ \Gamma, f : (x{:}A_1) \to\ A_2, x : A_1 \vDash \mathsf{injrng}\,(x{:}A_1) \to\ A_2\,\mathsf{for}\,x \\ \Gamma, f : (x{:}A_1) \to\ A_2 \vdash (x{:}A_1) \to\ A_2 \Leftarrow \mathsf{Type} \leadsto A_0\end{array}}{\Gamma \vdash \mathsf{rec}\,f\ x.a \Leftarrow A \leadsto (\mathsf{rec}\,f_{(x{:}A_1)\to\,A_2}\ x.a')_{\triangleright \mathsf{symm}\,v_1}}\,\textsc{ECRec}$$

$$\frac{\begin{array}{l}\Gamma \vdash A =^? [x : A_1] \to A_2 \leadsto v_1 \\ \Gamma, f : \bullet(x{:}A_1) \to\ A_2, x : A_1 \vdash a \Leftarrow A_2 \leadsto a' \\ \Gamma, f : \bullet(x{:}A_1) \to\ A_2, x : A_1 \vDash \mathsf{injrng}\ \bullet\,(x{:}A_1) \to\ A_2\,\mathsf{for}\,x \\ x\ \notin \mathsf{FV}\,(|a'|) \\ \Gamma, f : \bullet(x{:}A_1) \to\ A_2 \vdash \bullet(x{:}A_1) \to\ A_2 \Leftarrow \mathsf{Type} \leadsto A_0\end{array}}{\Gamma \vdash \mathsf{rec}\,f\ \bullet.a \Leftarrow A \leadsto (\mathsf{rec}\,f_{\bullet(x{:}A_1)\to\,A_2}\ \bullet_x.a')_{\triangleright \mathsf{symm}\,v_1}}\,\textsc{ECIRec}$$

Figure 5.5: Elaboration: functions and variables

$$\boxed{\Gamma \vdash a \Rightarrow a' : A}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \Leftarrow \mathsf{Type} \leadsto A' \\ \Gamma \vdash a \Leftarrow A' \leadsto a'\end{array}}{\Gamma \vdash a_A \Rightarrow a' : A'}\textsc{EIannot}$$

$$\frac{\begin{array}{c}|a| \leadsto^{i}_{\mathsf{cbv}} c \quad |b| \leadsto^{j}_{\mathsf{cbv}} c \\ \Gamma \vdash a = b \Leftarrow \mathsf{Type} \leadsto a' = b'\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} i\, j : a = b} \Rightarrow \mathsf{join}_{\leadsto_{\mathsf{cbv}} i\, j : a' = b'} : a' = b'}\textsc{EIjoinc}$$

$$\frac{\begin{array}{c}|a| \leadsto^{i}_{\mathsf{p}} c \quad |b| \leadsto^{j}_{\mathsf{p}} c \\ \Gamma \vdash a = b \Leftarrow \mathsf{Type} \leadsto a' = b'\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{p}} i\, j : a = b} \Rightarrow \mathsf{join}_{\leadsto_{\mathsf{p}} i\, j : a' = b'} : a' = b'}\textsc{EIjoinp}$$

$$\boxed{\Gamma \vdash a \Leftarrow A \leadsto a'}$$

$$\frac{\begin{array}{c}\Gamma \vdash a \Rightarrow a' : A \\ \Gamma \vdash A \overset{?}{=} B \leadsto v_1\end{array}}{\Gamma \vdash a \Leftarrow B \leadsto a'_{\triangleright v_1}}\textsc{ECinf}$$

$$\frac{\begin{array}{c}\Gamma \vdash A =^{?} (a = b) \leadsto v_1 \\ \Gamma \vdash a \overset{?}{=} b \leadsto v\end{array}}{\Gamma \vdash \mathsf{join} \Leftarrow A \leadsto v_{\triangleright \mathsf{symm}\, v_1}}\textsc{ECrefl}$$

Figure 5.6: Elaboration: equality

$$\boxed{\Gamma \vDash \ \mathsf{injrng}\, A\, \mathsf{for}\, v}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : A \quad \Gamma \vdash (x : A) \to\ B : \mathsf{Type} \\ \forall A' B'.(\Gamma \vDash ((x : A) \to\ B) = ((x : A') \to\ B')) \\ \qquad\qquad \text{implies } (\Gamma \vDash \{v/x\}\, B = \{v_{\triangleright v_0}/x\}\, B' \text{ where } \Gamma \vdash v_0 : A = A')\end{array}}{\Gamma \vDash \ \mathsf{injrng}\, (x : A) \to\ B\, \mathsf{for}\, v}\textsc{IRpi}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : A \quad \Gamma \vdash \bullet(x : A) \to\ B : \mathsf{Type} \\ \forall A' B'.(\Gamma \vDash (\bullet(x : A) \to\ B) = (\bullet(x : A') \to\ B')) \\ \qquad\qquad \text{implies } (\Gamma \vDash \{v/x\}\, B = \{v_{\triangleright v_0}/x\}\, B' \text{ where } \Gamma \vdash v_0 : A = A')\end{array}}{\Gamma \vDash \ \mathsf{injrng}\ \bullet (x : A) \to\ B\, \mathsf{for}\, v}\textsc{IRpi}$$

Figure 5.7: Core language injectivity restriction

$\Gamma \vDash A \overset{?}{=} B \rightsquigarrow v$, which checks A and B for equality and produces core-language proof v.

$\Gamma \vDash A =^? (x : B_1) \to B_2 \rightsquigarrow v$, which checks whether A is equal to some function type and produces that type and proof v.

$\Gamma \vDash A =^? (B_1 = B_2) \rightsquigarrow v$, which is similar to above, except for equality types.

For example, consider the rule for elaborating function applications, EIDAPP in Figure 5.5. In the corresponding declarative rule (IDAPP) the applied term a must have an arrow type, but this can be arranged by implicitly using ICAST to adjust a's type. Therefore, in the algorithmic system, the corresponding condition is that the type of a should be equal to an arrow type $(x : A) \to B$ modulo the congruence closure. Operationally, the typechecker will infer some type A_1 for a, then run the congruence closure algorithm to construct the set of all expressions that are equal to A_1, and check if the set contains some expression which is an arrow type. The elaborated core term uses the produced proof of $A_1 = (x : A) \to B$ in a cast to change the type of a.

At this point there is a potential problem: what if A_1 is equal to more than one arrow type? For example, if $A_1 = (x : A) \to B = (x : A') \to B$, then the elaborator has to choose whether to check b against A or A'. A priori it is quite possible that only one of them will work; for example the context Γ may contain an inconsistent equation like $\mathsf{Nat} \to \mathsf{Nat} = \mathsf{Bool} \to \mathsf{Nat}$. We do not wish to introduce a backtracking search here, because that could make type checking too slow.

This kind of mismatch in the domain type can be handled by extending the congruence closure algorithm. Note that things are fine if $\Gamma \vDash A = A'$, since then it does not matter if A or A' is chosen. So the issue only arises if $\Gamma \vDash (x : A) \to B = (x : A') \to B$ and not $\Gamma \vDash A = A'$. Fortunately, type constructors are injective in the core language (Section 3.6). Including injectivity as part of the congruence closure judgement (by the rule TCCINJDOM) ensures that it does not matter which arrow type is picked.

We also have to worry about a mismatch in the codomain type, i.e. the case when $\Gamma \vDash A_1 = (x : A) \to B$ and $\Gamma \vDash A_1 = (x : A') \to B'$ for two different types. At first glance it seems as if we could use the same solution. After all, the core language includes a rule for injectivity of the range of function types (rule TJINJRNG). There is an important difference between this rule and TJINJDOM, however, which is the handling of the bound variable x in the codomain B: the rule says that this can be closed by substituting any value for it. As a result, we cannot match this rule in the congruence closure relation, because the algorithm would have to guess that value. In other words, to match this rule in the congruence closure relation would mean to add a rule like

$$\frac{\Gamma \vDash (x : A) \to B = (x : A) \to B' \qquad \Gamma \vdash v : A}{\Gamma \vDash \{v/x\}\, B = \{v/x\}\, B'}$$

This proposed rule is an axiom schema, which can be instantiated for any value v.

Unfortunately, that makes the resulting equational theory undecidable. For example, the equational theory of SKI-combinators (which is undecidable) could be encoded as an assumption context containing one indexed datatype T and two equations:

```
data SK = S | K | App of SK SK

T : SK → Type
ax1 :   ((x y : SK) → T (App (App K x) y))
      = ((x y : SK) → T x)
ax2 :   ((f g x : SK) → T (App (App (App S f) g) x))
      = ((f g x : SK) → T (App (App f x) (App g x)))
```

Here, any value of type SK represents a combinator expression. In order to encode combinator equivalence problems as type checking problems we use a type variable T—it does not matter what the inhabitants of T is, just that it is indexed by SK. Finally, the two assumptions ax1 and ax2 represent the β-rules for K and S. For example, using the rule TJINJDOM twice, with the equation ax1 and the values v and u, proves T (App (App K u) v) = T u, and then TJINJDCON proves (App (App K u) v) = u. Together with the congruence and transitivity rules, we would have that two values of type SK are provably equal in this context iff they represent β-convertible combinator expressions.

As far as writing an elaborator goes, maybe it is ok that the injectivity axiom leads to undecidability when used with arbitrary values—after all, we only want to apply the axiom to the particular value v from the function application $a\ v$. However, there does not seem to be any natural way to write a declarative specification explaining what values v should be candidates.

Instead, we restrict the declarative language to forbid this problematic case. That is, the programmer is not allowed to write a function application unless all possible return types for the function are equal. Note that in cases when an application is forbidden by this check, the programmer can avoid the problem by proving the required equation manually and ensuring that it is available in the context.

In the fully-annotated core language we express this restriction with the rule IRPI (in Figure 5.7), and then lift this operation to partially annotated terms by rule EIRPI (Figure 5.3). Operationally, the typechecker will search for all arrow types equal to A_1 and check that the codomains with v substituted are equal in the congruence closure. This takes advantage of the fact that equivalence classes under congruence closure can be efficiently represented—although the rule as written appears to quantify over potentially infinitely many function types, the algorithm in Section 5.5 will represent these as a finite union-find structure which can be effectively enumerated. In the core language rule we need to insert a cast from A to A' to make the right-hand side well typed. By the rule TCCINJDOM that equality is always provable, so the typechecker will use the proof term v_0 that the congruence closure algorithm produced.

In the case of a simple arrow type $A \to B$, the range injectivity rule is unproblematic and we do include it in the congruence closure relation (TCCInjrng). So the application rule for simply-typed functions (EIApp) does not need the injectivity restriction. On the other hand, if the core language did not support injectivity for arrow domains, we could have used the same injectivity restriction for both the domain and codomain.

The rule for checking function definitions (ECRec in Figure 5.5) uses the same ideas that we saw in the application rule. First, while the declarative rule checks against a syntactic arrow type, the algorithmic system searches whether the type A is equivalent to some arrow type $(x : A_1) \to A_2$. Second, to avoid trouble if there is more than one such function type, we add an injrng restriction.

Thus the ECRec rule ensures that although there may be some choice about what type A_1 to give to the new variable x in the context, all the types that can be chosen are equal up to CC. We then design the type system so that all judgements respect CC-equivalence of typing contexts.

The rest of the elaborations rules hold few surprises. The rules for computationally irrelevant abstractions and applications (EIipi, EIidapp, and ECirec) are exactly analog to the rules for relevant functions.

On the checking side, the mode-change rule ECInf (in Figure 5.6) now needs to prove that the synthesized and checked types are equal. This rule corresponds to a direct call to the congruence closure algorithm, by the premise $\Gamma \mapsto A \overset{?}{=} B \leadsto v_1$ producing a proof term v_1. Note that the inputs are fully elaborated terms—in moving from the declarative to the algorithmic type system, we replaced the undecidable condition $\Gamma \vDash^{\exists} A = B$ with a decidable one.

Finally, the rule ECRefl (in Figure 5.6) elaborates checkable equality proofs (written join in the rule and as underscores in the concrete Zombie syntax). As in the rule for application, the typechecker does a search through the equivalence class of the ascribed type A to see if it contains any equations. If there is more than one equation it does not matter which one gets picked, because the congruence relation includes injectivity of the equality type constructor (TCCInjeq). In the elaborated term we need to prove $(a = b) = A$ given $A = (a = b)$. This can be done using Tjoinp (for reflexivity) and Tjsubst, and we abbreviate that proof term $\mathsf{symm}\, v_1$.

5.4.1 Properties of the congruence closure algorithm

It is attractive to base our type system around congruence closure because there exists efficient algorithms to decide it. But the correctness proof for the elaborator does not need to go into details about how that algorithm work. It only assumes that the congruence closure algorithm satisfies the following properties. (We show the statement of these properties for function types below, the others are similar.)

***Property* 15** (Soundness)**.** If $\Gamma \mapsto A \overset{?}{=} (x : B_1) \to B_2 \leadsto v$, then $\Gamma \vdash v : A = ((x : B_1) \to B_2)$ and $|v| = \mathsf{join}$ and $\Gamma \vDash A = (x\!:\!B_1) \to B_2$.

***Property* 16** (Completeness)**.** If $\Gamma \vDash A = (x\!:\!B_1) \to B_2$ then there exists a B_1', B_2' and v such that $\Gamma \mapsto A \overset{?}{=} (x\!:\!B_1') \to B_2' \leadsto v$.

***Property* 17** (Respects Congruence Closure)**.** If $\Gamma \vDash A = B$ and $\Gamma \mapsto B \overset{?}{=} (x\!:\!C_1) \to C_2 \leadsto v$ then $\Gamma \mapsto A \overset{?}{=} ((x\!:\!C_1') \to C_2') \leadsto v'$.

In other words, the algorithm should be sound and complete with respect to the $\Gamma \vDash A = B$ relation; it should generate correct core proof terms v; and the output should depend only on the equivalence class the input is in. In the next section we show how to implement an algorithm satisfying this interface.

5.5 Implementing congruence closure

Algorithms for congruence closure in the first-order setting are well studied, and our work builds on them. However, in our type system the relation $\Gamma \vDash a = b$ does more work than "classic" congruence closure: we must also handle erasure, terms with bound variables, (dependently) typed terms, the injectivity rules, the "assumption up to congruence" rule, and we must generate proof terms in the core language.

Our implementation proves an equation $a = b$ in three steps. First, we erase all annotations from the input terms and explicitly mark places where the congruence rule can be applied, using an operation called *labelling*. Then we use an adapted version of the congruence closure algorithm by Nieuwenhuis and Oliveras [97]. Our version of their algorithm has been extended to also handle injectivity and "assumption up to congruence", but it ignores all the checks that the terms involved are well-typed. Finally, we take the untyped proof of equality, and process it into a proof that a and b are also related by the typed relation. The implementation is factored in this way because the congruence rule does not necessarily preserve well-typedness, so the invariants of the algorithm are easier to maintain if we do not have to track well-typedness at the same time.

5.5.1 Labelling terms

In the $\Gamma \vDash a = b$ judgement, the rule TCCcongruence is stated in terms of substitution. But existing algorithms expect congruence to be applied only to syntactic

$\boxed{\Gamma \vdash^{\mathrm{L}} \ a = b}$

$$\frac{}{\Gamma \vdash^{\mathrm{L}} \ a = a}\textsc{LCCRefl} \qquad \frac{\Gamma \vdash^{\mathrm{L}} \ a = b}{\Gamma \vdash^{\mathrm{L}} \ b = a}\textsc{LCCSym} \qquad \frac{\Gamma \vdash^{\mathrm{L}} \ a = b \quad \Gamma \vdash^{\mathrm{L}} \ b = c}{\Gamma \vdash^{\mathrm{L}} \ a = c}\textsc{LCCTrans}$$

$$\frac{x : A \in \Gamma \quad \Gamma \vdash^{\mathrm{L}} \ A = ((- = -)\ a\ b)}{\Gamma \vdash^{\mathrm{L}} \ a = b}\textsc{LCCAssum}$$

$$\frac{\forall k.\ \ \Gamma \vdash^{\mathrm{L}} \ a_k = b_k}{\Gamma \vdash^{\mathrm{L}} \ F\,\overline{a_i} = F\,\overline{b_i}}\textsc{LCCCong} \qquad \frac{\Gamma \vdash^{\mathrm{L}} \ F\,a = F\,b \quad F\,\mathsf{injective}}{\Gamma \vdash^{\mathrm{L}} \ a = b}\textsc{LCCInj}$$

Figure 5.8: Untyped congruence closure on labelled terms

function applications: from $a = b$ conclude $f\ \ a = f\ \ b$. To bridge this gap, we preprocess equations into (erased) *labelled expressions*. A label F is an erased language expression with some designated holes (written $-$) in it, and a labelled expression is a label applied to zero or more labelled expressions, i.e. a term in the following grammar.

$$a ::= F\,\overline{a_i}$$

Typically a label will represent just a single node of the abstract syntax tree. For example, a wanted equation $f\ x = f\ y$ will be processed into $(-\,-)\ f\ x = (-\,-)\ f\ y$. The label $(-\,-)$ means this is a function application, which is a single node. However, for expressions involving bound variables, it can be necessary to be more coarse-grained. For example, given $a = b$ our implementation can prove $\mathsf{rec}\,f\ x.a + x = \mathsf{rec}\,f\ x.b + x$, which involves using $\mathsf{rec}\,f\ x.- + x$ as a label. In general, to process an expression a into a labelled term, the implementation will select the largest subexpressions that do not involve any bound variables.

The labelling step also deletes all annotations from the input expressions. This means that we automatically compute the congruence closure up to erasure (rule TCCErasure), at the cost of needing to do more work when we generate core language witnesses (Section 5.5.3).

Applying the labelling step simplifies the congruence closure problems in several ways. We show the simpler problem by defining the relation $\Gamma \vdash^{\mathrm{L}} \ a = b$ defined in Figure 5.8. Compared to Figure 5.2 we no longer need a rule for erasure, congruence is only used on syntactic label applications, all the different injectivity rules are handled generically, and we do not ensure that the terms are well-typed. In Section A.2.1 we formally define the label operation, and prove that it is complete in the following sense.

***Lemma* 18.** If $\Gamma \vDash a = b$ then $\mathsf{label}\,\Gamma \vdash^{\mathrm{L}} \ \mathsf{label}\,a = \mathsf{label}\,b$.

5.5.2 Untyped congruence closure

Next, we use an algorithm based on Nieuwenhuis and Oliveras [97] to decide the $\Gamma \vdash^{\mathrm{L}} a = b$ relation. The algorithm first "flattens" the problem by allocating *constants* c_i (i.e. fresh names) for every subexpression in the input. After this transformation every input equation has either the form $c_1 = c_2$ or $c = F(c_1, c_2)$, that is, it is either an equation between two atomic constants or between a constant and a label F applied to constants. Then follows the main loop of the algorithm, which is centered around three data-structures: a queue of input equations, a union-find structure and a lookup table. In each step of the loop, we take off an equation from the queue and update the state accordingly. When all the equations have been processed the union-find structure represents the congruence closure.

The union-find structure tracks which constants are known to be equal to each other. When the algorithm sees an input equation $c_1 = c_2$ it merges the corresponding union-find classes. This deals with the reflexivity, symmetry and transitivity rules. The lookup table is used to handle the congruence rule. It maps applications $F(c_1, c_2)$ to some canonical representative c. If the algorithm sees an input equation $c = F(c_1, c_2)$, then c is recorded as the representative. If the table already had an entry c', then we deduce a new equation $c = c'$ which is added to the queue.

In order to adapt this algorithm to our setting, we make three changes. First, we adapt the lookup tables to include the *richer labels* corresponding to the many syntactic categories of our core language. (Nieuwenhuis and Oliveras only use a single label meaning "application of a unary function.")

Second, we deal with *injectivity rules* in a way similar to the implementation of Coq's `congruence` tactic [39]. Certain labels are considered injective, and in each union-find class we identify the set of terms that start with an injective label. If we see an input equation $c = F(c_1, c_2)$ and F is injective we record this in the class of c. Whenever we merge two classes, we check for terms headed by the same F; e.g. if we merge a class containing $F(c_1, c_2)$ with a class containing $F(c_1', c_2')$, we deduce new equations $c_1 = c_1'$ and $c_2 = c_2'$ and add those to the queue.

Third, our implementation of the *extended assumption rule* works much like injectivity. With each union-find class we record two new pieces of information: whether any of the constants in the class (which represent types of our language) are known to be inhabited by a variable, and whether any of the constants in the class represents an equality type. Whenever we merge two classes we check for new equations to be added to the queue.

In Section A.3 we give a precise description of our algorithm, and prove its correctness, i.e. that it always terminates, and returns "yes" iff the wanted equation is in the $\Gamma \vdash^{\mathrm{L}} a = b$ relation.

First we prove that flattening a context does not change which expressions are equal in that context. Although the flattening algorithm itself is the same as in previous work, the statement of the correctness proof is refined to say that the new assumptions h are always used as plain assumptions $h_{\triangleright \mathsf{refl}}$, as opposed to the general assumption-up-to-CC rule $h_{\triangleright p}$. The distinction is important, because although the flattening algorithm will process every assumption that was in the original context, it does not go on to recursively flatten the new assumptions that it added. So for completeness of the whole algorithm we need to know that there is never a need to reason about equality between such assumptions.

Then the correctness proof of the main algorithm is done in two parts. The *soundness* of the algorithm (i.e. if the algorithm says "yes" then the two terms really are provably equal) is fairly straightforward. We verify the invariant that every equation which is added to the input queue, union-find structure, and lookup table really is provably true. For each step of the algorithm which extends these data structures we check that the new equation is provable from the already known ones. In fact, this proof closely mirrors the way the implementation in Zombie works: there the data structures contain not only bare equations but also the evidence terms that justify them (see section 5.5.3), and each step of the algorithm builds new evidence terms from existing ones.

The *completeness* direction (if $\Gamma \vdash^{\mathsf{L}} \ a = b$ then the algorithm will return "yes") is more involved. We need to prove that at the end of a run of the algorithm, the union-find structure satisfies all the proof rules of the congruence relation. For our injectivity rule and extended assumption rule this means properties like

- For all $\overline{a_i}$ and $\overline{b_i}$, if $F\,\overline{a_i} \approx_R F\,\overline{b_i}$ and F is injective, then $\forall k. a_k \approx_R b_k$.
- If $x : A \in \Gamma$ then for all a, b, if $A \approx_R (a = b)$ then $a \approx_R b$.

where $\approx_R$ denotes the equivalence relation generated by the union-find links. The proof uses a generalized invariant: while the algorithm is still running R satisfies the proof rules modulo the pending equations E in the input queue, e.g. the invariant for the assumption rule is

> If $x : A \in \Gamma$ then for all a, b, if $A \approx_R (a = b)$ then $a \approx_{E,R} b$.

However, the congruence rule presents some extra difficulties. The full congruence relation for a given context Γ is in general infinite (if $\Gamma \vdash^{\mathsf{L}} \ a = b$, then by the congruence rule we will also have $\Gamma \vdash^{\mathsf{L}} \ \mathsf{Suc}\ a = \mathsf{Suc}\ b$ and $\Gamma \vdash^{\mathsf{L}} \ \mathsf{Suc}\ (\mathsf{Suc}\ a) = \mathsf{Suc}\ (\mathsf{Suc}\ b)$ and $\ldots$). So at the end of the run of an algorithm the data structures will not contain information about all possible congruence instances, but only those instances that involve terms from the input problem.

Following Corbineau [38] we attack this problem in two steps. First we show that at the end of the run of the algorithm the union-find structure R *locally* satisfies the congruence rule in the following sense:

> If $a_i \approx_R b_i$ for all $0 \leq i < n$, and $F\,\overline{a_i}$ and $F\,\overline{b_i}$ both appeared in the list of input equations, then $F\,\overline{a_i} \approx_R F\,\overline{b_i}$.

We then need to prove that this local completeness implies completeness. This amounts to showing that if a given statement $\Gamma \vdash^{\mathrm{L}} \ a = b$ is provable at all, it is provable by using the congruence rule only to prove equations between subexpressions of Γ, a, and b. There are a few approaches to this in the literature. The algorithm by Nieuwenhuis and Oliveras [97] can be shown correct because it is an instance of Abstract Congruence Closure (ACC) [11], while the correctness proofs for ACC algorithms in general rely on results from rewriting theory. However, it is not immediately obvious how to generalize this approach to handle additional rules like injectivity. Corbineau [38] instead gives a semantic argument about finite and general models.

As it happens, in our development there is a separate reason for us to prove that local uses of the congruence rule suffice: we need this result to bridge the gap between untyped and typed congruence. This is the subject of Section 5.5.3, and we use lemma 21 from that section to finish the completeness argument. All in all, this yields:

***Lemma* 19.** The algorithm described above is a decision procedure for the relation $\Gamma \vdash^{\mathrm{L}} \ a = b$.

5.5.3 Typing restrictions and generating core language proofs

Along the pointers in the union-find structure, we also keep track of the evidence that showed that two expressions are equal. The syntax of the evidence terms is given by the following grammar. An evidence term p is either an assumption x (with a proof p that x's type is an equation), reflexivity, symmetry, transitivity, injectivity, or an application of congruence annotated with a label A.

$$p, q \quad ::= \quad x_{\triangleright p} \mid \mathsf{refl} \mid p^{-1} \mid p ; q \mid \mathsf{inj}_i \ p \mid \mathsf{cong}_{\,A}\, p_1 \mathinner{..} p_i$$

Next we need to turn the evidence terms p into proof terms in the core calculus. This is nontrivial, because the Nieuwenhuis-Oliveras algorithm does not track types. Not every equation which is derivable by untyped congruence closure is derivable in the typed theory; for example, if $f : \mathsf{Bool} \to \mathsf{Bool}$, then from the equation $(a : \mathsf{Nat}) = (b : \mathsf{Nat})$ we can not conclude $f \ a = f \ b$, because $f \ a$ is not a well-typed term. Worse still, even if the conclusion is well-typed, not every untyped *proof* is valid in the typed theory, because it may involve ill-typed intermediate terms. For example, let $\mathsf{Id} : (A : \mathsf{Type}) \to A \to A$ be the polymorphic identity function, and suppose we have some terms $a : A$, $b : B$, and know the equations $x : A = B$ and $y : a = b$. Then

$$(\mathsf{cong}_{\mathsf{Id}}\ x\ \mathsf{refl}); (\mathsf{cong}_{\mathsf{Id}}\ \mathsf{refl}\ y)$$

is a valid untyped proof of $\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ b$. But it is not a correct typed proof because it involves the ill-typed term $\mathsf{Id}\ B\ a$:

$$\dfrac{\dfrac{x : A = B \qquad a = a}{\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ a}\,\mathit{cong} \qquad \dfrac{B = B \qquad y : a = b}{\mathsf{Id}\ B\ a = \mathsf{Id}\ B\ b}\,\mathit{cong}}{\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ b}\,\mathit{trans}$$

Corbineau [39] notes this as an open problem. However, the above proof is unnecessarily complicated. We note that the same equation can be proved by a single use of congruence.

$$\dfrac{x : A = B \qquad y : a = b}{\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ b}\,\mathit{cong}$$

Furthermore, the simpler proof does not have any issues with typing: every expression occurring in the derivation is either a subexpression of the goal or a subexpression of one of the equations from the context, so we know they are well-typed.

Our key observation is that this trick works in general. The only time a congruence proof will involve expressions which were not already present in the context or goal is when transitivity is applied to two derivations ending in cong. We simplify such situations using the following CongTrans rule.

$$(\mathsf{cong}_A\ p_1 .. p_i); (\mathsf{cong}_A\ q_1 .. q_i) \mapsto \mathsf{cong}_A\ (p_1; q_1) .. (p_i; q_i)$$

This rule is valid in general, and it does not make the proof larger. We also need rules for simplifying evidence terms that combine transitivity with injectivity or assumption-up-to-CC, such as $\mathsf{inj}_i\ (\mathsf{cong}_A\ p_1 .. p_k)$ and $x_{\triangleright(r;\mathsf{cong}_=\ p\ q)}$, rules for pushing uses of symmetry $({}^{-1})$ past the other evidence constructors, and rules for rewriting subterms. The complete simplification relation $\mapsto$ is shown in Figure 5.9.

Applying the simplification rules enough times will produce an evidence term which is suitable for generating typed core proofs. Describing exactly what a *fully simplified evidence term* looks like is a little involved, but we can define a grammar as follows. We mutually define grammars for *synthesizable* terms pS, *checkable* terms pC, and *chained* terms p^* (containing zero or more ps—an empty chain denotes the term refl, and a nonempty chain denotes a sequence of right-associated uses of transitivity $p_1; (p_2; (\ldots; p_n))$). The metavariable p^*_{LR} ranges over chains that begin and end with a synthesizable term (as opposed to an empty chain or a chain with a pC at the beginning or end), and p^*_{R} over chains that end with a pS (but may have a pC at the

$\boxed{p \mapsto q}$

$$
\begin{array}{lcl}
p & \mapsto & p \\
\mathsf{refl}^{-1} & \mapsto & \mathsf{refl} \\
\mathsf{refl}; p & \mapsto & p \\
p; \mathsf{refl} & \mapsto & p \\
(p; q); r & \mapsto & p; (q; r) \\
p; p^{-1} & \mapsto & \mathsf{refl} \\
p^{-1}; p & \mapsto & \mathsf{refl} \\
p; (p^{-1}; r) & \mapsto & r \\
p^{-1}; (p; r) & \mapsto & r \\
 & & \\
p^{-1^{-1}} & \mapsto & p \\
(p; q)^{-1} & \mapsto & q^{-1}; p^{-1} \\
(\mathsf{cong}_{\,A}\, p_1 \mathinner{..} p_i)^{-1} & \mapsto & \mathsf{cong}_{A}\ {p_1}^{-1} \mathinner{..}\ {p_i}^{-1} \\
 & & \\
(\mathsf{inj}_i\ p)^{-1} & \mapsto & \mathsf{inj}_i\ (p^{-1}) \\
 & & \\
(\mathsf{cong}_{\,A}\, p_1 \ldots p_i); (\mathsf{cong}_{\,A}\, q_1 \ldots q_i) & \mapsto & \mathsf{cong}_{\,A}\, (p_1; q_1) \mathinner{..} (p_i; q_i) \\
 & & \\
\mathsf{inj}_k\ (\mathsf{cong}_{\,A}\, p_1 \ldots p_i) & \mapsto & p_k \\
\mathsf{inj}_k\ ((\mathsf{cong}_{\,A}\, p_1 \mathinner{..} p_i); r) & \mapsto & p_k; (\mathsf{inj}_k\ r) \\
\mathsf{inj}_k\ (r; (\mathsf{cong}_{\,A}\, p_1 \ldots p_i)) & \mapsto & (\mathsf{inj}_k\ r); p_k \\
x_{\triangleright(r;\mathsf{cong}_{\,=}\ p\ q)} & \mapsto & p^{-1}; (x_{\triangleright r}); q
\end{array}
$$

$$
\frac{p \mapsto p'}{x_{\triangleright p} \mapsto x_{\triangleright p'}}\textsc{assumption}
\qquad
\frac{\begin{array}{c} p \mapsto p' \\ q \mapsto q' \end{array}}{p; q \mapsto p'; q'}\textsc{trans}
$$

$$
\frac{\forall k.\ \ p_k \mapsto p_k'}{\mathsf{cong}_{\,A}\, p_1 \mathinner{..} p_i \mapsto \mathsf{cong}_{\,A}\, p_1' \mathinner{..} p_i'}\textsc{cong}
\qquad
\frac{p \mapsto p'}{\mathsf{inj}_k\ p \mapsto \mathsf{inj}_k\ p'}\textsc{inj}
$$

Figure 5.9: Simplification rules for evidence terms

beginning). Finally, x^o is an abbreviation for $x^o_{\triangleright \mathsf{refl}}$.

$$\begin{array}{lcl}
o & ::= & 1 \mid -1 \\
pS & ::= & x^o \mid x^o_{\triangleright p^*_{\mathsf{R}}} \mid \mathsf{inj}\ i\ pS \mid p^*_{\mathsf{LR}} \\
pC & ::= & \mathsf{cong}_{\ A}\, p^*_1 \mathinner{..} p^*_i \\
p^* & ::= & (pS \mid pC)^* \\
p^*_{\mathsf{R}} & ::= & (p^*; pS) \\
p^*_{\mathsf{LR}} & ::= & pS \mid (pS; p^*; pS)
\end{array}$$

There is one additional condition which is not shown in the grammar: there must never be two check-terms adjacent to each other in a chain.

Now, any evidence term p can be simplified into a normalized evidence term p^*. And given p^* it is easy to produce a corresponding proof term in the core language. The idea is that one can reconstruct the middle expression in every use of transitivity $(p; q)$, because at least one of p and q will be a synthesizable term, so it is specific enough to pin down exactly what equation it is proving. Formally, we define the judgement $\Gamma \vdash^{\mathrm{L}} p : a = b$ by adding evidence terms to the rules in Figure 5.8, and then prove:

***Lemma* 20.** If $\Gamma \vdash^{\mathrm{L}} p : a = b$ and $p \mapsto q$, then $\Gamma \vdash^{\mathrm{L}} q : a = b$.

***Lemma* 21.** If $\Gamma \vdash^{\mathrm{L}} p : a = b$, then there exists some p^* such that $p \mapsto^* p^*$.

***Lemma* 22.** If we have $\mathsf{label}\,\Gamma \vdash^{\mathrm{L}} p^* : \mathsf{label}\,a = \mathsf{label}\,b$ and $\Gamma \vdash a = b : \mathsf{Type}$, then $\Gamma \vDash a = b$.

Simplifying the evidence terms also solves another issue, which arises because of the TCCErasure rule. Because the input terms are preprocessed to delete annotations (Section 5.5.1), an arbitrary evidence term will not uniquely specify the annotations. For example, change the previous example by making the type parameter an *erased* argument of Id, and suppose we have assumptions $x : a = a'$ and $y : a' = b$. Then the evidence term $(\mathsf{cong}_{\,\mathsf{Id}\,\bullet\,\_}\ x); (\mathsf{cong}_{\,\mathsf{Id}\,\bullet\,\_}\ y)$ could serve as the skeleton of either the valid proof

$$\dfrac{\dfrac{x : a = a'}{\mathsf{Id}\ \ \bullet_A\ \ a = \mathsf{Id}\ \ \bullet_A\ \ a'}\,\mathit{cong} \qquad \dfrac{y : a' = b}{\mathsf{Id}\ \ \bullet_A\ \ a' = \mathsf{Id}\ \ \bullet_B\ \ b}\,\mathit{cong}}{\mathsf{Id}\ \ \bullet_A\ \ a = \mathsf{Id}\ \ \bullet_B\ \ B}\,\mathit{trans}$$

or the invalid proof

$$\dfrac{\dfrac{x : a = a'}{\mathsf{Id}\ \ \bullet_A\ \ a = \mathsf{Id}\ \ \bullet_B\ \ a'}\,\mathit{cong} \qquad \dfrac{y : a' = b}{\mathsf{Id}\ \ \bullet_B\ \ a' = \mathsf{Id}\ \ \bullet_B\ \ b}\,\mathit{cong}}{\mathsf{Id}\ \ \bullet_A\ \ a = \mathsf{Id}\ \ \bullet_B\ \ B}\,\mathit{trans}$$

Again, this issue is only due to the cong-trans pair. Simplifying the evidence term resolves the issue, because in a simplified term every intermediate expression is pinned down.

Putting together the labelling step, the evidence simplification step and the proof term generation step we can relate typed and untyped congruence closure. In the following theorem, the relation $\Gamma \vdash a = b$ is defined by similar rules as Figure 5.2 except that we omit the typing premises in TCCREFL, TTCERASURE and TTCCONGRUENCE.

***Theorem* 23.** Suppose $\Gamma \vdash a = b$ and $\Gamma \vdash a = b : \mathsf{Type}$. Then $\Gamma \vDash a = b$. Furthermore $\Gamma \vdash v : a = b$ for some v.

The computational content of the proof is how the elaborator generates core language proof terms for equalities, so this shows the correctness of the Zombie implementation. But it is also interesting as a theoretical result in its own right, and an important part of the proof of completeness of elaboration (Section 5.4).

5.6 Extensions

The full Zombie implementation includes more features than the surface language described in Section 5.3. We omitted them from the formal system in order to simplify the proofs, but they are important to make programming up to congruence work well.

5.6.1 Full application rule

The dependent application rule of the core language does not restrict its argument to be a value. Instead it includes a premise that requires that the substituted type is well-formed (Section 3.4):

$$\frac{\begin{array}{l}\Gamma \vdash a : (x\!:\!A) \to B\\ \Gamma \vdash b : A\\ \Gamma \vdash \{b/x\}\,B : \mathsf{Type}\end{array}}{\Gamma \vdash a\ b : \{b/x\}\,B}\ \textsc{Tapp}$$

This is also the rule implemented by the full Zombie surface language. In the elaborator, the premise $\Gamma \vdash \{b/x\}\,B : \mathsf{Type}$ requires attention when checking the injrng restriction. In the implementation we just change the test to say b instead of v:

$$\frac{\begin{array}{l}\Gamma \vdash b : A \quad \Gamma \vdash (x\!:\!A) \to B : \mathsf{Type}\\ \forall A'B'.(\Gamma \vDash ((x\!:\!A) \to B) = ((x\!:\!A') \to B'))\\ \qquad\qquad \text{implies } (\Gamma \vDash \{b/x\}\,B = \{b_{\triangleright v_0}/x\}\,B' \text{ where } \Gamma \vdash v_0 : A = A')\end{array}}{\Gamma \vDash \mathsf{injrng}\ (x\!:\!A) \to B\ \mathsf{for}\ b}$$

However, with a general expression instead of a value, this test is perhaps unnecessarily restrictive. Among the arrow types that are equal to the type of the applied functions, there may be some where the resulting type $\{b/x\}\,B$ is well-formed and others where it is not. Because the congruence closure relation only equates well-typed expressions, the current definition of injrng says that the application is only allowed if *all* possible function types would lead to a well-formed result. Perhaps one could instead search for *some* type which works—usually B will be a small expression, so the check for well-formedness can be done quickly. On the other hand, the question is somewhat academic, because in our experience the injrng condition never seems to fail in practical programs.

So implementing the general application rule is not hard. However, if the typing rules are generalized in this way, there is a gap in the proof of the completeness theorem (theorem 14). In order to prove that the elaborator can match any declarative typing derivation, we need to prove that for any expression $\Gamma \vdash b' : A'$ such that $|b'| = |b|$, we have $\Gamma \vDash \{b/x\}\,B = \{b'/x\}\,B'$. Provided that $\{b'/x\}\,B'$ is a well-formed type this is true, because $|\{b'/x\}\,B'| = |\{b_{\triangleright v_0}/x\}\,B'|$. If the expression b' is a value, we know that the type is well-formed by substitution. Even in the general case it seems it will always be well-formed—the only thing that can cause the substitution to fail is if it violates a value restriction, and we know that it does not because $\{b/x\}\,B$ is well-formed and erases to the same thing. But we have not formally proven this.

5.6.2 Datatypes

Although we do not include datatypes in the type system in this chapter, they are a part of the Zombie implementation, and an important component of any dependently-typed language. The rules for elaborating expressions manipulating datatypes follow the same pattern as the rules for functions and equalities; in particular the rules corresponding to the TDCON and TCASE need to search the congruence closure to see that the ascribed type and the type of the scrutinee are equal to some datatype. We also extend the congruence closure relation to include injectivity of datatype constructors.

The presence of congruence closure elaboration has a very positive effect on language design, because it means that the core language can use the "smart case" typing rule (Section 3.7). The downside to smart case has been that because equality information is recorded as an assumption in the context, it is more work for the programmer. However, with congruence closure, the type system is immediately able to take advantage of these equalities in each branch. Thus, the Zombie surface language has the convenience of a traditional unification-based rule, while the core language enjoys the simplicity of smart case.

Adding the elaboration rules related to datatypes to the implementation was straightforward. On the other hand, we omit them from the formalized type system in this chapter, because they would add a lot of extra work in the completeness theorem. Because the typing rules for datatypes are stated in terms of telescopes, they add several new judgement forms and lemmas (telescope well-formedness, weakening and substitution of telescopes, etc), and we would need to prove that each of these respect CC-equivalence of contexts.

5.6.3 Reduction modulo congruence

In the type system in this chapter, all β-reductions are introduced by expressions `join : a = b`. But in practice some additional support from the typechecker can make programming much more pleasant.

First, one often wants to evaluate some expression a “as far as it goes”. Then making the programmer write both sides of the equation $a = b$ is unnecessarily verbose. Instead we provide the syntax `unfold a in body`. The implementation reduces a to normal form, $a \rightsquigarrow a' \rightsquigarrow a'' \rightsquigarrow a'''$ (if a does not terminate the programmer can specify a maximum number of steps), and then introduces the corresponding equations into the context with fresh names. That is, it elaborates as

```
let _ = (join : a = a') in
let _ = (join : a' = a'') in
let _ = (join : a'' = a''') in
  body
```

Second, many proofs requires an interleaving of evaluation and equations from the context, particularly in order to take advantage of equations introduced by smart case. One example is `npluszero` in Section 5.1. The case-expression needs to return a proof of `n+0 = n`. If we try to directly evaluate `n+0`, we would reach the stuck expression `case n of Zero → 0; Succ m' → Succ (m' + 0)`, so instead we used an explicit type annotation in the `Zero` branch to evaluate `0+0`. However, the context contains the equation `n = Zero`, which suggests that there should be another way to make progress.

To take advantage of such equations, we add some extra intelligence to the way unfold handles CBV-evaluation contexts, that is expressions of the form $f\ a$ or $(\mathsf{case}\ b\ \mathsf{of} \ldots)$. When encountering such an expression it will first recursively unfold the function f, the argument a, or the scrutinee b (as with ordinary CBV-evaluation), and add the resulting equations to the context. However, it will then examine the congruence equivalence class of these expressions to see if they contain any suitable values—any value v is suitable for a, a function value $\mathsf{rec}\ f\ x.a_0$ for f, and a value headed by a data constructor for b—and then unfold the resulting expression $(\mathsf{rec}\ f\ x.a_0)\ v$. (If there are several suitable values, one is selected arbitrarily). This way unfolding can make progress where ordinary CBV-evaluation gets stuck.

In the Zombie implementation, there is one more way that unfold helps with stuck expressions. In Section 4.1.4 we noted the that applications $(\lambda x.a)\ b$ are stuck unless b is a value. In the full Zombie language it is possible to work around this, provided that b is known to be terminating. One rewrites `(join : (λx.a) b = ...)` to `let [y] = b in (join : (λx.a) y = ...)`. The expression form `let [y] = ...` is the computationally irrelevant version of `let`, so this transformation does not cause b to get evaluated at runtime. Unfold automatically introduces this type of erased variable bindings giving names to non-value expressions, in order to let β-redexes make progress.

Using the same machinery we also provide a "smarter" version of `join`, which first unfolds both sides of the equation, and then checks that the resulting expressions are CC-equivalent. This lets us omit the type annotations from `npluszero`:

```
npluszero n = case n [eq] of
                  Zero → smartjoin
                  Suc m → ...
```

The unfold algorithm does not fully respect CC-equivalence, because it only converts *into* values. For example, suppose the context contains the equation $f\ \ a = v$, Then unfold $g\ (f\ \ a)$ will evaluate $f\ \ a$ and add the corresponding equations to the context, but unfold $g\ v$ will not cause $f\ \ a$ to be evaluated. This gives the programmer more control over what expressions are run.

One drawback of the current unfold algorithm is that because each `join` expression only represents a single step, unfold can add a large number of equations to the context. This results in large core terms and slow type checking. In future work, it would be useful to optimize the algorithm by representing long chains of reductions as a single `join` expression with a step-count.

We have not studied the theory of the unfold algorithm, and indeed it is not a complete decision procedure for our propositional equality. If a subexpression of a does not terminate, unfold will spend all its reduction budget on just that subexpression (but this is OK, because the programmer decides what expression a to unfold). And if the context contains e.g. an equation between two unrelated function values, unfold will arbitrarily choose one of them (but it is hard to think of an example where this would happen). We have found unfold very helpful when writing examples.

Operational semantics of annotated terms While the idea of `unfold` is simple, implementing it requires some additional theory. For the plain `join` rule, the type-checker erases all annotations and then reduces the terms according to the step rules in Figure 3.4. We could try the same method for `unfold`, but then the final normal form a' would be unannotated, whereas the elaborator needs to produce a well-typed

$$\frac{\Gamma \vdash a : ((x{:}A) \to\ B) = ((x'{:}A') \to\ B')}{\Gamma \vdash \mathsf{domeq}\ a : A' = A}\ \textsc{domeq}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : ((x{:}A) \to\ B) = ((x{:}A') \to\ B') \\ \Gamma \vdash v : A \quad \Gamma \vdash v' : A' \\ |v| = |v'|\end{array}}{\Gamma \vdash \mathsf{raneq}\ a\ v\ v' : \{v/x\}\, B = \{v'/x\}\, B'}\ \textsc{raneq}$$

$$\frac{\Gamma \vdash a : D\ \overline{A_i} = D\ \overline{A_i}'}{\Gamma \vdash \mathsf{ntheq}_i\ a : A_i = A_i'}\ \textsc{injtcon} \qquad \frac{\Gamma \vdash a : A = B \quad \Gamma \vdash b : B = C}{\Gamma \vdash \mathsf{trans}\ a\ b : A = C}\ \textsc{transeq}$$

Figure 5.10: Derived forms for reasoning about equations

annotated core term. In order to implement `unfold`, we need to define a reduction relation $\rightsquigarrow_{\mathsf{a}}$ for annotated expressions.

The annotated reduction relation is the computational content of the type preservation proof. We want to ensure that if $\Gamma \vdash a : A$ and $|a| \rightsquigarrow_{\mathsf{cbv}} b'$, then there exists b such that $a \rightsquigarrow_{\mathsf{a}} b$ and $|b| = b'$ and $\Gamma \vdash b : A$.

In order to state the reduction rules, it is convenient to define abbreviations for some common operations on equations, as shown in figure 5.10. These rules are derivable in the core calculus.[9] The statement of the derived rules is less natural than the original typing rules (DOMEQ bakes in symmetry, RANEQ takes two values v and v') in order to make the reduction relation (figure 5.11) simpler.

The main challenge is to handle type casts. Since type casts are completely erased they do not impede reduction, so an expression like $(\lambda x.a)_{\triangleright b}\ v$ can step. At this point, the evaluator uses the injectivity rules for arrow types on the equality proof b.[10] For example in the rule AS_APP_REC, we know that $\mathsf{rec}\ f_A\ y.a$ has some arrow type $(y : A_1) \to\ B_1$ since it is a rec-expression, and that $(\mathsf{rec}\ f_A\ y.a)_{\triangleright b_0}$ has some arrow type $(y : A_2) \to\ B_2$ since it is applied to an argument v. So b_0 is a proof of $((y : A_1) \to\ B_1) = ((y : A_2) \to\ B_2)$. Then $\mathsf{domeq}\ b_0 : A_2 = A_1$ and $\mathsf{raneq}\ b_0\ v\ v' : \{v/x\}\, B_1 = \{v'/x\}\, B_2$, so the right-hand side of the step rule is well-typed.

The reduction rules are simple, but it took a few tries to get them to that state. In a previous version of the core language [120] the rules TCAST and TJSUBST were combined into a single rule which can eliminate several equations at once. That makes for shorter programs—but also a terribly complicated AS_CONV_CONV rule, which has

[9] For efficiency, the Zombie implementation actually includes these as primitive constructs in the core language. Particularly having trans available as a primitive is very helpful, because if we prove it as a lemma in terms of TJSUBST and TCAST, the lemma will need to take A, B, and C as arguments. Being able to omit those expressions generates much smaller congruence proof terms.

[10] This is similar to other languages with erased equality casts, for example Haskell Core.

$\boxed{a \leadsto_{\mathsf{a}} a'}$

$$\frac{a \leadsto_{\mathsf{a}} a'}{\mathcal{E}[a] \leadsto_{\mathsf{a}} \mathcal{E}[a']}\text{AS\_CTX}$$

$$\frac{}{(\mathsf{rec}\ f_A\ x.a)\ v \leadsto_{\mathsf{a}} \{\mathsf{rec}\ f_A\ x.a/f\}\ \{v/x\}\ a}\text{AS\_APP\_REC}$$

$$\frac{}{(\mathsf{rec}\ f_A\ \bullet_x.a)\ \bullet_v \leadsto_{\mathsf{a}} \{\mathsf{rec}\ f_A\ \bullet_x.a/f\}\ \{v/x\}\ a}\text{AS\_APP\_IREC}$$

$$\frac{v' = v_{\triangleright \mathsf{domeq}\ b_0}}{((\mathsf{rec}\ f_A\ y.a)_{\triangleright b_0})\ v \leadsto_{\mathsf{a}} (\{v'/y\}\ \{(\mathsf{rec}\ f_A\ y.a)/f\}\ a)_{\triangleright \mathsf{raneq}\ b_0\ v'\ v}}\text{AS\_APP\_CONV\_REC}$$

$$\frac{v' = v_{\triangleright \mathsf{domeq}\ b_0}}{((\mathsf{rec}\ f_A\ \bullet_y.a)_{\triangleright b_0})\ \bullet_v \leadsto_{\mathsf{a}} (\{b'/y\}\ \{(\mathsf{rec}\ f_A\ \bullet_y.a)/f\}\ a)_{\triangleright \mathsf{raneq}\ b_0\ v'\ v}}\text{AS\_APP\_CONV\_IREC}$$

$$\frac{a \leadsto_{\mathsf{a}} a'}{a_{\triangleright b} \leadsto_{\mathsf{a}} a'_{\triangleright b}}\text{AS\_CONV} \qquad \frac{}{(v_{\triangleright b})_{\triangleright c} \leadsto_{\mathsf{a}} v_{\triangleright(\mathsf{trans}\ b\ c)}}\text{AS\_CONV\_CONV}$$

$$\frac{}{\begin{array}{l}\mathsf{case}\ (d_{i\,D\,\overline{A_i}}\ \overline{v_i})_y\ \mathsf{of}\ \{\overline{d_j\ \Delta_j \Rightarrow a_j}^{j\in 1\ldots k}\} \\ \qquad \leadsto_{\mathsf{a}} \{\mathsf{join}_{\leadsto_{\mathsf{cbv}}(d_{i\,D\,\overline{A_i}}\ \overline{v_i})=(d_{i\,D\,\overline{A_i}}\ \overline{v_i})}/y\}\ \{\overline{v_i}/\Delta_i\}\ a_i\end{array}}\text{AS\_CASE\_DISPATCH}$$

$$\frac{\begin{array}{l}\mathsf{data}\ D\,\Delta\ \mathsf{where}\,\{\ \overline{d_j\ \mathsf{of}\ \Delta'_j}^{j\in 1\ldots k}\ \} \in \Gamma \\ \Gamma \vdash b_0 : D\ \overline{A_i} = D\ \overline{B_i} \\ \forall i.\quad v'_i = v_{i \triangleright\, \mathsf{ntheq}_i\ b_0}\end{array}}{\begin{array}{l}\mathsf{case}\ ((d_{i\,D\,\overline{A_i}}\ \overline{v_i})_{\triangleright b_0})_y\ \mathsf{of}\ \{\overline{d_j\ \Delta_j \Rightarrow a_j}^{j\in 1\ldots k}\} \\ \qquad \leadsto_{\mathsf{a}} \mathsf{case}\ (d_{i\,D\,\overline{B_i}}\ \overline{v_i}')_y\ \mathsf{of}\ \{\overline{d_j\ \Delta_j \Rightarrow a_j}^{j\in 1\ldots k}\}\end{array}}\text{AS\_CASE\_CONV}$$

Figure 5.11: Reduction of annotated terms

to do anti-unification to combine all the different equations involved.

Another issue, which doesn't come up in this simplified core calculus but becomes relevant in the full Zombie system, is that the full system contains facilities for optional termination checking (Chapter 7) and a predicative hierarchy of sorts Type_ℓ. Therefore we need subsumption rules saying that expressions of type Type_i can be used instead of Type_{i+1}, and that terminating expressions can be used in places expecting nonterminating expressions (TSub in Section 7.3.2). If it is just a question of creating an annotation regime in which the right erased terms can be given types, the simplest solution is to create separate (erasable) term constructors for subsumption rules like these. However, when defining $\leadsto_{\mathsf{a}}$ such constructors will need special stepping rules, so one is pushed towards a design which leaves them implicit and does more inference when checking core terms.

Implementing reduction of annotated expressions is somewhat subtle, particularly when using equations from the congruence closure to create values. But happily, it is done entirely by the elaborator. The core language only talks about reducing erased expressions, so it remains simple and trustworthy.

Chapter 6

Towards unification-based type inference

Even when the surface language includes congruence closure and reduction modulo, as described in the previous chapter, some programs are still unreasonably verbose. Mainstream dependently typed languages can infer term and type arguments using unification-based algorithms, which makes programs much less cluttered.

Such inference is of course helpful even without dependent types. In ML-like languages it lets the programmer omit type arguments to polymorphic functions, e.g. for functions dealing with lists the type checker can infer the list element type. But argument inference is even more important with dependent types, because more precise types lead to more arguments. If we move from lists to length-indexed vectors, the corresponding functions become parameterized over both element type and length. For example, consider an equation about appending length-indexed lists.

```
append [a] [plus i j] [k]
   (append [a] [i] [j] (VCons [i’] [_] x xs’) ys) zs
=
((VCons [plus (plus i’ j) k] [_] x (append [a] [plus i’ j] [k]
                                      (append [a] [i’] [j] xs’ ys) zs))
                  : Vector a (Succ (plus (plus i’ j) k)))
```

The equation is hard to read because it is cluttered by function arguments, most of which are fixed by the types of `x`, `xs’`, `ys` and `zs`. We can simplify it by using unification to propagate the information from those types:

```
append (append (VCons x xs’) ys) zs
=
VCons x (append (append xs’ ys) zs).
```

The Zombie implementation supports unification-based inference which allows the programmer to write the second expression instead of the first. This feature is a crucial part of creating a practically usable programming language, and it made a big difference in the ease of writing our example programs.

The theory of unification-based inference is less well developed than the material in previous chapters. Zombie features both unification-based inference and automatic reasoning using congruence closure, and we do not yet know precisely how much inference can be done by combining these two methods. However, in this chapter we bracket the problem in two directions.

In the first direction, we describe the current Zombie implementation. The type system is based on the bidirectional system from Chapter 5 extended with additional rules for inferable arguments (Section 6.1). To check types "up to congruence" it uses a heuristic algorithm to search for unifiers modulo the equations in the context (Section 6.2). If the inference fails the typechecker will complain about an uninstantiated unification variable and the programmer has to add an annotation. However, it works very well on our suite of example programs—the programs are not more cluttered than in other languages, and we can infer all the arguments that we would expect.

In the other direction, we present an undecidability result which states that we can not hope to infer *all* type arguments (Section 6.3). Unification modulo equations is a hard problem, and if our type system includes an "up to congruence" rule like CCAST in Chapter 5 then even if the rest of the type system is kept very minimal it is still possible to construct programs that require the type checker to solve arbitrary unification problems (Section 6.3.2). Even if one could restrict the language to rule out these situation, there are still open problems that would need to be solved to create a complete inference algorithm. In particular, although unification modulo equations has been investigated, previous work mostly deals with untyped terms, whereas we would need a system that handles dependently typed expressions (Section 6.3.1).

These two observation suggest a direction for future work (Section 6.4): explain the "unreasonable effectiveness" of the current Zombie implementation by defining a declarative types system which concisely explains what arguments unification can successfully infer. We suggest that this problem could be factored into two parts: first define a type system in terms of simultaneous equational unification, and then look for a decidable class of such unification problems.

6.1 When, where, and what to infer

In the past, different languages have taken different approaches to inferring arguments. For example, they differ in whether the programmer explicitly needs to turn on inference for a particular function, what to do when an argument is under-specified, and how far to propagate constraint information from a given site in the program text. The choices made by Zombie are not novel, but it is useful to indicate what they are.

6.1.1 "Equational" versus "inhabitational" arguments

Mainstream programming languages offer two different forms of inferable arguments, exemplified by *type parameters* in Haskell or ML, and *typeclass dictionaries* in Haskell. The difference is whether there is expected to be a unique solution or not. Type parameters are supposed to be completely determined by the context. On the other hand, for typeclass instances the exact way that the compiler constructs a dictionary does not matter as long as it has the right type.

The same distinction also holds true in dependently typed languages, but here more kinds of objects can be usefully inferred. In addition to type arguments, it is also often possible to infer *type indices*. Just like types, indices are an important part of the interface that a program exposes, so we do not want the typechecker to guess which index-expression was intended but we welcome the saved keystrokes if it can be inferred automatically. And in addition to typeclasses, one can also infer *proofs*. In mechanized mathematics it is usually the case that proofs are computationally irrelevant: we want to know e.g. that a divisor is nonzero or that a purported group operation is associative, but how the proof is constructed makes no difference for the rest of the program—only that there exists *some* proof term inhabiting the type. Coq provides a feature called "canonical structures" for automatically constructing inhabitants of certain types, which for example has been used to great effect to manage proofs about algebraic structures (including the famous proof of the Odd Order theorem) [61, 62].

Although Haskell keeps typeclass dictionaries and ordinary data values quite separate (they are in different namespaces and defined using different syntax), the current trend in language design seems to be to make them more similar. In Scala, the corresponding feature ("implicits" [101]) works on ordinary objects: any local variable can be marked as being part of the "implicit context", and the typechecker will try to instantiate "implicit function arguments" with an implicit variable of the right type. Similarly, recent versions of Agda has added "instance arguments" (written using double-braces, `{{ eq T }}`, in the concrete syntax), which are instantiated by any in-scope variable of the right type [48].

From a language design perspective, the problem with the inhabitation-style search is what happens if there is more than one possible variable of the right type. Agda and Scala will reject a program if there is more than one candidate in the context, forcing the programmer to specify explicitly which one should be picked. Another solution is to somehow ensure that the choice doesn't matter. For example, Haskell maintains as an invariant that there can only be one typeclass instance globally for any given type, so even if there is more than one way to build a dictionary from local variables, the choice will have no observable effect at runtime.

(It is tempting to draw a connection to computational irrelevance here, and say that computationally irrelevant arguments can always be guessed freely. However, even if the choice does not matter at runtime, in a dependent language it can still affect further typechecking. For example, suppose we are given a function with two irrelevant arguments, $\bullet(x:A) \to \bullet(y:A) \to (B\ x\ y)$, and we treat x and y as Agda-style instance arguments. It is possible that other parts of the program requires a type $B\ a_1\ a_2$ where a_1 and a_2 are definitionally equal, which would depend on the details of which instance arguments the typechecker picked. So Agda retains the requirement that there be a unique candidate for instance arguments, even for irrelevant arguments.)

In Zombie, we so far focus on the equational kind of inferable arguments. We have also added an experimental option (`--typeclassy`) which instantiates left-over unification variables by any variable of the right type from the context. This can be used to implement programs using typeclasses like as `Eq` and `Ord` (e.g. `member` in Chapter 2), but neither the implementation nor the language design is polished.

However, although we do not claim a general mechanism for typeclasses/implicits/instances in Zombie, there are some types for which we do try to infer any inhabitant: equations. Proofs of equations in Zombie never carry any interesting computational content (they always evaluate to the uninformative value `join`), so the run-time behavior of a program never depends on what proof was inferred. And as we saw in Section 3.6.2, they satisfy something close to proof irrelevance at typechecking-time: although not all expressions of equational type are equal, it is always possible to replace an arbitrary expression with just `join`, so there is never any reason for the programmer to write a type that depends in any interesting way on a particular proof. Inferring proofs of equations makes a lot of sense because we base the surface language around congruence closure, so there is a particular automatic theorem proving algorithm available to find such proofs.

6.1.2 Marking arguments as inferable

Another design decision is how the programmer indicates that a certain argument of a function should be inferred automatically. In ML, this is done through the distinction between terms and types: term arguments are explicit and type arguments inferred. The programmer does not need to annotate the program in any way in order to enable this.

Coq provides a feature (`Set Implicit Arguments`) which tries to provide similar automatic support in a dependent setting. Given a function type such as

```
cons : (A:Type) → (n:Nat) → A → Vec A n → Vec A (Suc n)
```

Coq will automatically realize that when cons is fully applied, the arguments `A` and `n` can be inferred from the types of the last argument, and so does not need to be given. However, a completely automatic system like this is not flexible enough. In order to deal with partial applications, or to provide redundant arguments for clarity, most languages seem to be moving towards making the programmer explicitly mark inferable arguments. Coq, Agda, and Twelf all allow the programmer to explicitly control inferability (by writing curly braces around inferable arguments and round parentheses around non-inferable ones).

This is also the solution we adopt in Zombie. However, instead of curly braces, inferable arguments are indicated by double arrows (as in Haskell typeclass arguments).[11] So the type of cons can be given as

```
cons : (A:Type) ⇒ (n:Nat) ⇒ A → Vec A n → Vec A (Suc n)
```

The inferable arguments get inserted when elaborating applications. The rule for checking an application $a\ b$ first synthesizes the type for a. If a's type begins with some inferable arrows, e.g. $a : (x : A) \Rightarrow (y : B) \Rightarrow C$, the elaborator creates new unification variables tagged with the expected types, in this case X_A and Y_B, and then goes on to check the application $a\ X_A\ Y_B\ b$. So for example, an application `cons x xs` behaves the same as `cons (?A : Type) (?n : Nat) x xs`. The programmer can also explicitly create new unification variables by underscores, so if the type of `cons` did not mention inferability, one could get the same effect by writing `cons _ _ x xs`. Finally, there is a way to turn off the automatic insertion of unification variables for a particular application, in case the inference would not work and the arguments need to be written out explicitly.

The distinction about whether a function argument is expected to be inferable by unification is orthogonal to whether it can be erased at runtime, so in the surface language there are four different arrow types:

$(x{:}A) \to B$	Ordinary functions
$\bullet(x{:}A) \to B$	E.g. preconditions, $\mathsf{safediv} : (x\ y : \mathsf{Int}) \to \bullet(y \neq 0) \to \mathsf{Int}$
$\bullet(x{:}A) \Rightarrow B$	E.g. polymorphism, $\mathsf{map} : \bullet(A\ B : \mathsf{Type}) \Rightarrow (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B$
$(x{:}A) \Rightarrow B$	E.g. typeclasses, $\mathsf{sort} : \bullet(A : \mathsf{Type}) \Rightarrow (\mathsf{Ord}\ A) \Rightarrow \mathsf{List}\ A \to \mathsf{List}\ A$

On the other hand, inferability does not affect the types in the core language. It is only used to guide elaboration.

[11] This is mostly a historical accident: it was easier to extend the parser with arrows than curly braces.

6.1.3 When to solve for unknown terms

The previous subsection explains when to create unification variables. We also need to explain when to solve them. Because we want to work up-to-congruence, we assume that we have a solver that takes a goal of the form

$$\Gamma \vDash A = B$$

and is allowed to instantiate any unification variables occurring in Γ, A and B in order to make the equation provable by congruence closure. (This is not necessarily easy, see Section 6.2, but we will take it on faith for now.) Then the question is when the typechecker should call the solver.

In the current implementation of Zombie, this is done in the "mode change" rule ECINF of the bidirectional system: when an expression synthesizes to a type A but is checked against the type B. The rule is exactly the same as in Chapter 5, we just assume that the algorithm implementing $\Gamma \mapsto A \overset{?}{=} B \rightsquigarrow v_1$ is generalized to also instantiate unification variables.

$$\frac{\begin{array}{c}\Gamma \mapsto a \Rightarrow a' : A \\ \Gamma \mapsto A \overset{?}{=} B \rightsquigarrow v_1\end{array}}{\Gamma \mapsto a \Leftarrow B \rightsquigarrow a'_{\triangleright v_1}}\ \textsc{ECInf}$$

When the typechecker encounters a unification variable, in most cases it does not do anything special. It trusts the annotation on the unification variable to be the type of the entire expression, and the elaborated form of the expression is just the variable itself.

$$\frac{}{\Gamma \mapsto X_A \Rightarrow X_A : A}\ \textsc{EIunivar}$$

However, we make one exception, for unification variables which are ascribed a type that is congruent to an equation. In that case, we immediately call the equation solver, and replace the variable with the corresponding proof.

$$\frac{\Gamma \mapsto A =^{?} (a = b) \rightsquigarrow v_1 \quad \Gamma \mapsto a \overset{?}{=} b \rightsquigarrow v}{\Gamma \mapsto X_A \Rightarrow v_{\triangleright \mathsf{symm}\, v_1} : A}$$

This rule generalizes the rule ECREFL from Chapter 5. There we treated the underscore as special syntax to invoke the congruence solver, but in the actual Zombie implementation underscores always create unification variables, and whether those variables are instantiated by unification or solved by congruence closure depends only on the ascribed type.

The mode-change rule combined with the usual bidirectional application rule suffices for most use-cases of inferred arguments. For example, we can typecheck the expression `cons (?X : Type) (?Y : Nat) x xs` as follows:

1. `cons` is a variable, so we synthesize its type by looking it up in the context:

$$cons : (x\!:\!\mathsf{Type}) \to\ (y\!:\!\mathsf{Nat}) \to\ x \to \mathsf{Vec}\ x\ y \to \mathsf{Vec}\ x\ (\mathsf{Suc}\ y)$$

2. In order to synthesize a type for $cons\ X$, we must check the argument X_{Type} against Type. It is a unification variable, so its type synthesizes to Type by EIunivar, so the synthesized and checked type trivially match. The type of the entire expression is

$$cons\ X : (y\!:\!\mathsf{Nat}) \to\ X \to \mathsf{Vec}\ X\ y \to \mathsf{Vec}\ X\ (\mathsf{Suc}\ y)$$

3. Similarly, we get

$$cons\ X\ Y : X \to \mathsf{Vec}\ X\ \ Y \to \mathsf{Vec}\ X\ (\mathsf{Suc}\ \ Y)$$

4. Now in order to synthesize $cons\ X\ Y\ x$ we need to check x against X. Term variables are synthesizable expressions, so we look up x's type in the context, and get e.g. $\Gamma \vdash x \Rightarrow \mathsf{Nat}$. In the rule ECinf we then need to solve the goal $\Gamma \vDash \mathsf{Nat} = X$, which can be done by instantiating X to Nat.

$$cons\ X\ Y\ x : \mathsf{Vec}\ \mathsf{Nat}\ \ Y \to \mathsf{Vec}\ \mathsf{Nat}\ (\mathsf{Suc}\ \ Y)$$

5. Similarly, the application $cons\ X\ Y\ x\ xs$ will instantiate the variable Y.

In this way, the inferred type of a subexpression will often contain unification variables, which may later get instantiated when typechecking a different part of the program. It is an error if any variables remain uninstantiated after the entire program has been elaborated, and this will be detected when the Zombie implementation typechecks the generated core term.

In our experience unification-based argument inference in Zombie works well in practice. It is hard to quantify this, since we do not have a benchmark for what expressions "should" be possible to infer, but subjectively it does not seem that we need more annotations than, say, Coq or Agda. One exception has to do with our heterogeneous equality. Often one writes down an equation where one side is the empty list, e.g. `f x = Nil`. In this case, Zombie can not infer the type parameter of the constructor, so in order to not get an uninstantiated unification variable the programmer has to add an type annotation:

```
f x = (Nil : List Nat)
```

This is slightly annoying, but of course it makes sense: because the Zombie equality is heterogeneous, the typechecker can not use the type of `f x` in order to infer the type of `Nil`. (An example of this can be seen with `EmptyTree a` in Section 2.1).

One **limitation** of this scheme is that although arguments can be inferred, there is no facility to infer the type of the applied expression. Zombie will not infer types of λ-expressions such as $\lambda x.b$ in general, because the elaborator requires that all function expressions be ascribed a function type which is used to figure out what the type of the variable x is. This means that variables in the context will usually not have unification variables in their types.

One could imagine a version of Zombie which instead tried to infer these, by introducing a binding $x : X$ into the context, and waiting to see what the unification variable X gets instantiated to. For first-order functions, this will generally work. However, for higher-order functions, when x may itself be applied to arguments, this reveals an interesting difference between type inference for simply-typed languages and dependently typed ones. To see the difference, consider explicitly formalizing the unification-constraints generated by typechecking a term a, as a judgement $\Gamma \vdash a : A \, \textsf{given} \, \Phi$ (where A and Φ are outputs). Such constraint-based systems are a standard way to develop the theory of type inference [110]. However, the dependent application rule requires a more powerful constraint language.

For a simply typed language, the application rule may look as follows.

$$\frac{\Gamma \vdash a : A \, \textsf{given} \, \Phi_1 \quad \Gamma \vdash b : B \, \textsf{given} \, \Phi_2}{\Gamma \vdash a \; b : Y \, \textsf{given} \, \Phi_1, \Phi_2, A = B \to Y} \textsc{ConTapp}$$

In words, this states that to apply a term, it much have a function type, and the domain of the function type must match the provided argument. When moving to dependent types, however, there is a complication, because the result type of the application now involves a substitution. So we can no longer say just that the function domain is some unknown term Y; instead it is some unknown *expression* involving a bound variable x, such that substituting b for x yields the result type of the application $a \; b$. This is not an ordinary first-order unification constraint. It is a *second order unification* problem, where some variables should be instantiated with types, but some variables should be instantiated with functions from types to types. In general, second order unification is undecidable.

Languages like Coq and Agda already implement (approximations to) full higher-order unification for their type inference, so this style of constraint can be easily accommodated by including a type-level function in the constraint:

$$\frac{\Gamma \vdash a : A \, \textsf{given} \, \Phi_1 \quad \Gamma \vdash b : B \, \textsf{given} \, \Phi_2}{\Gamma \vdash a \; b : (\lambda x.Y) \; b \, \textsf{given} \, \Phi_1, \Phi_2, A = (x\!:\!B) \to \; ((\lambda x.Y) \; x)} \textsc{ConTdapp}$$

However, the Zombie surface language was specifically designed to not include β-

reduction in its definitional equality, so we do not implement higher-order unification and it is hard to see how to write a fully general application rule. We can see the problem in the application rule EIdapp as it is currently implemented:

$$\frac{\begin{array}{l}\Gamma \mapsto a \Rightarrow a' : A_1 \\ \Gamma \mapsto A_1 =^{?} (x : A) \to B \leadsto v_1 \\ \Gamma \mapsto v \Leftarrow A \leadsto v' \\ \Gamma \vDash \mathsf{injrng}\,(x \!:\! A) \to\ B\ \mathsf{for}\ v'\end{array}}{\Gamma \mapsto a\ v \Rightarrow a'_{\triangleright v_1}\ v' : \{v'/x\}\,B}\ \textsc{EIdapp}$$

When solving the goal $\Gamma \mapsto A_1 =^{?} (x : A) \to B \leadsto v_1$, the implementation will look for an arrow type which is equal to A_1, but if A_1 happens to be a unification variable we will not create a new arrow type to instantiate the unification variable with. With just first-order unification constraints, there is no way to write down an arrow type that is sufficiently general.[12]

6.2 Solving equational constraints

After generating equational constraints, we also need to solve them. One of the lessons from the Zombie implementation is that rather simple syntactic unification (Section 6.2.1) suffices to typecheck most practical example programs.

However, one worry when doing so is that this would upset the programmer's mental model of the system. The type system based on bidirectional typing and congruence closure described in Chapter 5 has the pleasant property that all types are checked only up to congruence closure; the exact syntactic form does not matter. In order to get a theoretically well-behaved type inference system, we would want the constraint solver to satisfy the same property, i.e. the result of the query to the solver should only depend on the equivalence class of the inputs.

Therefore we are led to study unification with respect to a context of equality assumptions. That is, given two terms A and B which contain unification variables, find a substitution σ such that

$$\sigma\Gamma \vDash \sigma A = \sigma B$$

This generalization of the unification problem is called *rigid E-unification*. It was first studied by Gallier et al. [53, 54] who proved that it is decidable and NP-complete.

Because the problem is NP-hard it is difficult to find an efficient and complete algorithm. In Zombie we have experimented with two heuristic algorithms. The simplest

[12] For the same reason, type inference for System F is equivalent to second-order unification, and hence undecidable [112].

is to treat unification and congruence closure completely separately (Section 6.2.1); the other is a backtracking search that interleaves unification and congruence reasoning (Section 6.2.2). Neither algorithm is complete, but they perform well on our example programs.

6.2.1 Simple syntactic unification

The simplest approach to unification modulo congruence is to completely separate unification and congruence. Whenever we have to solve a problem $\Gamma \vDash A = B$, we first syntactically unify A and B "as much as possible", and then check if the resulting terms σA and σB are in the congruence closure of $\sigma\Gamma$.

"As much as possible" means that we proceed as in the usual recursive unification algorithm, except that we never return "not unifiable". In cases where were we are asked to unify two different term constructors, or when the occurs check fails, we just move on and hope that the two terms will later turn out to be provably equal to each under the assumptions Γ. In the following pseudocode, X denotes unification variables, and $F\,\overline{a_i}$ is a label application in the sense of Section 5.5.1 (i.e. F is a syntactic constructor in the core language).

```
$\mathsf{unify}(X, a)$ = $\mathsf{assignVar}(X, a)$
      when $X \notin \mathsf{fv}(a)$
$\mathsf{unify}(X, a)$ = $\mathsf{return}\ ()$
      otherwise. //failed occurs check.
$\mathsf{unify}(F\,\overline{a_i}, F\,\overline{b_i})$ = $\mathsf{zipWithM}\ \mathsf{unify}\ \overline{a_i}\ \overline{b_i}$
$\mathsf{unify}(F\,\overline{a_i}, G\,\overline{b_i})$ = $\mathsf{return}\ ()$
```

We have implemented this algorithm as an option in Zombie (`--cheap-unification`), and we find that it suffices for almost all the uses of unification in our set of test-cases. There are only four locations in the source code where the cheap unification fails and the more expensive default algorithm (Section 6.2.2) succeeds. These four cases fail in essentially the same way, so it suffices to look at one of them, e.g. the following definition of a function `lookup` which retrieves an element from a length-indexed list (`Vector`). In order to ensure that the index is in range, it is represented as a `Fin n`, i.e. a natural number strictly smaller than `n`. (The declaration of `Fin` ensures that $n > 0$, which forbids lookups into the empty vectors.)

```
data Fin (n : Nat) : Type where
  FZ of [m:Nat] [n = Succ m]
  FS of [m:Nat] [n = Succ m] (Fin m)

head : [A : Type] ⇒ [n:Nat] ⇒ Vector A (Succ n) → A
head = ...

tail : [A : Type] ⇒ [n:Nat] ⇒ Vector A (Succ n) → Vector A n
tail = ...

lookup : [A: Type] ⇒ [n:Nat] → Fin n → Vector A n → A
lookup = λ [A] .
  ind recFin [n] = λ f v .
    case f [f_eq] of
      FZ [m][m_eq] → head v
      FS [m][m_eq] fm → recFin [m] [ord m_eq] fm (tail v)
```

The function `lookup` is written in terms of helper functions `head` and `tail`. But with the cheap unification method, Zombie will fail to infer the argument `n` for the call `(tail v)` on the last line. The constraint that we need to solve is

```
Vector A n = Vector (?X : Type 0) (Succ (?Y : Nat))
```

Syntactic unification will instantiate $X := A$. However, both n and $\mathsf{Suc}\ Y$ are headed by (distinct) constructors, so from only looking at these two terms there is no way to know how to instantiate Y. In order to typecheck this expression, we need to note that the context contains the variables

$$\begin{array}{lcl} m & : & \mathsf{Nat} \\ m\_eq & : & n = \mathsf{Succ}\ m \end{array}$$

and use this to instantiate $Y := m$. In other words, we need to take equality assumptions in the context into account.

6.2.2 Unification on equivalence-classes

To deal with such cases, the default unification algorithm in Zombie does unification *after* congruence closure. That is to say, to solve a goal $\Gamma \vDash a = b$ we begin by running the congruence closure algorithm described in Section 5.5, which partitions all subexpressions of Γ, a and b into equivalence classes. If after that, a and b are not in the same equivalence class, we try to unify them. The unification algorithm operates on classes, as follows:

- If the equivalence class of a contains a unification variable X, then assign $X := b'$, where b' is some arbitrary member of the equivalence class of b (chosen to not contain X, in order to pass the occurs check). The two equivalence classes can now be merged.

- If neither class contains any unification variables, see if a is congruent to some expression $F\,\overline{a_i}$ and b is congruent to some expression $F\,\overline{b_i}$, where both expressions are headed by the same label F. If so, recursively try to unify a_i and b_i for all i.

We can see how this works for the problem

```
Vector A n = Vector (?X : Type 0) (Succ (?Y : Nat))
```

that we mentioned above. Neither of the two expressions are provably equal to a unification variable, but they *are* headed by the same constructor `Vector`, so we go on to recursively unify `A` with `?X` and `n` with `(Succ (?Y : Nat))`. The first subproblem can be solved by assigning `?X`. In the second subproblem, we see that the equivalence class of `n` contains `(Succ m)`, which is headed by the correct constructor. So we go on to unify `m` with `?Y` as desired.

This algorithm satisfies the property that whether two terms are unifiable only depends on what equivalence classes they are in, and it can solve all the unification problems that occur in our testcases. However, one can raise two complaints against it.

First, the above description does not specify what happens if there is more than one choice of label applications. For example, if we know $a = F\,\overline{a_i} = F'\,\overline{a_i}'$ and $b = F\,\overline{b_i} = F'\,\overline{b_i}'$, then we could choose to recursively try to unify either a_i with b_i or a_i' with b_i'. In this situation, the Zombie implementation will try all possibilities, in a backtracking search.

In the current set of example programs, this situation with multiple possible decompositions only happens a handful of times, which are all easy to resolve. However, in general there is no guarantee that the search will terminate quickly. In particular, the usual proof that rigid E-unification is NP-hard works by reducing Boolean satisfiability to rigid E-unification, as follows. We encode propositional formulas as syntactic formulas constructed from True, False, $\wedge$, and $\neg$, and work with respect to the following context Γ, which contains equations specifying the logical connectives:

$$\begin{aligned}\Gamma \equiv\quad & \mathsf{True} \wedge \mathsf{True} = \mathsf{True}, \mathsf{True} \wedge \mathsf{False} = \mathsf{False},\\ & \mathsf{False} \wedge \mathsf{True} = \mathsf{False}, \mathsf{False} \wedge \mathsf{False} = \mathsf{False},\\ & \neg\mathsf{True} = \mathsf{False}, \neg\mathsf{False} = \mathsf{True}\end{aligned}$$

Now a Boolean formula such as $\phi \equiv (X \wedge \neg Y) \wedge \neg(Y \wedge X)$ is satisfiable iff $\phi = \mathsf{True}$ has a unifier with respect to Γ. The only way to unify them is to search for an assignment of $\mathsf{True}/\mathsf{False}$ to the variables.

If we encode the above satisfaction problem as a Zombie program, the typechecker will indeed instantiate $X := \mathsf{True}$ and $Y := \mathsf{False}$—after doing a backtracking search over different ways to match True against $\mathsf{True} \wedge \mathsf{True}$ or against $\neg\mathsf{False}$ and so on. In general, this is a quite inefficient way to enumerate propositional assignments, and for large formulas it would be very slow.

Second, this is not a complete algorithm for rigid E-unification. Tiwari et al. [136] give the example of unifying $gfffgffX$ and $fffX$, given the equations $gX = X$ and $X = a$. Picking $X := fa$ is a solution, but one cannot find it just by equating two function arguments from the input problem, and indeed the Zombie typechecker is not able to solve this problem instance. To handle cases like this, all known complete algorithms for rigid E-unification include a rule of last resort, which tries to find a binding for a unification variable X by exhaustively trying each subexpression of the input problem in turn.

6.3 Challenges for complete type inference

Although it is now 25 years since rigid E-unification was first proposed, algorithms for solving the problem have still not been studied very intensely. Early algorithms were purely theoretical, i.e. they demonstrated that the problem was in NP but would not be practical to implement. Even more recent algorithms are often only proven correct, not implemented. Broadly speaking, there seems to be two styles of unification algorithms. On one hand, Goubault [63] and Tiwari, Bachmair, and Ruess [136] extend existing algorithms for congruence closure to also search for unifiers. Neither of these algorithms seems to have been implemented. On the other hand, there are algorithms inspired by *paramodulation* methods in equational/first-order theorem proving. These algorithms consist of a nondeterministic set of rewriting rules that search for a unifier, together with a constraint system which tries to cut down the search space by ensuring that the rewrite rules are only applied in a "decreasing" direction, Algorithms by Becher and Petermann [19] (implemented once by Grieser [64]) and by Degtyarev and Voronov [47] (implemented once by Franssen [52]) fall in this category.

However, these algorithms all deal with the untyped unification problem, in the context of theorem provers for first-order logic. Just as with congruence closure (Section 5.5), integrating rigid E-unification into a dependently typed programming language creates additional complications.

6.3.1 Typed rigid E-unification

First, most treatments of unification deal with untyped terms and attempt to find most general unifiers (which typically contain un-assigned unification variables). When using unification in a dependent language however, leaving unassigned unification variables is not acceptable, since they can correspond to unproved lemmas. So in order to accept a program as well-typed, the typechecker needs to find a ground substitution for every unification variable. This is a source of undecidability, since it is undecidable whether a given type has inhabitants or not.

Even if we do not require the algorithm to pick ground unifiers, in a typed setting we still need to me make sure to only produced well-typed unifiers. A priori, having type information available could either help or hurt. If we can rule out candidate substitutions because they don't have the right type, then types are helpful. On the other hand, if we must consider all the same candidate solutions as for the untyped problem, and then additionally check that they are well-typed, then the types slow us down.

Gallier and Isakowitz [55] studied rigid E-unification for a simply-typed system, and found that type information could be used to prune the search early. In that setting, when considering candidates for a unification variable X_B, we need only consider expressions of type B.

Unfortunately, with dependent types, we are not so lucky. Consider the context

```
f : [T:Type] → T → Type
A : Type,
B : Type,
y : A,
X : B,
h1 : A = f [A] y ,
h2 : f [B] X = B
```

It is not the case that $\Gamma \vdash y : B$. However, $\{y/X\}\Gamma \vdash y : B$. In other words, it is not sufficient to consider only expressions which have type B, because the type of the expression may change after we carry out the instantiation. This context is not inconsistent, one can inhabit it e.g. by setting A and B to Nat and f to the constant Nat function. So this style of example seems hard to avoid.

Performance The Zombie implementation is very cavalier about creating well-typed substitutions. While the elaborator tags each unification variable with an expected type, the unification code currently ignores those annotations, and it is possible to construct examples where the equational constraint get satisfied by an ill-typed term. (Ill-typed unifiers will be detected by the typechecker for the core language.) One annoying example which sometimes happens in practice is due to erased type casts: the terms $a_{\triangleright v}$ and a are propositionally equal but have different types, and the unifier can get confused about which one to pick for a given unification variable.

This limitation is partly due to performance concerns. In general, in order to correctly check that a has the right type one should take congruence classes into account.

(Requiring that types of a and the ascribed type of the unification variable are syntactically identical can rule out correct programs, e.g. if instantiating a unification variable would make the types equal.) However, since any subexpression in the context might be part of a unifier, checking the types up to congruence closure requires calculating the CC equivalence class of every subexpression, which can be very expensive.

The cost of computing the congruence closure of types is also relevant when implementing typeclasses/implicits/instance arguments (Section 6.1.1). This involves instantiating a unification variable with some expression of the right type, so if we want to work up to congruence, we need to compute the congruence closure of the types of all candidate expressions. But the situation here is better, because arbitrary subexpressions of the typing context are not candidates for instance arguments; only the variables in the context (Agda) or some special subset of those variables (Haskell and Scala). Similarly, to implement the assumption-up-to-CC rule (Section 5.2) we need to compute the congruence closure of types in the context, but not subexpressions of those types.

The following table shows the time it took to run the Zombie test suite with the typechecker instrumented to process different subsets of the context. While the absolute numbers are arbitrary (they depend on what example programs happened to be written in November 2013), the relative magnitudes give a rough indication of the cost of the different options.

Equations only	35.8s
Datatypes in context	36.3s
All types in context	83s
Every subexpression in context	180s

With just the "classic" assumption rule (Section 5.2), the congruence closure algorithm only needs to process equations in the context. To implement a Scala- or Haskell-style typeclass system up to congruence, we also need to process some other subset of the assumptions—in this example we processed every variable in the context that inhabits a datatype. With the full assumption-up-to-CC rule, and to implement an Agda-style system, we need to process the type of every variable in the context, since we do not know a priori which types will turn out to be equal to an equation. Finally, in order to track welltypedness of unifiers, we also want to process all subexpressions of types in the context, and the types of the types of the subexpressions, and so on.

It can be seen that the assumption-up-to-CC rule is quite costly, increasing the time by more than factor of 2. Tracking *every* type is worse still, and is not done by the current implementation.

6.3.2 Simultaneous rigid E-unification

There is an important mismatch between the rigid E-unification problem described above and the unification problems generated by type inference: when generating equality constraints by typechecking a program, not every constraint will be in the same context Γ. For example, most unification based type systems will generate at least a constraint for every application in the program, and not every application will be in the same typing context.

Thus we are led to consider a slightly generalized problem, namely

> Given m contexts $\Gamma_0 \dots \Gamma_m$ and m pairs of terms $A_1, B_1, \dots, A_m, B_m$ which contain unification variables, find a substitution σ such that
>
> $$\forall i.\ \sigma\Gamma_i \vDash \sigma A_i = \sigma B_i$$

This problem is known as *simultaneous rigid E-unification.* Unfortunately, it turns out to be undecidable [46].

If we insist that type inference be decidable, and still allow implicit use of congruence closure anywhere, one way to proceed would be to impose a restriction on the set of typeable programs. For example, one could insist that all constraints used to solve for a particular unification variable must be defined in a single typing context, and make the typechecker signal an error if there are still unsolved unification variables when a variable goes out of scope. However, any restriction which is strong enough to ensure decidability would also severely limit the power of type inference.

We can formalize this objection by fixing a minimal set of type rules which we want the typechecker to be able to handle, and then ask whether it is possible to create an algorithm which is complete for both those rules and the CC-conversion rule. A rather modest target may be to ask for a complete inference algorithm for the simply-typed lambda calculus. Disregarding congruence reasoning, it is very easy to write a unification based algorithm which decides whether a (Curry-style) λ-expression is typeable in STLC or not. In a practical programming language we will also want a way to ascribe specific types to expressions, so we should include a rule for type annotations. After adding the up-to-congruence rule, we end up with the following set of five rules. (This language is meant to be a subset of Zombie, so we assume the judgement $\Gamma \vDash A = B$ is defined by rules similar to Figure 5.2 in Chapter 5.)

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B} \qquad \frac{\Gamma \vdash a : A \to B \quad \Gamma \vdash b : A}{\Gamma \vdash a\ b : B}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a_A : A} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vDash A = B}{\Gamma \vdash a : B}$$

However, already with just these rules the problem of inferring arguments is undecidable. One can create arbitrary typechecking contexts by combining lambdas and type annotations, and then pose the unification problem $\Gamma \vDash A = B$ by asking if there exists some substitution of values for unification variables which makes the expression $(x_A)_B$ well typed.

In more detail, the reduction from simultaneous E-unification to type inference is done as follows. Suppose $\Gamma \equiv x_1 : A_1, x_2 : A_2, \ldots x_n : A_n$. Then we define the term

$$(\Gamma \vDash A = B)^* \quad \overset{\text{def}}{=} \quad (\lambda x_1\ x_2\ \ldots\ x_n\ y.(y_A)_B)_{A_1 \to A_2 \to \cdots \to A_n \to A \to B}$$

which is typeable under some substitution σ exactly if the corresponding congruence closure judgement holds. In order to verify that the reduction is correct, we need a few lemmas:

***Lemma* 24** (Context conversion)**.** If $\Gamma \vDash A = A'$, and $\Gamma, \Gamma_1, x : A, \Gamma_2 \vdash b : B$, then $\Gamma, \Gamma_1, x : A', \Gamma_2 \vdash b : B$.

Proof. By induction on the second judgement. In the variable case, if the variable is x we insert an implicit conversion. □

***Lemma* 25** (Inversion lemmas for typing)**.**

1. $\Gamma \vdash \lambda x.b : A \to B$ iff $\Gamma, x : A \vdash b : B$.

2. $\Gamma \vdash a\ b : B$ iff $\Gamma \vdash a : A \to B$ and $\Gamma \vdash b : A$ for some A.

3. $\Gamma \vdash a_A : A$ iff $\Gamma \vdash a : A$.

Proof. The proof of (1) is the most interesting. The right-to-left direction just states the typing rule. In the left-to-right direction we first use induction to prove a weaker statement: if $\Gamma \vdash \lambda x.b : A \to B$ then $\Gamma, x : A' \vdash b : B'$ for some A' and B' such that $\Gamma \vDash (A \to B) = (A' \to B')$. Then by injectivity for arrow types, we know $\Gamma \vDash A = A'$ and $\Gamma \vDash B = B'$. Conclude by context conversion and an implicit conversion. □

***Lemma* 26.** If $\Gamma \vdash (a_A) : B$, then $\Gamma \vDash A = B$

Proof. Induction, using the fact that congruence closure is transitive. □

***Lemma* 27** (Strengthening for congruence closure)**.** If $\Gamma, x : A \vDash a = b$, and it is not the case that $\Gamma \vDash A = (a_1 = b_1)$ for any a_1 and b_1, then $\Gamma \vDash a = b$.

***Theorem* 28.** Let $\cdot \vdash a : A$ be some arbitrary typeable expression. Suppose that $\{(\Gamma_i, A_i, B_i) \mid 1 \leq i \leq m\}$ is a problem instance of simultaneous rigid E-unification, such that it is never the case that $\Gamma_i \vDash A_i = (a = b)$ for some equation $(a = b)$. Then the expression

$$\sigma((\lambda x_1 \ldots x_m.a)\ (\Gamma_1 \vDash A_1 = B_1)^*\ \ldots\ (\Gamma_m \vDash A_m = B_m)^*)$$

is typeable iff σ is a solution of the unification problem.

Proof. By repeatedly applying the inversion rule for applications (lemma 25 (2)), we see that the entire expression is typeable iff $(\sigma\,\Gamma_i \vDash \sigma\,A_i = \sigma\,B_i)^*$ is typeable for each i.

By inversion for λ-expressions, that is true iff $\sigma\,\Gamma_i, x : \sigma\,A_i \vdash (x_{\sigma\,A_i})_{\sigma\,B_i} : \sigma\,B_i$. By inversion for type ascription, and lemma 26, that holds iff $\sigma\,\Gamma_i, x : \sigma\,A_i \vDash \sigma\,A_i = \sigma\,B_i$. Finally by weakening and strengthening (lemma 27), that holds iff $\sigma\,\Gamma_i \vDash \sigma\,A_i = \sigma\,B_i$, as required. □

Simultaneous rigid E-unification remains undecidable if all the expressions involved are known to not equal equations (in the first-order setting that is automatically true, since equations and terms are in different syntactic categories), so this theorem shows that the type inference problem is undecidable even for just STLC+congruence.

Another argument that a practical type inference system needs to simultaneously solve constraints from multiple typing contexts is that these come up in quite pedestrian functional programs. For example, consider the function `mapM` from the Haskell standard library, which uses nested lambda expressions in order to sequence evaluation using the monadic bind combinator:

```
mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM f [] = return []
mapM f (x:xs) = f x >>= (λ y →
                 mapM f xs >>= (λ ys →
                 return (y : ys)))
```

In this example, we want the typechecker to be able to infer the type of the variable `y`. However, imposing the rule that all constraints must live in the same context makes that impossible, because the constraint that fixes the type of `y` is not created until we check the body of `(λ ys → ...)`, i.e. it comes from a context which was extended by the variable `ys`.

6.4 Future work: a type system based on simultaneous rigid E-unification?

So far, we have mentioned three obstacles to writing a complete type inference system: one needs a rich constraint language to write a typing rule for dependent application, it is difficult to find well-typed unifiers, and in a realistic language we need to solve constraints simultaneously. It is intriguing to note that designing a type inference algorithm around simultaneous rigid E-unification (SREU), somewhat magically, seems to address all three problems at once.

The problem of finding **well-typed substitutions**, while still difficult, seems to at least not make the SREU problem much harder. One crucial part of the construction that shows that SREU is undecidable is that it is possible to write down equations which constrain any solution for a unification variable X to be a ground term over a given signature [46]. (For example, in order to force the solution for X to be a ground term constructed from the constants a, b, c and the unary function symbols f and g, one can add the equation $a = c, b = c, fc = c, gc = c \vdash X = c$ to the problem.) Thus, it seems that any algorithm for tackling SREU must already be part of the way towards finding only well-typed ground solutions.

The problem of finding **simultaneous unifiers** is undecidable, but as undecidable problem go it is not so bad: it turns out that SREU is equivalent to *second order* unification.[13] This is a special case of the higher-order unification problem which existing dependent languages routinely solve, so one can hope that the methods they use (such as pattern unification [88, 98]) can be adapted and work equally well in this setting.

Type inference of existing languages like Coq and Agda is typically approached by this two-step approach, by first defining a type system which generates higher-order unification problems, and then studying algorithms for decidable subclasses of the unification problem. We would hope that the same approach would still be effective when reducing the problem to SREU instead of HOU.

And finally, second order unification is exactly what is needed to state an **application rule** for a dependent language. In a language based around SREU, the rule may look something like

$$\frac{\Gamma \vdash a : A \;\mathsf{given}\; \Phi_1 \quad \Gamma \vdash b : B \;\mathsf{given}\; \Phi_2}{\Gamma \vdash a\ b : Z \;\mathsf{given}\; \Phi_1, \Phi_2, (\Gamma \vdash A = (x : B) \to\ Y), (\Gamma, h : x = b \vdash\ Y = Z)}\textsc{ConTdapp2}$$

Here, we record a context Γ with every constraint. The first-order variable Y represents the codomain part of the type of a, while Z represents the same codomain with the substitution b/x applied. The relation between Y and Z is encoded by requiring that $Y = Z$ given the assumption $x = b$.

Of course this is speculation, because there are no known algorithms for typed simultaneous rigid E-unification. But the above observations suggest the possibility that SREU could serve as a useful organizing principle for dependently typed languages, much like higher-order unification does in Coq, Agda, and Twelf.

[13] A reduction from SOU to SREU was found by Degtyarev and Voronkov [46], while one from SREU to SOU was found by Levy [77]. Veanes [137] additionally showed that the two problems are logspace-equivalent.

Chapter 7

Termination checking

The core calculus described in the previous sections covers all the major features of the Zombie core language except one: termination checking. In full Zombie, the programmer can declare certain expressions to be terminating, and the typechecker will enforce this.

The way the terminating checking is expressed in the type system is novel, and motivated by our interest in light-weight verification. Previous languages tend to either offer convenient support for writing total functions while making it inconvenient to define and reason about nonterminating functions (e.g. Coq and Agda, see Section 8.2.4), or they offer convenient general recursion but little or no support for termination checking and proof (e.g. Cayenne, see Section 8.2.1). We would like balance between proving and programming, allowing programmers to devote their verification budget to critical sections. So Zombie should support general recursion as natively as a functional programming language, yet at the same time should provide the reasoning capabilities of a constructive logic proof assistant.

To support this goal, we use the type system to track which programs are known to terminate and which are not, by indexing the typing judgment by an *consistency classifier* `log`/`prog`. Because the distinction is made through typing, the syntax and operational semantics of a program is kept the same whether it uses general recursion or not.

Further, the type system contains several features the which allow interaction between total and potentially nonterminating parts of the program:

- We define the logical language as a sublanguage of the programmatic language, so that all logical expressions can be used as programs.
- We allow uniform reasoning for logical and programmatic expressions through a heterogenous equality type. Two expressions can be shown to be equal based on their evaluation, which is the same for both fragments.

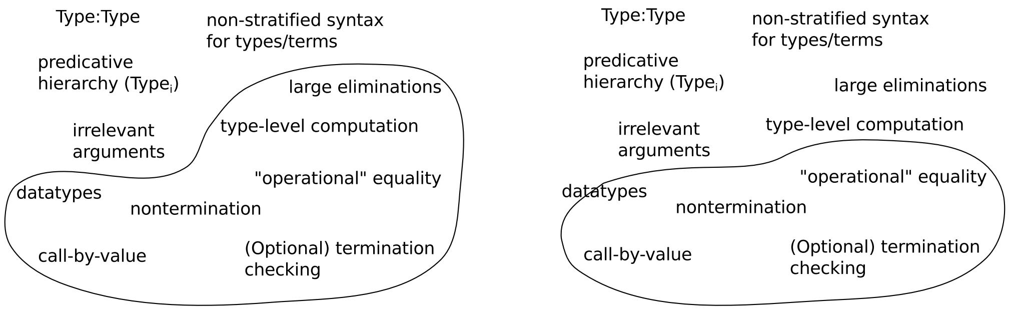

Figure 7.1: The calculi studied in Casinghino's thesis [30] (left) and in this chapter (right).

- We internalize the labeled typing judgment as a new type form $A@\theta$. This type can be used by either the prog or log fragment to manipulate values belonging to the other.
- We identify a set of "mobile types"—those whose values can freely move between the fragments.

In first half of this chapter, we present the general considerations that went into the termination checking design (Sections 7.1 and 7.2), and the detailed typing rules that they resulted in (Section 7.3). In particular, we aim to provide enough detail to make the example programs in Chapter 2 understandable.

When writing down typing rules and talking about their metatheory, we encounter the usual problem of this thesis: the full core language is too large to make proofs tractable. This is particularly true when it comes to normalization proofs, which are notoriously difficult. In fact, Casinghino dedicated an entire phd thesis to just the termination aspects of Zombie [30], and still had restrict attention to a subset (omitting general datatypes, collapsed syntax, universe hierarchies, erasure of function domains and function arguments, and large eliminations).

In this chapter we present a very cut-down version of the core language. Because Casinghino already studied termination checking of Zombie in depth, this thesis does not need to go into details about how the advanced features of Zombie affect the metatheoretic normalization proof. So the calculus in this chapter is even smaller than Casinghino's: we omit many features, but retain the novel typing rules which talk about termination. Figure 7.1 shows schematically how these calculi relate to the full Zombie language. For the most part the calculus in Section 7.3 does not contain novel findings beyond what was already presented in Casinghino's thesis, but additionally we extend it with subtyping (Section 7.3.5) and give a description of how the core calculus relates to the full Zombie language (Section 7.3.6).

However, this type system (which is implemented in the current version of Zombie) has two drawbacks, and addressing those is the focus of the second half of the chapter. First, the design of the typing rules allow typing derivations at `log` to contain subderivations at `prog`, which makes the normalization proof technically difficult (as we explain in Section 7.6.1). This in turn prevented us from supporting useful features in Zombie: intuitively there should be no issue with allowing $\mathsf{Type} : \mathsf{Type}$ or type-level general recursion as long as a program is marked as `prog`, but we were not able to integrate those features into our soundness proof for the `log` fragment (Section 7.7.2). Second, although the core language is suitably expressive it requires a lot of annotations, and it seems very difficult to find a complete algorithm to infer those annotations automatically (Section 7.7.3).

To deal with these two drawbacks, we design a second set of typing rules (Section 7.4), based on the idea of tracking termination using a standard type-and-effect system. The new type system looks superficially different from the calculus in Section 7.3, but we prove that it can type the same programs (Section 7.5). And the new type system leads to a much simpler metatheory, because termination can now by proved by a normal logical relations argument (Section 7.6). We hope that in future work this proof will be easier to extend to handle additional features in the nonterminating part of the type system, and also that it will support easier type inference.

Along the way we also make an observation about combining general recursion with reasoning about nonterminating programs. It turns out that one cannot at the same time allow recursive functions at arbitrary types, reduction at type-checking time, and "termination inversion" (Section 7.7.1). Zombie makes the novel choice of omitting the last one of this three features.

Mechanization Unlike the rest of the thesis, the proofs in this chapter have been mechanized in Coq. The proof scripts can be downloaded from the author's homepage. The proofs reuse some of the earlier development by Casinghino et al. [31], e.g. definitions of typing rules and lemmas about de Bruijn indices could be reused with only light adaptation.

7.1 Why termination checking?

In previous chapters we described a dependently typed language without any termination checking at all. This is certainly a respectable point in the design space: other languages making similar choices include Cayenne [10], Cardelli's Type:Type language [29], and ΠΣ [9]—and even recent versions of Haskell, if you squint a bit [50, 79]. Even so, for the full Zombie type system we believe that support for provably total functions is essential. There are two separate reasons for this.

7.1.1 Precision

The first reason is to better support program verification. Dependent types support both programming and verification in a single language, but without termination-checking, there are limitations about which properties are expressible as types. Most obviously, without termination checking one can prove partial but not total correctness properties. For example, we mentioned previously (Section 3.3) a function to match regular expressions:

```
match : (s:String) → (r:Regexp) → Maybe (Matches s r)
```

The type guarantees that if the function returns `Just m` then the string matches. On the other hand, without termination checking there is clearly no way to express the fact that the function always returns.

Phrasing it differently, a language without termination checking can prove *safety* but not *liveness* properties. For example, if we use dependent types to encode access control (as in Aura [70] or Aglet [92]), then without termination checking we can be sure that the `read` function will not give access to unauthorized users, but it may still get stuck in an infinite loop.

For lightweight verification, not being able to prove liveness is perhaps OK. In real applications we want to known not only that a function is terminating but that it terminates reasonably quickly—a function which uses exponential time or space is as vulnerable to denial-of-service attacks as one which does not terminate at all. For most projects, formally proving space/time bounds does not provide enough benefit to justify the cost, and the same could be said for termination proofs.

Rather, the main reason we are interested in termination checking is to make more *properties and proofs* expressible. For example, even putting aside the fact that `match` may not terminate, the type given above has another drawback: it leaves open the possibility that the `match` incorrectly returns `Nothing`. In order to ensure that the function classifies strings correctly we want to give it a more expressive type, such as

```
match : (s:String) → (r:Regexp) → Either (Matches s r)
                                          (Matches s r → False)
```

This type forces the programmer to construct a proof either that the string matches or that it *doesn't* match (using the standard definition of negation, $\neg P \equiv P \to \mathsf{False}$). But without termination checking, the type `Matches s r → False` is uninformative—it is inhabited by the trivial function $\lambda x.\mathsf{loop}()$—so types can only express properties which can be witnessed by some first-order datatype, such as `Matches`.

Without termination checking we could express *soundness* of `match`, but not *completeness*. The same pattern holds for many examples beyond regular expression matching. For example, one classic illustration of the power of dependent types is implementing correct-by-construction type checkers [86]. Here again, without termination checking one can express that the typechecker is sound but not that it is complete.

A particularly important example is verified SAT-solvers for propositional logic (either the scaled-down example in Section 2.4, or full-strength solvers like Versat [100]). Without termination checking we can express that the solver is sound, i.e. that if it returns a variable assignment then that assignment does satisfy the formula. But that is a very uninteresting property! The user can easily check that manually, by just evaluating the formula with the given assignment. The critical question is completeness: can we trust the solver when it declares a formula to be unsatisfiable (after a long search using subtle techniques to cut down the search space)? Some SAT solvers can construct explicit proofs of unsatisfiability (in some given logic), but such proof terms can be hundreds of megabytes in size and take longer to verify than to generate in the first place [130]. It is better to prove the SAT-solver itself partially correct. But the property "the formula ϕ is unsatisfiable" involves a function space (we can write it as the dependent implication $(\sigma : \mathsf{Assignment}) \to \mathsf{eval}\ \sigma\ \phi = \mathsf{false}$), so we need a type of total functions in order to express it.

Apart from stating properties, we also need termination-checking to prove them. Almost all interesting proofs involve *induction*. But the computational meaning of induction is just structural recursion; you are licensed to invoke the induction hypothesis on any structurally smaller term. The checks needed to make sure that an inductive proof is valid are exactly what is needed to termination-check a function.

To summarize, while having a way to prove programs terminate is nice, having implications and inductive proofs is crucial. In some language designs these would be two separate things. For example, in F* [133] types and propositional formulas are two syntactically separate categories, one of which is inhabited by program terms and the other automatically proved by an SMT solver. One of the benefits of making Zombie core a full-spectrum dependent language with unified syntax is that it avoids this duplication: by adding induction to the logic we get termination-checked programs for free.

7.1.2 Performance

The other reason to consider termination checking is to enable erasure. In the words of Randy Pollack, the point of writing a proof in a strongly normalizing calculus is that you don't need to normalize it. For example, in Section 3.3 we mentioned that a function like

```
safediv : Nat → (y:Nat) → (p: isZero y = false) → Nat
```

is still type safe if we call it with an infinite loop as the proof of the precondition; the expression `safediv 3 0 (loop())` simply diverges. However, that means that we actually keep the function argument in the compiled code and execute it at runtime. In the case of `safediv` this is doubly unsatisfying: first because we know that the implementation of `safediv` just throws away that argument, and second because it is an equation, so it can in any case only evaluate to the uninformative value `join`.

For function preconditions like these, we only need to know that the expression has *some* value, but it doesn't matter which one. Full Zombie handles this gracefully. We mark the precondition as an erased argument:

```
safediv : Nat → (y:Nat) → [p: isZero y = false] → Nat
```

Now, as long as proof `a` is known-terminating, Zombie allows it to be used as an implicit argument `safediv 3 0 [a]`. In this way, there is no trace of it at runtime.

The function `match` illustrates the dual property. In the previous subsection we gave it the type

```
match : (s:String) → (r:Regexp) → Maybe (Matches s r)
```

but in the case when it returns `Just m`, this requires constructing (and allocating memory for) an explicit witness `m`. The clients to `match` never care about the specific value returned, only that there exists *some* value witnessing the postcondition. For regular expression matching the overhead is perhaps tolerable, but as we mentioned above, explicit witnesses for propositional unsatisfiability can easily be hundreds of megabytes in size. Again, full Zombie handles this gracefully. We define a different datatype `EMaybe` where the argument is erased, and as long as `m` is known-terminating `match` can return `EJust [m]`, which erases to just a unit value at runtime.

Erasability in this sense is necessary for any practical dependently typed language. The precise design of erasure in Zombie is slightly ambitious because it internalizes information about erasiblity into the equational theory by making expressions which only differ in erased positions provably equal; in the terminology of Abel [4] this is "internal erasure". This feature is not available in e.g. Coq without axioms (see Section 8.1). But even Coq provides "external erasure": when using Coq's program extraction feature to compile a function, all proof arguments of sort Prop are erased from the runtime representation. Even just external erasure requires termination-checking.

On the other hand, Coq and Agda do not erase expressions when evaluating them at type-checking time (in order to ensure strong normalization of open terms). And practical experience with Coq and Agda illustrates how crucial proof erasure is for performance! For example, it is folklore that any dependently typed formalization of category theory will grind to a halt about halfway through, when the typechecker runs out of memory. The most natural way to phrase the definitions involves many type indices and Σ-types, both of which can cause the size of goals and proofs to blow up. Gross et al. [65] describe how to avoid some of the pitfalls through carefully designing the definitions and judiciously using abstraction and opaqueness. They also mention that extending Coq's definitional equality with irrelevant arguments would be very helpful, because Coq could then judge types equal without having to process the (large) proof terms embedded inside the type.

7.2 Design choices

Before moving on to the subsetted calculus, we briefly discuss some of the distinctive features of the full Zombie language. Every expression in Zombie is classified as either known-terminating or possibly nonterminating. Formally, the typing judgement is indexed by an additional parameter θ, which can be either L (for "logical", i.e. terminating) of P ("program").

$$\Gamma \vdash^{\theta} a : A$$

When θ is L the expression a is known to be total, so it is safe to erase, and by the Curry-Howard correspondence it can be read as a proof of the theorem A. When θ is P, a may make unrestricted use of nonterminating features. In the concrete syntax, the programmer marks definitions as either `log` or `prog`.

Allowing definitions to be either terminating or not is not so unusual. Both Idris and recent versions of Agda allows the programmer to write generally recursive functions, and the typechecker tracks which expressions used general recursion so that it can avoid reducing possibly-nonterminating functions during typechecking. However, the treatment of termination in Zombie has two features which sets it apart from mainstream dependently typed languages.

7.2.1 Type-based termination

The first difference is that the θ classifier is specified as part of the typing rules. Coq, Agda, and Idris (and the paper presentation of Martin-Löf type theory) treat type checking and termination as two separate processes. In these systems each function is first type-checked according to a set of typing rules which do not mention termination, and then there is a separate check that all recursive function calls happen at a smaller argument.

In Zombie, by contrast, the typing rules themselves enforce that recursion is structural. We can see this by comparing the typing rules for general recursive functions rec and structurally recursive functions ind:

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} (y{:}A) \to B : \mathsf{Type} \\ \Gamma, y :^{\mathsf{P}} A, f :^{\mathsf{P}} (y{:}A) \to B \vdash^{\theta} a : B\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\, y.a : (y{:}A) \to B}\ \textsc{TRec}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} (y{:}A) \to B : \mathsf{Type} \\ \Gamma, y :^{\mathsf{L}} A, f :^{\mathsf{L}} (x{:}A) \to \bullet(z{:}x \prec y) \to \{x/y\}B \vdash^{\mathsf{L}} a : B\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ind}\, f\, y.a : (y{:}A) \to B}\ \textsc{TInd}$$

The rule TRec is the ordinary rule for recursive functions, except that it specifies that the expression lives in P, because general recursion can be used to write infinite loops.

On the other hand, structurally recursive functions can safely be classified as L (by subtyping, they can be used at P also). But in order to enforce structural recursion, the recursive variable f is given a type which expects an extra argument, a *proof* that the size of y is smaller than x. Like equality types, $a \prec b$ is a primitive type in Zombie. It is introduced by two constructors, ord and ordtrans.

$$\frac{\Gamma \vdash^{\mathsf{L}} b : a = d\, \overline{A_i}\, \overline{a_i}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ord}\, b : a_i \prec a}\ \textsc{TOrd}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a : a_1 \prec a_2 \\ \Gamma \vdash^{\mathsf{L}} b : a_2 \prec a_3\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ordtrans}\, a\, b : a_1 \prec a_3}\ \textsc{TOrdtrans}$$

The rule TOrd says that the direct arguments to a data constructor are smaller than the entire constructor value. For example we can show that a natural number is smaller than its successor, $\mathsf{ord}\,\mathsf{join} : a \prec \mathsf{Suc}\, a$, because $\mathsf{join} : \mathsf{Suc}\, a = \mathsf{Suc}\, a$. TOrdtrans allows chaining together several $\prec$-judgements, which is useful e.g. for recursing on a value from a deep pattern match. The rule TOrd is written in terms of an equation in order to take advantage of the equation that the typing rule for case introduces. For example, we can write a function to add natural numbers as follows:

```
log add : Nat → Nat → Nat
ind add x y = case x [eq] of
   Zero   → y                  -- eq : x = Zero
   Suc x’ → add x’ [ord eq] y -- eq : x = Suc x’
```

This idea of integrating termination checking into the type system is known as *type-based termination*, and has been the subject of much research [2, 18, 23, 145]. Most existing systems employ "size types", in which each type is annotated by an ordinal-like size expression, which should decrease in recursive calls. Although they have not yet been adopted in a production language, size-types have been explored in the (experimental) MiniAgda system [3].

Type-based termination offers two advantages over a separate syntactic check. First, types are inherently much more resilient to program transformations and refactorings. Second, because the condition that the termination metric is decreasing is reified as an explicit proof term, the programmer is provided by an "escape hatch" to write terminating functions even if the termination proof is too subtle for the typechecker to find automatically. For example, consider division of natural numbers: to compute n/m, we make a recursive call on $(n-m)/m$. The term $n-m$ is not an immediate subterm of n. In the Zombie test cases we use the fact that $a \prec b$ is a first-class type to prove a lemma saying that the subtraction produces structurally smaller terms:

```
log minus_smaller :
  (x:Nat) → (y:Nat) → (eq x Zero = False) → (eq y Zero = False)
  → (minus x y ≺ x)
```

Coq and Agda satisfy the need to provide explicit proofs by separate libraries for well-founded induction. Functions written using the wellfounded recursion combinator look similar to Zombie's ind-functions—each recursive call takes a proof that the termination metric decreased—but the operational semantics are more complicated (if you reduce a function defined by well-founded recursion, you get a more complicated expression involving helper-functions from the library). So in order to reason about such recursive functions, the programmer instead uses library lemmas which provide the recursion equation as a propositional equation [17]. Because Zombie's version of structural recursion is a primitive feature, the ordinary features for reasoning about reduction (`join` and `unfold`) work even when the structural ordering proofs use additional lemmas.

Logical strength

While Zombie is expressive in that it exposes the ordering $a \prec b$ as a first-class type, it is quite limited in what ordering relations are actually provable. The $\prec$-relation is entirely generated by direct syntactic subarguments of data constructors. For a given datatype D, we can consider the function size : $D \to$ Nat, which recursively counts the number of constructors in a data value. The function size is monotone with respect to the $\prec$-order. So intuitively, the only functions that are definiable using ind are those that are terminating using a Nat-valued termination metric.

We can convert the intuition to a more formal claim: if we restrict the rule Tind to only allow $A \equiv$ Nat, the same functions are definable, i.e. there exists a way to encode programs from the full language into the restricted one. A first try, suggested by the previous paragraph, would be to first define a size-function for A, and then use an induction on the size of argument to the function, instead of on the structure of the argument. That does not quite work, because in the restricted system there is no way to write a terminating size-function. However, there is a simple trick to get around that: index the datatype itself by its size. For example, we can replace a function acting on the ordinary list datatype

```
data List (a : Type) : Type where
  Nil
  Cons of (x : a) (xs : List a)

map : [a b:Type] → (f:a → b) → List a → List b
map [a] [b] f = ind map xs =
  case xs [xs_eq] of
    Nil → Nil
    Cons x xs' → Cons (f x) (map xs' [ord xs_eq])
```

with a version that explicitly indexes the lists by their length, and recurses on the size index:

```
data List (a : Type) (n : Nat) : Type where
  Nil
  Cons of [m : Nat] [n = Succ m] (x : a) (xs : List a m)

map : [a b:Type] → (f:a → b) → [n:Nat] → List a n → List b n
map [a] [b] f = ind map [n] = λxs.
  case xs [xs_eq] of
    Nil → Nil
    Cons [m] [pf] x xs' → Cons [m] [pf] (f x) (map [m] [ord pf] xs')
```

Since course-of-values induction is provable from ordinary natural number induction, we could also without loss of generality omit the rule TORDTRANS, and only allow recursive calls on the immediate predecessor of a natural number.

Compared to other languages, the restriction to nat-valued termination metrics seems quite severe. For example, ACL2 explicitly uses an ordinal notation for its termination metrics. And the Coq and Agda termination checkers also use a more liberal notion of subterm. In Coq we can define the following datatype of infinitely-branching trees:

```
Inductive tree :=
   leaf : tree
 | branch : (nat → tree) → tree.
```

If we squint, we can view this as saying that a tree is either a leaf, or a branch with countably infinitely many children. Coq will consider e.g. `ts 42` to be a subterm of the tree `(branch ts)`, which allows functions defined by (transfinite) recursion on such trees even though the trees do not have a finite size. This feature is used in e.g.

Werner's axiomatization of ZFC set theory [143], which defines sets as well-founded trees, and it is also used by the well-founded recursion libraries.

The inability to do this kind of transfinite recursion in Zombie is a limitation, and remedying this would be a worthwhile line of future work. Doing so will require some thought: currently the typing rules are pleasantly simple because there is only a single type contructor $a \prec b$, which (like equality) is allowed to be heterogenously typed. Adding more proof-theoretically powerful induction principles may require being more careful about types, and also we would probably want to consider multiple different orderinga of a given type (e.g. different lexicographic orders of a product type). Furthermore, powerful induction principles make the metatheory more difficult (see e.g. Barthe et al. [18]), and our normalization proofs so far have only contemplated natural-number induction.

That said, the proof-strength of Zombie already goes quite far. In addition to just plain induction on a size, it is also possible (if somewhat clumsy) to do lexicographic induction by nesting several ind-expressions. (The same is true in Coq.) For example, we can define the Ackermann function as follows:

```
ack : (m:Nat) → (n:Nat) → Nat
ack = ind a m =
  case m [meq] of
    Zero    → λ n. Succ n
    Succ m' → ind am n =
      case n [neq] of
        Zero    → a m' [ord meq] 1
        Succ n' → a m' [ord meq] (am n' [ord neq])
```

Having to write the function as a nested function definition makes the program harder to read. Agda's termination checker is more programmer friendly, because for multiple-argument functions it automatically searches for a lexicographic ordering of the arguments. But replacing such multiple-argument functions with nested functions is still a fairly benign transformation: it can be done locally, function by function, and it preserves reduction behavior.[14]

One way to get a sense of whether the more powerful forms of induction are essential or not is to look at how Coq's well-founded induction feature get used in the Coq

[14] This is in contrast to more heavy-weight encodings of terminating functions. Using higher-order functions and recursion on natural numbers, one can write functions which require difficult termination metrics by using a Church-encoding of ordinal numbers. Using only simple types (as in Gödel's System T) one can encode any ordinal in in the series $\omega, \omega^{\omega}, \omega^{\omega^{\omega}}, \ldots$, and using dependent types one can go much higher still [51]. So with this limited set of features, it is already possible to write down almost any function imaginable. However, this has the usual drawbacks of Church encodings: while the functions has the right computational behavior, the function definitions do not look natural, and lack induction principles [56]. So while these constructions shed light on the proof-theoretic strength of dependent type theory, they are not useful for practical programming.

User Contributions.[15] This is a collection of 161 proof developments written in Coq, comprising 1.7 million lines of code. Out of these developments, 32 make use of the `well_founded_ind` recursion combinator to write recursive functions. So those developments are examples where Coq's built-in termination checker is too limiting. However, out of those 32:

23 use wellfounded induction either for strong induction over a nat-valued termination metric, or induction over a lexicographic product of nat-valued metrics. As indicated above Zombie directly supports strong induction, and lexicographic induction is possible in principle (but inconvenient).

2 developments take advantage of the fact that proofs of wellfoundedness in Coq are first-class objects that can be manipulated. `CompCert/lib/UnionFind.v` represents a union-find state as a pair of two elements: the pointer graph, and a proof that it is well-founded. The union operation has to update the proof component of the pair when it adds new arcs to the graph. Similarly, `ATBR` (a library for Kleene algebras, and an automata-based decision procedure) represents an automaton with ϵ-transitions as a transition relation alongside a proof that all chains of ϵ-transitions are wellfounded, and the automata constructions explicitly manipulate the wellfoundedness proofs.

These are elegant software engineering approaches to verifying the respective algorithms, but it would also be possible to write the same algorithms in a different way. For example `PersistentUnionFind` (a different contribution) shows how to implement Union-Find in terms of a nat-valued "pointer distance" relation.

1 development, `CompCert/common/Smallstep.v`, specifies lemmas about simulations of transition relations in terms of wellfoundedness. I am not certain what order is used when the lemma is eventually invoked.

4 developments prove theorems where the statement itself refers to wellfounded relations. `HoareTut` and `Random` prove total-correctness rules Hoare-logic rules for while loops, so the premises of the theorem states "for any termination metric…". Similarly, `WeakUpTo` proves the correctness of a proof technique for weak bisimilarity, under the assumption that the system is terminating (does not exhibit infinite sequences of silent transitions). And `ZornsLemma` proves a characterization of wellfoundedness itself, in terms of minimal elements. So in these cases, the theorem could not be stated in its general form without a notion of wellfounded order, but the developments do not instantiate the theorems with any concrete orderings.

3 developments directly make use of well-founded orders which are "bigger" than finite lexicographic products of ω. `CoLoR/HORPO` verifies termination of rewrite

[15] `svn://scm.gforge.inria.fr/svn/coq-contribs/branches/v8.4`

> systems that satisfy a “higher-order recursive path-ordering” condition, and the proofs use the multi-set order. `Buchberger` implements Buchberger’s algorithm for constructing Gröbner bases of sets of polynomials. In order to prove termination it uses the lexicographic exponential order on lists of ordered elements (in this case, polynomial coefficients). And finally, `Cantor` implements data structures for ordinals less than Γ_0, using Cantor and Veblen normal forms, which unsurprisingly needs fancy termination metrics.

From this we can conclude that “big” orderings (beyond ω^n for finite n) are used quite rarely. Even for proofs which would not in principle need a big ordering, it can be convenient to have a first-class notion of well-founded orders. But most of the developments either do not need anything except plain structural recursion, or can be done using just strong induction and lexicographic induction. So although support for more advanced termination arguments would not hurt, the features in Zombie as it exists today are not unrealistically weak—they suffice for most proofs.

7.2.2 Internalizing the termination classifier

The second way that Zombie differs from the built-in termination checking of Agda and Idris is that it reflects the termination of functions in their types.

In Idris, every function is simply classified as either terminating or not, similar to the classifier θ on the Zombie typing judgement. Arguments to higher-order functions in Idris are assumed to be terminating, but if a function is applied to a possibly diverging argument, then the application is considered possibly diverging also. Similarly, values of record type are considered possibly diverging unless all the components of the record are known to terminate. So there is no way to abstract over a mix of terminating and nonterminating components.

Zombie makes more fine-grained distinctions, because termination-classification can be expressed by a *type* which we write $A@\theta$. A value belongs to the type $A@\theta$ if it can be typed at type A and classifier θ. This way one can write functions which abstract over values of a known termination status. For example, we can write a function `composeP`, which composes nonterminating functions:

```
log composeP : [A B C : Type] →
  (B→C @prog) → (A→B @prog) → (A→C @prog)
composeP [A] [B] [C] f g = (λ x . f (g x))
```

Even though the arguments to `composeP` may not terminate when called, the function composition operator itself is known to terminate. Similarly we can accept and return datatypes (e.g. dependent pairs) which mix logical and programmatic values.

Being able to mix terminating and potentially nonterminating code is important for lightweight verification, because if a programmer elects to not spend effort proving

$$
\begin{array}{lll}
a,\ b,\ A,\ B & ::= & \mathsf{Type} \mid (x\!:\!A) \to B \mid a = b \\
 & \mid & \mathsf{Nat} \mid \Sigma x\!:\!A.B \mid A@\theta \\
 & \mid & x \mid \lambda x.a \mid \mathsf{rec}\ f\ x.a \mid \mathsf{ind}\,f\ x.t \mid a\ b \\
 & \mid & \mathsf{join} \mid \langle a, b\rangle \mid \mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} \\
 & \mid & 0 \mid \mathsf{Suc}\,a \mid \mathsf{ncase}_z\ a\ \mathsf{of}\ \{0 \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \\
 & & \\
v & ::= & \mathsf{Type} \mid (x\!:\!A) \to B \mid a = b \\
 & \mid & \mathsf{Nat} \mid \Sigma x\!:\!A.B \mid A@\theta \\
 & \mid & x \mid \lambda x.a \mid \mathsf{rec}\ f\ x.a \mid \mathsf{ind}\ f\ x.a \mid \mathsf{join} \\
 & \mid & \langle v_1, v_2\rangle \mid 0 \mid \mathsf{Suc}\,v
\end{array}
$$

Figure 7.2: Expressions and values

termination of some functions, they need a fine-grained way to specify what is and is not known terminating. A good example of this is again a SAT-solver. As noted above, it makes sense to not prove that the solver itself always terminates (for advanced solving algorithms, proving termination is quite subtle). But on the other hand, we need to be able to trust the proof of unsatisfiability that it returns. In Zombie we can assign it a type like the following:

```
prog solver :
  (f : Formula) → Either (Σ(v:Assignment). eval f v = true)
                          ((v:Assignment) → eval f v = false @log)
```

The marker `prog` expresses that the function `solver` itself may not return (it is checked in the P fragment). But if it returns a value `Right pf`, the marker `@log` ensures that value `pf` has been checked in the L fragment, and therefore is a trustworthy proof.

7.3 Core calculus: Non-termination as a possible world

To make the foregoing ideas precise, we now define a type system for a small subset of the full Zombie language. The syntax of expressions and values are shown in Figure 7.2. Compared to the core calculus in Chapter 3 there are few surprises. We still combine terms and types into a single syntactic category, and use the sort Type to see which are which. The types include function types $(x\!:\!A) \to B$ and equations $a = b$, but instead of general datatypes it has only Nat and dependent pair types $\Sigma x : A.B$. Also, we omit computational irrelevance. Finally, we add the type $A@\theta$ which internalizes the termination classifier.

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \cdot}\ \textsc{CNil} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}}{\vdash \Gamma, x :^{\theta} A}\ \textsc{CType}$$

$\boxed{\Gamma \vdash^{\theta} a : A}$

$$\frac{\begin{array}{l}(x :^{\theta} A) \in \Gamma \quad \vdash \Gamma \\ \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}\end{array}}{\Gamma \vdash^{\theta} x : A}\ \textsc{TVar} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type} \quad \mathsf{Mobile}\,(A) \\ \Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{L}} (x\!:\!A) \to B : \mathsf{Type}}\ \textsc{TArr}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} b : (x\!:\!A) \to B \\ \Gamma \vdash^{\theta} a : A \quad \Gamma \vdash^{\mathsf{L}} \{a/x\}B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\theta} b\ a : \{a/x\}B}\ \textsc{TApp} \qquad \frac{\begin{array}{l}\Gamma, x :^{\theta} A \vdash^{\theta} b : B \\ \Gamma \vdash^{\mathsf{L}} (x\!:\!A) \to B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\theta} \lambda x.a : (x\!:\!A) \to B}\ \textsc{TLam}$$

$$\frac{\begin{array}{l}\Gamma, f :^{\mathsf{P}} (x\!:\!A) \to B, x :^{\mathsf{P}} A \vdash^{\mathsf{P}} b : B \\ \Gamma \vdash^{\mathsf{L}} (x\!:\!A) \to B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\ x.a : (x\!:\!A) \to B}\ \textsc{TRec}$$

$$\frac{\begin{array}{l}\Gamma, x :^{\mathsf{L}} \mathsf{Nat}, f :^{\mathsf{L}} (y\!:\!\mathsf{Nat}) \to (z\!:\!\mathsf{Suc}\, y = x) \to B \vdash^{\mathsf{L}} b : B \\ \Gamma \vdash^{\mathsf{L}} (x\!:\!\mathsf{Nat}) \to B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{ind}\, f\ x.b : (x\!:\!\mathsf{Nat}) \to B}\ \textsc{TInd}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta_1} a : A \quad \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type} \\ \Gamma \vdash^{\theta_2} b : B \quad \Gamma \vdash^{\mathsf{L}} B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{L}} a = b : \mathsf{Type}}\ \textsc{TEq} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \\ \Gamma \vdash^{\theta} a : \{b_1/x\}A \\ \Gamma \vdash^{\mathsf{L}} \{b_2/x\}A : \mathsf{Type}\end{array}}{\Gamma \vdash^{\theta} a : \{b_2/x\}A}\ \textsc{TConv}$$

$$\frac{\begin{array}{l}a \leadsto_p^{*} c \quad b \leadsto_p^{*} c \\ \Gamma \vdash^{\mathsf{L}} (a = b) : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{join} : a = b}\ \textsc{TJoin}$$

Figure 7.3: Typing: variables, functions, and equations

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\dfrac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} 0 : \mathsf{Nat}}\textsc{TZero} \qquad \dfrac{\Gamma \vdash^{\theta} a : \mathsf{Nat}}{\Gamma \vdash^{\theta} \mathsf{Suc}\, a : \mathsf{Nat}}\textsc{TSuc}$$

$$\dfrac{\begin{array}{l}\Gamma \vdash^{\theta} a : \mathsf{Nat}\\ \Gamma, z :^{\mathsf{L}} 0 = a \vdash^{\theta} b_1 : B\\ \Gamma, x :^{\theta} \mathsf{Nat}, z :^{\mathsf{L}} (\mathsf{Suc}\, x) = a \vdash^{\theta} b_2 : B\\ \Gamma \vdash^{\mathsf{L}} B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\theta} \mathsf{ncase}_z\ a\ \mathsf{of}\ \{0 \Rightarrow b_1; \mathsf{S}\ x \Rightarrow b_2\} : B}\textsc{TNCase}$$

$$\dfrac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type} \quad \mathsf{Mobile}\,(A)\\ \Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{L}} \Sigma x\!:\!A.B : \mathsf{Type}}\textsc{TSigma} \qquad \dfrac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} \Sigma x\!:\!A.B : \mathsf{Type}\\ \Gamma \vdash^{\theta} a : A\\ \Gamma \vdash^{\theta} b : \{a/x\}B\\ \Gamma \vdash^{\mathsf{L}} \{a/x\}B : \mathsf{Type}\end{array}}{\Gamma \vdash^{\theta} \langle a, b\rangle : \Sigma x\!:\!A.B}\textsc{TPair}$$

$$\dfrac{\begin{array}{l}\Gamma \vdash^{\theta} a : \Sigma x\!:\!A_1.A_2 \quad \Gamma \vdash^{\mathsf{L}} B : \mathsf{Type}\\ \Gamma, x :^{\theta} A_1, y :^{\theta} A_2, z :^{\mathsf{L}} \langle x, y\rangle = a \vdash^{\theta} b : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} : B}\textsc{TPCase}$$

Figure 7.4: Typing: datatypes

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\dfrac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A}\textsc{TSub} \qquad \dfrac{\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}}{\Gamma \vdash^{\mathsf{L}} A@\theta : \mathsf{Type}}\textsc{TAt}$$

$$\dfrac{\Gamma \vdash^{\theta} v : A@\theta'}{\Gamma \vdash^{\theta'} v : A}\textsc{TUnboxVal} \qquad \dfrac{\begin{array}{l}\Gamma \vdash^{\theta} a : A\\ \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{P}} a : A@\theta}\textsc{TBoxP}$$

$$\dfrac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a : A\\ \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{L}} a : A@\theta}\textsc{TBoxL} \qquad \dfrac{\begin{array}{l}\Gamma \vdash^{\mathsf{P}} v : A\\ \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\textsc{TBoxLV}$$

Figure 7.5: Typing: subsumption and internalized consistency classification

$$\boxed{\mathsf{Mobile}\,(A)}$$

$$\frac{}{\mathsf{Mobile}\,(\mathsf{Nat})}\textsc{MobileNat} \qquad \frac{\mathsf{Mobile}\,(A) \quad \mathsf{Mobile}\,(B)}{\mathsf{Mobile}\,(\Sigma x\!:\!A.B)}\textsc{MobileSigma}$$

$$\frac{}{\mathsf{Mobile}\,(a = b)}\textsc{MobileEq} \qquad \frac{}{\mathsf{Mobile}\,(A@\theta)}\textsc{MobileAt}$$

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type} \quad \mathsf{Mobile}\,(A)}{\Gamma \vdash^{\mathsf{L}} v : A}\textsc{TMobileVal}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\theta} a : (\Sigma x\!:\!A_1.A_2)@\theta' \quad \Gamma \vdash^{\mathsf{L}} B : \mathsf{Type} \\ \Gamma, x :^{\theta'} A_1, y :^{\theta'} A_2, z :^{\mathsf{L}} \langle x, y\rangle = a \vdash^{\theta} b : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} : B}\textsc{TPCase'}$$

Figure 7.6: Typing: Mobile types and cross-fragment case expressions

The expressions are also mostly familiar, except there are now three different forms of function definitions: nonrecursive functions $\lambda x.a$, generally recursive functions $\mathsf{rec}\ f\,x.a$, and structurally recursive functions $\mathsf{ind}\ f\,x.a$. The remaining forms for expressions introduce and eliminate equations, pairs, and natural numbers.

Since this chapter is only concerned with type safety and logical soundness, not the design of the typechecking algorithm, we do not include any annotations in this language. In other words, this syntax describes *erased* expressions.

The typing rules for functions and equations are shown in figure 7.3. The typing judgement has the form $\Gamma \vdash^{\theta} a : A$, where contexts are lists of assumptions about the types of variables.

$$\Gamma ::= \cdot \mid \Gamma, x :^{\theta} A$$

Each variable in the context is tagged with θ to indicate its fragment, and this tag is checked in the TVar typing rule. (The assumption $\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}$ is redundant, since it is implied by $\vdash \Gamma$, but it simplied the metatheory slightly).

A type is well-kinded if $\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}$ (the consistency classifier is only interesting when checking terms; types are always checked at L). The kinding rule for arrows (TArr) is mostly the usual rule for dependent rule for dependent arrow types. However, it has an extra premise $\mathsf{Mobile}\,(A)$, which we will discuss in Section 7.3.4.

Next, the typing rules for function definitions track whether the function can diverge or not. Non-recursive lambda-expressions (TLam) do not themselves introduce nontermination, they will terminate as long as the function body only calls terminating functions, so the rules works parametrically over any θ. Generally recursive functions (TREC) may diverge, so the rule specifies P. Finally, the rule TIND is a simplified version of the rule for structural recursion in full Zombie (which we discussed in Section 7.2.1). It is restricted to only define recursion on natural numbers, and instead of a general structural subterm ordering, the recursive call must be on exactly the predecessor of x. As we discussed above this simplified version captures most of the expressivity of the full rule.

The next few rules support equational reasoning. The kinding rule for equations (TEQ) merely requires that the two expressions check at *some* θ_1 and θ_2. So just as in Chapter 3, the introduction rule TJOIN can prove two expressions a and b equal even if they diverge, as long as they have some common reduct c.

In this small calculus we add a restriction that A and B in TEQ have to be types, i.e. equations can only be between terms. This rules out equations between types, such as $((\lambda x.x \to x)\ \mathsf{Nat}) = (\mathsf{Nat} \to \mathsf{Nat})$. Similarly there are restrictions on the kinding rule for arrows (the domain and range has to be types, as opposed to kinds), and on the rules for case-expressions (they can produce terms but not types), which means that there are no interesting type-level equations to prove. Full Zombie lifts these restrictions using a predicative hierachy Type_ℓ to allow type-level computation.

Finally, the elimination rule for equations, TCONV, shows the payoff for tracking termination. Whereas the calculus in Chapter 3 required the erased proof b to be a value, in this version of the rule it is allowed to be any expression as long as it checks at L.

Figure 7.4 shows the rules for natural numbers and pairs. Except the premise $\mathsf{Mobile}\,(A)$ in the kinding rule for pair types, there are no surprises: the rules are exactly the general datatype rules from Chapter 3, specialized to these two types. As before, the case-rules bind an extra equation z in each branch.

7.3.1 Operational semantics

The operational semantics are almost identical to the semantics for the core language in Chapter 3 (Figure 3.4), so we do not explicitly show them here. The only new thing we need to discuss is how the to reduce ind-functions.

The recursive variable f in the rule TIND has a different type than the entire function (it takes an extra proof that the argument is getting smaller), so unlike the rule for rec-functions we cannot simply substitute the function definition into the body. Instead we substitute the function $\lambda y.\lambda z.(\mathsf{ind}\ f\ x.a)\ y$, which takes an extra argument z (the ordering proof), and then discards it:

$$\frac{v \rightsquigarrow_p v' \qquad a \rightsquigarrow_p a'}{(\mathsf{ind}\ f\ x.a)\ v \rightsquigarrow_p \{v'/x\}\{\lambda y.\lambda z.(\mathsf{ind}\ f\ x.a')\ y/f\}a'}\textsc{PInd}$$

7.3.2 Subsumption

The interesting part of termination-tracking in Zombie is that a single program can mix logical and programmatic expressions.

First, every logical expression can be safely used programmatically. We reflect this fact into the type system by the rule TSub (in Figure 7.5), which says that if a term a type checks logically, then it also checks programmatically. For example, a logical term can always be supplied to a function expecting a programmatic argument. This rule is useful to avoid code duplication. A function defined in the logical fragment can be used without penalty in the programmatic fragment.

7.3.3 Internalized termination classifier

To provide a general mechanism for logical expressions to appear in programs and programmatic values to appear in proofs, we introduce a type that internalizes the typing judgement, written $A@\theta$. Nonterminating programs can take logical proofs as preconditions (with functions of type $(x : A@\mathsf{L}) \to B$), return them as postconditions (with functions of type $(x : A) \to (B@\mathsf{L})$), and store them in data structures (with pairs of type $\Sigma x : A.(B@\mathsf{L})$). At the same time, logical lemmas can use @ to manipulate values from the programmatic fragment.

Intuitively, an expression a has type $A@\theta$ if it can be given the type A in the θ fragment. If the expression is a value, this is an exact equivalence:

$$\Gamma \vdash^{\theta_1} v : A@\theta_2 \qquad \text{iff} \qquad \Gamma \vdash^{\theta_2} v : A$$

For non-values we need to be slightly more careful, in order to maintain the property that any expression that checks at L normalizes. To ensure this we add value restrictions to some of the rules.

The introduction and elimination rules are shown in Figure 7.5. When introducing an @-type, the programmatic fragment can internalize any typing judgement (TBoxP), but in the logical fragment (TBoxL and TBoxLV) we sometimes need a restriction to ensure termination. Therefore, rule TBoxLV only applies when the subject of the typing rule is a value. (The rule TBoxL can introduce $A@\theta$ for any θ since logical terms are also programmatic.) Similarly, we can eliminate boxes freely if the expression is known to terminate (UnboxL) or if we are in the P fragment (UnboxP), but producing an expression in L from one in P is only allowed when the subject is a value (UnboxVal).

For example, a recursive function f can require an argument to be a proof by marking it $@\mathsf{L}$, forcing that argument to be checked in fragment L.

$$\dfrac{\Gamma \vdash^{\mathsf{P}} f : A@\mathsf{L} \to B \qquad \dfrac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A@\mathsf{L}}\,\textsc{TBoxP}}{\Gamma \vdash^{\mathsf{P}} f\ a : B}\,\textsc{TApp}$$

In the body of f the function argument is a variable (hence a syntactic value), so the rule UnboxVal applies and it can be used freely in the L fragment.

Similarly, a logical lemma g can be applied to a programmatic value by marking it $@\mathsf{P}$:

$$\dfrac{\Gamma \vdash^{\mathsf{L}} g : A@\mathsf{P} \to B \qquad \dfrac{\Gamma \vdash^{\mathsf{P}} v : A}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\,\textsc{TBoxLV}}{\Gamma \vdash^{\mathsf{L}} g\ v : B}\,\textsc{TApp}$$

Of course, g can only be defined in the logical fragment if it is careful to not use its argument in unsafe ways. For example, using TConv we can prove a lemma of type

```
    (n: Nat) → (f: (Nat → Nat)@P) → (f (plus n 0) = f n)
```

because reasoning about `f` does not require calling `f` at runtime.

Both introduction and elimination of @ is unmarked in the erased syntax, so the reduction behavior of an expression is unaffected by whether the type system deems it to be provably terminating or not.

7.3.4 Mobile types

The consistency classifier tracks which *expressions* are known to come from a normalizing language. For some types of *values*, however, the rules described so far can be unnecessarily conservative. For example, while a programmatic expression of type `Nat` may diverge, a programmatic value of that type is just a number, so we can treat it as if it were logical. On the other hand, we can not treat a programmatic function value as logical, since it might cause non-termination when applied.

The rule TMobileVal (Figure 7.6) allows values to be moved from the programmatic to the logical fragment. It relies on an auxiliary judgement $\mathsf{Mobile}\,(A)$. Intuitively, a type is mobile if the same set of values inhabit the type when $\theta = \mathsf{L}$ and when $\theta = \mathsf{P}$. In particular, these types do not include functions (though any type may be made mobile by tagging its fragment with @).

Concretely, the natural number type Nat is mobile, as is the primitive equality type (which is inhabited by the single value join). Any @-type is mobile, since it fixes a particular θ independent of the one on the typing judgement. Pair types are mobile if their component types are.

Even if a pair type is not mobile, it is always safe to do one level of pattern matching on one of its values, since such a value must start with the pair constructor. We reflect that in the rule TPCase', which generalizes TPCase from the previous section. This rule allows a scrutinee that type checks in one fragment θ' to be eliminated in another fragment θ. This lets the logical language reason by case analysis on programmatic values.

The mobile rule lets the programmer write simpler types, because mobile types never need to be tagged with logical classifiers. For example, without loss of generality we can give a function the type $(a = b) \to B$ instead of $((a = b)@\mathsf{L}) \to B$, since when needed, the body of the function can treat the argument as logical through TMobileVal. Similarly, multiple @'s have no effect beyond the innermost @ in a type. Values of type $A@\mathsf{P}@\mathsf{L}@\mathsf{P}@\mathsf{L}@\mathsf{P}$ can be used as if they had type $A@\mathsf{P}$.

In fact, the arguments to functions must always have mobile types. This restriction, enforced by rule TArr, means that higher-order functions must use @-types to specify which fragment their arguments belong to. For example, the type $(\mathsf{Nat} \to \mathsf{Nat}) \to A$ is not well-formed, so the programmer must choose either $((\mathsf{Nat} \to \mathsf{Nat})@\mathsf{L}) \to A$ or $((\mathsf{Nat} \to \mathsf{Nat})@\mathsf{P}) \to A$.

The reason that function arguments must be mobile is to account for contravariance. Through subsumption, we can introduce a function in the logical fragment and use it in the programmatic:

$$\dfrac{\dfrac{\Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} b : B}{\Gamma \vdash^{\mathsf{L}} (\lambda x.b) : (x : A) \to B}\,\textsc{TLam}}{\Gamma \vdash^{\mathsf{P}} (\lambda x.b) : (x : A) \to B}\,\textsc{TSub}$$

Here, the definition of b assumed x was logical, yet when the function is called it can be given a programmatic argument. For this derivation to be sound, we need to know that A means the same thing in the two fragments, which is exactly what $\mathsf{Mobile}\,(A)$ checks.

We also enforce the similar-looking condition that the first component of a Σ-type must be mobile. We do not know an example where this restriction is necessary for type safety, but it was convenient in the proofs, because this way the statements and proofs of lemmas about $\Sigma x : A.B$ look similar to corresponding lemmas about $(x : A) \to B$. More importantly, if we later wanted to extend the language to allow non-logical types (Section 7.7.2) this restriction would probably be necessary because A occurs contravariantly in the premise to the kinding rule TSigma. In any case, in full Zombie datatype definitions have a similar restriction for convenience reasons (Section 7.3.6), so the core calculus matches what the implementation is doing.

7.3.5 Subtyping

$$\boxed{A <: A'}$$

$$\frac{\theta \le \theta' \quad A <: A'}{A@\theta <: A'@\theta'}\textsc{SubAt} \qquad \frac{A <: A'}{A <: A'@\mathsf{P}}\textsc{SubAtP} \qquad \frac{A <: A'}{A@\mathsf{L} <: A'}\textsc{SubAtL}$$

$$\frac{\mathsf{Mobile}\,(A)}{A <: A@\theta}\textsc{SubMobile1} \qquad \frac{\mathsf{Mobile}\,(A)}{A@\theta <: A}\textsc{SubMobile2}$$

$$\frac{A' <: A \quad B <: B'}{(x\!:\!A) \to B <: (x\!:\!A') \to B'}\textsc{SubArr} \qquad \frac{B <: B'}{\Sigma x\!:\!A.B <: \Sigma x\!:\!A.B}\textsc{SubSigma}$$

$$\frac{}{(a = b) <: (a = b)}\textsc{SubEq} \qquad \frac{}{\mathsf{Nat} <: \mathsf{Nat}}\textsc{SubNat}$$

$$\boxed{A <:_{\mathsf{L}} A'}$$

$$\frac{A <: A'}{A <:_{\mathsf{L}} A'}\textsc{SubLSubtype} \qquad \frac{A <:_{\mathsf{L}} A'}{A <:_{\mathsf{L}} A'@\mathsf{L}}\textsc{SubLAtL} \qquad \frac{\begin{array}{c} A' <: A \\ B <:_{\mathsf{L}} B' \end{array}}{(x\!:\!A) \to B <:_{\mathsf{L}} (x\!:\!A') \to B'}\textsc{SubLArr}$$

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\begin{array}{l} \Gamma \vdash^{\theta} a : A \\ A <: A' \quad \Gamma \vdash^{\mathsf{L}} A' : \mathsf{Type} \end{array}}{\Gamma \vdash^{\theta} a : A'}\textsc{TSubtype} \qquad \frac{\begin{array}{l} \Gamma \vdash^{\mathsf{L}} a : A \\ A <:_{\mathsf{L}} A' \quad \Gamma \vdash^{\mathsf{L}} A' : \mathsf{Type} \end{array}}{\Gamma \vdash^{\mathsf{L}} a : A'}\textsc{TSubtypeL}$$

Figure 7.7: Typing: Subtyping

The subsumption rule TSub means that typing *judgements* at L are strictly better than at P. Further, the rules for @-types and mobile types mean that certain *types* are better than others. For example, any expression that checks at $A@\mathsf{L}$ also checks at $A@\mathsf{P}$:

$$\frac{\dfrac{\Gamma \vdash^{\mathsf{L}} a : A@\mathsf{L}}{\Gamma \vdash^{\mathsf{L}} a : A}\textsc{TUnboxL}}{\Gamma \vdash^{\mathsf{L}} a : A@\mathsf{P}}\textsc{TBoxL} \qquad \frac{\dfrac{\Gamma \vdash^{\mathsf{P}} a : A@\mathsf{L}}{\Gamma \vdash^{\mathsf{P}} a : A}\textsc{TUnboxP}}{\Gamma \vdash^{\mathsf{P}} a : A@\mathsf{P}}\textsc{TBoxP}$$

However, there are limits to what subsumption and unboxing can do. Given a function

$a : (x\!:\!A) \to (B@\mathsf{L})$, with the rules presented so far there is no way to give a the type $(x\!:\!A) \to (B@\mathsf{P})$. The best we can do is to η-expand the function definition:

$$\dfrac{\dfrac{\dfrac{\Gamma, x :^{\mathsf{L}} A \vdash^{\theta} a : (x\!:\!A) \to (B@\mathsf{L}) \quad \Gamma, x :^{\mathsf{L}} A \vdash^{\theta} x : A}{\Gamma, x :^{\mathsf{L}} A \vdash^{\theta} a\ x : B@\mathsf{L}}\ \textsc{TApp}}{\Gamma, x :^{\mathsf{L}} A \vdash^{\theta} a\ x : B@\mathsf{P}}\ \textsc{Unbox+Box}}{\Gamma \vdash^{\theta} \lambda x.a\ x : (x\!:\!A) \to (B@\mathsf{P})}\ \textsc{TLam}$$

Since the Zombie propositional equality does not include η-laws, modifying a program in this way could in turn require further adjustments elsewhere to make the types continue to line up. This is usually not a problem in practice, but it suggests that the type system is being unnecessarily fussy, by forcing the programmer to write one expression instead of different one with exactly the same operational behavior.

We remedy this situation by adding a subtyping relation to the language. The rule TSubtype (in Figure 7.7) allows an expression a to be used at any "worse" type. It relies on the auxilliary judgement $A <: A'$ which defines when a type is "better" than another type. The cases are mostly common-sense: $A@\mathsf{L}$ is better than $A@\mathsf{P}$ and better than A; A is better than $A@\mathsf{P}$; adding @-types to a mobile type A has no effect; and the subtyping rules extend underneath arrows and Σs.

In fact, even with this subtyping relation in place, the type system still makes some unneccessary distinctions. In particular, while it is possible to go from $\Gamma \vdash^{\mathsf{L}} a : A$ to $\Gamma \vdash^{\mathsf{L}} a : A@\mathsf{L}$ (by TBoxL), there is no rule that goes from $\Gamma \vdash^{\mathsf{L}} a : A \to B$ to $\Gamma \vdash^{\mathsf{L}} a : A \to (B@\mathsf{L})$. These two function types are equivalent, but only when checking a typing judgement at L. We can reflect this fact by adding a separate subtyping rule, TSubtypeL, which only applies in L contexts, and uses a more generous subtyping judgment $A <:_{\mathsf{L}} A'$.

These issues are mostly theoretical. The subtype judgements are valid semantically, and showing them gives a better understanding of what the @-types mean. But usually the programmer can avoid the need for them with rather small changes to the program text, e.g. η-expanding functions or adding an extra let-expression in order to make something a syntactic value. In fact, the Zombie implementation does not implement subtyping at all, and this has generally not been an issue when developing the example programs. (With one exception: as we describe in Section 7.7.3, the lack of something like the TSubMobile1/TSubMobile2 rules is quite annoying.)

7.3.6 Full Zombie: Polymorphism and Datatypes

The core calculus includes the crucial rules dealing with termination-checking, in the context of a simpler language than full Zombie. Most of the additional features of Zombie do not interact with the termination-checking rules. The main place where designing the Zombie type system required additional thought was deciding how the Mobile judgement should interact with the rest of the typing rules.

Polymorphism Unlike this simplified core calculus, full Zombie allows polymorphic functions. In a dependent calculus like this, polymorphism is particularly easy to add. One generalizes the sort Type to a hierarchy $\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 \ldots$ (types, kinds, superkinds, ...), and generalizes the arrow rule to allow arrows from kinds as well as from types:

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}_{\ell} \qquad \mathsf{Mobile}\,(A) \\ \Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} B : \mathsf{Type}_{\ell'}\end{array}}{\Gamma \vdash^{\mathsf{L}} (x\!:\!A) \to B : \mathsf{Type}_{\mathsf{max}(\ell,\ell')}}$$

So what about the mobile judgement? All the kinding rules in Zombie work at L; the only way to derive $\Gamma \vdash^{\mathsf{P}} A : \mathsf{Type}_{\ell}$ is to check A at L and then use subsumption. So the sort Type_{ℓ} classifies exactly the same values in L and in P. Accordingly, the sorts are considered mobile: $\mathsf{Mobile}(\mathsf{Type}_{\ell})$.

On the other hand, type variables (variables x : Type in the context) may eventually be instantiated to e.g. function types. Then they will classify a different set of values in L (terminating functions) and in P (any functions). So they are not considered mobile. Indeed, if they *were* considered mobile, we could define an obviously unsound function that coerces any value from P to L:

```
log bogus : (a : Type) → (a@prog) → (a@log)
bogus a x = x
```

(Here the body of the function would be checked by TUnboxVal, TMobileVal, TBoxL.)

The face that type variables are not mobile is a little awkward, because of the requirement that domains of arrow types must be mobile. So in the type of a polymorphic function like `map`, although the sort `Type` does not need an @-marker, the kinding rule forces us to mark not just function arguments, but also type variable arguments, with an explicit fragment:

```
map : [a b : Type] → (f : ((x:a@log) → b)@log) →
        (xs : List a) → List b
```

In practice, it does not matter very much that we picked `a@log` instead `a@prog`. If we want, we can apply `map` to a list of programmatic values by giving it an argument of type `List (T@prog)` and instantiating `a` with `T@prog`. Then the function being mapped should have type `(T@prog@log) → b`, which is essentially equivalent to `(T@prog) → b`.

In the Zombie implementation it is possible to hide some of the clutter by using a declaration `usually log` or `usually prog` which causes the implementation to insert a tag like @L whenever a non-mobile type occurs in a position where it is required to be mobile. For example, the type of composeP given in Section 7.2.2 is only valid if one implicitly inserts @prog for the function domains.

Datatypes The only datatypes in this small core calculus are natural numbers and dependent pairs. In prior work [31] we additionally treated sum types and iso-recursive types, the idea being that all inductive datatypes can be encoded using those features. In full Zombie, we instead treat datatypes as a primitive feature. Again, we should decide when a datatype is considered mobile. There are two options.

We could follow the style of Σ-types above, and compute whether a type is mobile or not based on whether its type arguments are. So for example, given a type definition

```
data List (a : Type) : Type where
  Nil
  Cons of (x : a) (xs : List a)
```

the type List T would be considered mobile exactly when T was mobile. Just like the second component of a Σ-type does not have to be mobile, there would be no requirement to tag the argument x to Cos with as P or L. On the other hand, this choice means that List a, where a is a type variable, would not be mobile. So the type of map given above would not we well-formed, and the argument would need to be marked e.g. @log.

Instead, we choose to make all datatypes mobile, by requiring the arguments to data constructors to be mobile (like the first components of Σ-types above). So in Zombie, in order to make the datatype declaration well-formed we need to tag the constructor argument x@log:

```
data List (a : Type) : Type where
  Nil
  Cons of (x : a@log) (xs : List a)
```

In return, the type List a is now mobile. This decision is mostly a matter of taste, but we found that on balance this style of declaration creates less clutter in the types. The difference would be more striking if, in future work, we allowed non-logical types (Section 7.7.2 below).

Type-level computation Finally, full Zombie includes type level computation. This again highlights a limitation in the Mobile (A) judgement. In Zombie this is defined completely syntactically, just by examining the form of A. In the examples that can be expressed in the limited core calculus that works very well. Once one adds type definitions, however, it can start to feel restrictive.

For instance, consider defining a less-than relation on natural numbers, `ltT m n`. We could do this in at least two ways. Either as an inductive datatype (similar to the `lt` type in the Coq standard library), or as a type definition referring to a boolean comparison function:

```
ltT : Nat → Nat → Type
ltT = λ  m n . (lt m n = True)
```

With the latter definition, `ltT n m` does not count as a mobile type, so it needs to be annotated with @s when used as a function argument type, even though it would be mobile if we inlined the definition. In other words, the $\mathsf{Mobile}\,(A)$ judgement does not respect β-equivalence.

One way to remedy this (in future work) would be to not check mobility syntactically, but instead track it in the type system. Maybe we should distinguish between sorts Type and $\mathsf{MobileType}$? This would also provide a way to write polymorphic functions quantifying over mobile types only, and thereby not have to specify $@\theta$ on type variables.

7.3.7 Previous publications

The calculus we described in this section is a small variation on another one, which was called λ^θ in Casinghino et al. [31] and LF^θ in Casinghino's thesis [30]. Compared to that system, the version in this chapter has been further simplified by removing some features which are not needed to illustrate the issues that we wanted to highlight. The changes are as follows:

- This version only has Σ-types, but omits sum- and recursive types. Without recursive types there is no need for type variables, so the well-formedness judgement $\vdash \Gamma$ only allows term variables.
- We add the premises $\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}$ and $\Gamma \vdash^{\mathsf{L}} B : \mathsf{Type}$ to the rule TEq which forbids equations between types (in any case without type-level functions there are not many interesting type-equalities). Without equations between types we do not need injectivity rules for type constructors (as in Section 3.9.4) for type safety.
- Like in the actual Zombie implementations, kinding assumptions have the form $\Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}$ instead of $\Gamma \vdash^{\theta} A : \mathsf{Type}$. (Without type-level computation this makes no difference.)
- We added a subtyping relation to the language.

Figure 7.8 shows schematically how the two variations of this calculus relate to the full set of features in Zombie. The calculus includes most of the novel typing rules

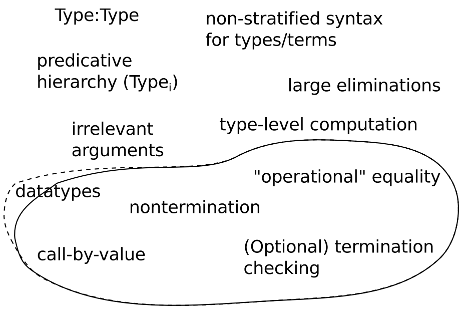

Figure 7.8: The previously published version of the calculus (dashed line) and the version in this chapter (solid line).

in Zombie (but not irrelevance), while it omits standard features which make the proof difficult. The biggest difference between the previous version and the one in this chapter is datatypes: Σ-, sum-, and recursive types together can encode general datatypes, while in this chapter we only show how to handle Σ-types.

7.4 Core calculus: Nontermination as an effect

The Zombie typing rules, and the smaller calculus in the previous subsection, were inspired by constructive versions of modal logic, which reason about statements whose truth varies in different "possible worlds".[16] In our case the two possible worlds are L and P. One could consider Zombie as two separate type systems, which happen to share the same program syntax, have mostly the same typing rules, and provide features for interoperability.

However, there is another way to think about nontermination. In that conception, there is a single language, and divergence is considered as an effect that can happen as a program executes, similar to raising an exception or printing output. This view is implicit in the long line of work, e.g. by Capretta [28], which treats nontermination in dependently typed languages as a monad.

If we think of nontermination as an effect, then the most natural way to formalize it is as a *type-and-effect system* [57]. By now the programming language community has arrived at a fairly standardized way to formulate effect systems for functional languages, parameterized over some abstract lattice of effects. In this section of the chapter, we instantiate this general approach in a small dependently typed calculus which tracks nontermination.

[16] Modal logic has previously been used to design type systems for distributed computation [69, 94]. In particular Zombie was inspired by ML5 [94], in which the typing judgement is indexed by what "world" (computer in a distributed system) a program is running on, and which includes a type $A@\theta$ internalizing that judgement.

$$
\begin{array}{rl}
T & ::= \mathsf{Nat} \mid (x\!:T) \overset{\theta}{\to} T' \mid \Sigma x\!:T.T' \mid t = t' \\
\\
t & ::= x \mid t\,t' \mid \lambda x.t \mid \langle t, t' \rangle \mid \mathsf{pcase}_z\ t\ \mathsf{of}\ \{(x,\ y) \Rightarrow t'\} \\
 & \mid\ \ 0 \mid \mathsf{Suc}\,t \mid \mathsf{join} \mid \mathsf{rec}\,f\ x.t \mid \mathsf{ind}\,f\ x.t \\
 & \mid\ \ \mathsf{ncase}_z\ t\ \mathsf{of}\ \{Z \Rightarrow t_1; \mathsf{S}\ x \Rightarrow t_2\} \\
\\
u & ::= x \mid \langle u_1, u_2 \rangle \mid 0 \mid \mathsf{Suc}\,u \\
 & \mid\ \ \lambda x.a \mid \mathsf{rec}\ f\ x.a \mid \mathsf{ind}\ f\ x.a \mid \mathsf{join} \mid \langle v_1, v_2 \rangle \mid 0 \mid \mathsf{Suc}\,v
\end{array}
$$

Figure 7.9: Effect-style calculus: Types, expressions, and values

The syntax of the calculus is shown in Figure 7.9. To keep things simple, we use separate syntactic categories for terms and types. Types include natural numbers, dependent functions $(x\!:T) \overset{\theta}{\to} T'$, dependent pairs $\Sigma x\!:T.T'$, and equations between terms. The difference compared to the possible-world-style calculus is that we no longer have @-types; instead each function type is tagged with a θ which is P if calling the function may cause nontermination.

The terms t in the system are exactly the same as the terms in the possible-world calculus, and their operational semantics are also the same. The types we assign them will be different, but since we are formalizing the erased version of the type system terms do not contain type annotations.

The typing rules are shown in Figures 7.10 and 7.11. To keep the two systems visually distinct, we write the effect-style typing judgement $\Gamma \vdash_\theta t : T$, with the θ as a subscript instead of a superscript, and the names of the effect-style typing rules include underscores.

The general principle is that the θ in the effect-style system tracks only effects from evaluating the particular expression in question, whereas the possible-world style system it also tracks what would happen if a client were to call that expression (when it is a function). We do not include an explicit subsumption rule in the type system, but the other rules are formulated so that this is admissible:

***Lemma* 29** (Subsumption)**.** If $\Gamma \vdash_{\mathsf{L}} t : T$ then $\Gamma \vdash_{\mathsf{P}} t : T$.

All the important differences between the possible-worlds and the effects calculus can be seen in the typing rules for variables, functions, and applications.

Variables only range over values, so referencing a variable in the context can never cause nontermination. So T_Var checks at any effect θ. This also means that there is no need to record an effect with variables in the context, so in this system contexts are just lists of types of variables:

$$\Gamma ::= \cdot \mid \Gamma, x : T$$

$\boxed{\Gamma \vdash T}$

$$\frac{\begin{array}{l}\Gamma \vdash T\\ \Gamma, x : T \vdash T'\end{array}}{\Gamma \vdash (x\!:T) \xrightarrow{\theta} T'}\textsc{k\_Arr} \qquad \frac{\begin{array}{l}\Gamma \vdash T\\ \Gamma, x : T \vdash T'\end{array}}{\Gamma \vdash \Sigma x\!:T.T'}\textsc{k\_Sigma} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Nat}}\textsc{k\_Nat} \qquad \frac{\begin{array}{l}\Gamma \vdash_\theta t : T\\ \Gamma \vdash_{\theta'} t' : T'\end{array}}{\Gamma \vdash t = t'}\textsc{k\_Eq}$$

$\boxed{\Gamma \vdash_\theta t : T}$

$$\frac{\begin{array}{l}(x : T) \in \Gamma \quad \vdash \Gamma\\ \Gamma \vdash T\end{array}}{\Gamma \vdash_\theta x : T}\textsc{T\_Var} \qquad \frac{\begin{array}{l}\Gamma \vdash_\theta t : (x\!:T') \xrightarrow{\theta'} T\\ \Gamma \vdash_\theta t' : T' \quad \theta' \le \theta\\ \Gamma \vdash \{t'/x\}T\end{array}}{\Gamma \vdash_\theta t\, t' : \{t'/x\}T}\textsc{T\_App} \qquad \frac{\begin{array}{c}\Gamma, x : T' \vdash_{\theta'} t : T\\ \Gamma \vdash (x\!:T') \xrightarrow{\theta'} T\end{array}}{\Gamma \vdash_\theta \lambda x.t : (x\!:T') \xrightarrow{\theta'} T}\textsc{T\_Lam}$$

$$\frac{\begin{array}{l}\Gamma, f : (x\!:T') \xrightarrow{\mathsf{P}} T, x : T' \vdash_{\mathsf{P}} t : T\\ \Gamma \vdash (x\!:T') \xrightarrow{\mathsf{P}} T\end{array}}{\Gamma \vdash_\theta \mathsf{rec}\, f\; x.t : (x\!:T') \xrightarrow{\mathsf{P}} T}\textsc{T\_Rec}$$

$$\frac{\begin{array}{l}\Gamma, x : \mathsf{Nat}, f : (y\!:\mathsf{Nat}) \xrightarrow{\mathsf{L}} (p\!:x = \mathsf{Suc}\, y) \xrightarrow{\mathsf{L}} T \vdash_{\mathsf{L}} t : T\\ \Gamma \vdash (x\!:\mathsf{Nat}) \xrightarrow{\mathsf{L}} T\end{array}}{\Gamma \vdash_\theta \mathsf{ind}\, f\; x.t : (x\!:\mathsf{Nat}) \xrightarrow{\mathsf{L}} T}\textsc{T\_Ind}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash_\theta 0 : \mathsf{Nat}}\textsc{T\_Zero} \qquad \frac{\Gamma \vdash_\theta t : \mathsf{Nat}}{\Gamma \vdash_\theta \mathsf{Suc}\, t : \mathsf{Nat}}\textsc{T\_Suc}$$

$$\frac{\begin{array}{l}\Gamma \vdash_\theta t : \mathsf{Nat}\\ \Gamma, z : 0 = t \vdash_\theta t_1 : T \quad \Gamma \vdash T\\ \Gamma, x : \mathsf{Nat}, z : (\mathsf{Suc}\, x) = t \vdash_\theta t_2 : T\end{array}}{\Gamma \vdash_\theta \mathsf{ncase}_z\; t\; \mathsf{of}\; \{Z \Rightarrow t_1; \mathsf{S}\; x \Rightarrow t_2\} : T}\textsc{T\_NCase}$$

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma x\!:T_1.T_2\\ \Gamma \vdash_\theta t_1 : T_1\\ \Gamma \vdash_\theta t_2 : \{t_1/x\}T_2 \quad \Gamma \vdash \{t_1/x\}T_2\end{array}}{\Gamma \vdash_\theta \langle t_1, t_2\rangle : \Sigma x\!:T_1.T_2}\textsc{T\_Pair} \qquad \frac{\begin{array}{l}\Gamma \vdash_\theta t : \Sigma x\!:T_1.T_2\\ \Gamma, x : T_1, y : T_2, z : \langle x, y\rangle = t \vdash_\theta t' : T\\ \Gamma \vdash T\end{array}}{\Gamma \vdash_\theta \mathsf{pcase}_z\; t\; \mathsf{of}\; \{(x,\; y) \Rightarrow t'\} : T}\textsc{T\_PCase}$$

$$\frac{\begin{array}{l}t \leadsto_p^* t_0 \quad t' \leadsto_p^* t_0\\ \Gamma \vdash_{\mathsf{P}} t : T \quad \Gamma \vdash_{\mathsf{P}} t' : T'\end{array}}{\Gamma \vdash_\theta \mathsf{join} : t = t'}\textsc{T\_Join} \qquad \frac{\begin{array}{l}\Gamma \vdash_\theta t : \{t_1/x\}T\\ \Gamma \vdash_{\mathsf{L}} t' : t_1 = t_2\\ \Gamma \vdash \{t_2/x\}T\end{array}}{\Gamma \vdash_\theta t : \{t_2/x\}T}\textsc{T\_Conv}$$

Figure 7.10: Effect-style kinding and typing

Similar reasoning holds for function definitions. A lambda-expression $\lambda x.t$ is already a value, so evaluating it can never diverge and the rule T_LAM can be checked at any θ. However, the body of the function may diverge when the function is called (it has effect θ', which may be P), and we record this information by the θ' in the function type. Similar reasoning holds for generally recursive functions (T_REC), where the function definition itself is terminating but the P in the function type $(x : T') \overset{\mathsf{P}}{\to} T$ records that it is dangerous to call. Structually recursive functions (T_IND) are given the type $(x : \mathsf{Nat}) \overset{\mathsf{L}}{\to} T$, with an L.

Function types are eliminated by the application rule TAPP, which is the classic type-and-effect style application rule. It combines three sources of nontermination: the final effect has to be P if the evaluation of t or t' may diverge (θ), or if t may diverge when called (θ').

All the other rules in the system follow the corresponding rules in the possible-world-style calculus quite closely, with the only difference being that variables in the context are not tagged with a θ.

7.4.1 Mixing L and P expressions in a program

The possible-world system used @-types (the BOX/UNBOX rules) and mobile types (the TMOBILEVAL rule) to mix expressions with different θs. In the effect-style calculus we have neither of these features, so the same use-cases are expressed slightly differently.

For example, we saw in Section 7.3.3 how a possibly nonterminating function f could require its argument to be a proof, by the type $f : A@\mathsf{L} \to B$. In the effect-style system the meaning of a type is always completely specified by the classifiers on the arrows inside it, so we instead express f's requirements in the type A itself. For example, suppose f wants a logical proof that some variable x is nonzero:

$$\Gamma \vdash^{\mathsf{P}} f : ((x = 0 \to \mathsf{False})@\mathsf{L}) \to B$$

In the effect-style system we write this by making f expect a terminating function:

$$\Gamma \vdash_{\mathsf{P}} f : (x = 0 \overset{\mathsf{L}}{\to} \mathsf{False}) \overset{\mathsf{P}}{\to} T$$

Conversely, we saw that a logical function g can be applied to a value in P, if it has the type $g : A@\mathsf{P} \to B$. In the effect style system this use-case is also supported, because if a value is typeable at all, it is typeable at L:

***Lemma* 30** (Values have logical effect)**.** If $\Gamma \vdash_\theta u : T$, then $\Gamma \vdash_{\mathsf{L}} u : T$.

So if a function t can be checked at L, and the function is logical (i.e. the latent effect in t's type is L), then the application to any value $t\,u$ can be checked at L also.

We also saw how Zombie allows generally recursive functions to return proofs, e.g. the function `solver` in Section 7.2.2. These examples also fit well into the effect-style system. We could give the SAT-solver a type like the following:

$$\begin{aligned}
&\texttt{solver : (f : Formula)} \xrightarrow{\mathsf{P}} \texttt{Either } (\Sigma\texttt{(v:Assignment). eval f v = true)}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{((v:Assignment)} \xrightarrow{\mathsf{L}} \texttt{eval f v = false)}
\end{aligned}$$

Here the termination behaviour can be read off the classifiers on the arrows: a call to `solve` might not terminate ($\xrightarrow{\mathsf{P}}$), but if it *does* terminate, then the function it returns is total ($\xrightarrow{\mathsf{L}}$).

Finally, the TMobileVal rule in the possible-world-style system expresses the payoff of CBV evaluation: no matter which fragment it came from, once you have evaluated it a number is a number (and an equality proof is an equality proof, etc). This part of the story works even better in the effect language. If we have

$$\begin{aligned}
&\texttt{foo : (x : Nat)} \xrightarrow{\mathsf{P}} \texttt{(2 = 2)}\\
&\texttt{bar : (2 = 2)} \xrightarrow{\mathsf{L}} \texttt{Nat}
\end{aligned}$$

we are allowed to directly form the application `bar (foo 3)` (which will be given the effect P). On the other hand, if `foo` was in P and we wanted to use the rule TMobileVal this would have to be explicitly sequenced as `let x = foo 3 in bar x` in order to create a syntactic value `x`.

As illustrated by the above example, we do not need to include an explicit rule for mobile values in the effect system. In some sense this is because *every* type is mobile: the meaning of function types is determined by the annotation on the arrow, rather than by the θ on the judgement, so types in the effect system always classify the same values in L as in P contexts.

This property seems promising if we consider (in future work) extending the system to include polymorphism. With just the typing rules presented in this section, the effect style system is a little more heavy on annotations than the possible-world-style system: here *every* function needs to be marked with a θ, whereas in the possible-world system, only functions with a non-mobile argument (i.e. higher-order functions) need an @θ annotation. But in the effect system every type is mobile, so we would not need to add annotations to type variables, and that should make types considerably less cluttered.

$\boxed{T <: T'}$

$$\frac{\begin{array}{l}\theta \le \theta' \\ T'_1 <: T_1 \quad T_2 <: T'_2\end{array}}{(x : T_1) \xrightarrow{\theta} T_2 <: (x : T'_1) \xrightarrow{\theta'} T'_2}\,\textsc{Sub\_Arr} \qquad \frac{T_2 <: T'_2}{\Sigma x : T_1 .\, T_2 <: \Sigma x : T_1 .\, T'_2}\,\textsc{Sub\_Sigma}$$

$$\frac{}{\mathsf{Nat} <: \mathsf{Nat}}\,\textsc{Sub\_Nat} \qquad \frac{}{(t_1 = t_2) <: (t_1 = t_2)}\,\textsc{Sub\_Eq}$$

$\boxed{\Gamma \vdash_\theta t : T}$

$$\frac{\begin{array}{c}\Gamma \vdash_\theta t : T \\ T <: T' \quad \Gamma \vdash T'\end{array}}{\Gamma \vdash_\theta t : T'}\,\textsc{T\_Subtype}$$

Figure 7.11: Effect-style typing: subtyping

7.4.2 Subtyping

Just like in the possible-world style system, we may wish to go further than just subsumption. A function $(x : T) \xrightarrow{\mathsf{L}} T'$ is strictly better than $(x : T) \xrightarrow{\mathsf{P}} T'$, even when it appears inside a larger type. In order to reflect this, we add a subtyping relation to the language (Figure 7.11). The rule Sub_Arr states that a function type is better if it has a better θ (i.e. if $\theta \le \theta'$). The rules also extend the subtyping relation underneath arrow types and Σ-types.

Compared to the subtyping relation for the possible-world language (Figure 7.7), the effect-style subtyping relation can be simpler, because there are fewer features (@-types, mobile types) to take into account.

7.5 Translating between the two systems

As we saw in Section 7.4.1 the ways to express certain programming idioms are quite different in the two calculi, so it is not immediately obvious whether the two systems are equally expressive. But they are in fact equivalent, and in a rather strong sense: exactly the same terms are typeable in both systems (but at different types).

The statement "exactly the same" needs some qualification, because the syntax of t-expressions and a-expressions are different; the latter also includes types and the sort Type. So we define functions $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ which inject expressions from one language into the expression syntax of the other. They leave terms unchanged but map all other expressions to 0 (to pick an arbitrary term):

$$\begin{array}{lcllcl}
\lceil x \rceil & = & x & \lfloor x \rfloor & = & x \\
\lceil \lambda x.t \rceil & = & \lambda x.\lceil t \rceil & \lfloor \lambda x.a \rfloor & = & \lambda x.\lfloor a \rfloor \\
\lceil \mathsf{rec}\, f\ x.t \rceil & = & \mathsf{rec}\, f\ x.\lceil t \rceil & \lfloor \mathsf{rec}\ f\ x.a \rfloor & = & \mathsf{rec}\, f\ x.\lfloor a \rfloor \\
\lceil \mathsf{ind}\, f\ x.t \rceil & = & \mathsf{ind}\ f\ x.\lceil t \rceil & \lfloor \mathsf{ind}\ f\ x.a \rfloor & = & \mathsf{ind}\, f\ x.\lfloor a \rfloor \\
\lceil t\, t' \rceil & = & \lceil t \rceil\ \lceil t' \rceil & \lfloor a\ b \rfloor & = & \lfloor a \rfloor\ \lfloor b \rfloor \\
\ldots \text{etc} \ldots & & & \ldots \text{etc} \ldots & & \\
 & & & \lfloor \mathsf{Type} \rfloor & = & 0 \\
 & & & \lfloor (x : A) \to B \rfloor & = & 0 \\
 & & & \lfloor a = b \rfloor & = & 0 \\
 & & & \lfloor \mathsf{Nat} \rfloor & = & 0 \\
 & & & \lfloor \Sigma x : A.B \rfloor & = & 0 \\
 & & & \lfloor A@\theta \rfloor & = & 0
\end{array}$$

The formal statement then is that for any expressions a and t, $\lfloor a \rfloor$ is typeable if a is, and $\lceil t \rceil$ is typeable if t is. Of course, the interesting question is how to translate a type A into some suitable type T and vice versa, in particular how to handle arrow types.

7.5.1 Translating from possible-world to effectful

First, consider translating possible-world style types to the effect-style system. The meaning of an arrow type $(x : A) \to B$ (whether it ranges over terminating or general functions) depends on whether it occurs in an L or a P context. So the type translation function $\lfloor A \rfloor_\theta$ is parameterized by a θ, which is changed by @-types:

$$\begin{array}{lcl}
\lfloor \mathsf{Nat} \rfloor_\theta & = & \mathsf{Nat} \\
\lfloor (x : A) \to B \rfloor_\theta & = & (x : \lfloor A \rfloor_{\mathsf{L}}) \xrightarrow{\theta} \lfloor B \rfloor_\theta \\
\lfloor \Sigma x : A.B \rfloor_\theta & = & \Sigma x : \lfloor A \rfloor_\theta . \lfloor B \rfloor_\theta \\
\lfloor A@\theta' \rfloor_\theta & = & \lfloor A \rfloor_{\theta'} \\
\lfloor a = b \rfloor_\theta & = & \lfloor a \rfloor = \lfloor b \rfloor
\end{array}$$

It is easy to check that this translation is compatible with the mobile and subtyping judgements:

***Lemma* 31.** If $\mathsf{Mobile}\,(A)$, then $\lfloor A \rfloor_{\mathsf{L}} = \lfloor A \rfloor_{\mathsf{P}}$.

***Lemma* 32.** If $A <: B$ and $\theta_1 \leq \theta_2$, then $\lfloor A \rfloor_{\theta_1} <: \lfloor B \rfloor_{\theta_2}$.

***Lemma* 33.** If $A <:_{\mathsf{L}} B$, then $\lfloor A \rfloor_{\mathsf{L}} <: \lfloor B \rfloor_{\mathsf{L}}$.

Because the kinding rule for arrows $(x : A) \to B$ requires $\mathsf{Mobile}\,(A)$, by lemma 31 it does not much matter at what θ we translate the domain A. In the definition of the translation function we use $\lfloor A \rfloor_{\mathsf{L}}$ because it makes the statement of lemma 32 simpler—this way the lemma holds for all types, not just well-kinded ones.

We extend the translation of types to contexts, $\lfloor \Gamma \rfloor$, by mapping each binding $x :^{\theta} A$ pointwise to $x : \lfloor A \rfloor_{\theta}$.

Then we prove that the translation preserves typing by induction on the $\Gamma \vdash^{\theta} a : A$ judgement. The proof is not difficult, but the two type systems were carefully crafted to make it go through—in particular, the proof uses subtyping in several places. Also, in order to have suitable induction hypotheses available, we go via an intermediate, equivalent definition of the type system which adds more kinding assumptions.

***Lemma* 34** (Typing rules with more kinding assumptions)**.** For any Γ, θ, a, and A, the typing judgement $\Gamma \vdash^{\theta} a : A$ holds iff the same typing holds in a system where the rules TUnbox*, TConv, and TSub have been modified by adding extra kinding assumptions as follows:

$$\frac{\begin{array}{l}\Gamma \vdash^{\theta} v : A@\theta' \\ \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type}\end{array}}{\Gamma \vdash^{\theta'} v : A}\textsc{TUnboxVal'} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a : A \\ \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type} \vee A \equiv \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{P}} a : A}\textsc{TSub'}$$

$$\frac{\begin{array}{ll}\Gamma \vdash^{\theta_1} b_1 : B_1 & \Gamma \vdash^{\mathsf{L}} B_1 : \mathsf{Type} \\ \Gamma \vdash^{\theta_2} b_2 : B_2 & \Gamma \vdash^{\mathsf{L}} B_2 : \mathsf{Type} \\ \Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 & \\ \Gamma \vdash^{\theta} a : \{b_1/x\}A & \\ \Gamma \vdash^{\mathsf{L}} \{b_2/x\}A : \mathsf{Type} & \end{array}}{\Gamma \vdash^{\theta} a : \{b_2/x\}A}\textsc{TConv'}$$

***Theorem* 35** (Translation of typing)**.**

1. If $\Gamma \vdash^{\theta} a : A$, then $\lfloor \Gamma \rfloor \vdash_{\theta} \lfloor a \rfloor : \lfloor A \rfloor_{\theta}$
2. If $\Gamma \vdash^{\theta} A : \mathsf{Type}$, then $\lfloor \Gamma \rfloor \vdash \lfloor A \rfloor_{\theta'}$ for all θ'.
3. If $\vdash \Gamma$, then $\vdash \lfloor \Gamma \rfloor$.

Proof. By mutual induction on the three judgements. The two most interesting cases are:

TSub This case illustrates why we need subtyping in the system. The given derivation looks like

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} a : A \\ \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type} \vee A \equiv \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{P}} a : A}\textsc{TSub'}$$

For example, if the type A is a function type $(x : A_1) \to A_2$, then we need to show $\lfloor \Gamma \rfloor \vdash_{\mathsf{P}} \lfloor a \rfloor : (x : \lfloor A_1 \rfloor_{\mathsf{L}}) \overset{\mathsf{P}}{\to} \lfloor A_2 \rfloor_{\mathsf{P}}$, and the IH says $\lfloor \Gamma \rfloor \vdash_{\mathsf{L}} \lfloor a \rfloor : (x :$

$\lfloor A_1 \rfloor_{\mathsf{L}}) \xrightarrow{\mathsf{L}} \lfloor A_2 \rfloor_{\mathsf{L}}$. So just the rule T_Sub is not enough; we must also change the θs on the arrow and inside A_2.

So we use subtyping (T_Subtype and lemma 32). In order to apply T_Subtype we need to know $\lfloor \Gamma \rfloor \vdash \lfloor A \rfloor_{\mathsf{P}}$. This follows by the mutual IH, using the extra kinding premise from lemma 34.

TMobileVal This illustrates how the benefits of mobile types can be emulated in the effect system. The given derivation looks like

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \Gamma \vdash^{\mathsf{L}} A : \mathsf{Type} \quad \mathsf{Mobile}\,(A)}{\Gamma \vdash^{\mathsf{L}} v : A}\textsc{TMobileVal}$$

The IH states $\lfloor \Gamma \rfloor \vdash_{\mathsf{P}} \lfloor v \rfloor : \lfloor A \rfloor_{\mathsf{P}}$. Because v is a value we have $\lfloor \Gamma \rfloor \vdash_{\mathsf{L}} \lfloor v \rfloor : \lfloor A \rfloor_{\mathsf{P}}$ (lemma 30), and then because A is mobile we have $\lfloor \Gamma \rfloor \vdash_{\mathsf{L}} \lfloor v \rfloor : \lfloor A \rfloor_{\mathsf{L}}$ (lemma 31). □

7.5.2 Translating from effectful to possible-world

To translate types in the other direction, we use the following translation function $\lceil T \rceil$. The interesting case is the one for function types: in order to encode the termination classification θ on the function we wrap the type in an $@\theta$.

$$\begin{array}{lcl}
\lceil \mathsf{Nat} \rceil & = & \mathsf{Nat} \\
\lceil (x : T_1) \xrightarrow{\theta} T_2 \rceil & = & ((x : \lceil T_1 \rceil) \to \lceil T_2 \rceil)@\theta \\
\lceil \Sigma x : T_1 . T_2 \rceil & = & \Sigma x : \lceil T_1 \rceil . \lceil T_2 \rceil \\
\lceil t_1 = t_2 \rceil & = & \lceil t_1 \rceil = \lceil t_2 \rceil
\end{array}$$

We translate contexts, $\lceil \Gamma \rceil$, by mapping each binding $x : T$ pointwise to $x :^{\mathsf{L}} \lceil T \rceil$.

Again, it is easy to verify that the type translation is compatible with the mobile and subtyping judgements:

***Lemma* 36.** For any T, we have $\mathsf{Mobile}\,(\lceil T \rceil)$.

***Lemma* 37.** For any T_1 and T_2, if $T_1 <: T_2$ then $\lceil T_1 \rceil <: \lceil T_2 \rceil$.

The translation of arrow types nicely illustrates how $@\theta$-types in the "possible world" system work. The $@\theta$-tag does not affect $\lceil T_1 \rceil$ or $\lceil T_2 \rceil$ (because these are mobile types), it but controls the interpretation on the function arrow. By contrast, if we were translating the effectful calculus into a type system based around monads, the monadic constructor would be applied to the codomain of the function, rather than to the entire function type.

Sometimes the type translation will create a type which contains "too many" @-tags, so we need to know that one can modify a typing derivation to use different, but equivalent, type assumptions. Note that the type context is contravariant:

***Lemma* 38** (Strengthening types in the context)**.** Suppose that $\Gamma \vdash^{\theta} x : A$ implies $\Gamma \vdash^{\theta'} x : A'$ for all Γ. Then $\Gamma_1, x :^{\theta'} A', \Gamma_2 \vdash^{\theta_1} b : B$ implies $\Gamma_1, x :^{\theta} A, \Gamma_2 \vdash^{\theta_1} b : B$.

***Theorem* 39** (Translation of typing)**.**

1. If $\Gamma \vdash_{\theta} t : T$, then $\lceil \Gamma \rceil \vdash^{\theta} \lceil t \rceil : \lceil T \rceil$.
2. If $\Gamma \vdash T$, then $\lceil \Gamma \rceil \vdash^{\mathsf{L}} \lceil T \rceil : \mathsf{Type}$.
3. If $\vdash \Gamma$, then $\vdash \lceil \Gamma \rceil$.

Proof. Mutual induction on the judgements. The most interesting case is:

TInd This is the case which motivates including the $A <:_{\mathsf{L}} A'$ relation. The given typing derivation looks like

$$\frac{\begin{array}{l}\Gamma, x : \mathsf{Nat}, f : (y\!:\!\mathsf{Nat}) \xrightarrow{\mathsf{L}} (p\!:\!x = \mathsf{Suc}\, y) \xrightarrow{\mathsf{L}} T \vdash_{\mathsf{L}} t : T \\ \Gamma \vdash (x\!:\!\mathsf{Nat}) \xrightarrow{\mathsf{L}} T\end{array}}{\Gamma \vdash_{\theta} \mathsf{ind}\, f\ x.t : (x\!:\!\mathsf{Nat}) \xrightarrow{\mathsf{L}} T}\ \textsc{T\_Ind}$$

The induction hypothesis for the first premise gives

$$\lceil \Gamma \rceil, x :^{\mathsf{L}} \mathsf{Nat}, f :^{\mathsf{L}} ((y\!:\!\mathsf{Nat}) \to ((p\!:\!y = \mathsf{Suc}\, x) \to \lceil T' \rceil)@\mathsf{L})@\mathsf{L} \vdash^{\mathsf{L}} \lceil t \rceil : \lceil T \rceil$$

whereas in order order to apply TIND we need

$$\lceil \Gamma \rceil, x :^{\mathsf{L}} \mathsf{Nat}, f :^{\mathsf{L}} (y\!:\!\mathsf{Nat}) \to (p\!:\!y = \mathsf{Suc}\, x) \to \lceil T' \rceil \vdash^{\mathsf{L}} \lceil t \rceil : \lceil T \rceil.$$

So we need to know that $(y\!:\!\mathsf{Nat}) \to (p\!:\!y = \mathsf{Suc}\, x) \to \lceil T' \rceil$ is a better type than $((y : \mathsf{Nat}) \to ((p : y = \mathsf{Suc}\, x) \to \lceil T' \rceil)@\mathsf{L})@\mathsf{L}$ in order to apply strengthening (lemma 38). This is not true in general, but it holds here because the binding for f is tagged as L. So we use the rule TSUBTYPEL. □

7.6 Normalization

To make sure that we defined the system correctly, we prove that every closed expression that checks at L terminates. By the translation theorems in the previous section, it equivalent to prove this for either the “possible world” calculus or the effect-style one. We choose the effect-style system, because the proofs are easier that way.

$$
\begin{array}{ll}
[\![\mathsf{Nat}]\!] & = \{u \mid \ u \text{ is of the form } \mathsf{Suc}^n\ 0\} \\
[\![(x : T_1) \overset{\mathsf{L}}{\to} T_2]\!] & = \{u \mid \mathsf{fv}\, u = \emptyset \text{ and } \forall u_1.\text{ if } u_1 \in [\![T_1]\!] \text{ then } (u\, u_1) \in \mathcal{C}[\![\{u_1/x\}\, T']\!]\} \\
[\![(x : T) \overset{\mathsf{P}}{\to} T']\!] & = \{u \mid \mathsf{fv}\, u = \emptyset\} \\
[\![\Sigma x : T_1 . T_2]\!] & = \{\langle u_1, u_2\rangle \mid \mathsf{fv}\, \langle u_1, u_2\rangle = \emptyset \text{ and } u_1 \in [\![T_1]\!] \text{ and } u_2 \in [\![\{u_1/x\}\, T_2]\!]\} \\
[\![t_1 = t_2]\!] & = \{\mathsf{join} \mid t_1 \curlyvee t_2\} \\
 & \\
\mathcal{C}[\![T]\!] & = \{t \mid \exists u. t \leadsto^* u \in [\![T]\!]\}
\end{array}
$$

Figure 7.12: Type interpretation

We use the standard proof method, *logical relations*, which was first introduced by Tait [134] to prove strong normalization for Gödel's System T (i.e. the simply-typed lambda calculus extended with recursion over natural numbers). The crux of this method is to define a "type interpretation": for each type T we define a set of terms $[\![T]\!]$ which can be used at that type without causing divergence. The main theorem then states that the interpretation is "sound", i.e. that any well-typed term is a member of its type's interpretation.

The proof method is well known (see e.g. Girard's textbook [60]). We adapt it to our language in two ways. First, instead of proving strong normalization (i.e. that any order of reduction terminates) we only prove that closed terms terminate under CBV-order. This simplifies the proof because one only needs to consider values, not open "neutral" expressions. Proving CBV-normalization is enough to establish type safety and logical consistency, and in the full Zombie language open expressions may in any case diverge (because of irrelevant type casts, Section 3.6). Accordingly, the sets $[\![T]\!]$ in our definition (Figure 7.12) contain only values. We mutually define the set $\mathcal{C}[\![T]\!]$ as the set of all expressions that evaluate to values in $[\![T]\!]$.

Second, our calculus differs from System T because it includes dependent types and propositional equality. So we need to pick an interpretation for the equality type. Just as we did in the type safety proof in Chapter 3, we interpret equations by joinability under parallel reduction: the set $[\![t_1 = t_2]\!]$ contains the single value join if $t_1 \curlyvee t_2$, and is empty otherwise. The interpretations for arrow- and Σ-types also differ from the simply-typed system because the codomain type is closed by a substitution, $\{u_1/x\}\, T_2$.

Apart from these two points, this is a standard logical relation for normalizations. The interpretation function $[\![T]\!]$ is defined by recursion on the depth of the type T, where any terms occurring inside the type are considered to have depth zero. So substituting a term into a type does not change the depth of the type—this is one way in which having syntactically separate terms and types is more convenient for the proof than having collapsed syntax.

The soundness theorem relies on a few key lemmas. First, we need the same facts about joinability as for the type safety proof, as well a lemma stating that joinability of natural number values coincides with syntactic equality.

***Lemma* 40** (Properties of $\curlyvee$)**.**

1. $\curlyvee$ is an equivalence relation.
2. If $t_1 \leadsto_{\mathsf{cbv}} t_2$, then $t_1 \curlyvee t_2$.
3. If $t_1 \curlyvee t_2$ then $\{t_1/x\}t \curlyvee \{t_2/x\}t$.
4. If $t_1 \curlyvee t_2$ and $\{t_1/x\}t \curlyvee \{t_1/x\}t'$, then $\{t_2/x\}t \curlyvee \{t_2/x\}t'$.

***Lemma* 41** (Joinable natural number constants are equal)**.** If $u_1 \curlyvee u_2$ and both u_1 and u_2 are of the form $\mathsf{Suc}^n\ 0$, then $u_1 = u_2$.

Then we need some lemmas about the type interpretation.

***Lemma* 42** (Type interpretation respects joinability)**.** If $t \curlyvee t'$ then $[\![\{t/x\}T]\!] = [\![\{t'/x\}T]\!]$.

Proof. Induction on T. The only interesting case is for equality types, which uses lemma 40 part (4). □

***Lemma* 43** (Type interpretation respects subtyping)**.** If $T <: T'$, then $[\![T]\!] \subseteq [\![T']\!]$.

Finally, although we are ultimately only interested in closed expressions, for the induction to go through we need to generalize the soundness theorem to also talk about open expressions. So the theorem is stated in terms of multi-substitutions ρ, and we define the judgement $\Gamma \models \rho$ saying that ρ maps every variable in the context Γ to a value in the appropriate interpretation:

$$\frac{}{\cdot \models \emptyset}\textsc{ENil} \qquad \frac{\Gamma \models \rho \qquad u \in [\![T]\!]}{\Gamma, x : T \models \rho[x \mapsto u]}\textsc{ECons}$$

We maintain the invariant that all the values occurring in the above definitions are closed:

- If $u \in [\![T]\!]$ then $\mathsf{fv}\, u = \emptyset$.
- If $(x : T) \in \Gamma$ and $\Gamma \models \rho$ then $\mathsf{fv}\,(\rho\, x) = \emptyset$.
- If $\Gamma \vdash_\theta t : T$ and $\Gamma \models \rho$ then $\mathsf{fv}\,(\rho\, t) = \emptyset$.

This is to make it easy to manipulate expressions involving substitutions. For example we always have $\rho\,(\{t_1/x\}t_2) = \{\rho\, t_1/x\}\rho\, t_2$, because x is not free in ρ.

We can now prove soundness. Normalization and logical consistency are immediate corollaries.

***Theorem* 44** (Soundness)**.** If $\Gamma \vdash_{\mathsf{L}} t : T$ and $\Gamma \models \rho$, then $\rho\, t \in \mathcal{C}[\![\rho\, T]\!]$.

Proof. Induction on $\Gamma \vdash_\theta t : T$. The two most interesting cases are:

TApp This case illustrates the extra complication that come from working with a dependent language instead of a simply-typed one.

We are given $\Gamma \vdash_{\mathsf{L}} t_1\, t_2 : \{t_2/x\}T$. From the IHs we know $\rho\, t_1 \rightsquigarrow^* u_1 \in [\![\rho\,(x : T') \xrightarrow{\mathsf{L}} T]\!]$ and $\rho\, t_2 \rightsquigarrow^* u_2 \in [\![\rho\, T']\!]$. So by the definition for $[\![\rho\,(x : T') \xrightarrow{\mathsf{L}} T]\!]$ we know that $t_1\, t_2 \rightsquigarrow^* u_1\, u_2 \rightsquigarrow^* u \in [\![\rho\,\{u_2/x\}T]\!]$.

But we actually need to show $u \in [\![\rho\,\{t_2/x\}T]\!]$. At this point, note that $\rho\,\{t_2/x\}T = \{\rho\, t_2/x\}\rho\, T$ and $\rho\,\{u_2/x\}T = \{u_2/x\}\rho\, T$. We know $\rho\, t_2 \rightsquigarrow^* u_2$, so by lemma 40 we have $\rho\, t_2 \curlyvee u_2$, and hence by lemma 42 $[\![\{\rho\, t_2/x\}\rho\, T]\!] = [\![\{u_2/x\}\rho\, T]\!]$.

TInd This is Tait's trick: we use an inner induction to handle the natural number recursor. Because of the way we formulated the rule TIND using type-based termination, this involves reasoning about equations.

We are given $\Gamma \vdash_\theta \mathsf{ind}\, f\, x.t : (x : \mathsf{Nat}) \xrightarrow{\mathsf{L}} T$, and we need to show that $(\mathsf{ind}\, f\, x.\rho\, t)\, u \in \mathcal{C}[\![\{u/x\}T]\!]$ for any $u \in [\![\mathsf{Nat}]\!]$. So consider such a u; by the definition of $[\![\mathsf{Nat}]\!]$ we know $u = \mathsf{Suc}^n\ 0$ for some n. We now show

$$\forall n. u = \mathsf{Suc}^n\ 0 \implies (\mathsf{ind}\, f\, x.\rho\, t)\, u \in \mathcal{C}[\![\{u/x\}\rho\, T]\!]$$

by an inner induction on n. The inductive case and the base case are both quite similar. The reduction rule for ind states

$$\begin{aligned}(\mathsf{ind}\, f\, x.\rho\, t)\, u \ &\rightsquigarrow \{u/x\}\{\lambda y.\lambda z.(\mathsf{ind}\, f\, x.\rho\, t)\, y/f\}\rho\, t \\ &\equiv \rho[x \mapsto u][f \mapsto \lambda y.\lambda z.(\mathsf{ind}\, f\, x.\rho\, t)\, y]\, t.\end{aligned}$$

So by the IH for t it suffices to show that

$$\Gamma, x : \mathsf{Nat}, f : (y : \mathsf{Nat}) \xrightarrow{\mathsf{L}} (p : x = \mathsf{Suc}\, y) \xrightarrow{\mathsf{L}} T \models \rho[x \mapsto u][f \mapsto \lambda y.\lambda z.(\mathsf{ind}\, f\, x.\rho\, t)\, y],$$

which amounts to proving

$$\lambda y.\lambda z.(\mathsf{ind}\, f\, x.\rho\, t)\, y \in [\![(y : \mathsf{Nat}) \xrightarrow{\mathsf{L}} (p : u = \mathsf{Suc}\, y) \xrightarrow{\mathsf{L}} \{y/x\}\rho\, T]\!]$$

which in turn is equivalent to

$$\forall u_1 \in [\![\mathsf{Nat}]\!]. \forall u_2 \in [\![u = \mathsf{Suc}\, u_1]\!].(\mathsf{ind}\, f\, x.\rho\, t)\, u_1 \in \mathcal{C}[\![\{u_1/x\}\rho\, T]\!].$$

Now, from the assumption $u_2 \in [\![u = \mathsf{Suc}\, u_1]\!]$ and lemma 41 we know that $u = \mathsf{Suc}\, u_1$. So in the base case of the inner induction (when $u = 0$) the implication is vacuously true. And in the step case, when $u = \mathsf{Suc}^{n'+1}\, 0$, we know that $u_1 = \mathsf{Suc}^{n'}\, 0$, so we apply the inner IH. □

***Corollary* 45** (Normalization)**.** *If $\cdot \vdash_{\mathsf{L}} t : T$, then there exists some u such that $\rho\, t \leadsto_c^* u$.*

Proof. Instantiate the soundness theorem with $\rho := \emptyset$. □

***Corollary* 46** (Consistency)**.** *There is no term t such that $\cdot \vdash_{\mathsf{L}} t : \mathsf{Suc}\, 0 = 0$.*

Proof. Suppose there was such a t. From the soundness theorem instantiated with $\rho := \emptyset$ we know that $[\![\mathsf{Suc}\, 0 = 0]\!]$ is nonempty, so $\mathsf{Suc}\, 0 \curlyvee 0$. But then by lemma 41 we would have $\mathsf{Suc}\, 0 = 0$, which is absurd. □

7.6.1 Normalization for the possible-world style calculus

Thanks to the translation in Section 7.5, Theorem 44 also implies that the "possible world"-style calculus enjoys normalization and consistency. Previously, we published a direct proof of normalization for a very close variation of that calculus [31]. However, the proof in this chapter, which goes via the effect style calculus, is simpler and more flexible.

The key point is that in the proof of Theorem 44 above, we only had to consider cases where the typing derivation checks at L. In every case where we have to prove an informative statement, the interesting premises of the rule also check at L. For example, the case for T_Rec is completely trivial: because the label on the arrow is P we only need show that $\rho\, (\mathsf{rec}\, f\ x.t)$ is a closed value.

In the "possible world" system, we could hope to proceed similarly, by defining a type interpretation $[\![A]\!]$, and proving the theorem

If $\Gamma \vdash^{\mathsf{L}} a : A$ and $\Gamma \models \rho$, then $\rho\, a \leadsto^* v \in [\![\rho\, A]\!]$.

However, if we try to prove this by induction on the typing derivation, we get stuck in the rules TUnboxVal and TMobileVal. For instance, one can use TUnboxVal to unpack a logical theorem from a programmatic value:

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A@\mathsf{L}}{\Gamma \vdash^{\mathsf{L}} v : A}\textsc{TUnboxVal}$$

With the above theorem statement these is no way to make progress here; we need to show $v \in [\![A]\!]$ but the premise to the rule does not give any induction hypothesis, because it checks at P.

The proof for the "possible world" system [30, 31] instead defines two different type interpretations $[\![A]\!]^{\mathsf{L}}$ and $[\![A]\!]^{\mathsf{P}}$, which coincide for mobile types A. The theorem statement for typing judgements in P is a partial correctness statement: if $\rho\, a \leadsto^* v$ then $v \in [\![A]\!]^{\mathsf{P}}$.

Unfortunately, this approach means that we need to define an interpretation which is sound for all programs in P! Defining a sound logical relation for generally recursive functions is already quite tricky (the reduction rule for $(\mathsf{rec}\ f\ x.b)\ v$ substitutes the same function into the body b, so when proving soundness there is no induction hypothesis for it). But in addition the programmatic fragment of the language should have enough features to support ordinary functional programming, e.g. in Casinghino et al. [31] it included general recursive μ-types. Because $[\![A]\!]^{\mathsf{P}}$ must account for recursive functions and recursive types it cannot be defined by recursion on A, so instead that work used a quite sophisticated definition with combines a step-indexed logical relation for P with a ordinary logical relation for L.

The hybrid step-indexing idea is interesting in itself, but the proofs are more difficult to set up than the "vanilla" logical relation above. Also, the step-indexed proof is less easily extensible, because every new feature in the P fragment needs to be reflected in the $[\![A]\!]^{\mathsf{P}}$ interpretation. There are several programming features beyond recursive types which we considered, but did not include because handling them in the normalization proof was too difficult (Section 7.7.2). By contrast, in the effect-style calculus, we can add any typing rule which concludes at effect P to the system without changing the normalization proof at all.

7.7 Limitations and future work

The termination-checking in Zombie is fairly sophisticated, and is sufficient for a wide selection of different use-cases (Chapter 2). Even so, there are features we would like but do not yet know how to include in a sound way.

7.7.1 Termination inversion and Fixpoint induction

First, Zombie is currently somewhat weak when it comes to external proofs about nonterminating functions. The basic reasoning principle is TJOIN: evaluate an expression a fixed finite number of steps according to the operational semantics. Together with the ability to define structurally recursive functions, this is all that is needed to write proofs about functions in L. However, for functions in P it is not always enough.

Termination inversion

As the simplest example, define natural number addition by general recursion and try to prove that addition is commutative. Of course, in full Zombie it is very easy to define addition in L by structural recursion, and then the proof of commutativity is also easy. However, more complicated programs may not be so easy to write in a manifestly terminating way, and we use addition as a toy example.

```
prog plus : Nat → Nat → Nat
rec plus n m =
  case n of
    Zero → m
    Succ n' → Succ (plus n' m)

log plusAssoc : (i j k : Nat) → plus i (plus j k) = plus (plus i j) k
plusAssoc = ???
```

We would like to proceed by induction on i. However, the proof does not go through. In the step case (when `i = Succ i'`) we need to prove

```
    plus (Succ i') (plus j k) = plus (plus (Succ i') j) k
```

The expression `(plus (Succ i' j)` reduces to `Succ (plus i' j)`, so we can simplify the right-hand side. But then we are stuck: as we discussed in Section 4.1.4, the left-hand side cannot CBV-reduce because `(plus j k)` is not a value, and similarly the expression `(plus (Succ (plus i' j)) k)` cannot CBV-reduce because `plus i' j` is not a value.

The problem is that some reasoning principles are only sound for terminating terms. Proving equations by CBV-reduction is one of them; another is proof by induction over the structure of values. In order to keep track of when such reasoning is valid, many languages (e.g. Nuprl and Sep^3 [74]) add a predicate Terminates a, stating that the expression a is known to terminate. Because Zombie uses call-by-value evaluation, where variables range over values, we do not have to add this as a primitive predicate. We define

$$\mathsf{Terminates}\ a \ :=\ \Sigma x\!:\!A.x = a$$

which can be read as "there exists some value x which is equal to a".

However, having just a way to state that expressions terminate does not go very far, because the language does not have enough ways to prove that something terminates. For example, we can try to save the above theorem by weakening it to only talk about terminating computations:[17]

[17] This idea was investigated by Nathan Collins.

```
log plusAssoc' :
  (i j k : Nat) → Terminates (plus i j) → Terminates (plus j k)
                → plus i (plus j k) = plus (plus i j) k
```

The adjusted statement allows the base case of the induction to go through, but not the step case. We know that `Terminates (plus (Succ i') j)`, and hence (by just substituting equals for equals) that `Terminates (Succ (plus i' j))`. But the Zombie language does not provide a way to go from that to `Terminates (plus i' j)`.

Of course, it is in fact true that `plus i' j` terminates whenever `Succ (plus i' j)` does, because the data constructor `Succ` forces its argument. Some languages, such as Nuprl, include a typing rule reflecting this fact. It applies to any strict evaluation context $\mathcal{C}$:

$$\frac{\Gamma \vdash^{\mathsf{L}} b : \mathsf{Terminates}\ \ \mathcal{C}[a]}{\Gamma \vdash^{\mathsf{L}} b : \mathsf{Terminates}\ \ a}\textsc{TCtxTerm}$$

With this rule in place, we can prove the weakened version of the theorem.

Termination case

The weakened version `plusAssoc'` is a bit disappointing, because `plus` would be associative even if it *did* sometimes diverge. For example, if `plus 0 n` diverges, then both sides of the equation are diverging terms, which means that they are contextually equivalent. So we may wish for a stronger reasoning principle which can prove the associativity theorem outright.

In previous work we proposed *termination-case* [74], which expresses the idea "for all expressions a, either a terminates or it is equivalent to the diverging expression abort". Formally we add a new expression abort to the syntax, with the reduction rule $\mathcal{C}[\mathsf{abort}] \leadsto_{\mathsf{cbv}} \mathsf{abort}$ for any evaluation context $\mathcal{C}$. Then for all expressions a we add the statement $(\mathsf{Terminates}\ \ a) \vee (a = \mathsf{abort})$ as an axiom.

For example, in the base case of the proof of `plusAssoc` we do a case on whether `plus 0 j` is a value or diverges—in either case the equation is provable by just reduction (join).

Another idea, which we already described in Section 4.1.4, is to make the propositional equality coarser, by allowing unrestricted β-reduction instead of just CBV-reductions (even though this makes it equate some terms which are not contextually equivalent). In fact, for functions which use their arguments strictly in the function body—e.g. `plus` above—termination-case is as good as unrestricted β-reduction. To see this, consider how we could prove an equation $(\lambda x.a)\ b = \{b/x\}a$. Using termination-case, it suffices consider two cases, when b is a value and when b diverges.

- In the first case we must prove $(\lambda x.a)\ v = \{v/x\}a$. This is just ordinary CBV reduction.

- In the second case we must prove $(\lambda x.a)$ abort $= \{\text{abort}/x\}a$. In a CBV language we always have $(\lambda x.a)$ abort $=$ abort. So we are left with the proof obligation $\{\text{abort}/x\}a =$ abort which states that the function body is strict in x. Note that, happily, this obligation will not be provable for the problematic example `safediv 3 0 (loop()) = div 3 0`, where the two sides are not contextually equivalent.

The termination-case rule is strictly stronger than termination-inversion. We prove the implication (termination-case) $\Longrightarrow$ (termination-inversion) in two steps. First note that the reduction rule for abort lets us prove that abort $= v$ is a contradiction (i.e. given a proof that abort is equal to a value we may conclude anything), using the same trick as in Section 4.1.5. Write a function which takes $((\lambda x.0)\ v) = ((\lambda x.1)\ v)$ as a precondition and returns False, then satisfy the precondition by changing its type to $((\lambda x.0)$ abort$) = ((\lambda x.1)$ abort$)$. Then we can prove termination-inversion: given a proof of Terminates $\mathcal{C}[a]$, we use termination-case to consider two cases,:

- If a is a value, we are done.
- If a is abort, then by reduction we know $\mathcal{C}[a]$ is equal to abort, which is a contradiction.

Induction principles for programs

During the Trellys project we have thought a lot about reasoning about termination because, as we have seen, in a CBV language termination assertions are needed even for simple equational reasoning. However, in order to prove more complicated properties about nonterminating programs one needs even stronger reasoning principles: some form of *induction*.

A good example which demonstrates the issue is trying to use Zombie to formalize some calculus, say untyped λ-calculus. There are two natural ways to represent reduction behaviour. We could use the same approach as one would in Coq or Agda, and define a big-step evaluation relation as an inductive datatype:

```
data Eval (t : Tm) (t' : Tm) : Type where
  ...
```

But since Zombie supports general recursion, we can also directly define a function to evaluate terms to normal form:

```
prog eval : Tm → Tm
```

It is then natural to ask how these two formalizations are related. In one direction, we can show that `Eval t v` implies `eval t = v` by an induction on the derivation `Eval t v`. However, in the current version of Zombie, there is no way to show the other direction. We would need to do an induction on something, but what? In the case of `plusAssoc` the argument `i` was decreasing in every call, but the same is not true for `t`.

Here is an intuition for why the theorem should be provable. If `eval t = v` holds, then the call to `eval t` did not diverge. So it evaluated to a value by some number of steps, and each recursive call inside it evaluated by some smaller number of steps. So an induction "on the number of steps of evaluation" would give us an IH for each recursive call. In the literature, there are at least two proposals for making this intuition precise.

Fixpoint induction First, many systems inspired by Scott-style denotational semantics (e.g. LCF) provide a principle called *fixpoint induction*, also known as *Scott induction*. It is motivated by the following fact about domains:

> Let D be a domain, $F : D \to D$ a continuous function, and $P \subseteq D$ an admissible predicate. Then $P(\mathsf{fix}(F))$ whenever $P(\bot)$ and $\forall x \in D.P(x) \implies P(Fx)$.

In the most common case the fixpoint is defining a function, so $F : (A \to B) \to (A \to B)$ and $\mathsf{fix}(F){:}\,A \to B$, and the predicate has the form $P(f) = \forall x \in A.Q(f\ x)$. Then the conditions to prove are $Q(\bot)$ and

$$\forall f \in (A \to B).(\forall x \in A.Q(f\ x)) \implies (\forall x \in A.Q(F\ f\ x).$$

Stating this in words, to prove that the predicate Q holds about a function, one may assume it as an induction hypothesis for the recursive calls $f\ x$.

Fixpoint induction only applies to properties P that hold for the completely undefined computation $\bot$. In practice this can be worked around by instead proving the weaker claim $x \neq \bot \implies P(x)$. In fact one often has to qualify statements in this way anyway, because properties like disjointness of data constructors do not hold when the constructor argument may diverge (see e.g. Paulson's verification of unification in LCF [104]).

Also, it only applies to *admissible* properties P. In denotational semantics, a set $P \subseteq D$ is admissible if $\forall n.P(x_n)$ implies $P(\sqcup_n x_n)$, for all increasing sequences $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$. Crary [40] defines a similar notion of admissibility using operational semantics—if we write F^n for the expression $F(F(F \ldots F(\mathsf{diverge})))$, then a predicate P is admissible if for all a and F, $P(\{\mathsf{fix}(F)/x\}a)$ holds whenever $\exists k.\forall n > k.P(\{F^n/x\}a)$ holds.

In either case, it is undecidable whether an arbitrary property is admissible or not. Instead, one can define a grammar of logical formulas, such that any property defined by a formula in that grammar is admissible. Some examples of admissible properties of functions are "wherever the function is defined it is equal to an even number", or "is more defined than some other function". An example of a nonadmissible property is "is undefined for some argument".

Computational induction Smith [122] proposed a different induction principle, which even more directly captures the intuition of gettting an induction hypothesis for each terminating recursive call. Given a big-step operational semantics, he defines b *induces* a to mean that whenever b is evaluated, a is necessarily evaluated also. As a simple example, $\mathcal{C}[a]$ induces a for any evaluation context $\mathcal{C}$. Because this is a semantic notion one can also come up with more involved examples, e.g. $\mathsf{if}\ e_1\ \mathsf{then}\ f(x)\ \mathsf{else}\ f(x)$ induces $f(x)$.

Now, the computational induction principle states that in order to prove "if a terminates, then $P(a)$", one can soundly assume an induction hypothesis for any subexpression induced by a. Or, stated more formally:

> If $\forall b. \mathsf{Terminates}\ \ b \implies (\forall b'. b \text{ induces } b' \implies P(b')) \implies P(b)$,
> then $\forall a. \mathsf{Terminates}\ \ a \implies P(a)$

This can seen as a general recipe for coming up with induction principles to add to a language: whenever we can find a syntactic criterion which implies the "induces" relation, we can add a corresponding version of the computational induction principle to the language. In particular, the termination-inversion rule (Section 7.7.1) is one such instance, with "induces" specialized to "is in an evaluation context" and P specialized to $\mathsf{Terminates}$. Nuprl also includes a version which proves statements about general recursive functions [123].

Smith's paradox

Termination-inversion, fixpoint induction, and computational induction make perfect sense when reasoning about simply typed programs. However, rather suprisingly, in a dependently typed language they can lead to inconsistency!

The problem is related to admissibility in the fixpoint induction sense. Suppose we try to define the meaning of a recursive function of type $(x : T) \to T'$ as the limit of longer and longer unfoldings. If the type T' is not sufficiently wellbehaved, it may be the case that the result of any finite unfolding is in T', but the finished unfolding is not.

Smith [123] used this intuition to construct a logical contradiction. (Interesting reasoning steps are underlined.) Define a Σ-type T of functions which are not total, and recursively define a pair p which inhabits T.

$$\begin{aligned}
\text{Total}\ (f : \mathbb{N} \to \overline{\mathbb{N}}) &\overset{\text{def}}{=} (n : \mathbb{N}) \to \mathsf{Terminates}\ (f\ n)\\
T &\overset{\text{def}}{=} \Sigma(f : \mathbb{N} \to \overline{\mathbb{N}}).\text{Total}\ f \to \text{False}\\
(p : T) &\overset{\text{def}}{=} \text{fix}\ (\lambda p.\langle \lambda x.\text{if}\ x = 0\ \text{then}\ 0\ \text{else}\ \pi_1(p)(x-1), \lambda h.\text{—}\rangle)
\end{aligned}$$

The type $\mathbb{N} \to \overline{\mathbb{N}}$ is Nuprl's type of possibly-nonterminating functions. The first component of the pair p is just a complicated way to write a constant-zero recursive function:

$$g \overset{\text{def}}{=} \lambda x.\text{if}\ x = 0\ \text{then}\ 0\ \text{else}\ g(x-1)$$

The dash in the second component of the pair is an (elided) proof which sneakily derives a contradiction using $\pi_2(p)$ and the hypothesis h that g is total. In more detail we reason as follows. We are given the assumptions

$$\begin{array}{lll}
\mathsf{prog}\ \ p & : & T\\
\mathsf{log}\ \ \ h & : & \mathsf{Total}(\lambda x.\text{if}\ x = 0\ \text{then}\ 0\ \text{else}\ \pi_1(p)(x-1))
\end{array}$$

By instantiating h at (e.g.) $x := 1$, we get $\mathsf{Terminates}\ \pi_1(p)(0)$. So by termination-inversion, $\mathsf{Terminates}\,p$. This means that we are licensed to use p as a logical assumption. So by $\pi_2(p)$, to get a contradiction it suffices to show that $\pi_1(p)$ terminates for all arguments y, which is easy by instantiating h at $x := \mathsf{Suc}\ y$.

On the other hand, once we have defined p we can give a separate proof that $\pi_1(p)$ *is* total and that $\pi_1(p)(x) = 0$ for all x. We reason by induction on x. In the step case, we use typechecking-time reduction with the definition of p to show

$$\pi_1(p)(\mathsf{Suc}\ x') = \pi_1(p)(x')$$

and then appeal to the IH. Now $\pi_1(p)$ is both total and not total, which is a contradiction.

Zombie has almost all the ingredients needed for this paradox. Instead of a recursively defined pair we can use a recursive function Unit $\to T$, and we can encode $\mathsf{Terminates}\ a$ as $\Sigma(y : A).a = y$. The only thing lacking is termination-inversion: using our encoding, a function $(\mathsf{Terminates}\ \pi_1(p)) \to (\mathsf{Terminates}\ p)$ would have to magically guess the second component of a pair knowing only the first component. If we assume such a function as an axiom, we can encode the paradox and derive inconsistency.[18]

The paradox uses several different features: general recursion at arbitrary types (including Σ-types), typechecking-time reduction, and termination-inversion. Since we derived false, it is clear that we cannot support all three at the same time. Since

[18] This is `Admiss2.trellys` in the test suite.

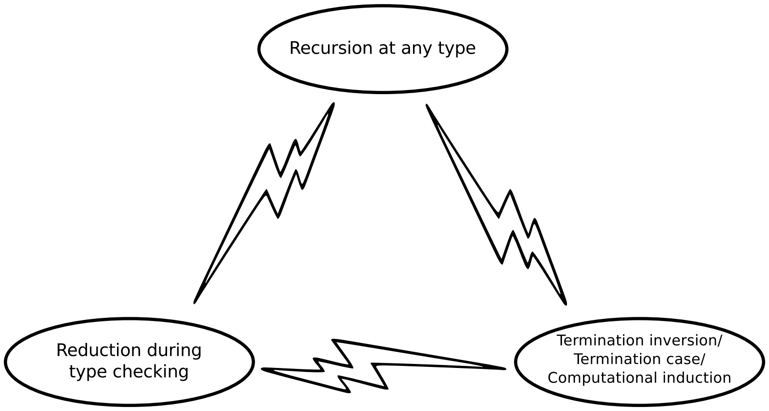
Recursion at any type
Reduction during type checking
Termination inversion/ Termination case/ Computational induction

Figure 7.13: Pick any two.

termination-case and computational induction both imply termination-inversion, we know that they also lead to inconsistency in the same way.

A priori it is not obvious whether removing some of these features will result in a consistent logic, but previous work on Nuprl [40, 123] and Hoare Type Theory [132] shows that one can get a consistent language by omitting either of the first two. Zombie completes the triple by showing that omitting termination-inversion also works. So the situation is as depicted in Figure 7.13.

Recursion at arbitrary types Nuprl's take on the problem is that the *type* of p is bad. The property "is not a total function" is not admissible, so one should not be allowed to form the fixpoint. The general recursion operator in Nuprl can only be used when the type of the function is deemed admissible, according to a conservative syntactic criterion. Notably, the problem only arises with Σ-types—all the other type constructors preserve admissibility. Crary [40] provides an expressive axiomatization of admissible types, but these conditions can lead to significant proof obligations.

From the perspective of lightweight verification, the biggest weakness of this approach is that it restricts recursion even in the programmatic fragment of the language, e.g. the pair p above was defined in P. So even a programmer who does not care about logical consistency still needs to be aware of the admissibility condition.

Compile-time reduction Hoare Type Theory instead restricts type-checking time *reduction*. There is a fixpoint combinator which can be used at any type, but in the Coq implementation it is implemented as an axiom, without any reduction rule. This is sufficient to ensure consistency [132].

Omitting reduction blocks the above paradox because p has to be treated opaquely, so there no way to prove that $\pi_1(p)(x)$ evaluates to 0. In general, this means that one can not write "external style" proofs about programmatic functions. Instead, the programmer must use Hoare-style "internal" reasoning, where the function is made to return a Σ-type stating the desired postcondition.

Termination-inversion The consistency proof in Section 7.6 (and Casinghino's proof for a larger fragment of Zombie) shows a third alternative: we can support both unrestricted recursion and compile-time reduction by not providing the termination-inversion principle.

Without termination inversion there are limitations on the external proofs we can write, as we saw in previous subsections. The rule TJoin lets us write proofs as long as they can be proved by a finite number of unfoldings. If this is not enough, we can take the same approach as Hoare Type Theory, and make the function return a Σ-type. For example, we could change the type of `eval` from Section 7.7.1 to return a proof as well as the value:

```
prog eval : (t:Tm) → Σ (t':Tm). Eval t t'
```

The lack of termination-inversion and fixpoint induction is sometimes limiting. However, this choice of features seems appealing particularly for a language targeting lightweight verification, where we do not expect very heavy proofs, but want to make sure that working in the programmatic fragment is easy.

Of course, if a function is defined in `log` we can reason about it by induction without any restrictions. The same is true if it is defined in `prog` but *could* be defined in `log` (i.e. there exists a suitable termination metric, which we can do induction over). So in this sense Zombie is strictly better than Coq and Agda, where only terminating function definitions can be written down.

7.7.2 "Non-logical" types

The current type system for Zombie is also limited in the set of features it supports for dependently typed *programming* (as opposed to theorem-proving). Currently it handles general recursive function definitions while safeguarding against logical inconsistency. But there are two other features which would be useful for the programmer but which also jepardize consistency.

Type:Type Like Coq and Agda, the current Zombie language uses a predicative hierarchy of types, indexed by levels $\ell = 0, 1, 2, \ldots$. We include typing rules stating that $\mathsf{Type}_\ell : \mathsf{Type}_{\ell+1}$, and that Type_ℓ is a subtype of $\mathsf{Type}_{\ell'}$ for $\ell \leq \ell'$. Simply dispensing with the levels and allowing $\mathsf{Type} : \mathsf{Type}$ leads to inconsistency via Girard's paradox [37, 59].

However, allowing $\mathsf{Type} : \mathsf{Type}$ is easier for the programmer than keeping track of the levels, particularly when doing type-generic programming. For example, Weirich and Casinghino [142] describe a generalization of the ordinary `zipWith` function which takes an argument a vector of types, e.g. `< Nat, Nat, Bool >`, and then expects a corresponding number of arguments, e.g.

```
(Nat → Nat → Bool) → List Nat → List Nat → List Bool
```

They note that $\mathsf{Type} : \mathsf{Type}$ would let the programmer use the ordinary vector type (which is polymorphic, parameterized over Type_0) to package this list of types, whereas Zombie would require a duplicate version of vectors parameterized over Type_1.

So for programs in the P fragment of Zombie, which is in any case not logically consistent, we would like to collapse all the type levels.

Type-level computation with general recursion Second, although the full Zombie language allows types to be defined using recursive functions and large eliminations, it only allows doing this via terminating structural recursion. At least when working in the programmatic fragment, this seems like a needless restriction: if we do not spend effort proving that the term-level definitions terminate, why should we do so for the type-level definitions? On the other hand, using generally recursive type definitions one can encode generally recursive types (including negative occurrences), so it is a source of logical inconsistency.

In both cases, the inconsistency comes from "bad types". That is, although the type looks plausible it can be used to write nonterminating expressions, so for soundness it needs to be rejected as not well-formed. On the other hand, it does not affect type safety—we know this because the core language in Chapter 3 imposes none of these restrictions and is still type safe.

Now, Zombie uses collapsed syntax, so type well-formedness is expressed by the typing judgement $\Gamma \vdash^{\theta} A : \mathsf{Type}$. In the current type system, all kinding rules provide and require $\theta \equiv \mathsf{L}$. A rather natural extension would be to add rules which admits the above forms of types as well-kinded, but at $\theta \equiv \mathsf{P}$. For example, $\mathsf{Type} : \mathsf{Type}$ can be added by a rule

$$\frac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{P}} \ \mathsf{Type}_{\ell} : \mathsf{Type}_0}$$

Similarly, while all typing rules with kinding assumptions currently require kinding at L, we would selectively relax that restriction as appropriate.

Non-logical types for non-logical programs

When considering adding non-logical types to the language, we can imagine two levels of support.The first, more tractable one, is to allow *programs* in P to have *types* in P.

For an example of how this could be useful, we need to look no further than the standard example of dependently typed programming with large eliminations: type-generic programming. Suppose we want to implement a type generic decision procedure for equality comparisons (i.e. what `deriving Eq` provides in Haskell). The steps to do this are standard: define a datatype (here, `code`) which represents the types the generic function should be able to handle (e.g. booleans and product types); provide a function (`decode`) which uses large eliminations to translates from the codes to the actual types they represent; write a type-level function (`eqdecT`) which computes the type our generic function should have when instantiated for a particular concrete type; and finally write the generic function itself (`eqdec`). Using Haskell-like syntax, and omitting unfold statements for `eqdecT`, the function definition may look as follows:

```
data code = cBool | cPair of code code | ...

prog decode : code → Type
decode cBool = Bool
decode (cProd c1 c2) = (decode c1) × (decode c2)
...

prog eqdecT : code → Type
eqdecT c = (decode c) → (decode c) → Bool

prog eqdec : (c:code) → eqdecT c
eqdec cBool = λ b1 b2 . eqdecBool b1 b2
eqdec (cProd c1 c2) = λ (a1, b1) (a2, b2) .
                          eqdec c2 a1 a2 && eqdec c2 b1 b2
```

This would be a valid function in the core language defined in Chapter 3. However, the current Zombie implementation rejects it, because the type-level function `eqdecT` is tagged as P, and the current kinding rule for arrow types require both sides of the arrow to be well-kinded at L. (In this case it is easy to make the program acceptable to Zombie by declaring the functions `decode` and `eqdecT` as `log`. But the question is whether we can allow type-level functions in `prog` in general.)

Extending the typing rules to allow this kind of program is easy in both the "possible world" and the "effect" setup. For example, in the possible-world system, we would generalize the rules for arrow types and recursive functions:

$$\frac{\Gamma \vdash^{\theta} A : \mathsf{Type} \quad \mathsf{Mobile}\,(A) \qquad \Gamma, x :^{\theta} A \vdash^{\theta} B : \mathsf{Type}}{\Gamma \vdash^{\theta} (x\!:\!A) \to B : \mathsf{Type}}\ \textsc{TArr'}$$

$$\frac{\Gamma, f :^{\mathsf{P}} (x\!:\!A) \to B, x :^{\mathsf{P}} A \vdash^{\mathsf{P}} b : B \qquad \Gamma \vdash^{\mathsf{P}} (x\!:\!A) \to B : \mathsf{Type}}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec}\, f\, x.a : (x\!:\!A) \to B}\ \textsc{TRec'}$$

By changing the kinding premise in TREC to be at P, we allow the type of the function to make use of non-logical features. (In addition, compared to the core calculus in this chapter we also have to adapt the rules for equality to permit equations between types (for `eqdecT`), as we do in full Zombie).

We conjecture that this extension would be sound, and in an effect-style system it should not make the metatheory much more difficult. This is because the effect-style system maintains the property that if a judgement concludes at L, then its subderivations are at L also, so the added kindings should not interact much with the normalization proofs.

On the other hand, in an possible-world style system it is more complicated to accomodate non-logical types. The problem is that, as we mentioned in Section 7.6.1, our proof requires a type interpretation for types of both logical and programmatic expressions. Defining a logical interpretation for, e.g., types defined by general recursion is nontrivial. So we omitted non-programmatic types from our metatheoretic proofs, and therefore also from the implementation.

Logical theorems quantifying over non-logical types

One of the nice things about Zombie is that it allows both programs returning proofs and proofs about programs. So if we extend the type system to allow functions like `eqdec`, it is natural to ask whether we can also write proofs about it. Is it possible to prove a theorem stating that the decision procedure is sound?

```
log thm : (c : code) → (a1 a2 : decode c)
          → (eqdec c a1 a2 = True) → a1=a2
```

So far we have not been able to define a type system which allows this, because the metatheory gets challenging. However, the question is interesting because it shows one way in which the “possible world”-style system seems preferable to the effect-style system.

In order to avoid all the paradoxes from non-logical types, we need to enforce that functions which check at L can only be classified by types that are well-formed at L.

This already rules out the type of thm above, because the kinding rule for arrow types at L require the domain and codomain to be well-formed at L, but decode c appears in the domain.

In an effect-style system, that is the end of the story—there is not any obvious way to fit a function like thm into the type system. However, in the possible-world-style system, there is an intriguing possibility. The @-types internalize a shift to a separate type system; so why not make the well-formedness judgement track that also?

$$\frac{\Gamma \vdash^{\theta} A : \mathsf{Type}}{\Gamma \vdash^{\mathsf{L}} A@\theta : \mathsf{Type}}\textsc{TAt'}$$

Now we can write down a type for thm, as long as we place the arguments a1 and a2 in P.

```
log thm : (c : code) → (a1 a2 : decode c @prog)
               → (eqdec c a1 a2 = True) → a1=a2
```

Although the *type* of the theorem is now well-formed, it is not immediately obvious that it is provable. Such a proof has to follow the structure of the definition of eqdec—first do a case split on c, and then in the case when it codes a pair, destruct a1 and a2, and check whether their first and second components are equal. The latter step requires a logical expression to look at the contents of a programmatic pair.

Fortunately, this is exactly what the typing rule TMobileVal allows, at least if we pick a suitable definition of the pair type:

$$A \times B \equiv \Sigma x\!:\!(A@\mathsf{P}).(B@\mathsf{P})$$

We add the @P to ensure that $A \times B$ is a mobile type (as mentioned in Section 7.3.6 above, full Zombie already treats general datatypes in general this way).

Now, in the branch where c = (cPair c1 c2) we can use TCast to change the type of a1 in the context from decode (cProd c1 c2) to (decode c1) × (decode c2) At this point we can use TMobileVal for Σ-types to treat the P variable a1 as an L variable, which allows a case-expression scrutinizing it even in an L expression. The key thing that makes this possible is that a1 is a variable, and therefore a value, so value-restricted rules like TMobileVal apply.

In summary, a natural extension of the "possible world"-style system provides a way to write logical expressions quantifying over programmatic types. This is intriguing, and in fact this was one of the main reasons we chose to pursue "possible worlds" rather than effect-style for the full Zombie language. However, we do not yet have a proof of logical consistency for the extended system.

7.7.3 Surface language concerns

We have presented two different styles of type systems, one inspired by modal logic, and one formulated as a type-and-effect system. Considered as core languages, there is no big difference between them: they can express the same programs (Section 7.5), although perhaps the modal one is more promising for future extensions (reasoning about terms of non-logical type, Section 7.7.2).

On the other hand, when designing a surface language elaborating into the current core language, we noticed several points where the system with @-types does not work smoothly, and these trouble-spots could be avoided by using an effect-style system instead.

@-types everywhere Because domains of function types and arguments to data constructors are required to be mobile, almost every type in the context is mobile. (The main exception in the current Zombie implementation is top-level definitions.) Some types (e.g. datatypes) are automatically mobile, but because of the handling of type variables, argument to polymorphic functions and parameterized types need to be tagged with an $@\theta$.

In practice, this means that variable references more often than not need to use the rule TUnboxVal to eliminate the @-qualifier. Similarly, arguments to polymorphic functions need to be checked using TBox*. So these rules can create a lot of syntactic clutter. If they were only used occasionally we could use explicit (erasable) annotations, e.g. `unbox x`, but to make programs readable it is important to be able to infer these.

Failure of local completeness Given a bidirectional type system like the one in Chapter 5, the most obvious way to infer uses of TBox* is to make them checking rules. For example, the surface-language rule corresponding to TBoxP would be:

$$\frac{\begin{array}{c}\Gamma \vdash^{\theta} a \Leftarrow A \\ \Gamma \vdash^{\mathsf{L}} A \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathsf{P}} a \Leftarrow A@\theta}\,\text{CBoxP}$$

Since this is a checking ($\Leftarrow$) rule, we do not need an explicit box-annotation on a.

However, after implementing the above rule and experimenting with writing programs using it becomes clear that it does not always work. The problem is that our Box/Unbox rules do not satisfy *local completeness.*

A deduction system is locally complete [108] if the elimination-rules are "strong enough" in the following sense: whenever there exists a derivation of a formula A, there exists a derivation ending in an introduction rule for that formula's connective.

The rules for @-types satisfy this as long as the subject is a value, but it fails in general. Suppose $\Gamma \vdash^{\mathsf{P}} a : A@\mathsf{L}$. The only @-introduction rule that could prove that formula is TBoxP, and to apply that we need to prove the premise $\Gamma \vdash^{\mathsf{L}} a : A$. But if a is a non-value, then TUnboxVal does not apply, and there is no way to derive that.

The failure of local completeness complicates typechecking, because one can not always eagerly apply the box rule. For example, given $\Gamma \vdash^{\mathsf{P}} f : \mathsf{Nat} \to (A@\mathsf{L})$ and $\Gamma \vdash^{\mathsf{P}} g : (A@\mathsf{L}) \to \mathsf{Nat}$, the expression $g\ (f\ 0)$ should be typeable. However, naively using the above checking rule gets stuck:

$$\dfrac{\Gamma \vdash^{\mathsf{P}} g \Rightarrow (A@\mathsf{L}) \to \mathsf{Nat} \quad \dfrac{\dfrac{???}{\Gamma \vdash^{\mathsf{L}} (f\ 0) \Leftarrow A}}{\Gamma \vdash^{\mathsf{P}} (f\ 0) \Leftarrow A@\mathsf{L}}\,\textsc{CBoxP}}{\Gamma \vdash^{\mathsf{P}} g\ (f\ 0) \Rightarrow \mathsf{Nat}}\,\textsc{IApp}$$

Things are still more complicated when working up-to-congruence. For example, suppose that in the above example f instead had the type $\mathsf{Nat} \to B$. Then it is not clear whether the implementation should search for a proof that $\Gamma \vDash B = (A@\mathsf{L})$, or if it should first apply TBoxP and then search for a proof that $\Gamma \vDash B = A$.

The current Zombie implementation uses a rather poorly motivated hack. When checking an expression a against an @-type $A@\theta$, we look at the syntactic form of a; if it is a variable or an application, we do not apply the box rule, but instead synthesize a type for a and try to prove that the type is CC-equivalent to either $A@\theta$ or A. The intuition is that applications and variables do not have checking rules, so there is no value in pushing the type A in and we instead synthesize straight away. This heuristic can fail in at least two ways: there may be other expression forms that would also benefit from being synthesized rather than checked; and when checking against a nested type $A@\theta_1@\theta_2$ even applications or variables really should be checked by peeling off one θ at a time.

This syntactic condition also interacts with the other typechecking rules. For example, when checking a case-expression against an @-type, we should first apply the case-rule and check each branch against the @-type, as opposed to eagerly applying the box-rule, because the branches could consist of applications/variables. On the other hand, the rules CBox should fire before ECrec, because we do not want to try to prove an @-type is equal to a function type.

To summarize, it is hard to make the box/unbox rules completely invisible in a bidirectional type system, and harder still when adding congruence closure. One appealing thing about the effect-style system is that these rules are not necessary there.

Mobile types and higher-order functions Another issue with @-types which becomes apparent when writing programs is an interaction between mobile types and higher-order functions. Consider a polymorphic higher-order function such as `map`. In order make its type well-formed, type variables must specify some θ, e.g. L:

```
map : [a b : Type] ⇒ (f : (x:a@log) → b @log) →
      (xs : List a) → List b
```

On the other hand, functions which operate on mobile types do not have to tag their arguments with @-types, which would in any case not matter. For example, the `Nat` type is mobile, so natural number addition can be given simply the type

```
plus : Nat → Nat → Nat
```

While this makes the type of `plus` less cluttered, it has an unwelcome consequence: `plus` can no longer be used as an argument to `map`. That is, in the application

```
map (plus 1) lst
```

the expression `(plus 1)` has type `Nat → Nat`, while `map` expects `(A@log) → B` for some `A` and `B`. The types do not match.

We saw an example of this problem in the DPLL-solver in Section 2.4. There, the functions `interp_lit` and `interp_clause` were used as arguments to the higher-order functions `any` and `all`, and there types had a spurious `@log` qualifying the mobile type `List`.

Semantically, this is not a big problem. As we did above, we can include subtyping rules (SubMobile1 and SubMobile2 in Figure 7.7) which express that Nat and Nat@θ are equivalent types. One could also contemplate adding a rule stating that these two types are propositionally equal, rather than just equivalent.

However, writing a typechecker for a surface language including such rules is a harder problem. In general, including subtyping in a language tends to make type checking complicated. And even the equational version is not straightforward when combined with unification-based inference. In the above example the implicit type arguments of `map` generates two unification variables X and Y, and we try to match Nat $\to$ Nat with $(X@\mathsf{L}) \to Y$. Just syntactic unification cannot solve this goal, because it will match Nat against $X@\mathsf{L}$ and the two expressions have different top-level constructors. On the other hand, the rule for @-qualified mobile types can only fire *after* the variable X has been instantiated. So the typechecker would have to interleave ordinary unification with operations which depend on the semantics of the language.

Subtyping constraints versus equality constraints In the future, one may consider moving from the current bidirectional type system (i.e. local typechecking) to a general constraint-based system (Section 6.4). Usually such systems are easier to design if they can be phrased in terms of equality constraints, because those constraints can be solved by unification. On the other hand, asymmetric rules that state that one type should subsume another require more ingenuity. In particular, naively adapting the unification technique to inequality constraints creates semi-unification problems [67], which are undecidable [73].

When creating a constraint-based type system, all the issues about inferring uses of TBox/TUnbox/SubMobile that we mentioned above would become relevant, because these features all involve "asymmetric" constraints (e.g. if the surface language includes implicit unboxing, then $A@\mathsf{L}$ is a "better" type than A). Since the effect-style system does not include these rules, it may be an easier target for elaboration.

Of course, somewhat tempering this optimism is the fact that the effect-style calculus includes a subtyping relation, which also creates asymmetrical constraints. However, these may be more tractable, for two reasons. First, it may be that subtyping is not necessary in practice. The current Zombie implementation does not implement subtyping, and it can still check our example programs. In the translation of a possible-world program that does not use subtyping, the only use of subtyping is when the subsumption rule TSub is applied to a function type, e.g.

$$\frac{\Gamma \vdash^{\mathsf{L}} f : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}}{\Gamma \vdash^{\mathsf{P}} f : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}}\ \textsc{TSub}$$

In the corresponding effect-style derivation we use subtyping to say that a function of type $\mathsf{Nat} \xrightarrow{\mathsf{L}} \mathsf{Nat} \xrightarrow{\mathsf{L}} \mathsf{Nat}$ can also be used at type $\mathsf{Nat} \xrightarrow{\mathsf{P}} \mathsf{Nat} \xrightarrow{\mathsf{P}} \mathsf{Nat}$. But in the typical case this is more than is needed; even if the function has the original L type, the application rule T_App still lets it be applied in a P context. Similarly, if f was used as an argument to a higher-order function, the programmer could work around the need for subtyping by instead η-expanding f.

Second, even if we want subtyping in the langauge, the effect-style subtyping relation $T <: T'$ is easier to handle than the possible-world style relation $A <: A'$. The difference is that while the possible-world relation includes $A@\mathsf{L} <: A$, the effect-style relation never changes the top-level constructor of a type expression, only the θ-annotations on arrows. So it should still be possible to use unification to determine the shape of type arguments, leaving only residual constraints about the values of the θs.

7.8 What was gained by the effect-style system?

To summarize, in this chapter we showed that it is possible to reformulate the existing Zombie type system to treat nontermination as an effect instead of as a modal-logic-like world, without losing any expressivity. In doing so, we gained several things.

First, the metatheory became simpler. Because typing derivations in the effect-style system do not change back and forth between L and P, we can use a simpler logical relation to show normalization (Section 7.6).

Second, the effect-style types are simpler and less cluttered. In the effect-style system, every type contains L/P annotations to fully specify its meaning. In the possible-world system the meaning of a function arrow changes when it is used in an L or a P context, and the programmer disambiguates by using @-types in the places which require fully-annotated ("mobile") types. In the small calculus in this chapter, this is a win for the possible-world system—the @-types are only needed for higher-order functions, which are comparatively rare. However, as we described in Section 7.3.6, in the full language the tradeoff changes, because for polymorphic functions all uses of type variables need to be disambiguated. Then it is more economical to place the annotation on the function arrow rather than in the surrounding context.

The possible-world calculus also has the drawback some syntactically different types describe the same sets of values, e.g. $(\Sigma x : A.B)@\mathsf{L}$ vs $\Sigma x : (A@\mathsf{L}).(B@\mathsf{L})$ or (in a larger calculus with sum types) $(A + B)@\mathsf{L}$ versus $(A@\mathsf{L}) + (B@\mathsf{L})$. The translation in Section 7.5.1 maps these to the same type, which in turn makes rest of the type system simpler (we can use the simpler case-rule TPCase instead of the "cross-fragment" rule TPCase'). And most importantly it eliminates the distinction between $\mathsf{Nat}@\theta$ and Nat, which is an annoyance in our current system (as we described in Section 7.7.3).

Finally, perhaps the biggest advantage is that in the effect-style system, there is no need for the elaborator to infer where to use the TBox/TUnbox rules. As we described in Section 7.7.3, these are difficult to handle in a principled way, and in the effect-style system they can be completely avoided.

With all these advantages, one may ask why we designed Zombie in the possible-world style in the first place. The answer, as explained in Section 7.7.2, is that the possible-world style provides a natural syntax for writing theorems that quantify over non-logical types. So far we have been unable to extend our logical consistency proof to that feature, however, so for now nothing is lost in the effect-style system.

Chapter 8

Related work

The key features of the Zombie core language are erasure, nontermination, and heterogeneuous "operational-semantics based" equality, while in the surface language the main innovation is the use of congruence closure. None of these features are completely new, but they have never before been combined in a dependently typed language. In Sections 8.1 to 8.4 of this chapter we consider related work for each of these four features in turn.

8.1 Computational irrelevance

Zombie's irrelevant arguments solve an important problem in dependently typed programming, namely how to deal with all the extra expressions needed to verify (rather than just execute) a program. Other languages use different features to address the same problem, and the differences between them can be quite subtle. This section compares the style of irrelevant arguments adopted by Zombie to the corresponding features in other languages.

8.1.1 Prop and Set in Coq

In Coq, the distinction between Prop and Set plays a similar role to irrelevant arguments. The Coq program extraction mechanism will replace any expressions whose type has kind Prop with an uninformative unit value, just like the erasure operation deals with implicit arguments in Zombie. Furthermore, by assuming the standard *proof irrelevance* axiom ("if A is a proposition, any two terms of type A are equal"), Coq users get a coarsened term equivalence at typechecking time, similar to what is provided by the use of erasure in Zombie's TJOIN rule. For example, if $a, b : A : \mathsf{Prop}$, then the equality $f a = f b$ is propositionally true by the proof irrelevance axiom.

In order to support erasure and proof irrelevance, Coq restricts pattern matches on proofs of propositions to occur only inside other proofs of propositions, exactly like pattern matches on erased terms in Zombie are only allowed inside other erased terms.

On the other hand, unlike Zombie, the proof irrelevance axiom does not help with coarsening the *definitional* equality. Adding proof irrelevance to Coq's definitional equality would break strong normalization of open terms. For example, in order to define functions by well-founded recursion, Coq programmers will encode a function $f : A \to B$ by adding an extra argument $f : (x : A) \to \mathsf{Acc}\ x \to B$, a proof that there is no infinite descending chain starting at x. The Coq typechecker will only unfold a call to f if the accessibility proof has the right syntactic form. Definitional proof irrelevance would equate all accessibility proofs, so f would unfold infinitely.

Irrelevant arguments in the style used by Zombie were partly invented as a reaction to limitations of Coq's Prop/Set distinction [90]. By tying erasability to the type of an expression, the Coq design assumes that expressions of the same type will either always be used relevantly or irrelevantly. But that is not the case; a common example is natural numbers. Obviously many programs use natural numbers in a relevant way, e.g. adding them and returning the result, but equally many programs use numbers specificationally, e.g. as indices to data structures like Vec. We would like to recognize this distinction, but with just a single Nat datatype we are forced to classify it as Set and never erase any numbers. This phenomenon pushes the Coq programmer towards duplicating datatypes. For example, in Coq the datatypes for conjunctions $A \wedge B$ and products $A \times B$ are identical except that the first is marked erasable and the second is not. Irrelevant arguments avoid this issue by leaving the relevant/irrelevant distinction to the consumer of an expression rather than the producer.

8.1.2 Irrelevant arguments in Ynot

One interesting case study, which illustrates both the extra convenience that irrelevant arguments offer beyond Coq's Prop/Set system, and also how they make the metatheory tricky, is the Ynot project. Ynot embeds a program logic into Coq by postulating appropriately typed primitives for loops and memory references. Side effects are represented by an indexed monad `Cmd P Q`, "a computation which can be run in a heap satisfying precondition P, and returns a value while leaving the heap satisfying postcondition Q". For example [80], a function to insert a value into a B+ tree can be given a the following type:

```
insert : forall (h : handle) (k : key) (v : value) (m : AssocList),
           Cmd (rep h m)
               (fun res : option value ⇒
                 rep h (specInsert v m) * [res = specLookup k m]).
```

Here `rep h m` is a predicate stating "the handle `h` points to an in-memory tree representing the same key-value mapping as the association list `m`", and the postcondition states that the tree has been updated to instead represent the association list `(specInsert v m)`.

However, giving the function the above type does not quite work. If we use Coq's program extraction to compile `insert`, then `m` will be treated as computationally relevant and kept around at runtime—in a database system implemented using B+ trees, we would be carrying around an extra copy of the entire database represented as a linked list. But the intention is that `m` should only be part of the specification, by expression the relation between the pre- and postcondition (in other words, `m` is a "ghost variable").

So some method is required to make specificational variables erasable. Ideally we would like to mark `m` as an irrelevant argument, but Coq does not offer that feature. Early versions of Ynot [95] worked around the problem by changing the type of the postcondition Q, so that instead of a unary predicate over heaps it was a binary relation between the heap before and after the function call. This way ghost variables can be replaced with existential quantification in the postcondition. However, in the experience of the Ynot implementers this led to clumsy proofs [33], and they instead decided to use unary postconditions while encoding irrelevant arguments using a new type and an axiom:

```
Inductive inhabited (A:Type) : Prop :=
  inhabits : A → inhabited A.

Axiom pack_injective : forall (T : Set) (x y : T),
  inhabits x = inhabits y → x = y.
```

The idea is that the constructor `inhabits` can inject any value into `inhabited A`. Since this type has sort Prop, the argument to `inhabits` will be erased by program extraction. So we can make `m` irrelevant by giving it the type `m : inhabited AssocList`. In the proof of the postcondition we need to use the axiom to "project out" `m` again.

Unfortunately, this axiom was eventually found to be inconsistent! If we pick $A :=$ Prop, then `inhabits` provides an injection from Prop to the type `inhabited` Prop, which itself is in Prop. This is as good as having Prop : Prop, so Girard's paradox applies.[19]

Recent versions of Ynot continue to use the axiom, but the user is admonished to be careful when using it. Since the intended effect of `inhabited` is only to guide extraction, rather than to lower the universe levels, it is possible to check that a development is sound by temporarily changing the definition of `inhabited` to the

[19] For details, see e.g. the message "Re: [Coq-Club] problem with tactic-generated terms" sent by Robert Dockins to the mailing list coq-club@inria.fr on September 12, 2014.

identity function (so `pack_injective` becomes trivially provable), and make sure that the development still compiles.

8.1.3 The Implicit Calculus of Constructions

Our treatment of implicit arguments is based on Miquel's ICC [89], with the annotation regime following ICC* [15] and EPTS [91]. However, there is one difference between ICC and Zombie, which is that we leave a placeholder to mark the site of an implicit lambda or application. That is, we have $|\lambda^{\bullet}_{x:A}.b| = \lambda^{\bullet}.|b|$ and $|a\ \bullet_b\ | = |a|\ \bullet$, while ICC* defines $|\lambda^{\bullet}_{x:A}.b| = |b|$ and $|a\ \bullet_b\ | = |a|$.

The reason we leave placeholders is to ensure that syntactic values get erased to syntactic values. Since we make type casts irrelevant this invariant is needed for type-safety [119]. For example, using a hypothetical equality h we can type the term

$$\lambda^{\bullet}_{h:\mathsf{Bool}=\mathsf{Nat}}.1 + \mathsf{true}_{\triangleright h} : \bullet(h\!:\!\mathsf{Bool} = \mathsf{Nat}) \to \ \mathsf{Nat}.$$

In Zombie this term erases to the value $\lambda^{\bullet}.1 + \mathsf{true}$. On the other hand, if it erased to the stuck expression $1 + \mathsf{true}$ then progress would fail.

We also conjecture that retaining these placeholders may simplify the metatheory. In ICC, a constant like 42 can be given not only the type Nat, but also function types like $\forall P.\forall Q.\mathsf{Nat}$, and one of the challenges in proving consistency and normalization is to find a semantics which can handle this. Although we have not developed a normalization proof, it possible that the added information in the term would help with this.

8.1.4 Pfenning-style irrelevance

A slightly different version of irrelevance was studied by Pfenning [107] and Reed [111]. They add the restriction that the bound variable x in an irrelevant arrow type $\bullet(x : A) \to \ B$ is only allowed to occur in erased positions in B. This restriction forbids types like

$$\bullet(x\!:\!\mathsf{Bool}) \to \ \mathsf{if}\ x\ \mathsf{then\ Unit\ else\ Nat}$$

which are well-formed in our language.[20]

Agda's implementation of irrelevance adopts the same restriction. The reason is that this works better with Agda's definitional equality. Agda uses a type-directed algorithm [4] to decide $\beta\eta$-equivalence (with η-laws for functions, records, unit types,

[20] ICC did not include large eliminations, and the language we studied for termination-checking [31] does not include irrelevance, so there is still no normalization proof which handles this form of types. On the other hand, in Chapter 3 we at least proved that they will not break type safety.

etc). It takes goals of the form $\Gamma \vdash a : A \equiv \Gamma' \vdash a' : A'$ containing two typed terms to be checked for equivalence, and breaks them down into smaller subgoals using rules like

$$\frac{\Gamma \vdash a : A \to B \equiv \Gamma' \vdash a' : A' \to B' \qquad \Gamma \vdash b : A \equiv \Gamma' \vdash b' : A'}{\Gamma \vdash a\ b : B \equiv \Gamma' \vdash a'\ b' : B'}$$

$$\frac{}{\Gamma \vdash a : \mathsf{Unit} \equiv \Gamma' \vdash a' : \mathsf{Unit}} \qquad \frac{\Gamma, x : A \vdash a : B \equiv \Gamma', x : A' \vdash a' : B'}{\Gamma \vdash \lambda x.a : A \to B \equiv \Gamma' \vdash \lambda x.a' : A' \to B'}$$

However, this algorithm interacts poorly with types defined by irrelevant arguments and large eliminations. Abel [4] gives the following example. Define a type constructor $A\ x \overset{\mathsf{def}}{=} \mathsf{if}\,x\,\mathsf{then\,Unit\,else\,Nat}$, and suppose we are in a context containing $B : \bullet(x : \mathsf{Bool}) \to (A\ x \to A\ x) \to \mathsf{Type}$. Then we may expect

$$\Gamma \vdash B\ \bullet_{\mathsf{false}}\ (\lambda x_{\mathsf{Nat}}.x) : \mathsf{Type} \equiv \Gamma \vdash B\ \bullet_{\mathsf{true}}\ (\lambda x_{\mathsf{Unit}}.\mathsf{tt}) : \mathsf{Type}$$

to be provable up to η and erasure—after all, the two λ-abstractions are both identity functions, on Nat and Unit respectively. However, if we apply the algorithm rules above we get first the subgoal

$$\Gamma \vdash (\lambda x_{\mathsf{Nat}}.x) : \mathsf{Nat} \to \mathsf{Nat} \equiv \Gamma \vdash \lambda x_{\mathsf{Unit}}.\mathsf{tt} : \mathsf{Unit} \to \mathsf{Unit}$$

and from there

$$\Gamma, x : \mathsf{Nat} \vdash x : \mathsf{Nat} \equiv \Gamma, x : \mathsf{Unit} \vdash \mathsf{tt} : \mathsf{Unit}$$

This last goal (comparing an arbitrary natural number x against the unit value) should certainly not be provable, or by transitivity all natural numbers would be equal. So the usual algorithm is not complete.

To summarize, it is not clear how to create an algorithm that decides η-equivalence in the presence of irrelevant arguments and large eliminations. In the usual type-directed algorithm, the types on the left and right sides of the $\equiv$ get "out of sync". Zombie does not run into this problem, because it does not support η-laws in the first place (section 3.6).

8.1.5 Intersection and union types

Intuitively, irrelevant arguments appear quite similar to intersection and union types. If we consider only runtime representation, we have the equivalences

$$A \wedge B \quad \approx \quad \bullet(x{:}\mathsf{Bool}) \to \ \mathsf{if}\,x\,\mathsf{then}\,A\,\mathsf{else}\,B$$

$$A \vee B \quad \approx \quad \Sigma\,\bullet_{x:\mathsf{Bool}}\,.\mathsf{if}\,x\,\mathsf{then}\,A\,\mathsf{else}\,B.$$

However, the typing rules for irrelevant arguments are less general than the ones for intersection/union types. In particular, consider the introduction rule for intersections:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash a : B}{\Gamma \vdash a : A \wedge B}$$

This rule can not be emulated using the corresponding irrelevant arrow type. If we start by using an irrelevant abstraction, we end up trying to prove $\Gamma, x : \mathsf{Bool} \vdash a : \mathsf{if}\, x\, \mathsf{then}\, A\, \mathsf{else}\, B$. The only way to make progress is to make a case-split on x, which is forbidden, since that would be a computationally relevant use of x.

The fact that irrelevant arguments are more restrictive seems very helpful in practice. Type inference for intersection types is difficult. Depending on what formulation one considers, union types can have problems with type preservation [13]. But irrelevant arguments are easy to implement and use, and cause no problems in the type safety proof.

8.1.6 Truncation in HoTT

Homotopy Type Theory [135] includes a *truncation* operator $\|A\|$, which builds a type where all inhabitants of A are considered (propositionally) equal. So the type $(x : \|A\|) \to B$ in HoTT is similar to an irrelevant arrow type $\bullet(x : A) \to B$, in that all function arguments are considered equal.

To support this there is a restriction on pattern matches on terms belonging to truncated types. However, the rule is more generous than the rule for irrelevant arguments. An irrelevant function can only pattern match on its argument x in irrelevant positions; this is a syntactic restriction. In HoTT, a variable x belonging to a truncated type can be pattern-matched on anywhere, but the programmer needs to supply a proof that the result of the match-expression is the same no matter what x is.

This more semantic treatment allows more programs, but unlike irrelevant arguments there is no direct connection with erasure and program extraction. One example which illustrates the difference is that in HoTT it is possible to write a function g which inhabits the type

$$\|\Sigma x : \mathsf{Nat}. f x = 0\| \to (\Sigma y : \mathsf{Nat}. f y = 0).$$

The trick is that g first destructs the provided proof that f has a zero, then tries every number $0 \leq y \leq x$ and returns the smallest one that works. Since the returned y is always the least zero of f it does not depend on x, so the pattern match is allowed. But the compiled code makes use of x at runtime, so g's input argument can not be erased.

$$\begin{array}{rcl} x & \in & \text{term variables} \\ \alpha & \in & \text{type variables} \\ \gamma & \in & \text{coercion variables} \end{array}$$

$$\begin{array}{llcl} \text{types} & A, B & ::= & \alpha \mid A = B \mid A \to B \mid A\ B \mid \forall \alpha.A \mid \forall \gamma.A \mid \ \ldots \\ \text{terms} & a, b, & ::= & x \mid \lambda x.b \mid a\ b \\ & & \mid & \lambda \alpha.b \mid a\ A \\ & & \mid & \lambda \gamma.b \mid a\ p \\ & & \mid & a_{\triangleright p} \mid \ \ldots \\ \text{coercions} & p & ::= & \gamma \mid \mathsf{refl} \mid p^{-1} \mid p ; q \mid \mathsf{inj}_i\ p \mid \mathsf{cong}_{\,A}\, p_1 \mathinner{..} p_i \mid \ \ldots \end{array}$$

Figure 8.1: Part of the GHC core language [139]

8.1.7 GHC Core

As mainstream programming languages add verification features to their type systems, they will likely have to pay more attentions to issues of erasure. One interesting example can be seen in how the core language of the Glasgow Haskell Compiler has evolved.

Early versions of GHC compiled into a core language inspired by System F. This has two syntactic categories, terms and types, which directly control erasure: term expressions have runtime representation after compilation, while type expressions do not. This is similar to e.g. Standard ML.

Adding type-level functions ("type families" in Haskell terminology) and indexed datatypes ("generalized algebraic datatypes", GADTs) required extending the core language, because these features introduce equations between syntactically different type expressions. The new core language FC (for "System F with Coercions") now has three syntactic classes: terms, types, and coercions (i.e. equality proofs).

Adapting the figures from Vytiniotis et al. [139] to use a more Zombie-like syntax, the system is as shown in Figure 8.1. The language of coercions is quite similar to the language of congruence proofs that the Zombie elaborator generates. In particular, it involves no computation: the elaborator is responsible for generating coercion *values* justifying any required type equations. This way, one never has to worry about nontermination in equality proofs, and the FC cast rule works exactly as the Zombie one, with the coercion being computationally irrelevant. So in this system it is still apparent from the syntax which parts of an expression are erased, namely types and coercions but not terms.

While the coercion language can be seen as a class of particularly tame equality proofs, Haskell also provides ways to manipulate equations as first-class objects. First, in the core language constructors of GADTs take equations are arguments (just like the parameters-only formulation of datatypes in Zombie). Second, recent versions of GHC feature "constraint kinds" [24], which lets functions return equations directly. These two features form a link between the coercion language and the term language; pattern matching on a term of GADT type will bring a coercion variable into scope. The programmer can use this to write recursive functions that compute proofs (albeit somewhat clumsily, and without termination checking) [79].

First-class equations bring up the issue that we described in Section 3.3: nontermination, laziness and erasure do not mix well. So although Haskell in general is lazy, it needs to ensure that any computation of equations gets evaluated strictly. However, adding first-class equations was not as big a change as one might think, because earlier versions of GHC already included suitable machinery in order to handle so-called unboxed integers. Unlike ordinary values of the ordinary integer types, which are always represented as pointers into the heap (pointing either to a number or to a lazy thunk), values of the of the unboxed integer type are stored directly in registers, which necessitates evaluating them strictly. The same trick is used for equations: there is a type of first-class equations, which is implemented as a wrapper around the type of unboxed equations. Coercion variables can only range over unboxed equations, so a core term which wants to use a first-class equation must first unbox it, which forces the computation [139].

8.2 Nontermination and dependent types

Throughout the design of Zombie, we have assumed that any expression may fail to terminate anywhere. Other dependent language designs make more fine-grained distinctions than this, because they distinguish between termination of (closed, compiled) terms at runtime, and of (open) terms at type-checking time. These two conditions fill quite different purposes. We need compiled expressions to terminate in order to read them as constructive proofs, for liveness guarantees, and for erasure to be a sound optimization (Section 7.1). On the other hand, we need open expressions to be strongly normalizing if we want the typechecker to reduce them in order to decide $\beta\eta$-convertibility (Chapter 5). Zombie is unusual in not enforcing termination at type-checking time. Most dependently typed languages offer facilities for writing programs that will compile into general recursion, but they limit the ways the typechecker can observe their reduction behavior.

8.2.1 Potential nontermination both at typecheck- and runtime

Like Zombie, Augustsson's language Cayenne [10] also allows nontermination everywhere. The typechecker will automatically reduce expressions in types up to some maximum number of steps (which can be specified as a global option to the type checker). Augustsson reports that this works well in practice.

Cardelli's Type:Type language [29] and ΠΣ [9] are core calculi comparable to the type system in Chapter 3. They allow $\mathsf{Type} : \mathsf{Type}$ and general recursion, but have no support for erasure or optional termination checking. Cardelli's language (like Zombie Core) does not specify what reduction strategy the typechecker should use. ΠΣ includes a novel system of "boxed expressions" classified by "boxed types", which delay reduction. By carefully using these, the programmer can e.g. encode recursive datatypes as recursive large eliminations while preventing the typechecker from getting trapped in a loop forever unfolding them.

Nuprl [35, 40, 123] has very complete support for generally recursive functions, mixing terminating and nonterminating code, and erasure. As described in Section 7.7.1, the way Nuprl handles general recursion makes slightly different tradeoffs compared to Zombie, with a more powerful induction principle but more restricted function definitions.

All the above languages are call-by-name. Zombie is call-by-value, so we adopt slightly different rules for reduction and recursion. Another difference is the techniques we use for our metatheoretic proofs. The proof of type safety for the Type:Type language uses denotational semantics, however the proof can not handle case-expressions and dependent elimination. The type safety and consistency proof for Nuprl uses a logical interpretation into PERs [66]. The Nuprl proof is in some ways more powerful than the methods we use (in Sections 3.9 and 7.6) because it also justifies η-laws and functional extensionality, but it is more difficult to set up. As far as we know, no results have been proven about Cayenne, ΠΣ, and Idris.

8.2.2 Terminating both at typecheck- and runtime

The default expectation in mainstream dependently typed languages like Coq and Agda is that all functions terminate. This is enforced by a built-in termination checker, which checks that all recursive calls are made at a structurally smaller argument [1, 58]. One drawback is that because the check is completely automatic there is no way for the programmer to provide hints to it, so over the years it tends to grow more and more complicated in order to cover as many interesting programs as possible.

As we mentioned in Section 7.2.1, for terminating functions which go beyond the kinds of structurally recursion the termination-checker can recognize, these languages also provide standard-library support for wellfounded recursion, although this is not as convenient as ordinary function definitions.

8.2.3 Terminating at runtime only

Even if the programmer is ultimately only interested in writing total functions, the requirement that all expressions normalize at typechecking time still comes at a cost. This is because some language features can create nontermination in expressions with free variables, even though every closed program would terminate.

For example, in Section 3.6 we noted that Coq and Agda treat type casts as computationally relevant for reductions in order to ensure normalizations in inconsistent contexts. However, the only closed proofs of equations are the uninformative equality proof `eq_refl`, so when compiling the program into ML or Haskell one can omit equality proofs and replace all type casts with `Obj.magic/unsafeCoerce`. In Zombie we drop the requirement that all open expression normalize, so this computational irrelevance can be reflected in the type system.

Another example is Coq's treatment of coinduction. Coq's typechecker will only reduce an expression that has a coinductive type if it is the scrutinee of a case expression. This limitation is annoying for the programmer, and also causes type preservation to fail. In the metatheory this is handled by dropping the restriction on reduction and allowing any expression to reduce, and then noting that the resulting "idealized" language enjoys type safety and logical consistency (but not normalization) [58].

8.2.4 Terminating at typechecking-time only

Many dependent languages allow writing nonterminating programs, but impose restrictions to ensure that *types* remain terminating during type-checking. One interesting question then is how much of the operational semantics of the programs is made available as equational reasoning principles for the programmer to use.

At one end of the scale, many languages forbid potentially nonterminating expressions from occurring inside types at all. They do this either by making the type language completely separate from the expression language (e.g. DML [147], ATS [146], Sep^3 [74], Ωmega [116], Haskell with GADTs [106]), or by restricting dependent application to values or "pure" expressions (e.g. DML [78], F* [133], Aura [70], Deputy [34], and Ou et al.'s language [103]).

One step up are languages where expressions from the nonterminating fragment may appear in types, but are treated opaquely by the typechecker (i.e. there are no typechecking-time reduction rules). Idris [27] permits non-total definitions, but functions that do not pass a conservative structural recursion criterion are flagged as potentially nonterminating, and will not be reduced by the typechecker. Similarly, Hoare Type Theory (implemented in the Ynot system) [95, 132] provides combinators for general recursion and for mutating memory, but no reduction rules.

In both cases, this means that the one cannot write external proofs about functions that were defined using general recursion. Programs in this system rely on the "internal" style, where an effectful function returns a Σ-type stating a postcondition. Sometimes the best way to accomplish this is to define an auxiliary pure function (which *can* be reasoned about externally), and use that to state the postcondition. For example, in Section 8.1.2 the effectful function `insert` was specified in terms of `specInsert`.

On the other end of the scale are systems that provide both a way to define recursive function and accompanying equational reasoning principles for them. For example, Bertot and Komendantsky [21] describe a way to embed general recursive functions into Coq by defining a datatype partial A that is isomorphic to the usual Maybe A but is understood as representing a lifted CPO $A_{\perp}$, and then use classical logic axioms to provide a fixpoint combinator fixp. When defining a recursive function the user must prove continuity side-conditions. Since the recursive functions are defined non-constructively they can not be reduced directly (the evaluation gets stuck on the axiom), but it is possible to prove a a fix-point equation as a propositional equality.

One of the most popular approaches to embedding generally recursive programs into total dependent languages is by corecursion. Capretta [28] defined a coinductive type for computations:[21]

```
CoInductive Computation (A:Set) : Type :=
| return : A → Computation A
| step : Computation A → Computation A.
```

Values of type `Computation A` are either finite computations `(step (step ...step (return a)))`, or the infinitely stepping term `(step (step ...))`. Capretta showed that it is possible to implement a CBV-style fixpoint operator and a monadic bind:

$$\begin{array}{lcl} Y & : & (A \to \mathsf{Computation}\ B) \to A \to \mathsf{Computation}\ B \\ \mathit{bind} & : & (A \to \mathsf{Computation}\ B) \to \mathsf{Computation}\ A \to \mathsf{Computation}\ B \end{array}$$

In practice, programmers do not typically use Y to construct recursive programs; instead they are directly written as Coq/Agda functions returning Computation A, and the productivity rules for corecursive functions allow recursive calls as long as they are guarded by enough step constructors (e.g. Danielsson [41]). To ensure that

[21] For uniformity, all the code examples in this subsection have been re-written in Coq, although some were originally in Agda.

functions are deemed productive even if they make non-tail recursive calls, it can be better to make `bind` one of the constructors of the datatype instead of `step` [87].

Using the intuition that expressions of coinductive type are like processes that respond to observations, the above encoding can be read as transforming a potentially infinite computation into an interactive object: each time you prod it, it promptly responds either "I'm done" (`return`) or "I need more time" (`step`). So this use of coinduction fits well with the usual uses of coinduction to model reactive systems like web servers or operating systems. The pattern matching rules for coinductive expressions in Coq and Agda allow any finite observation, but do not support induction. This is similar to the reasoning principles provided by Zombie—a Zombie program in P can be unfolded a finite number of steps using `join`, but there is no support for fixpoint induction/computational induction.

The coinductive approach also scales to more effects than just nontermination. Setzer and Hancock [114] and Setzer [113] propose to parameterize the monad by a set of commands and responses:

```
CoInductive IO (C : Set) (R: C → Set) (A : Set) : Type :=
| do : (c : C) → (f : R c → IO C R A) → IO C R A
| return : (a : A) → IO C R A.

Inductive ConsoleCommands : Set :=
| putStrLn : String → ConsoleCommands
| getLn : ConsoleCommands.

Definition ConsoleResponses (c : ConsoleCommand) : Set :=
  match c with
  | puStrLn s ⇒ Unit
  | getLn ⇒ String
  end.

Definition IOConsole : Set → Set :=
  IO ConsoleCommands ConsoleResponses.
```

Just like in Haskell, this conception of IO can be viewed from two perspectives. A function of type $A \to \mathsf{IO}\ B$ can be viewed both as an effectful function from A to B, or as a pure function computing a value of type $\mathsf{IO}\ B$. In this case one can even write some external proofs about programs by pattern-matching on values of the IO type, e.g. to see what the first issued command is. Of course, the values of type Computation/IO that are manipulated at type-checking time are only to be considered a mathematical model of program behavior. When compiling the program, they should be replaced with native facilities which do not allocate memory for step and which actually print to the console.

Compared to Zombie's direct support of general recursion, the coinductive approach favors total functions and treats nontermination less conveniently. Nonterminating programs must be written using monadic combinators (and are therefore never syntactically equal to pure programs). The use of combinators is even more inconvenient in a dependent language than in a simply-typed one. For simple types, monads and type-and-effect systems are exactly equivalent [141], with the bind combinator corresponding to applying an effectful function, but in a dependent system there is no direct way to "apply" a dependent function to a monadic value. So theorems about effectful programs instead need to explictly propagate the type dependency using a predicate transformer like `Lift` [41]:

```
CoInductive Lift (P : A → Prop) : Computation A → Prop :=
  ...

Lemma bindCong : Lift P x → (forall (x:A), P x → Lift Q (f x))
                → Lift Q (x >>= f).
```

The $\mathsf{Computation}$ monad provides recursive function definitions but not general recursive types. Our modal-style type system [31] can support general μ-types in the P fragment.

Finally, the coinductive approach requires a separate notion of equivalence to reason about partial programs. In, e.g., Coq, one would compare pure expressions according to the standard operational semantics, but define a coarser equivalence relation for partial terms that ignores the number of steps they take to normalize. Equations like $((\mathsf{rec}\, f\ x.b)\ v) = \{v/x\}\, \{\mathsf{rec}\, f\ x.b/f\}\, b$ do not hold with the usual Coq equality because the step counts differ. Conveniently programming with equivalence relations like this, which are not directly justified by the reduction behavior of expressions, is an active area of research involving topics such as setoids [16], extensional equality for coinductive types [85], quotient types, and the univalence axiom [135].

8.3 Propositional equality

The usual equality type in Coq and Agda's standard libraries is homogenous and has a computationally relevant elimination rule. These languages also provide the heterogenous JMeq [84], which we discussed in Section 3.6.

Extensional Type Theory, which is the basis for Nuprl [35], is similar to our language in that conversion is computationally irrelevant and completely erased. ETT terms are similar to our unannotated terms, while our annotated terms correspond to ETT typing derivations. (Unlike the usual presentation of ETT, however, our $\mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a=b}$ term contains step-counts, so derivations can involve many steps of computation without the annotated term growing very big. This is important to be able to efficiently implement so-called proof by reflection.) On the other hand, the equational theory of ETT is different from our language, in particular it can prove extensionality while our equality cannot.

Guru [128], like our language, is "very heterogeneous" (one can eliminate equalities where the two sides have different types), and equalities are proved by joinability without any type-directed rules. However, unlike our language the equality formation rule does not require that the equated expressions are even well-typed. This can be annoying in practice, because simple programmer errors are not caught by the type system. Guru does not have our n-ary congruence rule TJSUBST.

GHC Core [131, 139] is similar to our core language in not having a separate notion of definitional and propositional equality. Instead, all type equivalences—which are implicit in Haskell source—must be justified by the typechecker by explicit proof terms. The Haskell type checker builds these proofs by constraint-solving rules which automatically use assumptions in the context, similar to our automatic use of congruence closure.

8.3.1 Propositional equality and congruence closure

The idea of using congruence closure is not limited to this particular version of propositional equality. Below, we discuss how the nonstandard features of Zombie's equality interact with congruence closure and suggest how the algorithm could be adapted to other settings.

Our equality is *heterogeneous*, but congruence closure will work just as well with a conventional homogeneous equality. In fact, in one way a conventionally typed equality would work better, because if would allow a more expressive congruence rule. In first-order logic, a term is either an atom or an application, so there is just a single congruence rule, the one for applications. One might expect that our relation would have one congruence rule for each syntactic form (i.e. for $a = b$ and $(x : A) \to\ B$ and $\mathsf{rec}\ f\ x.a$ etc). However, we do not do that, because it would lead to problems for terms with variable-binding structure. For those, one would expect the congruence rules to go under binders, e.g.:

$$\frac{\Gamma, x : A \vDash b = b'}{\Gamma \vDash (\lambda x_A.b) = (\lambda x_A.b')}$$

However, adding this rule is equivalent to adding functional extensionality, which is not compatible with our "very heterogeneous" treatment of equality (Section 3.6.1). Instead we adopt the rule TCCCONGRUENCE, which is phrased in terms of substitution. This rule in particular subsumes the usual congruence rule for application, but it additionally allows changing subterms under binders, as long as the subterms do not mention the bound variables.

Second, we use an *n-ary congruence rule*, while most theories only allow eliminating one equation at a time. For congruence closure to work equality must be a congruence, e.g. given $a = a'$ and $b = b'$ we should be able to conclude $f\ a\ b = f\ a'\ b'$. Our n-ary rule supports this in the most straightforward way possible. An alternative (used in some versions of ETT [40]) would be to use separate n-ary congruence rules for each syntactic form. Systems that only allow rewriting by one equation at a time require some tricks to avoid ill-typed intermediate terms (e.g. Bertot and Castéran [20] Section 8.2.7).

Finally, because elimination of propositional equality is *erased*, equations like $a_{\triangleright b} = a$ are considered trivially true. Having such equations available is important, because the elaborator inserts casts automatically, without detailed control by the programmer. In Coq that would be problematic, because an inserted cast could prevent two terms from being equal. However, making the cast erasable is not the only possible approach. For example, in Observational Type Theory [8] the casts are computationally relevant but the theory includes $a_{\triangleright b} = a$ as an axiom. In that system one can imagine the elaborator would use the axiom to make the elaborated program type-check.

8.3.2 Stronger equational theories

The theory of congruence closure is one among a number of related theories. One can strengthen it in various ways by adding more reasoning rules, in order to get a more expressive programming language. However, doing so may endanger type inference, or even the decidability of type checking.

One obvious question is whether we could extend the relation $\Gamma \vDash a = b$ to do both congruence reasoning and β-reduction at the same time. Unfortunately, this extension causes the relation to become undecidable.

This is clearly the case in our language, which directly includes general recursive function definitions. But even if we allowed only terminating functions, the combination of equality assumption and lambdas can be used to encode general recursion. For example, reasoning in a context containing

```
f : Nat → Nat
h : f = (λx. if (even x) then f (n/2) else f (3*n+1))
```

is equivalent to having available a direct recursive definition

```
f x = if (even x) then f (n/2) else f (3*n+1)
```

There may exist restricted versions of the problem that are still decidable. For example, Altenkirch [7] conjectured that the equational theory of β+congruence is decidable as long as all assumed equations are between first-order values (formed from pairs and constants).

Another natural generalization of the equality relation is to allow *rewriting by axiom schemes*, i.e. instead of only using ground equations $a = b$ from the context, also instantiate and use quantified formulas like $\forall xyz.a = b$. In general this generalization (the "word problem") is also not decidable, e.g. it is easy to write down an axiom scheme for the equational theory of SKI-combinators. However, there are semi-decision procedures such as *unfailing completion* [12] which form the basis of many automated theorem provers. Such provers can achieve completeness by systematically deriving all consequences of a theory in a saturating way. They therefore tend be good at "small but deep" problems.

Other automatic theorem provers give up completeness and instead treat axiom schemes heuristically. In particular, SMT solvers such as Z3 [44] manage the instantiation of formulas through *triggers*. A trigger is a set of patterns, i.e. first-order terms with free variables, associated with a particular quantified formula. When a new fact is added, the solver checks whether it matches a trigger pattern, and when all patterns are matched the corresponding formula is instantiated and added as a fact. Because the reduction behavior of functions can be axiomatized, such systems can to some extent also be used to reason up to β-equivalence; for example the Dafny programming language [76] uses an SMT-solver backend for verification, and compiles functions into first-order axiom schemes.

However, choosing what parts of a formula should be its trigger can be subtle. For example (taken from [93]), consider the following typical axiomatization of a pair constructor and its two selector functions:

$$\forall xy.\mathsf{fst}(\mathsf{pair}(x, y)) = x \land \mathsf{snd}(\mathsf{pair}(x, y)) = y$$

If we consider this as a general reasoning principle about pairs, we might add the trigger $\{\mathsf{pair}(x, y)\}$. In that case, an assumption like $\mathsf{pair}(0, a) = \mathsf{pair}(1, a)$ will trigger it, eventually yielding $0 = 1$ through congruence closure. On the other hand, if we consider the formula as the definition of the function fst it is natural to use the trigger $\{\mathsf{fst}(\mathsf{pair}(x, y))\}$, in which case the trigger would not fire. Overly general triggers hurts performance by filling up the database with useless facts, while overly specific triggers miss useful reasoning. Modern SMT solvers typically supports both automatically guessing a suitable trigger, or allowing the user to specify the triggers of an axiom explicitly. Axiomatizing a theory in a way that allows the most effective triggers is something of an art [93], with a similar flavor to logic programming.

Even when preserving decidability and completeness one can still extend congruence closure to know about specific axioms schemes, such as for natural numbers with successor and predecessor [97] or lists [96] or injective data constructors [39].

Clearly one could design a programming language around a more ambitious theory than just congruence closure. Many languages, such as Dafny [76] and Dminor [22] call out to an off-the-shelf theorem prover in order to take advantage of all the theories that the prover implements. One reason we focus on a simple theory is that it makes *unification* easier, which seems to offer promising avenues for future work on type inference. As we describe in Chapter 6, unification modulo congruence closure (rigid E-unification) is NP-complete and algorithms for it have been studied. Unification modulo other equational theories (E-unification) must be handled on a theory-by-theory basis, and it is not an operation exposed by most provers.

8.4 Congruence closure

There is a large literature about the theory and applications of congruence closure. Two topics in particular are related to its use in Zombie.

8.4.1 Simplifying congruence proofs

Our evidence simplification rules are quite natural, and in fact the same rules has been studied before for a different reason. For efficiency, users of congruence closure want to make proofs as small as possible by taking advantage of simplifications like $\mathsf{refl}; p \mapsto p$ or $p^{-1}; p \mapsto \mathsf{refl}$.

Stump and Tan [127] already described the first 11 simplification rules in Figure 5.9 (the ones to do with just reflexivity, symmetry, and transitivity). They go further than we do and also prove that these rules form a convergent rewrite system, so applying them in any order will produce a normal-form evidence term. By contrast, we only proved that the particular simplification strategy that the Zombie implementation uses produces normal-form terms.

The CongTrans simplification rule, which is key to producing well-typed equality proofs, has also been studied in the context of optimizing proofs for size. There, the issue is that uses of cong can block the other simplification rules. De Moura et al. define the same CongTrans rule and give the following example [45]. Given assumptions $h_1 : a = b, h_2 : b = d, h_3 : c = b$, consider the proof term

$$(\mathsf{cong}_f\,(h_1; {h_3}^{-1})); (\mathsf{cong}_f\,(h_3; h_2)) : f a = f d$$

We can get rid of the assumption h_3 by doing the rewrite

$$(\mathsf{cong}_f\,(h_1; {h_3}^{-1})); (\mathsf{cong}_f\,(h_3; h_2)) \mapsto \mathsf{cong}_f\,(h_1; {h_3}^{-1}; h_3; h_2).$$

They note that it is always beneficial to apply the rule if one of the sub-proofs of cong uses strictly fewer assumptions than the other.

8.4.2 Dependent programming with congruence closure

CoqMT [126] aims to make Coq's definitional equality stronger by including additional equational theories, such as Presburger arithmetic, so that for example the types $\mathsf{Vec}\ 0$ and $\mathsf{Vec}\ (n \times 0)$ can be used interchangeably. The prototype implementation only looks at the types themselves, but the metatheory also considers using assumptions from the context. This is complicated because CoqMT still wants to consider types modulo β-convertibility, and in contexts with inconsistent assumptions like $\mathsf{true} = \mathsf{false}$ one could write nonterminating expressions. Therefore CoqMT imposes restrictions on where an assumption can be used. Unfortunately, the restrictions also rule out interesting programs, for example using reduction modulo when evaluating types defined by type-level computation.

VeriML makes the definitional equality user-programmable [124], and as an example builds a "stack" combining congruence closure, β-reduction, and potentially other theorem proving.

Neither CoqMT or VeriML prove that their implementation is complete with respect to a declarative specification. For example, the VeriML application rule requires that the applied function has the type $T \to T'$ and then checks that T is definitionally equal to the type of the argument, but there is no attempt to also handle declarative derivations which require definitional equality to create an arrow type.

The Guru language includes a tactic `hypjoin` [105] similar to our `smartjoin` and `unfold`. However, instead of using equations from the context, the programmer has to write an explicit list of equations, and unlike `unfold` it normalizes the given equations.

Chapter 9

Conclusion and future work

We set out to develop a dependently typed language for lightweight verification. That goal led us into some previously unexplored territory, and we found several new and interesting things:

- We proved type safety for a dependently typed language with both **nontermination** and **erasure** (Chapter 3). Combining the two shows the places where assurances of termination are needed to ensure type safety, including a rather subtle interaction between erasure and error-effects (Section 4.1.5).
- One of the most novel parts of the core language is the treatment of **propositional equality**, which is erasure-based, weakly typed, and heterogeneous. This makes quite different tradeoffs from other dependently typed languages: it omits type-directed reasoning such as functional extensionality, but in return it makes it much easier for the programmer to use equations (Section 3.6). Another benefit is that this formulation makes the metatheory simple, since equality types can be interpreted as just parallel-reduction-joinability of closed expressions (Sections 3.9 and 7.6). And because we have irrelevant elimination and n-ary elimination, our equality works well with automatic theorem proving algorithms such as congruence closure (Section 8.3.1).
- We showed how to design a dependent type system around **congruence closure** instead of $\beta\eta$-equivalence (Chapter 5). This is a quite unconventional setup, particularly suitable for programs that make heavy use of general recursion and comparatively light use of equational reasoning. Adapting congruence closure from untyped first-order logic to dependently typed terms required several new ideas, both for the specification of the congruence relation (Section 5.2) and for the algorithm to implement it (Section 5.5).
- And finally, we studied several ways to combine optional **termination checking** with the nonterminating language, without treating nonterminating as a

second-class citizen (Chapter 7). The particular formulation used in Zombie makes a novel tradeoff which omits termination-inversion in order to support both typechecking-time reduction and general recursion without side conditions (Section 7.7.1).

9.1 How close are we to a language for lightweight verification?

The ideal we aspire to is that a programmer should be able to start out programming just as in a conventional functional programming language, and then gradually evolve the program to include more formal verification where appropriate. So we should compare Zombie to both existing functional and existing dependent languages.

When compared to functional languages, Zombie comes close. It has essentially all the features in the Haskell98 core language Of course, the Haskell *surface* language has various other features (e.g. typeclasses), but supporting those require no theoretical advances, only implementation effort.

Compared to Standard ML the gap is slightly bigger, because Zombie does not have mutable references or catchable exceptions. Other languages such as F* and Deputy are able to support those by only providing "value dependent" or "purely dependent" types (Section 4.2). Similar ideas could probably be used to design a dialect of Zombie with memory effects.

On the other hand, if we compare program verification in Zombie to similar developments in Coq or Agda (Chapter 2), the developments in Zombie are more tedious. Our decision to omit β-reduction completely from the surface language (Chapter 5) is well-behaved theoretically, but results in verbose programs. In order to make verification more pleasant, it would probably be worthwhile to introduce some automatic β-reductions also. The experience with Cayenne [10] was that even a simple scheme with a global cut-off for how many steps the typechecker takes was enough to typecheck many practical programs. The drawback of such a system would be that there is no simple declarative specification of which programs are well-typed.

9.2 Future work and future impact

As we have seen throughout the thesis, there are various loose ends and unanswered questions which could be tidied up. For example: Can we find a different declarative presentation of the surface language which gets rid of the injrng restriction (Section 5.4)? Can we define unification-based type inference as a constraint problem

(Section 6.4)? Can we integrate stronger forms of induction (Section 7.2.1)? Can we support non-logical types in the programmatic fragment, and can we provide ways to externally reason about such programs (Section 7.7.2)? Would elaboration work better with an effect-style type system (Section 7.7.3)?

Further in the future, it would be interesting to incorporate more ambitious automatic theorem proving. Congruence closure is perhaps the simplest equational theory, so it is interesting to what it is like to program using it alone. But as we describe in Section 8.3.2 there are stronger theories. It would be particularly interesting to see if one could add restricted versions of β-convertibility which would preserve decidability while still convering the most used cases. Existing languages like Coq sometimes do "too much" β-reduction, and provide various features to *prevent* expressions from reducing (both for performance, and because proof-construction tactics like `auto` do not respect β), so having more detailed control over reduction could be welcome. Similarly, unification modulo congruence (Chapter 6) could useful not only for type inference, but also for automatic proof search.

We also hope that this research will be useful as a building block for programming language designers in the future. This could be both when adding more support for functional programming to dependent languages, or more support for dependent types to functional languages.

In the first direction, for example Idris [27] is a dependently typed language which aims to be useful for practical programming. Like Zombie, Idris allows the programmer to write general recursive programs directly. However, unlike Zombie the Idris typechecker will never reduce non-total functions. This choice is mainly because the implications of nontermination were poorly understood, and treating these expressions opaquely is the conservative choice.[22] The metatheoretic proofs for Zombie (Sections 3.9 and 7.6) show that typechecking-time reduction can be added the a language without endangering desirable properties like type safety and consistency, and could embolden future language designers.

For an example in the other direction, the last decade has seen more and more features inspired by dependent type systems added to Haskell, and that trend seems set to continue. The Haskell typechecker automatically makes use of equality assumptions from the context to typecheck programs. Although as far as we know nobody has characterized exactly how the Haskell constraint solving rules compare to the congruence closure problem, the two problems seem quite similar. As Haskell adds more and more dependently typed features (such as first-class equations), we expect that our adaption of congruence closure to a dependently-typed setting (Chapter 5) will become more relevant.

[22] Edwin Brady, personal communication.

Appendix A

Proofs related to Chapter 5

A.1 Assumptions

A.1.1 Assumptions about the annotated core language

The following properties of the core language were proved in our prior work [120], so in this paper we assume them without proof.

Assumption **47** (Weakening for annotated language)**.** If $\Gamma \vdash a : A$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash a : A$.

Assumption **48** (Strengthening for annotated language)**.** If $\Gamma, \Gamma' \vdash b : B$ and $\mathsf{FV}\,(b) \subseteq \mathsf{dom}\,(\Gamma)$, then $\Gamma \vdash b : B$.

Assumption **49** (Inversion for type well-formedness)**.**

1. If $\Gamma \vdash a = b : C$, then $\Gamma \vdash a = b : \mathsf{Type}$ and there exists A and B such that $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$.
2. If $\Gamma \vdash (x : A) \to B : C$, then $\Gamma \vdash (x : A) \to B : \mathsf{Type}$ and $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma, x : A \vdash B : \mathsf{Type}$.
3. If $\Gamma \vdash \bullet(x : A) \to B : C$, then $\Gamma \vdash \bullet(x : A) \to B : \mathsf{Type}$ and $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma, x : A \vdash B : \mathsf{Type}$.

Assumption **50** (Substitution for fully-annotated language)**.** If $\Gamma, x : A \vdash b : B$ and $\Gamma \vdash v : A$, then $\Gamma \vdash \{v/x\}\, b : \{v/x\}\, B$.

Assumption **51** (Regularity for fully-annotated language)**.**

If $\vdash \Gamma$ and $x : A \in \Gamma$, then $\Gamma \vdash A : \mathsf{Type}$.

If $\Gamma \vdash a : A$ then $\vdash \Gamma$ and $\Gamma \vdash A : \mathsf{Type}$

A.1.2 Algorithmic congruence closure relations

Next we specify what assumptions we make about the congruence closure algorithm. Calls to it are represented as judgements:

- $\Gamma \vDash A =^? (x : A') \to B' \leadsto v$
- $\Gamma \vDash A =^? [x : A'] \to B' \leadsto v$
- $\Gamma \vDash A =^? (A' = B') \leadsto v$
- $\Gamma \vDash A \overset{?}{=} B \leadsto v$

Here, A and B are inputs, while A', B' and v are outputs. For example, $\Gamma \vDash A =^? (x : A') \to B' \leadsto v$ means "find A' and B' such that $\Gamma \vDash A = (x\!:\!A') \to\ B'$, and a v such that $\Gamma \vdash v : A = ((x\!:\!A') \to\ B')$". Note that the judgement $\Gamma \vDash A =^? A' \to B' \leadsto v$ is syntactic sugar for the dependently-typed version, $\Gamma \vDash A =^? (x : A') \to B' \leadsto v$.

By using the congruence closure algorithm presented in Section 5.5 these relations can be straightforwardly implemented: one constructs the congruence closure of all equations in the context, and then checks whether the equivalence class of A contains any members with the right form (and return the first one found if there are several). Note that this algorithm will give the same answer for two inputs which are in the same equivalence class (but with a different proof v). We formalize that observation as the following assumption.

***Assumption* 52.** (Respects CC) If $\Gamma \vDash A = B$

- $\Gamma \vDash B \overset{?}{=} C \leadsto v_1$ then $\Gamma \vDash A \overset{?}{=} C \leadsto v_2$.
- $\Gamma \vDash B =^? (x : C_1) \to C_2 \leadsto v_1$ then $\Gamma \vDash A =^? (x : C_1) \to C_2 \leadsto v_2$.
- other forms of types

Furthermore, the algorithm can also generate terms in the core language that prove that the required equation holds. We write this as $\Gamma \vDash A =^? (x : B_1) \to B_2 \leadsto v$, etc. It is also convenient to specify that the inferred proof always erases to just join. That is, we assume the following interface.

***Assumption* 53.** (CC soundness for function types) If $\Gamma \vDash A =^? (x : B_1) \to B_2 \leadsto v$, then $\Gamma \vdash v : A = ((x\!:\!B_1) \to\ B_2)$ and $|v| = \mathsf{join}$ and $\Gamma \vDash A = ((x\!:\!B_1) \to\ B_2)$.

Similar assumptions are required for the other versions of the relation.

Finally we use the elaborating relation $\Gamma \vDash A \overset{?}{=} B \leadsto v$, which decides whether $\Gamma \vDash A = B$ (both A and B are inputs), and if so produces a core proof term v for the equation.

***Assumption* 54.** (CC soundness) If $\Gamma \mapsto A \overset{?}{=} B \rightsquigarrow v$, then $\Gamma \vdash v : A = B$ and $|v| = \mathsf{join}$ and $\Gamma \vDash A = B$.

***Assumption* 55.** (CC completeness) If $\Gamma \vDash A = (x\!:\!B_1) \to B_2$ then there exists a $(x\!:\!B_1') \to B_2'$ and v such that $\Gamma \mapsto A =^? (x : B_1) \to B_2 \rightsquigarrow v$ succeeds.

Similar assumptions are required for the other versions of the relation.

A.2 Proofs about the congruence closure relation

A.2.1 Properties of typed congruence closure relation

$$\frac{|a| = |b|}{\Gamma \vdash a = b}\textsc{CCrefl} \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash b = a}\textsc{CCsym} \qquad \frac{\Gamma \vdash a = b \quad \Gamma \vdash b = c}{\Gamma \vdash a = c}\textsc{CCtrans}$$

$$\frac{\begin{array}{l} x : A \in \Gamma \\ \Gamma \vdash A = (a = b) \end{array}}{\Gamma \vdash a = b}\textsc{CCassumption} \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash \{a/x\}\, c = \{b/x\}\, c}\textsc{CCcongruence}$$

$$\frac{\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vdash A_1 = B_1}\textsc{CCinjdom} \qquad \frac{\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vdash A_2 = B_2}\textsc{CCinjrng}$$

$$\frac{\Gamma \vdash (\bullet A_1 \to\ B_1) = (\bullet A_2 \to\ B_2)}{\Gamma \vdash A_1 = B_1}\textsc{CCiinjdom} \qquad \frac{\Gamma \vdash (a_1 = a_2) = (b_1 = b_2)}{\Gamma \vdash a_k = b_k}\textsc{CCinjeq}$$

$$\frac{\Gamma \vdash (\bullet A_1 \to\ B_1) = (\bullet A_2 \to\ B_2)}{\Gamma \vdash A_2 = B_2}\textsc{CCiinjrng}$$

Figure A.1: Untyped congruence closure

This subsection gives the proofs for the results described in Sections 5.5.1 and 5.5.3. The main result is a theorem relating the typed congruence closure relation $\Gamma \vDash a = b$ with an untyped variation $\Gamma \vdash a = b$. The latter is defined in Figure A.1.

***Definition* 56** (Injective labels)**.** We define the judgement $F\,\mathsf{injective}$ to mean that F is one of $- \to -$ or $\bullet\!- \to\ -$ or $- = -$.

***Lemma* 57** (Weakening for congruence closure)**.** If $\Gamma \vDash a = b$ and $\vdash \Gamma, \Gamma'$, then $\Gamma, \Gamma' \vDash a = b$.

Proof. Easy induction on $\Gamma \vDash a = b$. All cases except TCCAssumption are direct by the IH. □

***Lemma* 58** (Regularity for congruence closure)**.**
If $\Gamma \vDash a = b$ then $\Gamma \vdash a = b : \mathsf{Type}$.

Proof. Induction on $\Gamma \vDash a = b$. The cases are:

TCCrefl, TCCcongruence, TCCerasure These rules have a typing assumption which proves $\Gamma \vdash a = b : \mathsf{Type}$.

TCCsym,TCCtrans Direct by IH.

TCCinjrng By the IH, we get that $\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)$. Applying kinding inversion (lemma 49) twice we find $\Gamma \vdash A_1 : \mathsf{Type}$ and $\Gamma, x : A_1 \vdash B_1 : \mathsf{Type}$, and similarly for A_2 and B_2. Since x is not free in B_1 (this is a simple type), by strengthening (lemma 48) we know $\Gamma \vdash B_1 : \mathsf{Type}$. Similarly, $\Gamma \vdash B_2 : \mathsf{Type}$. So we have $\Gamma \vdash B_1 = B_2 : \mathsf{Type}$ as required.

TCCinjdom,TCCiinjdom,TCCiinjrng,TCCinjeq Similar to the previous case.

□

We define the judgement $\Gamma \vdash^{\llcorner} p : a = b$ ("p is evidence that $\Gamma \vdash a = b$") in the obvious way, by adding evidence terms to each inference rule in the definition in $\Gamma \vdash^{\llcorner}\ a = b$. The resulting rules are shown in Figure A.2. The grammar of evidence terms (which was also shown in the main paper) is as follows

$$p, q \quad ::= \quad x_{\triangleright p} \mid \mathsf{refl} \mid p^{-1} \mid p; q \mid \mathsf{inj}_i\ p \mid \mathsf{cong}_A\, p_1 \mathinner{..} p_i$$

Note that the notation $^{-1}$ (symmetry) and ; (transitivity) are simply syntactic constructors of evidence terms, as opposed to functions operating on evidence terms.

***Lemma* 59** ($\Gamma \vdash^{\llcorner} p : a = b$ is deterministic)**.** If $\Gamma \vdash^{\llcorner} p : a = b$ and $\Gamma \vdash^{\llcorner} p : a' = b'$, then $a = a'$ and $b = b'$.

Proof. Simple induction on p. We implicitly assume that Γ only has one binding for any given variable. □

The evidence simplification relation $p \mapsto q$ was already shown in Figure 5.9, but we repeat it in Figure A.3 in order to give names to the rules so we can conveniently refer to them. We write $\mapsto^*$ for the transitive closure of $\mapsto$.

***Lemma* 60.** If $\Gamma \vdash^{\llcorner} p : a = b$ and $p \mapsto q$, then $\Gamma \vdash^{\llcorner} q : a = b$.

Proof. Induction on on $p \mapsto q$, then do inversion on the assumption $\Gamma \vdash^{\llcorner} p : a = b$.

$$\frac{}{\Gamma \vdash^{\mathrm{L}} \mathsf{refl} : a = a}\textsc{CCPrefl} \quad m\frac{\Gamma \vdash^{\mathrm{L}} p : a = b}{\Gamma \vdash^{\mathrm{L}} p^{-1} : b = a}\textsc{CCPsym} \quad m\frac{\begin{array}{l}\Gamma \vdash^{\mathrm{L}} p : a = b \\ \Gamma \vdash^{\mathrm{L}} q : b = c\end{array}}{\Gamma \vdash^{\mathrm{L}} p; q : a = c}\textsc{CCPtrans}$$

$$\frac{\begin{array}{l}x : A \in \Gamma \\ \Gamma \vdash^{\mathrm{L}} p : A = ((- = -)\ a\ b)\end{array}}{\Gamma \vdash^{\mathrm{L}} x_{\triangleright p} : a = b}\textsc{CCPassumption} \qquad \frac{\forall k.\ \ \Gamma \vdash^{\mathrm{L}} p_k : a_k = b_k}{\Gamma \vdash^{\mathrm{L}} \mathsf{cong}\, F\, \overline{p_k}^{\,k} : F\, \overline{a_i} = F\, \overline{b_i}}\textsc{CCPcong}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathrm{L}} p : F\, \overline{a_i} = F\, \overline{b_i} \\ F\, \mathsf{injective}\end{array}}{\Gamma \vdash^{\mathrm{L}} \mathsf{inj}_i\ p : a_i = b_i}\textsc{CCPinj}$$

Figure A.2: (Untyped, labelled) congruence closure, tracking the evidence terms

In InvTrans1 and InvTrans2, we use lemma 59 to know that the two occurrences of p prove the same equation. □

Next, we define a syntactic class of of *fully simplified evidence term*, as follows.We define grammars for *synthesizable* term pS, *checkable* terms pC, and *chained* terms p^* (containing zero or more ps—an empty chain denotes the term refl, and a nonempty chain denotes a sequence of right-associated uses of transitivity $p_1; (p_2; (\ldots; p_n)))$. The metavariable p^*_{LR} ranges over chains that begin and end with a synthesizable term (as opposed to an empty chain or a chain with a pC at the beginning or end), and p^*_{R} over chains that end with a pS (but may have a pC at the beginning). Finally, x^o is an abbreviation for $x^o_{\triangleright\mathsf{refl}}$.

$$\begin{array}{lcl}
o & ::= & 1 \mid\ -1 \\
pS & ::= & x^o \mid x^o_{\triangleright p^*_{\mathsf{R}}} \mid \mathsf{inj}\ i\ pS \mid p^*_{\mathsf{LR}} \\
pC & ::= & \mathsf{cong}_{\,A}\, p^*_1 \mathinner{..} p^*_i \\
p^* & ::= & (pS \mid pC)^* \\
p^*_{\mathsf{R}} & ::= & (p^*; pS) \\
p^*_{\mathsf{LR}} & ::= & pS \mid (pS; p^*; pS)
\end{array}$$

There is one additional condition which is not shown in the grammar: there must never be two check-terms adjacent to each other in a chain.

***Lemma* 61.** If $\Gamma \vdash^{\mathrm{L}} p : a = b$, then there exists some p^* such that $p \mapsto^* p^*$.

Proof. As a first step, we use the Inv* rules to push uses of symmetry to the leaves of the evidence term. That is, the symmetry rule is only applied the uses of assumptions from the contexts. So without loss of generality we can assume that the evidence term

p	$\mapsto$	p	null
refl^{-1}	$\mapsto$	refl	InvRefl
$\mathsf{refl}; p$	$\mapsto$	p	ReflTrans1
$p; \mathsf{refl}$	$\mapsto$	p	ReflTrans2
$(p; q); r$	$\mapsto$	$p; (q; r)$	TransTrans
$p; p^{-1}$	$\mapsto$	refl	InvTrans1
$p^{-1}; p$	$\mapsto$	refl	InvTrans2
$p; (p^{-1}; r)$	$\mapsto$	r	InvCTrans1
$p^{-1}; (p; r)$	$\mapsto$	r	InvCTrans2
$p^{-1^{-1}}$	$\mapsto$	p	InvInv
$(p; q)^{-1}$	$\mapsto$	$q^{-1}; p^{-1}$	InvTrans
$(\mathsf{cong}_{\,A}\, p_1 \,..\, p_i)^{-1}$	$\mapsto$	$\mathsf{cong}_{\,A}\, {p_1}^{-1} \,..\, {p_i}^{-1}$	InvCong
$(\mathsf{inj}_i\;\, p)^{-1}$	$\mapsto$	$\mathsf{inj}_i\;\, (p^{-1})$	InvInj
$(\mathsf{cong}_{\,A}\, p_1 \,...\, p_i); (\mathsf{cong}_{\,A}\, q_1 \,...\, q_i)$	$\mapsto$	$\mathsf{cong}_{\,A}\, (p_1; q_1) \,..\, (p_i; q_i)$	CongTrans
$\mathsf{inj}_k\;\, (\mathsf{cong}_{\,A}\, p_1 \,...\, p_i)$	$\mapsto$	p_k	InjCong1
$\mathsf{inj}_k\;\, ((\mathsf{cong}_{\,A}\, p_1 \,..\, p_i); r)$	$\mapsto$	$p_k; (\mathsf{inj}_k\;\, r)$	InjCong2
$\mathsf{inj}_k\;\, (r; (\mathsf{cong}_{\,A}\, p_1 \,...\, p_i))$	$\mapsto$	$(\mathsf{inj}_k\;\, r); p_k$	InjCong3
$x_{\triangleright(r;\mathsf{cong}_{\,=}\, p\; q)}$	$\mapsto$	$p^{-1}; (x_{\triangleright r}); q$	AssumCong

$$\frac{p \mapsto p'}{x_{\triangleright p} \mapsto x_{\triangleright p'}}\,\textsc{assumption} \qquad \frac{\begin{array}{c} p \mapsto p' \\ q \mapsto q' \end{array}}{p; q \mapsto p'; q'}\,\textsc{trans}$$

$$\frac{\forall k.\quad p_k \mapsto p'_k}{\mathsf{cong}_{\,A}\, p_1 \,..\, p_i \mapsto \mathsf{cong}_{\,A}\, p'_1 \,..\, p'_i}\,\textsc{cong} \qquad \frac{p \mapsto p'}{\mathsf{inj}_k\;\, p \mapsto \mathsf{inj}_k\;\, p'}\,\textsc{inj}$$

Figure A.3: Simplification rules for evidence terms (with names for rules)

p belongs to the following subgrammar.

$$p \quad ::= \quad x^{o}_{\triangleright p} \mid \mathsf{refl} \mid p;q \mid \mathsf{inj}_i\ p \mid \mathsf{cong}_{\,A}\,p_1 \mathinner{..} p_i$$

Next, we proceed by induction on the structure of p. In each case, we must show there exists some evidence chain p^* such that $p \mapsto^* p^*$.

- The term is $x^{o}_{\triangleright p}$. By IH, we know $p \mapsto^* p^*$.

  If p^* is empty (refl) or ends with a synthesizable term, then the term $x^{o}_{\triangleright p^*}$ is a valid chain and we are done.

  Otherwise, p^* ends with a use of cong, i.e. p^* is $r^*;\mathsf{cong}_{\,A}\,q_1 \mathinner{..} q_i$. However, by the assumption we know that $\Gamma \vdash^{\llcorner} x^{o}_{\triangleright(r^*;\mathsf{cong}_{\,A}\,q_1\,\ldots\,q_i)} : a = b$. Assuming (wlog) that $o = 1$, this means that $\Gamma \vdash^{\llcorner} (r^*;\mathsf{cong}_{\,A}\,q_1 \ldots q_i) : A = (a = b)$. By inversion we know that the label A is $=$ and there are exactly two subterms q_1 and q_2, so we can simplify using AssumCong:

  $$x_{\triangleright(r^*;\mathsf{cong}_{\,=}\,q_1\;q_2)} \mapsto {q_1}^{-1}; (x_{\triangleright r^*}); q_2$$

  which is a valid chain. Similarly, in the case $o = -1$ we can simplify using AssumCong and InvTrans:

  $$x^{-}_{\triangleright(r^*;\mathsf{cong}_{\,=}\,q_1\;q_2)}1 \mapsto {q_2}^{-1}; (x_{\triangleright r^*})^{-1}; q_1$$

- The term is refl. This is already a valid (empty) chain.

- The term is $p;q$. By the IHs for p and q we know that there are chains p^* and q^*. We must now show that $p^*;q^*$ can be simplified into a valid chain r^*.

  If p^* is the empty chain refl, then by ReflTrans1 we can just return q^*. Similarly if q^* is empty, then by ReflTrans2 we can return p^*.

  If both p^* and q^* are nonempty, we use TransTrans to reassociate $p^*;q^*$ into a right-associated chain. However, we must also ensure that the resulting chain does not contain two adjacent pCs. That would happen if p^* ends with a use of cong and q^* begins with cong. In that case, after reassociation we end up with a subproof of the form

  $$(\mathsf{cong}_{\,A}\,p_1 \mathinner{..} p_i); (\mathsf{cong}_{\,B}\,q_1 \mathinner{..} q_j)$$

  By assumption we know this is evidence for some equation $a = b$. By inversion on the judgement

  $$\Gamma \vdash^{\llcorner} (\mathsf{cong}_{\,A}\,p_1 \mathinner{..} p_i); (\mathsf{cong}_{\,B}\,q_1 \mathinner{..} q_j) : a = b$$

we see that we must have $A = B$ and $i = j$, and $a = b$ must be ${}_A\,\overline{a_i} =_A \overline{b_i}$. Then we can use CongTrans to simplify to a single use of cong.

- The term is $\mathsf{inj}_i\ p$. By IH we know $p \mapsto^* p^*$.

  Now, $\mathsf{inj}\,i\,p^*$ may not be a valid normalized evidence term, because it may violate the condition that p^* begins and ends with a pS. Let $p^* = q_1^*; q_2^*; q_3^*$, such that q_1^* and q_3^* consists only of checkable terms and q_2^* begins and ends with a synthesizable term. Now apply InjCong2 and InjCong3 repeatedly to simplify q_1^* and q_3^*. We get

  $$\mathsf{inj}\,i\,p^* \mapsto^* r_1^*; (\mathsf{inj}\,i\,q_2^*); r_3^*$$

  where r_1^* consists of subterms from the cong-expressions in q_1^*, and similarly for q_3^*.

  Finally, at this point r_1^* and r_3^* may contain adjacent cong-terms, so we need to simplify them using CongTrans as in the previous case.

- The term is $\mathsf{cong}_A\, p_1 \mathinner{..} p_i$. By the IHs, we know $p_k \mapsto^* p_k^*$. Then

  $$\mathsf{cong}_A\, p_1 \mathinner{..} p_i \mapsto^* \mathsf{cong}_A\, p_1^* \mathinner{..} p_i^*$$

  which is a valid chain.

□

Intuitively, the label function recursively decomposes a term a into a first-ordered "labelled" expression $F(a_1, \ldots, a_k)$, where F is the least nontrival linear multi-hole context that agrees with a. The label function takes an expression a, and returns a label F together with a list of subexpressions ak. We write this as

$$\mathsf{label}\,a = F\,\overline{a_i}$$

The function label is defined in turns of a helper function $\mathsf{label}_S\ a$, which takes as argument a set of variables S and an expression a and also returns ${}_A\,\overline{a_i}$, with the additional constraint that label_S tries to select the smallest label F such that $\mathsf{FV}\,(a_k) \cap S = \emptyset$. The two functions are quite similar (in the Haskell implementation there is just one function which takes an extra boolean argument); the difference is that label_S can return the trivial context which is just a single hole, whereas label always chooses a label that contains at least one syntactic constructor.

$$\begin{array}{lcl}
\mathsf{label}\,\mathsf{Type} & = & (\mathsf{Type}) \\
\mathsf{label}\,x & = & (x) \\
\mathsf{label}\,(\mathsf{rec}\,f_A\ x.a) & = & (\mathsf{rec}\ f\ x.F)\ \overline{a_i} \\
 & & \text{where } \mathsf{label}_{\{f,x\}}\ a = F\,\overline{a_i} \\
\mathsf{label}\,(\mathsf{rec}\,f_A\ \bullet_x.a) & = & (\mathsf{rec}\ f\ \ \bullet.F)\ \overline{a_i} \\
 & & \text{where } \mathsf{label}_{\{f,x\}}\ a = F\,\overline{a_i} \\
\mathsf{label}\,(a\ b) & = & (-\,-)\ (\mathsf{label}\,a)\ (\mathsf{label}\,b) \\
\mathsf{label}\,(a\ \bullet_b) & = & (-\ \bullet)\ (\mathsf{label}\,a) \\
\mathsf{label}\,((x\,{:}\,A) \rightarrow\ B) & = & (x : -) \rightarrow F)\ (\mathsf{label}\,A)\ \overline{B_i} \\
\quad \text{where } \mathsf{label}_{\{x\}}\ B = F\,\overline{B_i} & & \\
\mathsf{label}\,(\bullet(x\,{:}\,A) \rightarrow\ B) & = & \bullet(x : -) \rightarrow F)\ (\mathsf{label}\,A)\ \overline{B_i} \\
\quad \text{where } \mathsf{label}_{\{x\}}\ B = F\,\overline{B_i} & & \\
\mathsf{label}\,(a = b) & = & (- = -)\ (\mathsf{label}\,a)\ (\mathsf{label}\,b) \\
\mathsf{label}\,\mathsf{join}_\Sigma & = & (\mathsf{join}) \\
\mathsf{label}\,(a_{\triangleright b}) & = & \mathsf{label}\,a
\end{array}$$

$$\begin{array}{lcl}
\mathsf{label}_S\ a & = & (-)\ (\mathsf{label}\,a) \\
\quad \text{when } \mathsf{FV}\,(a) \cap S = \emptyset & & \\
\text{Otherwise:} & & \\
\mathsf{label}_S\ x & = & (x) \\
\mathsf{label}_S\ (\mathsf{rec}\,f_A\ x.a) & = & (\mathsf{rec}\ f\ x.F)\ \overline{a_i} \\
\quad \text{where } \mathsf{label}_{S\cup\{f,x\}}\ a = F\,\overline{a_i} & & \\
\mathsf{label}_S\ (\mathsf{rec}\,f_A\ \bullet_x.a) & = & (\mathsf{rec}\ f\ \ \bullet.F)\ \overline{a_i} \\
\quad \text{where } \mathsf{label}_{S\cup\{f,x\}}\ a = F\,\overline{a_i} & & \\
\mathsf{label}_S\ (a\ b) & = & (F\ G)\ \overline{a_i}\ \overline{b_i} \\
\quad \text{where } \mathsf{label}_S\ a = F\,\overline{a_i} & & \\
\quad \text{and } \quad \mathsf{label}_S\ b = G\,\overline{b_i} & & \\
\mathsf{label}_S\ (a\ \bullet_b) & = & (F\ \bullet)\ \overline{a_i} \\
\quad \text{where } \mathsf{label}_S\ a = F\,\overline{a_i} & & \\
\mathsf{label}_S\ ((x\,{:}\,A) \rightarrow\ B) & = & ((x : F) \rightarrow G)\ \overline{A_i}\ \overline{B_i} \\
\quad \text{where } \mathsf{label}_S\ A = F\,\overline{A_i} & & \\
\quad \text{and } \quad \mathsf{label}_{S\cup\{x\}}\ B = G\,\overline{B_i} & & \\
\mathsf{label}_S\ (\bullet(x\,{:}\,A) \rightarrow\ B) & = & (\bullet(x : F) \rightarrow G)\ \overline{A_i}\ \overline{B_i} \\
\quad \text{where } \mathsf{label}_S\ A = F\,\overline{A_i} & & \\
\quad \text{and } \quad \mathsf{label}_{S\cup\{x\}}\ B = G\,\overline{B_i} & & \\
\mathsf{label}_S\ (a = b) & = & (F\ G)\ \overline{a_i}\ \overline{b_i} \\
\quad \text{where } \mathsf{label}_S\ a = F\,\overline{a_i} & & \\
\quad \text{and } \quad \mathsf{label}_S\ b = G\,\overline{b_i} & & \\
\mathsf{label}_S\ \mathsf{join}_\Sigma & = & (\mathsf{join}) \\
\mathsf{label}_S\ (a_{\triangleright b}) & = & \mathsf{label}_S\ a
\end{array}$$

We also define the "inverse" function $\mathsf{unlabel}$, which simply substitutes away all the label applications. The function $\mathsf{unlabel}$ is defined by recursion on the labelled term:

$$\mathsf{unlabel}\,(F\,\overline{a_i}) = \{\mathsf{unlabel}\,a_1/x_1\}\ldots\{\mathsf{unlabel}\,a_j/x_j\}F$$

when the holes in F are named x_1 through x_j.

Lemmas 62–65 are all proved by inductions on the term a.

***Lemma* 62** ($\mathsf{unlabel}$-label)**.** For any a, we have $\mathsf{unlabel}\,(\mathsf{label}\,a) = |a|$.

***Lemma* 63** (Substituting into a label)**.** Suppose $\mathsf{label}\,a' = F\,\overline{a_i}$ where the holes in F are named $x_1 \ldots x_i$. Then $|a'| = |\{\mathsf{unlabel}\,a_1/x_1\}\ldots\{\mathsf{unlabel}\,a_j/x_k\}\,F|$.

***Lemma* 64** (label does not let bound variables escape)**.**

- If $\mathsf{label}\,a = F\,\overline{a_i}$, then for every k we have $\mathsf{FV}\,(a_k) \subseteq \mathsf{FV}\,(a)$.
- If $\mathsf{label}_S\ a = F\,\overline{a_i}$, then for every k we have $\mathsf{FV}\,(a_k) \subseteq (\mathsf{FV}\,(a) \setminus S)$.

***Lemma* 65** (label decides erasure)**.** For any expressions a and b, we have $|a| = |b|$ iff $(\mathsf{label}\,a) = (\mathsf{label}\,b)$

***Lemma* 66.** For all a, b and c such that $\mathsf{FV}\,(a) \cap S = \emptyset$ and $\mathsf{FV}\,(b) \cap S = \emptyset$, if $\mathsf{label}_S\ \{a/x\}\ c = F\,\overline{a_i}$ and $\mathsf{label}_S\ \{b/x\}\ c = G\,\overline{b_i}$, then $F = G$, and there exists $\overline{c_i}$ such that for all k, $a_k = \mathsf{label}\,\{a/x\}\ c_k$ and $b_k = \mathsf{label}\,\{b/x\}\ c_k$.a

Proof. Induction on the structure of c.

c **is** x Then since we assumed that a and b have no free variables in S, $\mathsf{label}_S\ \{a/x\}\ c = (-)\ (\mathsf{label}\,a)$ and $\mathsf{label}_S\ \{b/x\}\ c = (-)\ (\mathsf{label}\,b)$, so the labels are equal and we can take the list to be just $c_0 = x$.

c **is** Type Then $\mathsf{label}_S\ \{a/x\}\ c = \mathsf{label}_S\ \{b/x\}\ c = (\mathsf{Type})$, so the labels are indeed equal, and we can take the empty list for $\overline{c_i}$.

c **is some variable** $y \neq x$ Similar to the previous case.

c **is** join_Σ Similar to the previous case.

c **is** $\mathsf{rec}\,f_A\ y.c_0$ Let $\mathsf{label}_S\ \{a/x\}\ c_0 = F\,\overline{a_i}$ and $\mathsf{label}_S\ \{b/x\}\ c_0 = G\,\overline{b_i}$. By the IH we know $F = G$, and there is a list $\overline{c_i}$.

Now, $\mathsf{label}_S\ \{a/x\}\ c = (\,\mathsf{rec} f\ y.F)\,\overline{a_i}$ and $\mathsf{label}_S\ \{b/x\}\ c = (\,\mathsf{rec} f\ y.G)\,\overline{b_i}$. So the labels are indeed equal, and the list of expressions is just $\overline{c_i}$.

c **is** $\mathsf{rec}\,f_A\ \bullet_y.c_0$ Similar to the previous case.

c is $(y\!:C_1) \to\ C_2$ Let

$$\begin{array}{lcl}\mathsf{label}_S\ \{a/x\}\,C_1 & = & F\,\overline{a_i}\\ \mathsf{label}_S\ \{b/x\}\,C_1 & = & G\,\overline{b_i}\\ \mathsf{label}_{S\cup\{y\}}\ \{a/x\}\,C_2 & = & F'\,\overline{a_i}'\\ \mathsf{label}_{S\cup\{y\}}\ \{b/x\}\,C_2 & = & G'\,\overline{b_i}'\end{array}$$

Since we can choose the bound variable y fresh, the disjointness condition on S is still satisfied. So by the IHs we get that $F = G$ and $F' = G'$, and also suitable lists $\overline{c_i}$ and $\overline{c_i}$.

Now, $\mathsf{label}_S\ \{a/x\}\,((y\!:C_1) \to\ C_2) = ((y : F) \to F')\ \overline{a_i}\ \overline{a_i}'$ and $\mathsf{label}_S\ \{b/x\}\,((y\!: C_1) \to\ C_2) = ((y : F) \to F')\ \overline{b_i}\ \overline{b_i}'$. So the label is indeed the same for both applications, and $\overline{c_i}\ \overline{c_i}'$ is a suitable list.

c is $\bullet(y\!:C_1) \to\ C_2$ or $c_1\ c_2$ or $c_1 = c_2$ Similar to the previous case.

c is $c_1\ \bullet_{c_2}$ Let

$$\begin{array}{l}\mathsf{label}_S\ \{a/x\}\,c_1 = F\,\overline{a_i}\\ \mathsf{label}_S\ \{b/x\}\,c_1 = G\,\overline{b_i}\end{array}$$

The IH gives $F = G$ and a list $\overline{c_i}$. The label we return is $(F\ \bullet)$, and the argument list is $\overline{c_i}$.

c is $c_{1\triangleright c_2}$ Similar to the previous case.

□

***Lemma* 67** (CC implies LCC, the congruence case)**.** For all c, if $\Gamma \vdash^{\mathrm{L}}\ \mathsf{label}\,a = \mathsf{label}\,b$ then $\Gamma \vdash^{\mathrm{L}}\ \mathsf{label}\,\{a/x\}\,c = \mathsf{label}\,\{b/x\}\,c$

Proof. Simultaneous induction on the structure of c. Most of the cases are similar, so we show only some representative ones.

c is x we must show $\Gamma \vdash^{\mathrm{L}}\ \mathsf{label}\,a = \mathsf{label}\,b$, which we have as an assumption.

c is Type . Then both $\mathsf{label}\,\{a/x\}\,c$ and $\mathsf{label}\,\{b/x\}\,c$ are just (Type), so LCCREFL proves the required equation.

c is a variable $y \neq x$ Similar to previous case.

c is some variable join_Σ Similar to the previous case.

c is $\mathsf{rec}\,f_A\ y.c_0$ By lemma 66, there is some F and $\overline{c_i}$ such that $\mathsf{label}_{\{f,y\}}\ \{a/x\}\,c_0 = F\,\overline{a_i}$ and $\mathsf{label}_{\{f,y\}}\ \{b/x\}\,c_0 = F\,\overline{b_i}$, and furthermore $a_k = \mathsf{label}\,\{a/x\}\,c_k$ and $b_k = \mathsf{label}\,\{b/x\}\,c_k$.

By the IH, we know that $\forall k.\ \ \Gamma \vdash^{\mathrm{L}}\ a_k = b_k$. Note that $\mathsf{label}\,\{a/x\}\,(\mathsf{rec}\,f_A\ y.c_0)$ is $(\,\mathsf{rec}\,f\ y.F)\,\overline{a_i}$, and $\mathsf{label}\,\{b/x\}\,(\mathsf{rec}\,f_A\ y.c_0)$ is $(\,\mathsf{rec}\,f\ y.F)\,\overline{b_i}$. So by LCCCONG using the label $(\,\mathsf{rec}\,f\ y.F)$ we have $\Gamma \vdash^{\mathrm{L}}\ \{a/x\}\,c = \{b/x\}\,c$ as required.

c is $\mathsf{rec}\ f_A\ \bullet_y.c_0$ Similar to the previous case.

c is $(y\!:C_1) \to\ C_2$ By lemma 66, there is some F and $\overline{c_i}$ such that $\mathsf{label}_S\ \{a/x\}\ C_2 = F\,\overline{a_i}$ and $\mathsf{label}_S\ \{b/x\}\ C_2 = F\,\overline{b_i}$, and furthermore $a_k = \mathsf{label}\,\{a/x\}\ c_k$ and $b_k = \mathsf{label}\,\{b/x\}\ c_k$.

Now $\mathsf{label}\,\{a/x\}\ c = ((y : \text{-}) \to F)\ (\mathsf{label}\,\{a/x\}\ C_1)\overline{a_i}$. and
$\mathsf{label}\,\{b/x\}\ c = ((y : \text{-}) \to F)\ (\mathsf{label}\,\{b/x\}\ C_1)\overline{b_i}$.

By the IHs we get $\Gamma \vdash^{\mathrm{L}}\ \mathsf{label}\,\{a/x\}\ C_1 = \mathsf{label}\,\{b/x\}\ C_1$ and also $\forall k.\ \Gamma \vdash^{\mathrm{L}}\ a_k = b_k$. So we conclude by LCC<span style="font-variant:small-caps">cong</span> using the label $((y : \text{-}) \to F)$.

c is $\bullet(y\!:C_1) \to\ C_2$ **or** $c_1\ c_2$ **or** $c_1 = c_2$ Similar to the previous case.

c is $c_1\ \bullet_{c_2}$ Then $\mathsf{label}\,\{a/x\}\ c$ is $(\text{-}\ \bullet)\ (\mathsf{label}\,\{a/x\}\ c_1)$ and
$\mathsf{label}\,\{b/x\}\ c$ is $(\text{-}\ \bullet)\ (\mathsf{label}\,\{b/x\}\ c_1)$. By the IH we have $\Gamma \vdash^{\mathrm{L}}\ \mathsf{label}\,\{a/x\}\ c_1 = \mathsf{label}\,\{b/x\}\ b_1$. So we conclude by LCCCONG using the label $(\text{-}\ \bullet)$.

c is $c_{1 \triangleright c_2}$ Similar to the previous case.

□

***Lemma* 68** (label preserves CC)**.** If $\Gamma \vdash a = b$, then $\mathsf{label}\,\Gamma \vdash^{\mathrm{L}}\ \mathsf{label}\,a = \mathsf{label}\,b$.

Proof. Induction on $\Gamma \vdash a = b$. The cases are

CCrefl We are given

$$\frac{|a| = |b|}{\Gamma \vdash a = b}\textsc{CCrefl}$$

From $|a| = |b|$ and lemma 65 we know $\mathsf{label}\,a = \mathsf{label}\,b$. So apply LCCREFL.

CCsym,CCtrans These follow directly by IH.

CCassumption We are given

$$\frac{\begin{array}{l} x : A \in \Gamma \\ \Gamma \vdash A = (a = b) \end{array}}{\Gamma \vdash a = b}\textsc{CCassumption}$$

Since $x : A \in \Gamma$ we know $x : \mathsf{label}\,A \in \mathsf{label}\,\Gamma$. And by the IH we have $\mathsf{label}\,\Gamma \vdash \mathsf{label}\,A = \mathsf{label}\,(a = b)$. Since $\mathsf{label}\,(a = b)$ is the same as $\mathsf{label}\,a = \mathsf{label}\,b$, we conclude by LCCASSUMPTION.

CCcongruence We are given

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash \{a/x\}\ c = \{b/x\}\ c}\textsc{CCcongruence}$$

Apply lemma 67.

CCinjdom We are given

$$\frac{\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vdash A_1 = B_1}\text{CC}\textsc{injdom}$$

The IH gives $\mathsf{label}\,\Gamma \vdash \mathsf{label}\,(A_1 \to B_1) = \mathsf{label}\,(A_2 \to B_2)$, which is the same as $\mathsf{label}\,\Gamma \vdash (- \to -)\ (\mathsf{label}\,A_1)\ (\mathsf{label}\,B_1) = (- \to -)\ (\mathsf{label}\,A_2)\ (\mathsf{label}\,B_2)$. And $(- \to -)$ is an injective label, so we conclude by CC<span style="font-variant:small-caps">Injectivity</span>.

CCinjrng,CCiinjdom,CCiinjrng,CCinjeq These cases are similar to the previous one.

□

***Lemma* 69** (Label arguments arise from well-typed subexpressions)**.**

- If $\Gamma \vdash a' : A$, and $\mathsf{label}\,a' = F\,\overline{a_i}$, then for every a_k there exists a'_k such that $\Gamma \vdash a'_k : A_k$ and $a_k = \mathsf{label}\,a'_k$.
- If $\Gamma \vdash a' : A$, and $\mathsf{label}_S\ a' = F\,\overline{a_i}$, then for every a_k there exists a'_k such that $\Gamma \vdash a'_k : A_k$ and $a_k = \mathsf{label}\,a'_k$.

Proof. (Strong) induction on the structure of a'. Most of the cases of the induction are similar, so we do not show all of them. A few representative cases for label are:

a' **is** Type **or some variable** x Then $\mathsf{label}\,a'$ is a nullary label-application, so $\overline{a_i}$ is empty and the lemma is vacuously true.

a' **is** $\mathsf{rec}\,f_A\ x.b'$ There is only one typing rule for rec-expressions, so from the judgement $\Gamma \vdash a' : A$, we know that

$$\Gamma, f : (x\!:\!A_1) \to A_2, x : A_1 \vdash b' : A_2$$

From the definition of label we know that $\mathsf{label}\,a'$ is $(\mathsf{rec}\ f_{:A}\ x.F)\ \overline{a_i}$, where $\mathsf{label}_{\{f,x\}}\ b = F\,\overline{a_i}$. So by the IH for a' we know that there exists a'_k such that $a_k = \mathsf{label}\,a'_k$ and $\Gamma, f : (x : A_1) \to A_2, x : A_1 \vdash a'_k : A'$. By lemma 64 we know that f and x are not free in a'_k, so by strengthening (lemma 48) we have $\Gamma \vdash a'_k : A'$ as required.

a' **is** $b'\ c'$ Then $\mathsf{label}\,a'$ is $(-\ -)\ \overline{b_i}\ \overline{c_i}$. The expression a_k must belong to one of the lists $\overline{b_i}$ or $\overline{c_i}$, so by the IH for b' or c' we get a corresponding b'_k or c'_k.

A few representative cases for label_S are:

a' **has no free variables in** S Then $\mathsf{label}_S\ a' = (-)\ (\mathsf{label}\,a')$. So there is only a single a_k, which must be $(\mathsf{label}\,a')$. Thus we can take $a'_k = a'$.

a' is Type or some variable x Similar to the corresponding case for label: $\mathsf{label}_S\ a'$ is a nullary label-application and the lemma is vacuously true.

a' is $\mathsf{rec}\, f_A\ x.b'$ As in the case for label, we know that

$$\Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash b' : A_2$$

and $\mathsf{label}_S\ a'$ is $(\mathsf{rec}\ f_{:A}\ x.F)\ \overline{a_i}$, where $\mathsf{label}_{S\cup\{f,x\}}\ b = F\,\overline{a_i}$. Conclude by IH and strengthening as in the above case.

a' is $b'\ c'$ Then $\mathsf{label}_S\ a'$ is $(F\ G)\ \overline{b_i}\ \overline{c_i}$. The expression a_k must belong to one of the lists $\overline{b_i}$ or $\overline{c_i}$, so by the IH for b' or c' we get a corresponding b'_k or c'_k.

□

***Lemma* 71** (Inversion for label)**.**

- If $(\mathsf{label}\, A') = (- = -)\ a\ b$, then there exists a' and b' such that $A' = (a' = b')$ and $(\mathsf{label}\, a') = a$ and $(\mathsf{label}\, b') = b$.
- If $\mathsf{label}\, A' = (- \to -)\ a_1\ a_2$, then there exists a'_1 and a'_2 such that $A' = (a'_1 \to a'_2)$, and $(\mathsf{label}\, a'_1) = a_1$ and $(\mathsf{label}\, a'_2) = a_2$.
- Similar for $\bullet a'_1 \to\ a'_2$
- Similar fpr $a'_1 = a'_2$.

Proof. Immediate from considering cases for A' and examining the definition of label.

□

***Lemma* 71** (Normalized untyped CC implies typed CC)**.**

- If $\mathsf{label}\, \Gamma' \vdash^{\perp} pS : a = b$, then there exists annotated core expressions a', b' such that $a = \mathsf{label}\, a'$ and $b = \mathsf{label}\, b'$ and $\Gamma' \vDash a' = b'$.
- If $\mathsf{label}\, \Gamma' \vdash^{\perp} pC : \mathsf{label}\, a' = \mathsf{label}\, b'$ and $\Gamma' \vdash a' = b' : \mathsf{Type}$, then $\Gamma' \vDash a' = b'$.
- If $\mathsf{label}\, \Gamma' \vdash^{\perp} p^* : \mathsf{label}\, a' = \mathsf{label}\, b'$ and $\Gamma' \vdash a' = b' : \mathsf{Type}$, then $\Gamma' \vDash a' = b'$.
- If $\mathsf{label}\, \Gamma' \vdash^{\perp} p^*_{\mathsf{R}} : \mathsf{label}\, a' = b$ and $\Gamma' \vdash a' : A$, then there exists an b' such that $b = \mathsf{label}\, b'$ and $\Gamma' \vDash a' = b'$.

Proof. We proceed by mutual induction on the sizes of pS and p^*. The cases for pS are:

The evidence is $x_{\triangleright p^*_{\mathsf{R}}}$ By examining the definition of $\Gamma \vdash^{\perp} p : a = b$, we see that the only rule that applies is CCPassumption, so we know we have

$$\begin{array}{c} x : A \in (\mathsf{label}\, \Gamma') \\ \mathsf{label}\, \Gamma' \vdash^{\perp} p^*_{\mathsf{R}} : A = ((- = -)\ a\ b) \end{array}$$

From $x : A \in (\mathsf{label}\,\Gamma')$ we know that $A = \mathsf{label}\,A'$ for some $x : A' \in \Gamma'$.

Then from the mutual IH for p^*_{R} we known that there exists some B' such that $((- = -)\ a\ b) = \mathsf{label}\,B'$ and $\Gamma' \vDash A' = B'$.

Further, by lemma 70 we know that $B' = \mathsf{label}\,(a' = b')$ for some expressions a' and b' such that $a = \mathsf{label}\,a'$ and $b = \mathsf{label}\,b'$. So we have shown $\Gamma' \vDash A' = (a' = b')$. Now apply TCCAssumption to conclude $\Gamma' \vDash a' = b'$ as required.

The evidence is $(x_{\triangleright p^*_{\mathsf{R}}})^{-1}$ By reasoning as in the previous case we get some a' and b' such that $a = \mathsf{label}\,a'$ and $b = \mathsf{label}\,b'$ and $\Gamma' \vDash a' = b'$. Then apply TCCSym to conclude $\Gamma' \vDash b' = b'$ as required.

The evidence is $\mathsf{inj}\,i\,pS$ By examining the definition of the $\Gamma \vdash^{\perp} p : a = b$ judgement we see that the only rule that applies is CCPInj. So we must have

$$\begin{array}{c} \mathsf{label}\,\Gamma' \vdash^{\perp} pS : F\,\overline{a_i} = F\,\overline{b_i} \\ F\,\mathsf{injective} \end{array}$$

Recall that $F\,\mathsf{injective}$ means that F is either $- \to -$, $- = -$, or $\bullet\!- \to -$.

We consider the case when it is $- \to -$ and i is 1; the other cases are similar. That is, the assumed derivation looks like

$$\frac{\mathsf{label}\,\Gamma' \vdash pS : (- \to -)\ a_1\ a_2 = (- \to -)\ b_1\ b_2}{\mathsf{label}\,\Gamma' \vdash^{\perp} \mathsf{inj}\,i\,pS : a_1 = b_1}$$

From the IH we get expressions A' and B' such that $(- \to -)\ a_1\ a_2 = \mathsf{label}\,A'$ and $(- \to -)\ b_1\ b_2 = \mathsf{label}\,B'$, and $\Gamma' \vDash A' = B'$. By lemma 70 we then know $A' = (a'_1 \to a'_2)$ and $B' = (b'_1 \to b'_2)$ Then apply TCCInjDom to conclude $\Gamma' \vDash a'_1 = b'_1$ as required.

The evidence is a chain p^*_{LR} From the grammar for p^*_{LR} that means that it is either a single terms pS (which we dealt with in the above cases), or it is a chain starting and ending with a synthesizable term, that is p^*_{LR} is $pS; q^*; rS$.

In the latter case, by the IH for pS and rS we get terms c'_1 and c'_2 such that $\Gamma \vDash \mathsf{label}\,a' = \mathsf{label}\,c'_1$ and $\Gamma \vDash \mathsf{label}\,c'_1 = b'$.

Now we can apply the mutual induction hypothesis for the chain r^*, to get $\Gamma' \vDash c'_1 = c'_2$.

Finally, apply transitivity (CCPtrans) twice to conclude $\Gamma' \vDash a' = b'$ as required.

The only case for pC is when **the evidence term is a use of congruence,** $\mathsf{cong}\,F\,p_1 \mathinner{..} p_i$. The only rule that applies is CCPcong, so the assumed derivation is

$$\frac{\forall k.\ \ \mathsf{label}\,\Gamma' \vdash^{\llcorner} p_k : a_k = b_k}{\mathsf{label}\,\Gamma' \vdash^{\llcorner} \mathsf{cong}\, F\, p_1 \mathinner{..} p_i : F\,\overline{a_i} = F\,\overline{b_i}}$$

By assumption we know that $(F\,\overline{a_i}) = (\mathsf{label}\, a')$ and $(F\,\overline{b_i}) = (\mathsf{label}\, b')$.

From the assumption $\Gamma \vdash a' = b' : \mathsf{Type}$ we know a' and b' are well typed, so by lemma 69 we know that for every a_k there exists a well typed a'_k such that $a_k = \mathsf{label}\, a'_k$, and similarly for b_k.

So from IH for p_k we know $\forall k.\ \ \Gamma' \vDash a'_k = b'_k$.

By lemma 63 we know that $|a'| = |\{\mathsf{unlabel}\, a_1/x_1\} \ldots \{\mathsf{unlabel}\, a_i/x_i\}\, F|$. Since $\mathsf{unlabel}$ is inverse to label (lemma 62) this means $|a'| = |\{a'_1/x_1\} \ldots \{a'_i/x_i\}\, F|$. Similarly, $|b'| = |\{b'_1/x_1\} \ldots \{b'_i/x_i\}\, F|$. Finally, we know that $\Gamma' \vdash a' = b' : \mathsf{Type}$ by the assumption to the theorem.

So by TCCcongruence, $\Gamma' \vDash a' = b'$ as required.

The cases for p^* are:

The empty chain (refl) The only rule that can apply is CCPrefl, so we know that $(\mathsf{label}\, a') = (\mathsf{label}\, b')$. By lemma 65 this implies that $|a'| = |b'|$. We know as an assumption to the lemma that $\Gamma' \vdash a' = b'$, apply TCCErasure to conclude $\Gamma' \vDash a' = b'$ as required.

A chain consisting of a single term, p The evidence term p must be either a checkable are a synthesizable term. In the case when it is a pC we directly appeal to the mutual IH.

In the case when it is a pS, by the mutual IH we know that there are a'' and b'' such that $a = \mathsf{label}\, a''$ and $b = \mathsf{label}\, b''$ and $\mathsf{label}\,\Gamma' \vDash a'' = b''$.

Since $\mathsf{label}\, a' = \mathsf{label}\, a''$, by lemma 65 we know $|a'| = |a''|$, and similarly $|b''| = |b'|$. So by two uses each of TCCErasure and TCCTrans we get $\mathsf{label}\,\Gamma' \vDash a' = b'$, as required.

A chain of length > 1, starting with synthesizable term, $pS; q^*$ The only rule that applies is CCPtrans, so we must have

$$\begin{gathered}\mathsf{label}\,\Gamma' \vdash^{\llcorner} p : a = c \\ \mathsf{label}\,\Gamma' \vdash^{\llcorner} q^* : c = b\end{gathered}$$

From the mutual IH for pS we know that there is some a'' and c'' such that $a = \mathsf{label}\, a''$, $c = \mathsf{label}\, c''$, and $\mathsf{label}\,\Gamma' \vDash a'' = c''$. By reasoning as in the previous case we also know that $\mathsf{label}\,\Gamma' \vDash a' = a''$.

Now by the IH for q^* we know $\mathsf{label}\,\Gamma' \vDash c'' = b'$.

So by transitivity (TCCTrans) we get $\mathsf{label}\,\Gamma' \vDash a' = b''$ as required.

A chain of length > 1, starting with a checkable term, $pC; qS; r^*$ The definition of chains stipulates that there must never be two adjacent pCs, so we know that the second evidence term in the chain, qS, is synthesizable.

The only rule that applies is CCP<span style="font-variant:small-caps">trans</span>, so we must have

$$\begin{array}{l} \mathsf{label}\,\Gamma' \vdash^{\llcorner} pC : a = c_1 \\ \mathsf{label}\,\Gamma' \vdash^{\llcorner} qS : c_1 = c_2 \\ \mathsf{label}\,\Gamma' \vdash^{\llcorner} r^* : c_2 = b \end{array}$$

By the mutual IH for qS we get suitable c_1' and c_2'. Then apply the IHs for pC and r^*.

The cases for p_{R}^* are similar to the reasoning for general chains p^*. □

***Lemma* 72** (Core proof terms for $\Gamma \vDash a = b$)**.** If $\Gamma \vDash a = b$, then there exists some value v in the annotated core language such that $\Gamma \vdash v : a = b$.

Proof. Induction on the judgement $\Gamma \vDash a = b$.

TCCerasure The assumed derivation looks like

$$\frac{\begin{array}{l} |a| = |b| \\ \Gamma \vdash a : A \quad \Gamma \vdash b : B \end{array}}{\Gamma \vDash a = b}\textsc{TCCerasure}$$

From the regularity assumptions $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$ we know $\Gamma \vdash a = b :$ Type. So the equation follows from a use of join:

$$\frac{|a| \leadsto_{\mathsf{cbv}}^{0} |a| \quad |b| \leadsto_{\mathsf{cbv}}^{0} |a| \quad \Gamma' \vdash a = b : \mathsf{Type}}{\Gamma' \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} 00 : a = b} : a = b}$$

TCCrefl Similar to the previous case.

TCCsym By IH we get $\Gamma \vdash v : a = b$. From regularity (lemma 58) we know that a is typeable, so $\Gamma \vdash a = a : \mathsf{Type}$. then we can prove $b = b$ using T<span style="font-variant:small-caps">cast</span>, T<span style="font-variant:small-caps">subst</span> and T<span style="font-variant:small-caps">joinc</span>, as follows:

$$\frac{\dfrac{\Gamma \vdash v : a = b}{\Gamma \vdash \mathsf{join}_{\sim v = a} : (a = a) = (b = a)} \quad \dfrac{\Gamma \vdash a = a : \mathsf{Type}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} 00 : a = a} : a = a}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} 00 : a = a \triangleright \mathsf{join}_{\sim v = a}} : b = a}$$

TCCtrans The assumed derivation looks like

$$\frac{\Gamma \vDash a = b \quad \Gamma \vDash b = c}{\Gamma \vDash a = c}\textsc{TCCtrans}$$

The IHs are $\Gamma \vdash v_1 : a = b$ and $\Gamma \vdash v_2 : b = c$. We can then prove $a = c$ using TJCAST and TJSUBST:

$$\dfrac{\Gamma \vdash v_1 : a = b \quad \dfrac{\Gamma \vdash v_2 : b = c}{\Gamma \vdash \mathsf{join}_{a=\sim v_2} : (a = b) = (a = c)}}{\Gamma \vdash v_{\triangleright \mathsf{join}_{a=\sim v_2}} : a = c}$$

TCCassumption The assumed derivation looks like

$$\dfrac{\begin{array}{l} x : A \in \Gamma \\ \Gamma \vDash A = (a = b) \end{array}}{\Gamma \vDash a = b}\textsc{TCCassumption}$$

The IH gives $\Gamma \vdash v : A = (a = b)$, so $\Gamma \vdash x_{\triangleright v} : a = b$.

TCCcongruence The assumed derivation looks like

$$\dfrac{\begin{array}{l} \Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k \\ |A = B| = |\{a_1/x_1\} \ldots \{a_j/x_j\}\ c = \{b_1/x_1\} \ldots \{b_j/x_j\}\ c| \end{array}}{\Gamma \vDash A = B}\textsc{TCCcongruence}$$

The IH gives $\overline{v_i}$ such that $\forall k.\ \Gamma \vdash v_k : a_k = b_k$. By the regularity assumption to the rule we know that the equation is well-typed. So by TJSUBST we have

$$\Gamma \vdash \mathsf{join}_{\{\sim v_1/x_1\} \ldots \{\sim v_j/x_j\} c : A = B} : A = B$$

as required.

TCCinjdom From the IH we have $\Gamma \vdash v : ((x : A_1) \to B_1) = ((x : A_2) \to B_2)$. So apply TJINJDOM to get $\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\, v} : A_1 = A_2$ as required.

TCCinjrng, TCCiinjdom, TCCiinjrange, TCCinjeq Similar to the TCCINJDOM case.

□

***Theorem* 73** (Typed CC from untyped CC)**.** Suppose $\Gamma \vdash a = b$ and $\Gamma \vdash a = b : \mathsf{Type}$. Then $\Gamma \vDash a = b$, and furthermore $\Gamma \vdash v : a = b$ for some v.

Proof. From $\Gamma \vdash a = b$, by lemma 68 we get $\Gamma \vdash \mathsf{label}\, a = \mathsf{label}\, b$. By evidence simplification (lemma 61) we get $\Gamma \vdash^{\llcorner} p^* : \mathsf{label}\, a = \mathsf{label}\, b$. From this, and the fact that $\Gamma \vdash a = b : \mathsf{Type}$, by lemma 71 we get $\Gamma \vDash a = b$ as required. Finally, by lemma 72 there is some v such that $\Gamma \vdash v : a = b$. □

***Lemma* 74.** If $\Gamma \vDash a = b$ then $\Gamma \vdash a = b$.

Proof. Induction on $\Gamma \vDash a = b$.

TCCrefl,TCCerasure By CCRefl.

TCCsym,TCCtrans,TCCassumption By IH, then using CCSym, or CCTrans CCAssumption.

TCCcongruence The given derivation looks like

$$\frac{\begin{array}{l}\Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \ \Gamma \vDash a_k = b_k \\ |A = B| = |\{a_1/x_1\}\ldots\{a_j/x_j\}\ c = \{b_1/x_1\}\ldots\{b_j/x_j\}\ c|\end{array}}{\Gamma \vDash A = B}\ \text{TCCcongruence}$$

From the IHs we know $\forall k.\ \ \Gamma \vdash a_k = b_k$, so by applying CCcongruence j times we get

$$\Gamma \vdash \{a_1/x_1\}\ldots\{a_j/x_j\}\ c = \{b_1/x_1\}\ldots\{b_j/x_j\}\ c$$

Then use CCRefl and CCTrans to get $\Gamma \vdash a = B$.

TCCinjrng The IH gives $\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)$. Then apply CCInjrng.

TCCinjdom,TCCiinjdom,TCCiinjrng,TCCinjeq Similar to previous case.

□

Putting together two lemmas we get this version which is quoted in the paper:

***Corollary* 75** (TCC implies LCC)**.** If $\Gamma \vDash a = b$ then $\mathsf{label}\,\Gamma \vdash^{\mathsf{L}}\ \mathsf{label}\,a = \mathsf{label}\,b$.

Proof. By lemma 74 we have $\Gamma \vdash a = b$, then by lemma 68 we get $\mathsf{label}\,\Gamma \vdash^{\mathsf{L}}\ \mathsf{label}\,a = \mathsf{label}\,b$. □

***Lemma* 76** (Untyped CC ignores annotations in Γ)**.** If $\Gamma \vdash a = b$ and $|\Gamma| = |\Gamma'|$ then $\Gamma' \vdash a = b$.

Proof. By induction on $\Gamma \vdash a = b$. All the cases are immediate by IH except CCAssumption, were we are given

$$\frac{\begin{array}{l}x : A \in \Gamma \\ \Gamma \vdash A = (a = b)\end{array}}{\Gamma \vdash a = b}\ \text{CCAssumption}$$

By the IH we know $\Gamma' \vdash A = (a = b)$. From the assumption $|\Gamma| = |\Gamma'|$ we know that there is some $x : A' \in \Gamma'$ with $|A'| = |A|$. By CCRefl we have $\Gamma' \vdash A' = A$, so by CCTrans we know $\Gamma \vdash A' = (a = b)$. Then conclude by CCAssumption. □

***Lemma* 77** (Untyped CC ignores annotations)**.** If $\Gamma \vdash a = b$ and $|\Gamma| = |\Gamma'|$ and $|a| = |a'|$ and $|b| = |b'|$, then $\Gamma' \vdash a' = b'$.

Proof. By lemma 76 we know $\Gamma' \vdash a = b$, and by CCREFL we know $\Gamma' \vdash a' = a$ and $\Gamma' \vdash b = b'$. Then conclude by CCTRANS. □

***Lemma* 78** (CC doesn't look at type annotations)**.** Suppose $\Gamma \vDash a = b$, and $|\Gamma'| = |\Gamma|$, $|a'| = |a|$ and $|b'| = |b|$, and $\Gamma' \vdash a' : A'$ and $\Gamma' \vdash b' : B'$. Then $\Gamma' \vDash a' = b'$.

Proof. From $\Gamma \vDash a = b$ by lemma 74, we get $\Gamma \vdash a = b$. By lemma 77 we get $\Gamma' \vdash a' = b'$. Then by theorem 73 we get $\Gamma' \vDash a' = b'$. □

A.3 The untyped congruence closure algorithm and its correctness

The following section gives a precise mathematical definition of our algorithm to decide the $\Gamma \vdash^{\text{L}} \; a = b$ relation, and a correctness proof. The algorithm was described informally in Section 5.5.2.

A.3.1 Flattening

Developing the main rewriting algorithm is easier if the input problem is in a simple, restricted form. So following Nieuwenhuis and Oliveras [97] we first "flatten" the problem by introducing a fresh name for each subterm that occurs in it. We assume that we have an infinite set of *atomic constants* c_i available. The basic idea is that for any context Γ, we can construct an equivalent context with named subterms, e.g. a given assumption $h : f(g\ a) = b$ can be replaced with the set of assumptions

$$\begin{array}{rll}
h_1 : & f & = c_1 \\
h_2 : & g & = c_2 \\
h_3 : & a & = c_3 \\
h_4 : & c_2\ c_3 & = c_4 \\
h_5 : & c_1\ c_4 & = c_5 \\
h_6 : & b & = c_6 \\
h : & c_5 & = c_6
\end{array}$$

In the `Zombie` implementation, the flattening pass works directly on core language expressions. Constants are just integers, and the output of the flattening pass consists of a list of equations in the following Haskell datatype:

```
data EqConstConst = EqConstConst Constant Constant
data EqBranchConst  = EqBranchConst Label [Constant] Constant
type Equation = Either EqConstConst EqBranchConst
```

In addition, there is a table keeping track of additional information about each constant—in particular, whether that constant represents a type which is inhabited by a variable in the context. This is needed to handle the "assumption up to congruence" rule.

In order to reason about the correctness of this process, we need to formalize the input and output of the flattening. We aim to verify the algorithm, not the implementation, so we abstract away from the exact datastructures and instead represent the flattening stage as a transformation from contexts to context. The input is a context where each member is a labelled term (as defined in section A.2.1). The output is a context containing all the equations (the hs above), and also variable declarations encoding the information about being inhabited. For simplicitly, whenever a constant is marked as inhabited we assume that there is an inhabitant for both the constant and the expression that it names. (When generating core language proofs all constants are replaced with the original core language expressions they named). For a more complete example, the labelled context

$$\begin{array}{rl} x: & F\ a\ b \\ y: & F\ a\ b = G \end{array}$$

will be transformed into the flat context

$$\begin{array}{rll} h_1: & a & = c_1 \\ h_2: & b & = c_2 \\ h_3: & F\ c_1\ c_2 & = c_3 \\ h_4: & G & = c_4 \\ h_5: & (c_3 = c_4) & = c_5 \\ x_1: & c_3 & \\ x_2: & F\ c_1\ c_2 & \\ y_1: & c_5 & \\ y_2: & (c_3 = c_4) & \end{array}$$

where the h_i represent the list of equations that the algorithm outputs, and the x_i and y_i represent the information that c_3 and c_5 are inhabited.

The treatment of flattening is a bit more subtle than in previous work about first-order logic. In first-order systems, terms and equations are syntactically distinct categories, and one can maintain the invariant that every non-atomic subterm appearing in the flat context has a name. But in our setting there are two sources of equations in the flat context and only some of them have names; in the above example the equation x from the input context has been given the name c_5, but the flat context also contains the new assumptions h_i, and we do not allocate constants naming them (which would lead to infinite regress).

To be precise, the output of the flattening phase is a *flat context*, in the sense of the following definiton.

***Definition* 79** (Flat term)**.** A term is *flat* if it is either an atom a, or a label application $F\,\overline{a_i}$ such that each a_i is an atom.

***Definition* 80** (Flat term over Γ)**.** Let Γ be a context. We say that a term a is *flat over* Γ if a is either an atom, or it is a label applied to a list of atoms $F\,\overline{a_i}$ which is the left-hand-side of an equation in Γ.

***Definition* 81** (Flat context)**.** A context Γ is flat if each variable binding in it is either:

- $x : a$ where a is a flat term over Γ.
- $x : a = b$ where a and b are atoms.
- $x : F\,\overline{a_i} = b$, where a_i and b are atoms, and satisfying the following property: there exists a variable $y : F\,\overline{a_i} \in \Gamma$ iff there exists a variable $z : b \in \Gamma$.

In the above example, the first bullet point corresponds to the x_i and y_i, and the second two bullet points correspond to the h_i.

Given any context Γ we can create an equivalent flat context Γ' by repeatedly picking a subexpression b which is is not yet a left-hand-side of an equation, picking a fresh name x for it, and replacing b with x throughout the context and goal. This procedure is exactly the same as the one by Nieuwenhuis and Oliveras.

However, the proof of its correctness is slightly more complicated. The following lemmas show that this this operation does not change what equations are provable. But in addition, we sharpen the result slightly to specify what the proofs look like: the new equations (the h_i in the above example) can be used as-is as assumptions, there is no need to for the more general assumption-up-to-CC rule. We need the sharpened result to justify that the flattening algorithm is complete even though it only works on the original input context, and does not go on to recursively flatten the new equations that it introduced.

***Lemma* 82.** For any labelled context Γ, and any labelled terms a and b, we have $\Gamma, h : x = b \vdash^{L} p : a = \{b/x\}\,a$. Furthermore, every use of h in the evidence term p is of the form $h_{\triangleright\mathsf{refl}}$.

Proof. Induction on the structure of a.

- It is the variable x (a nullary label application). By CCAssumption we have $\Gamma, h : x = b \vdash^{L} h_{\triangleright\mathsf{refl}} : x = b$, as required.
- It is some other application $F\,\overline{a_i}$. Then $\{b/x\}\,(F\,\overline{a_i}) = F\;\overline{\{b/x\}\,a_i}$. By IH we get $\Gamma, h : x = b \vdash^{L} p_i : a_i = \{b/x\}\,a_i$ and hence by congruence we have $\Gamma, h : x = b \vdash^{L} \mathsf{cong}\,F\,\overline{p_i}^{\,i} : F\,\overline{a_i} = \{b/x\}\,(F\,\overline{a_i})$.

□

***Lemma* 83** (Naming subterms)**.** Suppose that x does not occur in b, and h is completely fresh. Then there exists p such that $\{b/x\}\,\Gamma \vdash^{\llcorner} p : \{b/x\}\,a_1 = \{b/x\}\,a_2$ iff there exists p' such that $\Gamma, h : x = b \vdash^{\llcorner} p' : a_1 = a_2$. Futhermore, any use of h in p' is of the form $h_{\triangleright \mathsf{refl}}$.

Proof. We prove the two directions by separate inductions. For the the "⇒" direction, the cases are:

CCPrefl We know that $\{b/x\}\,a_1 \equiv \{b/x\}\,a_2$. Apply lemma 82 to get $\Gamma, h : x = b \vdash^{\llcorner} p_1 : a_1 = \{b/x\}\,a_1$ and $\Gamma, h : x = b \vdash^{\llcorner} p_2 : \{b/x\}\,a_2 = a_2$, then conclude by transitivity.

CCPsym, CCPtrans Directly by IH.

CCPassumption We are given the derivation

$$\frac{y : \{b/x\}\,A \in \{b/x\}\,\Gamma \qquad \{b/x\}\,\Gamma \vdash^{\llcorner} q : \{b/x\}\,A = (\{b/x\}\,a_1 = \{b/x\}\,a_2)}{\{b/x\}\,\Gamma \vdash^{\llcorner} y_{\triangleright q} : \{b/x\}\,a_1 = \{b/x\}\,a_2}$$

(Where $y : A \in \Gamma$). By the IH, we have $\Gamma, h : x = b \vdash^{\llcorner} q' : A = (a_1 = a_2)$. Then apply CCPassumption again.

CCPcong We are given derivation $\{b/x\}\,\Gamma \vdash^{\llcorner} \mathsf{cong}\,F\,\overline{p_k}^{\,k} : \{b/x\}\,(F\,\overline{a_i}) = \{b/x\}\,(F\,\overline{b_i})$. Note that $\{b/x\}\,(F\,\overline{a_i}) \equiv F\,\overline{\{b/x\}a_i}$, then apply the IHs for the p_k.

CCPinj Similar to the previous case.

The cases for the "⇐" direction are:

CCPrefl Directly by CCPrefl.

CCPsymm, CCPtrans Immediate from IH.

CCPassumption We are given the the derivation

$$\frac{y : A \in (\Gamma, h : x = b) \qquad \Gamma, h : x = b \vdash^{\llcorner} q : A = (a_1 = a_2)}{\Gamma, h : x = b \vdash^{\llcorner} y_{\triangleright q} : a_1 = a_2}$$

There are two cases. If $x \equiv h$, we know $\Gamma, h : x = b \vdash^{\llcorner} q : (x = b) = (a_1 = a_2)$. By the IH we have $\{b/x\}\,\Gamma \vdash^{\llcorner} q' : (b = b) = (\{b/x\}\,a_1 = \{b/x\}\,a_2)$ By CCPinj we get $\{b/x\}\,\Gamma \vdash^{\llcorner} q_1 : b = \{b/x\}\,a_1$ and $\{b/x\}\,\Gamma \vdash^{\llcorner} q_2 : b = \{b/x\}\,a_2$. Then conclude by symmetry and transitivity.

Otherwise, $y : A \in \Gamma$. By the IH we have $\{b/x\}\,\Gamma \vdash^{\llcorner} q' : \{b/x\}\,A = (\{b/x\}\,a_1 = \{b/x\}\,a_2)$, so $\{b/x\}\,\Gamma \vdash^{\llcorner} y_{\triangleright q'} : \{b/x\}\,a_1 = \{b/x\}\,a_2$, as required.

CCPcong,CCPinj From IH, using the fact that $\{b/x\}\,(F\,\overline{a_i}) \equiv F\,\overline{\{b/x\}a_i}$.

In the assumption case for the, we are given that $a_1 = a_2 \in (\Gamma, h : x = b)$. If the equation used was h itself we must prove $\{b/x\}\, x = \{b/x\}\, b$ which is certainly true.

Otherwise, we have $(a_1 = a_2) \in \Gamma$, and we must prove $\{b/x\}\, \Gamma \vdash \{b/x\}\, a_1 = \{b/x\}\, a_2$; this follows directly by the assumption rule. □

***Lemma* 84** (Redundant equal assumptions)**.** If $\Gamma \vdash^{L}\ b_1 = b_2$, then $\Gamma, x_1 : b_1 \vdash a_1 = b_2$ iff $\Gamma, x_1 : b_1, x_2 : b_2 \vdash a_1 = a_2$

Proof. The "⇐" direction is a trivial induction. The "⇒" direction is by induction on $\Gamma, x_1 : b_1 \vdash a_1 = b_2$. The only interesting case is the assumption case, in the case when x_2 is the used assumption. Then we are given the derivation

$$\frac{\Gamma, x_1 : b_1, x_2 : b_2 \vdash b_2 = (a_1 = a_2)}{\Gamma, x_1 : b_1, x_2 : b_2 \vdash a_1 = a_2}$$

By IH we get $\Gamma, x_1 : b_1 \vdash b_2 = (a_1 = a_2)$. Then by transitivity we have $\Gamma, x_1 : b_1 \vdash b_1 = (a_1 = a_2)$, and conclude by using assumption x_1. □

***Lemma* 85** (Flattening contexts)**.** For any triple (Γ, a_1, a_2), we can find a triple (Γ', a_1', a_2'), where Γ' contains two sets of assumptions x_i and h_i, which satisfies the following:

1. For all $x_i : A \in \Gamma'$, the expression A is a flat term over Γ'.
2. For all $h_i : A \in \Gamma'$, A is an equation of the form mentioned in one of the second two bullet points of definition 81.
3. There exists some p such that $\Gamma \vdash^{L} p : a_1 = a_2$ if and only if there exists some p' such that $\Gamma' \vdash^{L} p' : a_1' = a_2'$. Furthermore, every use in p' of an assumption from the set h_i has the form $h_{i \triangleright \mathsf{refl}}$ (i.e. the conversion is just refl).

In particular, (1) and (2) implies that Γ' is a flat context.

Proof. We begin with the context Γ, and let the assumptions in it be the original set of assumptions x. Then we repeatedly use lemma 83 to add additional equations h until properties (1) is satisfied, while maintaining (2) and (3) as invariants. We write $\Gamma_0, \Gamma_1, \ldots$ for the intermediate contexts.

The original context $\Gamma_0 \equiv \Gamma$ trivially satisfies (2) and (3), since the set of assumptions h_i is empty.

Now let Γ_k be some intermediate context. If all the assumptions $x_i : A \in \Gamma_k$ are already over flat terms over Γ_k then we are done. Otherwise, A is a labelled term, so we pick a subterm of it of the form $b \equiv F\, \overline{a_i}$ with a_i atomic, pick a fresh atom c, and replace all occurances of b with c everywhere in Γ_k, a_1 and a_2. Call the resulting context Γ_k', so that $\Gamma_k \equiv \{b/c\}\Gamma_k'$. The next context is then $\Gamma_{k+1} \equiv \Gamma_k', h : b = c, x' : b$ if b occured as an assumption in Γ_k, and b$\Gamma_{k+1} \equiv \Gamma_k', h : b = c$ otherwise. We check that Γ_{k+1} still satisfies the invariants. For (1), x' is indeed a flat term over the context (thanks to h). For (2), the new equation is of the application-constant form, and either neither side is inhabited, or x and x' inhabit the two sides.

For (3), we consider the case where $\Gamma_{k+1} \equiv \Gamma'_k, h : b = c, x' : b$ (the case when there is no assumption x' is simpler). We need to show

> There exists some p such that $\Gamma_k \vdash^{\mathsf{L}} p : a_1 = a_2$ if and only if there exists some p' such that $\Gamma'_k, h : b = c, x' : b \vdash^{\mathsf{L}} p' : a'_1 = a'_2$ (with uses of hs restricted).

Lemma 83 gives us that $\Gamma_k \vdash^{\mathsf{L}} p : a_1 = a_2$ iff $\Gamma'_k, h : b = c \vdash^{\mathsf{L}} p' : a'_1 = a'_2$. And lemma 84 gives $\Gamma'_k, h : b = c \vdash^{\mathsf{L}} p' : a'_1 = a'_2$ iff $\Gamma'_k, h : b = c, x' : b \vdash^{\mathsf{L}} p' : a'_1 = a'_2$, because $x : c \in \Gamma'_k$. □

A.3.2 Main Algorithm

The state of the algorithm consists of:

- A list E of *pending equations* to be processed.
- A *representatives* table, which maps each constant c to its Union-Find representative $c' = r(c)$. Along which each representative, we store information about that equivalence class:
  - The *equality list*, $Q(c)$. The set of pairs of constants (a, b) such that $a = b$ is in this equivalence class of c'.
  - The *injectivity list*, $I(c)$. The set of tuples $(Ax_1, \ldots x_n)$ such that A is injective and $A\ x_1 \ldots x_n$ is in the equivalence class of c'.
  - The *use list*, $U(c)$: the set of input equations $y = A\ x_1 \ldots x_n$ such that c' is the representative of one of the x_i.
  - The *assumption flag*, $A(c)$. A Boolean tracking any member of the equivalence class that was inhabited by a variable in the context.

  We will overload notation slightly to let $Q(a)$ mean $Q(r(a))$ when a is not the representative of its class, and similar for I, U, and A.
- The *lookup table* (a.k.a signature table), S: maps tuples $(A, x_1, \ldots x_n)$ to an input equation $y = A\ x_1 \ldots x_n$, if such an equation exists, or to the undefined value $\perp$ otherwise.

Of these, $I(c)$, $Q(c)$, and $A(c)$ are additions which were not in the Nieuwenhuis-Oliveras algorithm.

The algorithm is initialized as follows:

$$\begin{array}{lll}
E_0 & = & \text{All the given equations in } \Gamma \\
r_0(c) & = c & \text{for all constants } c \text{ in the problem} \\
Q_0(c) & = \emptyset & \text{for all constants } c \\
I_0(c) & = \emptyset & \text{for all constants } c \\
U_0(c) & = \emptyset & \text{for all constants } c \\
A_0(c) & = \mathsf{true} \text{ iff } x : c \in \Gamma & \\
S_0(F, a_1, \ldots, a_n) & = \bot & \text{for all labels and constants}
\end{array}$$

The algorithm then proceeds by considering the pending equations one by one, updating the state and sometimes adding additional pending equations. We can show it symbolically as a transition system between tuples containing the state. (In the "merge" rule, we show the case where a rather than b is picked as the representative by the union operation, but this choice does not affect correctness, and in practice the implementation will choose one or the other depending on the size of the equivalence classes).

$$\begin{array}{rl}
\textsc{Trivial} & (E \cup \{a = b\}, r, Q, I, U, A, S) \\
\Longrightarrow & (E, r, Q, I, U, A, S) \\
 & \quad \text{when } r(a) = r(b) \text{ already}
\end{array}$$

$$\begin{array}{rl}
\textsc{Merge} & (E \cup \{a = b\}, r, Q, I, U, A, S) \\
\Longrightarrow & (E \cup \{a_i = b_i \mid (F\ a_1 \ldots a_n) \in I(a) \text{ and } (F\ b_1 \ldots b_n) \in I(b)\} \\
 & \quad \cup U(b) \\
 & \quad \cup \{c = c' \mid (c, c') \in Q(a) \wedge A(b) \wedge \neg A(a))\} \\
 & \quad \cup \{c = c' \mid (c, c') \in Q(b) \wedge A(a) \wedge \neg A(b))\}, \\
 & r', Q', I', U, A', S) \\
 & \text{where } r'(b) = r(a),\ Q'(a) = Q(a) \cup Q(b),\ I'(a) = I(a) \cup I(b), \\
 & \quad \text{and } A'(a) = A(a) \vee A(b)
\end{array}$$

$$\begin{array}{rl}
\textsc{Update1} & E \cup \{F\ a_1 \ldots a_n = a\}, r, Q, I, U, A, S) \\
\Longrightarrow & (E', r, Q', I', U', A, S') \\
 & \text{where } S'(F, a_1, \ldots, a_n) = (F\ a_1 \ldots a_n = a) \\
 & \text{when } S(F, a_1, \ldots, a_n) = \bot
\end{array}$$

$$\begin{array}{rl}
\textsc{Update2} & (E \cup \{F\ a_1 \ldots a_n = a\}, r, Q, I, U, A, S) \\
\Longrightarrow & (\quad E' \cup \{a = b\}, r, Q', I', U', A, S) \\
 & \text{when } S(F, a_1, \ldots, a_n) = (F\ b_1 \ldots b_n = b)
\end{array}$$

Where in the UPDATE1 and UPDATE2 rules,

$$\begin{array}{ll} E' & = E \cup \{a_i = b_i \mid (F\ b_1 \ldots b_n) \in I(a)\} \cup \{c = c' \mid \text{if } F\ a_1 \ldots a_n \text{ is } c = c' \text{ and } A(a)\ \} \\ Q'(a) & = Q(a) \cup \{c = c' \mid \text{if } F\ a_1 \ldots a_n \text{ is } c = c'\} \\ I'(a) & = I(a) \cup \{F\ a_1 \ldots a_n \mid \text{if } F \text{ is injective}\} \\ U'(a_i) & = U(a_i) \cup (F\ a_1 \ldots a_n = a) \quad \text{for } 1 \leq i \leq n \end{array}$$

A.3.3 Soundness

***Lemma* 86** (Invariants for soundness)**.** Suppose $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ is the initial state correponding to a flat context Γ, and $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (E, r, Q, I, U, A, S)$. Then

1. If $(a = b) \in E$, then $\Gamma \vdash^{\mathrm{L}} a = b$.
2. If $r(a) = b$, then $\Gamma \vdash^{\mathrm{L}} a = b$.
3. If $(a = b) \in Q(c)$, then $\Gamma \vdash^{\mathrm{L}} c = (a = b)$.
4. If $F\,\overline{a_i} \in I(c)$, then $\Gamma \vdash^{\mathrm{L}} c = (F\,\overline{a_i})$ and F is injective.
5. If $U(c) = (F\,\overline{a_i} = a)$, then $\Gamma \vdash^{\mathrm{L}} F\,\overline{a_i} = a$.
6. If $S(F, a_1, \ldots, a_n) = (F\,\overline{b_i} = b)$, then $\Gamma \vdash^{\mathrm{L}} F\,\overline{b_i} = b$ and $\Gamma \vdash^{\mathrm{L}} F\,\overline{a_i} = b$.
7. If $A(c) = \mathsf{true}$, then there exists some $x : A \in \Gamma$ such that $\Gamma \vdash^{\mathrm{L}} c = A$.

Proof. First, all invariants hold in the initial state $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$. (1) is true because each equation in E_0 is a hypothesis from Γ. (2) is true because r_0 is just the reflective relation. (3–6) are vacuously true since the sets Q,I,U and S are all empty. And (7) holds because of how A was initialized.

Next, we check that all the transitions of the algorithm preserves the invariants. In the TRIVIAL transition the only component of the state that changes is E, and $E' \subset E$ gets smaller so the invariant is trivially preserved. MERGE and UPDATE1/2 add additional equations, but it is easy to see that they are justified by the $\Gamma \vdash a = b$ relation.

(In the implementation, the datastructures for E,r and S store not only the terms a and b, but also proof terms $\Gamma \vdash^{\mathrm{L}} p : a = b$. Each transition constructs new proof terms from the old). □

A.3.4 Completeness

The completeness proof follows the same strategy as the proof by Corbineau [38]. We prove that at the end of a run of the algorithm, the union-find structure r has

$$\frac{}{a \approx_E a}\,\textsc{EQrefl} \qquad \frac{\begin{array}{c} a \approx_E b \\ b \approx_E c \end{array}}{a \approx_E c}\,\textsc{EQtrans} \qquad \frac{a = b \in E}{a \approx_E b}\,\textsc{EQassumption} \qquad \frac{a \approx_E b}{b \approx_E a}\,\textsc{EQsymm}$$

Figure A.4: The equivalence relation generated by a set of equations E

enough links to validate all the proof rules of the $\Gamma \vdash^{\mathrm{L}} a = b$ relation—in particular the assumption, congruence, and injectivity rules.

The invariant properties of the relation are stated in terms of the equivalence relations generated by a sets of equations. We let the letters E and R range over lists of equations,

$$E, R \quad ::= \quad \cdot \mid E, a = b$$

and write $a \approx_E b$ for the equivalence relation generated by such a list. In other words, the relation defined by the rules in Figure A.4.

The equivalence relations satisfies some simple properties:

***Lemma* 87.** If $b \approx_{(E,a=a')} b'$, then either $b \approx_E b'$, or $b \approx_E a$ and $a' \approx_E b$, or $b \approx_E a'$ and $a \approx_E b$.

Proof. Induction on the judgement $b \approx_{(E,a=a')} b'$. □

***Lemma* 88.** If $a \approx_E a'$, then $b \approx_{(E,a=a')} b'$ iff $b \approx_E b'$.

Proof. The "⇐" direction is an easy induction. For the "⇒" direction, by lemma 87 either we have $b \approx_E b'$ (and we are done), or else the equation was used. If the equation was used we have either $b \approx_E a$ and $a' \approx_E b$, or $b \approx_E a'$ and $a \approx_E b$. Either way, the conclusion follows by transitivity and symmetry. □

***Lemma* 89.** If $a \approx_E b$ or $b \approx_E a$, and the is is not an instance of reflexivity (i.e. $a \not\equiv b$), then E contains some equation of the form $a = c$ or $c = a$.

Proof. Easy induction. □

In a given a state (E, r, Q, I, U, A, S) of the algorithm, we write E for the set of equations occuring in the first component, and we write R to denote the content of r and S interpreted as a a set of equations according to the following scheme:

- One equation equation $c = c'$ whenever $r(c) = c'$.
- One equation equation $F\,\overline{a_i} = b$ whenever $\forall k.r(a_k) = a'_k$ and $S(F, a'_1, \ldots, a'_n) = (F\,\overline{b_i} = b)$.

Note that R is finite, because both r and S have finite domains. We use the notation $E \setminus E'$ to denote set-difference.

In all the following we assume that the list E has no duplicates, so we can equivocate between treating it as a set and as a list. This makes it easier to state the invariants of the algorithm (in particular invariant 2 below). In practice, if the list *does* contain duplicates they will eventually be discarded by the rule Trivial, so when implementing the algorithm there is no need to preprocess the list to remove them.

***Lemma* 90** (Monotonicity of $\approx_{E,R}$)**.** If $(E, r, Q, I, U, A, S) \Longrightarrow (E', r', Q', I', U', A', S')$ and $c_1 \approx_{E,R} c_2$, then $c_1 \approx_{E',R'} c_2$.

Proof. We consider each of the transitions in turn.

Trivial We already had the equation $a = b \in R$, so $E \cup R \equiv E' \cup R'$.

Merge We deleted the equation $a = b$ from E, and added the equation $r(a) = r(b)$ to R. By transitivity we can derive $a \approx r(a) \approx r(b) \approx b$. Then appeal to lemma 88.

Update1 We deleted the equation $F\,\overline{a_i} = a$ from E and added it to R, so $E \cup R \equiv E' \cup R'$.

Update2 By the definition of R, we already had $F\,\overline{a_i} = b \in R$. Now we deleted $F\,\overline{a_i} = a$ from E, and instead added $a = b$. By transitivity we can derive $F\,\overline{a_i} \approx b \approx a$. Then appeal to lemma 88.

□

We can now state the invariants of the algorithm.

***Lemma* 91** (Invariants for completeness of CC algorithm)**.**
Suppose $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ is the initial state correponding to a flat context Γ, and $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (E, r, Q, I, U, A, S)$. Then

1. If $x : A \in \Gamma$ then for all a, b, if $A \approx_R (a = b)$ then $a \approx_{E,R} b$.
2. If for all $0 \leq i < n$ we have $a_i \approx_R b_i$, and both $F\,\overline{a_i}$ and $F\,\overline{b_i}$ are left-hand-sides of equations in $E_0 \setminus E$, then $F\,\overline{a_i} \approx_{E,R} F\,\overline{b_i}$.
3. For all $\overline{a_i}$ and $\overline{b_i}$, if $F\,\overline{a_i} \approx_R F\,\overline{b_i}$ and F is injective, then $\forall k. a_k \approx_{E,R} b_k$.
4. If $F\,\overline{a_i} = b \in (E_0 \setminus E)$, then for all $0 \leq i < n$ we have $(F\,\overline{a_i} = b) \in U(a_i)$.
5. If $F\,\overline{a_i} = a \in (E_0 \setminus E)$ and $r(a_k) = a'_k$, then $S(F\,\overline{a_i}') = (F\,\overline{b_i} = b)$ for some equation such that $b \approx_{E,R} a$ and $b_k \approx_R a_k$. And conversely, if $S(F\,\overline{a_i}') = (F\,\overline{b_i} = b)$, then the equation $F\,\overline{a_i} = a \in (E_0 \setminus E)$ and $b \approx_{E,R} a$ and $b_k \approx_R a_k$.
6. If $c \approx_R (a = b)$, then $(a' = b') \in Q(c)$, for some constants a' and b' such that $a \approx_R a'$ and $b \approx_R b'$.

7. if $c \approx_R F\,\overline{a_i}$ for some injective label F, then $F\,\overline{a_i} \in I(c)$.
8. $A(c)$ iff $c \approx_R A$ for some A such that $x : A \in \Gamma$.
9. All equations in E, S and U are between flat terms. Also, if an equation has the form the form $F\,\overline{a_i} = a$ (label application vs atomic constant), then that equation was present in E_0, and there exists a variable $x : F\,\overline{a_i} \in \Gamma$ iff there exists a variable $y : a \in \Gamma$

Proof. First, these invariants hold for the initial state $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$.

1. In the initial state R is just the reflexive relation, so the statement simplifies to "if $x : a = b \in \Gamma$ then $a \approx_{E,R} b$". In the initial state corresponding to Γ, we have $(a = b) \in E_0$, so this is true.
2. $E_0 \setminus E$ is empty, so vacuously true.
3. R is the reflexive relation, so the only case we worry about is a reflexive equation $F\,\overline{a_i} \approx_R F\,\overline{a_i}$. Then we certainly also have $a_k \approx_{E,R} a_k$.
4. $E_0 \setminus E$ is empty, so vacuously true.
5. Both $E_0 \setminus E$ and S are empty, so both directions are vacuously true.
6. R is the reflexive relation, so we can never have an atom $\approx$ a label application.
7. Similar to invariant 6.
8. R is the reflexive relation, so $A(c)$ should be inhabited if the constant c itself is inhabited by a variable. This is exactly how A is initialized.
9. S_0 and U_0 are empty, so we only need to consider the equations in E_0. For these, the invariant is just restating part of the assumption that Γ is a flat context (definition 81).

Next, we check that the invariants are preserved by each transition

$$(E, r, Q, I, U, A, S) \Longrightarrow (E', r', Q', I', U', A', S').$$

The cases are:

Trivial Here $E = E', a = b$ and $R = R'$. By the precondition to the rule we know $a \approx_R b$, so by lemma 88 the relations $\approx_{E,R}$ and $\approx_{E',R'}$ coincide. And since $R = R'$ the relations $\approx_R$ and $\approx_{R'}$ coincide trivially. Finally, the set of expressions $F\,\overline{a_i}$ which appear as left-hand sides in $E_0 \setminus E$ and $E_0 \setminus E'$ are the same (since the only equation that changed was an atom-atom equation). It is then easy to see that all the invariants are preserved.

Merge In this transition, S is unchanged and we added one link to r. So $R' = (R, b = a')$, where we write $a' = r(a)$.

1. We are given some A, c_1, c_2 such that $A \approx_{R'} (c_1 = c_2)$, and we must show $c_1 \approx_{E',R'} c_2$.

   By lemma 87, there are two cases. Either the new equation was not used, i.e. $A \approx_R (c_1 = c_2)$. Then by the IH for the previous step we have $c_1 \approx_{E,R} c_2$. By monotonicity (lemma 90) $c_1 \approx_{E',R'} c_2$ as required.

   Otherwise the new equation was used, so we have $A \approx_R b$ and $a' \approx_R (c_1 = c_2)$ (or the symmetric $A \approx_R a'$ and $b \approx_R (c_1 = c_2)$; we show the first case w.l.o.g.). By invariant 8 we know that $A(b) = \mathsf{true}$, and by invariant 6 we know that $(c_1' = c_2') \in Q(a')$ for $c_1' \approx_R c_1$ and $c_2' \approx_R c_2$.

   Now proceed by cases on the value of $A(a')$. If $A(a') = \mathsf{true}$, then by invariant 8 we know that there is some $y : A' \in \Gamma$ such that $A' \approx_R a'$. So by invariant 1 we have $c_1 \approx_{E,R} c_2$. By monotonicity (lemma 90) $c_1 \approx_{E',R'} c_2$ as required.

   Otherwise, $A(a') = \mathsf{false}$. We have $A \approx_R b$, so by invariant 8 we known $A(b) = \mathsf{true}$. In other words, we have $A(b) \wedge \neg A(a)$. So the transition rule MERGE will add the equation $c_1' = c_2'$ to E'. Then $c_1 \approx_{E',R'} c_2$ using that new equation.

2. We are given some $F\,\overline{c_i}$ and $F\,\overline{c_i}' \in E_0 \setminus E'$, such that $\forall i. c_i \approx_{R'} c_i'$, and we need to show $F\,\overline{c_i} \approx_{E',R'} F\,\overline{c_i}'$.

   Apply lemma 87 to each of the $c_i \approx_{R'} c_i'$. Suppose that all of them fall in the first the first case, so the new equation was not used and we have $c_i \approx_R c_i'$. Then by invariant 2 we have $F\,\overline{c_i} \approx_{E,R} F\,\overline{c_i}'$. By monotonicity (lemma 90) $F\,\overline{c_i} \approx_{E',R'} F\,\overline{c_i}'$ as required.

   Otherwise, there is at least one k such that the new equation $b = a'$ was used. That is, we have $c_k \approx_R b$ and $a' \approx_R c_k'$ (or the symmetric case $c_k \approx_R a'$ and $b \approx_R c_k'$; we show the first case w.l.o.g.). So in particular c_k and b have the same representative. Now $E_0 \setminus E \supseteq E_0 \setminus E'$, so $F\,\overline{c_i} \in E_0 \setminus E$. Then by invariant 4 we have $(F\,\overline{c_i} = c_0) \in U(b)$. So by the transition rule MERGE we have $(F\,\overline{c_i} = c_0) \in E'$, contradicting the assumption that $F\,\overline{c_i} \in E_0 \setminus E'$.

3. We are given $F\,\overline{c_i} \approx_{R'} F\,\overline{c_i}'$ and must show $c_k \approx_{E',R'} c_k'$. By lemma 87 we must consider two cases.

   Either $F\,\overline{c_i} \approx_R F\,\overline{c_i}'$. Then by invariant 3 we have $F\,\overline{c_i} \approx_{E,R} F\,\overline{c_i}'$, and by monotonicity (lemma 90) $F\,\overline{c_i} \approx_{E',R'} F\,\overline{c_i}'$ as required.

   Otherwise we have $F\,\overline{c_i} \approx_R b$ and $a' \approx_R F\,\overline{c_i}'$ (or the symmetric case). So by invariant 7 we have $F\,\overline{c_i} \in I(b)$ and $F\,\overline{c_i}' \in I(a)$. So by the transition rule MERGE the equation $c_k = c_k'$ is explicitly added to E', and we have

$c_k \approx_{E',R'} c'_k$ as required.

4. $F\,\overline{a_i} = a \in (E_0 \setminus E')$. The only equation which changed was an atom-atom equation, so we also have $F\,\overline{a_i} = a \in (E_0 \setminus E)$. Then appeal to invariant 4 for the previous state.

5. For the first direction, suppose $F\,\overline{a_i} = a \in (E_0 \setminus E')$. The only equation which changed was an atom-atom equation, so we also have $F\,\overline{a_i} = a \in (E_0 \setminus E)$. Then by invariant 5 for the previous state, we have $S(F\,\overline{a_i}) = (F\,\overline{b_i} = b)$ which are suitably $\approx_{E,R}$. By monotonicity they are still $\approx_{E',R}$.

   For the converse direction, suppose that $(F\,\overline{b_i} = b)$ is in the range of S. Since the transition rule did not change S, it must still be in the range of S in the previous state. So by the invariant $F\,\overline{b_i} = b \in (E_0 \setminus E)$, and the subterms are suitably $\approx_{E,R}$. Similar to the previous paragraph, it must also be in $(E_0 \setminus E')$, and by monotonicity the subterms are still $\approx_{E',R'}$.

6. We are given some atoms c, c_1, c_2 such that $c \approx_{R'} (c_1 = c_2)$, and we must show $c'_1 = c'_2 \in Q'(c)$.

   By lemma 87, there are two cases. Either the new equation was not used, i.e. $c \approx_R (c_1 = c_2)$. Then by the IH for the previous step we have $(c'_1 = c'_2) \in Q(c)$ and hence in $Q'(c)$.

   Otherwise the new equation was used, so we have $c \approx_R b$ and $a' \approx_R (c_1 = c_2)$ (or the symmetric $A \approx_R a'$ and $b \approx_R (c_1 = c_2)$; we show the first case w.l.o.g.). By invariant 6 we have $(c1' = c2') \in Q(a)$, and hence in $Q'(c) \equiv Q'(a') \equiv Q(a) \cup Q(b)$.

7. Similar to invariant 6.

8. Similar to invariant 6.

9. The transition leaves S and U unchanged. The equations added to E are either atom-atom, or they came from U and therefore have the required form by invariant 9 for the previous state.

Update1 In this case

$$\begin{array}{lcl} E' & = & (E \setminus (F\,\overline{a_i} = a)) \cup \{c = c' \mid \text{if } F\ a_1 \ldots a_n \text{ is } c = c' \text{ and } A(a)\ \} \\ R' & = & R, F\,\overline{a_i} = a \end{array}$$

1. We are given some $A, c1, c2$ such that $A \approx_{R'} (c_1 = c_2)$, and we must show $c_1 \approx_{E',R'} c_2$.

   By lemma 87, there are two cases. Either the new equation was not used, i.e. $A \approx_R (c_1 = c_2)$. Then by invariant 1 we have $c_1 \approx_{E,R} c_2$. By monotonicity (lemma 90) $c_1 \approx_{E',R'} c_2$ as required.

Otherwise the new equation was used, which can happen in two ways.

- We have $A \approx_R F\,\overline{a_i}$ and $a \approx_R (c_1 = c_2)$.

  By lemma 89, unless $A \equiv F\,\overline{a_i}$ that means that R must contain some equation mentioning $F\,\overline{a_i}$. However, this is impossible: each equation in R comes either from r (but this only relates constants, not label applications) or from S (but we know as a premise to the rule that $S(F\,\overline{a_i}) = \bot$).

  On the other hand, if $A \equiv F\,\overline{a_i}$, then the assumption says that $x : (F\,\overline{a_i}) \in \Gamma$, so by invariant 9 we know that $x : a \in \Gamma$. So by invariant 1 we know $c_1 \approx_{E,R} c_2$, and hence by monotonicity $c_1 \approx_{E',R'} c_2$.

- We have $A \approx_R a$ and $F\,\overline{a_i} \approx_R (c_1 = c_2)$.

  By invariant 8 we then have $A(a) = \mathsf{true}$. So by the transition rule Update1 the equation $c_1 = c_2$ is explicitly added to E', and we have $c_1 \approx_{E',R'} c_2$ as required.

2. We are given some $G\,\overline{c_i}$ and $G\,\overline{c_i}' \in E_0 \setminus E'$, such that $\forall i.c_i \approx_{R'} c_i'$, and we need to show $G\,\overline{c_i} \approx_{E',R'} G\,\overline{c_i}'$.

   Apply lemma 87 to all the $c_i \approx_{R'} c_i'$. If the new equation was not used for any of them, we have $\forall i.c_i \approx_R c_i'$. Using the assumption $G\,\overline{c_i} \in E_0 \setminus E'$, invariant 5, and the fact that $S(F\,\overline{a_i}) = \bot$ we know that $G\,\overline{c_i} \not\equiv F\,\overline{a_i}$. This means that we must also have $G\,\overline{c_i} \in E_0 \setminus E$, and similar for $G\,\overline{c_i}'$. Hence by invariant 2 for the previous state and monontonicity we get $G\,\overline{c_i} \approx_{E',R'} G\,\overline{c_i}'$.

   Otherwise, the new equation $F\,\overline{a_i} = a$ was used for at least one c_k, which can happen in two ways.

   - We have $c_k \approx_R F\,\overline{a_i}$ and $a \approx_R c_k'$.

     By lemma 89, unless $A \equiv F\,\overline{a_i}$ that means that R must contain some equation mentioning $F\,\overline{a_i}$. However, this is impossible: each equation in R comes either from r (but this only relates constants, not label applications) or from S (but we know as a premise to the rule that $S(F\,\overline{a_i}) = \bot$).

     So we must have $c_k \equiv (F\,\overline{a_i})$. That means that $G\,\overline{c_i}$ has the form $G\ c_1 \ldots (F\,\overline{a_i}) \ldots c_n$. However, according to invariant 9, $G\,\overline{c_i}$ should be a flat term, so this also cannot happen.

   - We have $c_k \approx_R a$ and $F\,\overline{a_i} \approx_R c_k'$.

     The reasoning in this case is similar, using c_k' instead of c_k.

3. We are given some injective G such that $G\,\overline{c_i} \approx_{R'} G\,\overline{c_i}'$, and we must show $c_k \approx_{E',R'} c'_k$.

   By lemma 87, the new equation is either used or not. If not, we have $G\,\overline{c_i} \approx_R G\,\overline{c_i}'$, so by invariant 3 we get $c_k \approx_{E,R} c'_k$ and hence by monotonicity (lemma 90) $c_k \approx_{E',R'} c'_k$ as required.

   Otherwise the equation is used and we have either $G\,\overline{c_i} \approx_R F\,\overline{a_i}$ and $a \approx_R G\,\overline{c_i}'$, or the symmetric situation. W.l.o.g. we consider the first case.

   By lemma 89, unless $G\,\overline{c_i} \equiv F\,\overline{a_i}$, there must be some equation in R involving $F\,\overline{a_i}$. But that is impossible by invariant 5, since by the premise to the rule UPDATE1 we know that $S(F\,\overline{a_i}) = \bot$.

   On the other hand, if $G\,\overline{c_i} \equiv F\,\overline{a_i}$, then we are given a new equation $F\,\overline{c_i} = a$ and we know $a \approx_R F\,\overline{c_i}'$. So by invariant 7 we know $F\,\overline{c_i} \in I(a)$. So the transition rule UPDATE1 adds the equation $ck = ck'$ to E', and we have $c_k \approx_{E',R'} c'_k$.

4. Supposed $(G\,\overline{c_i} = c) \in (E_0 \setminus E')$. The set $E_0 \setminus E$ contains all equations in $E_0 \setminus E'$ except for $F\,\overline{a_i} = a$. So there are two cases. If $(G\,\overline{c_i} = c) \not\equiv (F\,\overline{a_i} = a)$, then we also have $(G\,\overline{c_i} = c) \in (E_0 \setminus E)$, and can appeal to invariant 4 for the previous state. MERGE transition. Otherwise, if $(G\,\overline{c_i} = c) \equiv (F\,\overline{a_i} = a)$, then the transition rule explicitly adds the equation to U'.

5. For the first direction, suppose $G\,\overline{c_i} = c \in (E_0 \setminus E')$. The set $E_0 \setminus E$ contains all equations in $E_0 \setminus E'$ except for $F\,\overline{a_i} = a$. So there are two cases. If $(G\,\overline{c_i} = c) \not\equiv (F\,\overline{a_i} = a)$, then we also have $(G\,\overline{c_i} = c) \in (E_0 \setminus E)$. So we can use similar reasoning as in the corresponding case for the MERGE transition. Otherwise, if $(G\,\overline{c_i} = c) \equiv (F\,\overline{a_i} = a)$, then in the new state we have $S'(F, a_1, \ldots, a_n) = (F\,\overline{a_i} = a)$. Certainly $a_k \approx_{R'} a_k$ and $a \approx_{E',R'} a$, as required.

   For the converse direction, suppose that $(G\,\overline{c_i} = c)$ is in the range of S'. Again there are two cases. If is was already in the range of S, we reason similarly to the corresponding case for the MERGE transition. Otherwise, if it is the new equation, then by invariant 9 that equation is in E_0, and by the transition rule it is no longer in E', so it is in $(E_0 \setminus E')$ as required.

6. We are given some $c \approx_{R'} (c_1 = c_2)$, and must show that some suitable $(c'_1 = c'_2) \in Q'(c)$.

   By lemma 87, the new equation from S' is either used or not. If not, we have $c \approx_R (c_1 = c_2)$, and get $(c_1 = c_2) \in Q(c)$ by invariant 6 for the previous state. Otherwise the equation was used, which can happen in two ways:

- $c \approx_R F\,\overline{a_i}$ and $a \approx_R (c_1 = c_2)$. But we know that c and $F\,\overline{a_i}$ are different (one is an atom and one is a label application), so by lemma 89 that would mean that R contains an equation mentioning $F\,\overline{a_i}$, which is impossible since $S(F\,\overline{a_i} = \bot$.
- $c \approx_R a$ and $F\,\overline{a_i} \approx_R (c_1 = c_2)$. By reasoning similar to the previous paragraph this can only happen if $F\,\overline{a_i} \equiv (c_1 = c_2)$. In that case we have $(c_1 = c_2) \in Q'(c) \equiv Q'(a)$, since it was explicitly added by the transition rule Update1.

7. Similar to invariant 6.
8. Similar to invariant 6.
9. We modify S and U by adding the equation $F\,\overline{a_i} = a$; this equation comes from E so by the invariant from the previous state it is good. And all the new equations in E' are atom-atom.

Update2 In this transition $R' = R$.

1. We are given som A, c_1, c_2 such that $A \approx_{R'} (c_1 = c_2)$. So $A \approx_R (c_1 = c_2)$. Then by invariant 1 and monotonicity (lemma 90) we have $c_1 \approx_{E',R'} c_2$ as required.
2. We are given some $G\,\overline{c_i}$ and $G\,\overline{c_i}' \in E_0 \setminus E'$, and we need to show $G\,\overline{c_i} \approx_{E',R'} G\,\overline{c_i}'$.

   By assumption we have $\forall i. c_i \approx_{R'} c_i'$. So $\forall i. c_i \approx_R c_i'$.

   If $G\,\overline{c_i} \not\equiv F\,\overline{a_i}$, then we must also have $G\,\overline{c_i} \in (E_0 \setminus E)$ (since only one equation was removed from E), and similarly for $G\,\overline{c_i}'$. So then by invariant 2 and monotonicity we have $G\,\overline{c_i} \approx_{E',R'} G\,\overline{c_i}'$ as required.

   Otherwise, we are given $\forall i. a_i \approx_R c_i'$, and we need to prove $F\,\overline{a_i} \approx_{E',R'} F\,\overline{c_i}'$. From the premise to the rule we know $S(F, r(a_1), \ldots, r(a_n)) = (F\ b_1, \ldots, b_n = b)$, so by invariant 5 we know that there is some equation $F\,\overline{b_i} = b \in (E_0 \setminus E)$ where $b_k \approx_R a_k$. So by transitivity we have $b_k \approx_R a_k$. Then by invariant 2 we have $F\,\overline{b_i} \approx_{E,R} F\,\overline{c_i}'$, and by monotonicity (lemma 90) $F\,\overline{b_i} \approx_{E,R} F\,\overline{c_i}'$. By the definition of R we have $F\,\overline{a_i} = b \in R$. So by transitivity $F\,\overline{a_i} \approx b \approx F\,\overline{b_i} \approx F\,\overline{c_i}'$ as required.
3. We are given some injective G such that $G\,\overline{c_i} \approx_{R'} G\,\overline{c_i}'$ and we must show $c_k \approx_{E',R'} c_k'$. By invariant 3 we know $c_k \approx_{E,R} c_k'$. Then apply monotonicity (lemma 90).
4. Similar to the case for Update1
5. For the first direction, suppose $G\,\overline{c_i} = c \in (E_0 \setminus E')$. The set $E_0 \setminus E$ contains all equations in $E_0 \setminus E'$ except for $F\,\overline{a_i} = a$. So there are two cases. If

$(G\,\overline{c_i} = c) \not\equiv (F\,\overline{a_i} = a)$, then we also have $(G\,\overline{c_i} = c) \in (E_0 \setminus E)$. So we can use similar reasoning as in the corresponding case for the Merge transition. Otherwise, if $(G\,\overline{c_i} = c) \equiv (F\,\overline{a_i} = a)$, then in the new state we have $S'(F, a_1, \ldots, a_n) = (F\,\overline{b_i} = b)$, and we need to prove $a_k \approx_{R'} b_k$ and $b \approx_{E',R'} a$. We get $a_k \approx_{R'} b_k$ from invariant 5 for the previous state, and we get $a \approx_{E',R'} b$ from the equation that this transition added.

For the converse direction, suppose that $(G\,\overline{c_i} = c)$ is in the range of S'. Since $S = S'$ it is was already in the range of S, we reason similarly to the corresponding case for the Merge transition.

6. We are given some $c \approx_{R'} (c_1 = c_2)$, and must show that some suitable $(c_1' = c_2') \in Q'(c)$.

   By lemma 87, the new equation from S' is either used or not. If not, we have $c \approx_R (c_1 = c_2)$, and get $(c_1 = c_2) \in Q(c)$ by invariant 6 for the previous state. Otherwise the equation was used, which can happen in two ways:

   - $c \approx_R F\,\overline{a_i}$ and $a = (c_1 = c_2)$. Then by invariant 6 for the previous state we have $(c_1 = c_2) \in Q(a)$, and hence in $Q'(c)$.
   - $c \approx_R a$ and $F\,\overline{a_i} \approx_R (c_1 = c_2)$. The only equations mentioning $F\,\overline{a_i}$ in R are those arising from $S(F, a_1, \ldots, a_n)$, so this can only happen in two ways. Either $(F\,\overline{a_i}) \equiv (c_1 = c_2)$, in which case the transition rule explicitly adds $(c_1 = c_2)$ to $Q'(a)$. Or else the transition was via b, i.e. we had $F\,\overline{a_i} \approx_R b \approx_R (c_1 = c_2)$. In this case we know $(c_1' = c_2') \in Q(b)$ from invariant 6 for the previous state, and hence it is also in $Q'(c)$.

7. Similar to invariant 6.
8. Similar to invariant 6.
9. Similar to the corresponding case for the Update1 transition.

□

The invariants in lemma 91 shows that the equivalence relation $\approx_R$ constructed by the algorithm is "locally" complete: it satifies the congruence rule as long as the conclusion of the rule only contains subterms from the context E_0. In order to show that it is "globally" complete, we need to know that all provable equations are provable using only subterms of the problem. One way to do that is to use the notion of normal-form evidence terms which we introduced previously.

***Lemma* 92** (Completeness for normal-form evidence terms)**.** Suppose Γ is a context of the form described in lemma 85, and let $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ be the initial state of the algorithm for Γ, and suppose $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (\cdot, r, Q, I, U, A, S)$. Then:

- If $\Gamma \vdash^{\llcorner} pS : A = B$, then A and B are flat terms over Γ and $A \approx_R B$.
- If $\Gamma \vdash^{\llcorner} pC : A = B$ and A and B are flat terms over Γ, then $A \approx_R B$.
- If $\Gamma \vdash^{\llcorner} p^* : A = B$ and A and B are flat terms over Γ, then $A \approx_R B$.
- If $\Gamma \vdash^{\llcorner} p^*_{\mathsf{R}} : A = B$ and A is a flat term over Γ, then B is a flat term over Γ and $A \approx_R B$.

provided that every use of assumptions h in the proofs pS,pC, p^* and p^*_{R} either refer to an assumption $h : A \in \Gamma$ where A is a flat term over Γ, or are of the form $h_{\triangleright \mathsf{refl}}$.

Proof. We proceed by induction on the structure of the given evidence term. The cases for pS are:

The evidence is $x_{\triangleright p^*_{\mathsf{R}}}$ From the premises to the rule we know we have $x : A \in \Gamma$ and $\Gamma \vdash^{\llcorner} p^*_{\mathsf{R}} : A = (a = b)$. By the assumptions to there are two possibilities for A

Either A is a flat term over that context Then by the mutual IH for p^*_{R}, $a = b$ is flat as well (as required), and $A \approx_R (a = b)$. By invariant 1 we have $a \approx_R b$ as required.

Or else, $p^*_{\mathsf{R}} \equiv \mathsf{refl}$, so $A \equiv (a = b)$. By the definition of flat context (definition 81) A can one of three things: either a flat term (so this is a label application of the label "=", and a and b are atoms), or an equation between atoms (so a and b are atoms), or a application-atom equation (so a is a label application, and by virtue of this precise equation it is a flat term over Γ). In either of the three cases a and b are flat terms over Γ as required, and by invariant 1 we have $a \approx_R b$ as required.

The evidence is $(x_{\triangleright p^*_{\mathsf{R}}})^{-1}$ Similar to the above case we get $a \approx_R b$, and therefore $b \approx_R a$ by symmetry.

The evidence is $\mathsf{inj}\, i\, pS$ By the IH for pS, we know pS proves an equation between flat terms. Since the injectivity rule applies, they must be two label applications, $\Gamma \vdash^{\llcorner} pS : F\, \overline{a_i} = F\, \overline{b_i}$. So the conclusion of the rule is an equation $a_k = b_k$ between two atoms, and atoms are flat over any context.

By the IH we also know $F\, \overline{a_i} \approx_R F\, \overline{b_i}$, so by invariant 3 we get $a_k \approx_R b_k$.

The evidence is a chain p^*_{LR} In other words, it is either a single term pS, which we dealt with in the previous cases, or it is is a chain starting and ending with a synthesizable term, that is p^*_{LR} is $pS; q^*; rS$. In the latter case we use the IHs for pS and rS to see that two two sides of the equation q^* are flat terms, appeal to the mutual IH for q^*, and use transitivity to chain together the three equations.

The only case for pC is when **the evidence term is a use of congruence,** $\mathsf{cong}\, F\, p_1 \mathinner{..} p_i$. The only rule that applies is CCPcong, so the equation in the conclusion must be between two label applications, $F\,\overline{a_i} = F\,\overline{b_i}$. By assumption we know that they are flat terms over Γ, i.e. both label applications appear as left-hand sides of equations in Γ and all the a_i and b_i are atoms.

Since a_i and b_i are atoms they are per definition flat over Γ, so the IHs apply and give $a_i \approx_R b_i$.

The initial context E_0 contains all equations in Γ, in particular it contains the defining equations for $F\,\overline{a_i}$ and $F\,\overline{b_i}$. So by invariant 2 we get $F\,\overline{a_i} \approx_R F\,\overline{b_i}$ as required.

The cases for p^* are:

The empty chain (refl) We then have $a \approx_R a$ by reflexivity of $\approx$.

A chain consisting of a single term, p The evidence term p must be either a checkable are a synthesizable term, so we appeal to the corresponding mutual IH.

In the case when it is a pS,

A chain of length > 1 The definition of chains stipulates that there must never be two adjacent pCs, so we know that the either the first or the second evidence term in the chain is a pS. This is similar to the case for p^*_{LR} above.

The cases for p^*_{R} are similar to the case for p^*_{LR} above. □

***Lemma* 93** (Termination of the CC algorithm)**.** If $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ is the initial state corresponding to some (flat) context Γ, there exists some final state with an empty list of pending equations such that $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (\cdot, r, Q, I, U, A, S)$.

Proof. Consider the (finite) set X of all flat terms occuring in Γ. The termination metric is the lexicographic order on (Number of equivalence classes on X induced by R)×(Number of application-atom equations in E)×(Number of atom-atom equations in E).

None of the rules can increase the number of equivalence classes. Trivial leaves number of app-atom equations unchanged and decreases atom-atom equations. Merge adds all kinds of equations, but reduces the number of equivalence classes. Update1/2 adds atom-atom equations but decrease the number of app-atom equations. □

***Theorem* 94** (Correctness of the CC algorithm)**.** Suppose Γ is any context, Γ' is the flattened version of Γ, and $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ is the initial state of the algorithm corresponding to Γ'. Then $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (\cdot, r, Q, I, U, A, S)$, and for any atomic a and b, we have $\Gamma \vdash a = b$ iff $a \approx_R b$.

$$\frac{}{\vDash \emptyset : \Gamma = \Gamma}\textsc{EEsame} \qquad \frac{\begin{array}{l}\vDash \sigma : \Gamma = \Gamma' \\ \Gamma \vdash A : \mathsf{Type} \\ \Gamma' \vDash A' = \sigma\, A \\ \Gamma' \vdash v : A' = \sigma\, A\end{array}}{\vDash \sigma \left\{x_{\triangleright v}/x\right\} : \Gamma, x : A = \Gamma', x : A'}\textsc{EEcons}$$

Figure A.5: Context equivalence

Proof. By lemma 93 we know the algorithm will terminate in a state with E empty. In that state, if a and b have the same r-representative then by lemma 86 invariants 2 and 6 we know $\Gamma \vdash a = b$.

Conversely, suppose that $\Gamma \vdash a = b$, so $\Gamma \vdash^{\llcorner} p : a = b$ for some p. By lemma 85 we know that $\Gamma' \vdash^{\llcorner} p' : a = b$ for some proof p' where every assumption is either a flat term or plain assumption $h_{\triangleright \mathsf{refl}}$. (We know that a and b are not changed by the flattening step since they were assumed to be atoms). By lemma 61 we have $\Gamma' \vdash^{\llcorner} p^* : a = b$ for some p^*, and inspecting the proof of that lemma we see that p^* still obeys the restricton on assumptions. Then by lemma 92 we have $a \approx_R b$. □

Requiring a and b to be atoms is not a serious restriction: if we want to check some non-atomic terms a' and b' for equality we can pick fresh constants a and b, and add the equations $a = a'$ and $b = b'$ to the context. Also, checking whether $a \approx_R b$ is a cheap operation. Since they are both atoms, the wanted equation is true iff in the final state of the algorithm a and b are in the same union-find class (have the same r-representative).

A.4 Proofs about the core language

A.4.1 Equivalent contexts

The next properties concern a new relation $\vDash \sigma : \Gamma = \Gamma'$, defined in Figure A.5, which uses a substitution σ as the witness to the equivalence of two contexts.

***Lemma* 95** (Regularity for context equivalence)**.** If $\vDash \sigma : \Gamma = \Gamma'$, then $\vdash \Gamma$ and $\vdash \Gamma'$.

Proof. Induction on $\vDash \sigma : \Gamma = \Gamma'$. In the EEsame case we have this as a premise. In the EEcons case, we have $\Gamma \vdash A : \mathsf{Type}$ as a premise, and get $\Gamma' \vdash A' : \mathsf{Type}$ from regularity of the congruence closure relation (lemma 58). □

***Lemma* 96** (Variables in equivalent contexts)**.** If $y : C \in \Gamma$, and $\vDash \sigma : \Gamma = \Gamma'$, then there exists C' such that $y : C' \in \Gamma'$ and $\Gamma \vDash C' = \sigma\, C$.

Proof. Induction on $\vDash \sigma : \Gamma = \Gamma'$. The EEsame case is trivial.

In the EEcons case, the rule looks like

$$\frac{\begin{array}{l}\vDash \sigma : \Gamma = \Gamma' \\ \Gamma \vdash A : \mathsf{Type} \\ \Gamma' \vDash A' = \sigma\, A \\ \Gamma' \vdash v : A' = \sigma\, A\end{array}}{\vDash \sigma \left\{x_{\triangleright v}/x\right\} : \Gamma, x : A = \Gamma', x : A'}\,\textsc{EEcons}$$

There are two cases. If $x = y$, so $A = C$, then we can pick $C' := A'$, and we have $\Gamma' \vDash C' = \sigma\, C$ as a premise. By weakening (lemma 57) we have $\Gamma', x : A' \vDash C' = \sigma\, C$ as required.

If $x \neq y$, then $y : C \in \Gamma$, so by IH we have $y : C' \in \Gamma'$ with $\Gamma' \vDash C' = \sigma\, C$. Again, use weakening to get $\Gamma', x : A' \vDash C' = \sigma\, C$. □

***Lemma* 97** (Context conversion preserves erasure)**.** If $\vDash \sigma : \Gamma = \Gamma'$, then for any expression a we have $|\sigma\, a| = |a|$.

Proof. Examining the definition of $\vDash \sigma : \Gamma = \Gamma'$ we see that the substitution only adds type casts, which are erased. □

***Lemma* 98** (Context conversion for annotated language, var case)**.**
If $x : A \in \Gamma$ and $\vDash \sigma : \Gamma = \Gamma'$, then $\Gamma' \vdash \sigma\, x : \sigma\, A$.

Proof. Induction on the length of Γ.

Γ is empty This contradicts the assumption that $x \in \Gamma$.

Γ is $\Gamma_0, y : B$ for some $y \neq x$ Then by considering the possible derivations of $\vDash \sigma : \Gamma = \Gamma'$ we know we have $\vDash \sigma_0 : \Gamma_0 = \Gamma'_0$ (and so on). By the IH we have $\Gamma'_0 \vdash \sigma_0\, x : \sigma_0\, A$. So by weakening (lemma 47) we have $\Gamma' \vdash \sigma_0\, x : \sigma_0\, A$. Since x is a bound variable we can pick it to not be in the domain of σ_0, and since $\Gamma_0 \vdash A : \mathsf{Type}$ we know $x \ \notin \mathsf{FV}(A)$. So the is equivalent to $\Gamma' \vdash \sigma\, x : \sigma\, A$.

Γ is $\Gamma_0, x : A$ By considering the possible derivations of $\vDash \sigma : \Gamma = \Gamma'$ we know that we must have

$$\frac{\begin{array}{l}\vDash \sigma : \Gamma = \Gamma' \\ \Gamma \vdash A : \mathsf{Type} \\ \Gamma' \vDash A' = \sigma\, A \\ \Gamma' \vdash v : A' = \sigma\, A\end{array}}{\vDash \sigma \left\{x_{\triangleright v}/x\right\} : \Gamma, x : A = \Gamma', x : A'}\,\textsc{EEcons}$$

So in particular we know $\sigma\, x$ is $x_{\triangleright v}$, which by Tcast has the type $\sigma\, A$.

□

***Lemma* 99** (Context conversion for annotated language)**.**
If $\Gamma \vdash a : A$ and $\vDash \sigma : \Gamma = \Gamma'$, then $\Gamma' \vdash \sigma\, a : \sigma\, A$.

Proof. Induction on $\Gamma \vdash a : A$.

Tvar By lemma 98.

Ttype Trivial.

Tpi The IH for A gives $\Gamma' \vdash \sigma\, A : \mathsf{Type}$.

By TCCREFL we have $\Gamma' \vDash \sigma\, A = \sigma\, A$, and it is easy to pick some identify proof v such that $\Gamma' \vdash v : \sigma\, A = \sigma\, A$. Then by EECONS, $\vDash \sigma\, \{x_{\triangleright v}/x\} : \Gamma, x : \sigma\, A = \Gamma', x : \sigma\, A$.

So by the IH, we get $\Gamma', x : \sigma\, A \vdash \sigma\, B : \mathsf{Type}$.

Now apply TPI to get $\Gamma' \vdash (x : \sigma\, A) \to \sigma\, B : \mathsf{Type}$ as required.

Tipi Similar to the previous case.

Trec By the IH we get $\Gamma' \vdash (x : \sigma\, A_1) \to \sigma\, A_2 : \mathsf{Type}$.

By reasoning similar to the TPI case we get

$$\begin{aligned} \vDash \sigma\, \{f_{\triangleright v_1}/f\}\, \{x_{\triangleright v_2}/x\} : \quad & \Gamma, f : (x : \sigma\, A_1) \to \sigma\, A_2, x : \sigma\, A_1 \\ = \quad & \Gamma', f : (x : \sigma\, A_1) \to \sigma\, A_2, x : \sigma\, A_1 \end{aligned}$$

and hence by IH we get $\Gamma', f : (x : \sigma\, A_1) \to \sigma\, A_2, x : \sigma\, A_1 \vdash \sigma\, a : \sigma\, A_2$.

Now apply TREC to get $\Gamma' \vdash \mathsf{rec}\ f_{(x\sigma\, A_1) \to \sigma\, A_2}\ x.\sigma\, a : (x : \sigma\, A_1) \to \sigma\, A_2$ as required.

Tirec Similar to the previous case.

Tdapp By the IHs for a and v we get $\Gamma' \vdash \sigma\, a : \sigma\, (x : A) \to B$ and $\Gamma' \vdash \sigma\, v : \sigma\, A$. Then apply TDAPP.

Tapp,Tidapp, Teq Similar to the previous case.

Tjoinc By the IH we get $\Gamma' \vdash \sigma\, a = \sigma\, b : \mathsf{Type}$. Context equivalences preserve erasure (lemma 97), so $|\sigma\, a| = |a|$, and therefore we still have $|\sigma\, a| \leadsto^{i}_{\mathsf{cbv}} c$. Similarly, $|\sigma\, b| \leadsto^{i}_{\mathsf{cbv}} c$. Then apply TJOINC.

Tjoinp Similar to the previous case.

Tjinjdom By the IH we have $\Gamma' \vdash \sigma\, v : ((x : \sigma\, A_1) \to \sigma\, B_1) = ((x : \sigma\, A_2) \to \sigma\, B_2)$. Then by TJINJDOM we do indeed have $\Gamma' \vdash \mathsf{join}_{\mathsf{injdom}\, \sigma\, v} : \sigma\, A_1 = \sigma\, A_2$

Tjinjrng,Tjiinjdom,Tjiinjrng. Tinjeq Similar to the previous case.

Tjsubst The IHs give $\forall k.\ \Gamma' \vdash \sigma\, v_k : \sigma\, a_k = \sigma\, b_k$ and $\Gamma' \vdash \sigma\, B : \mathsf{Type}$. Since context equivalence preserves erasure (lemma 97) the premise $|B| = |(\{a_1/x_1\} \ldots \{a_j/x_j\}\ c = \{b_1/x_1\} \ldots \{b_j/x_j\}\ c)|$ is unchanged. Then apply TJSUBST.

Tcast The IHs give $\Gamma' \vdash \sigma\, a : \sigma\, A$ and $\Gamma' \vdash \sigma\, v : \sigma\, A = \sigma\, B$ and $\Gamma' \vdash \sigma\, B : \mathsf{Type}$. Then by TCAST we have $\Gamma' \vdash (\sigma\, a)_{\triangleright \sigma\, v} : \sigma\, B$ as required.

□

Lemma* 100** (Context conversion for congruence closure).***
If $\vDash \sigma : \Gamma = \Gamma'$, then $\Gamma \vDash a = b$ implies $\Gamma' \vDash \sigma\, a = \sigma\, b$.

Proof. By induction on $\Gamma \vDash a = b$. The cases are

TCCrefl By context conversion for the annotated language (lemma 99), we have $\Gamma' \vdash \sigma\, a : \sigma\, A$. Then apply TCCREFL again.

TCCerasure By context conversion for the annotated language (lemma 99), $\sigma\, a$ and $\sigma\, b$ are well-typed in Γ'. And applying a context equivalence σ does not affect the erasure of a term (lemma 100). Then apply TCCERASURE again.

TCCsym Direct by IH.

TCCtrans Direct by IH.

TCCassumption The rule looks like

$$\frac{\Gamma \vDash C = (a = b) \qquad y : C \in \Gamma}{\Gamma \vDash a = b}$$

By the IH we know $\Gamma' \vDash \sigma\, C = \sigma\,(a = b)$.

By lemma 96 there exists $y : C' \in \Gamma'$ with $\Gamma' \vDash C' = \sigma\, C$. So by transitivity (TCCTRANS) we have $\Gamma' \vDash C = \sigma\,(a = b)$. Note that $\sigma\,(a = b) \equiv (\sigma\, a = \sigma\, b)$ Apply TCCASSUMPTION.

TCCcongruence The given rule looks like

$$\frac{\begin{array}{c}\Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k \\ |A = B| = |\{a_1/x_1\} \ldots \{a_j/x_j\}\ c = \{b_1/x_1\} \ldots \{b_j/x_j\}\ c|\end{array}}{\Gamma \vDash A = B}\ \text{TCCCONGRUENCE}$$

By IH we know $\forall k.\ \Gamma' \vDash \sigma\, a_k = \sigma\, b_k$.

By context conversion for the annotated language (lemma 99) we know $\Gamma' \vdash \sigma\, A = \sigma\, B : \mathsf{Type}$. And since context equivalences do not affect the erasure of terms (lemma 97) we still have

$$|\sigma\, A = \sigma\, B| = \{\sigma\, a_1/x_1\} \ldots \{\sigma\, a_j/x_j\}\ c = \{\sigma\, b_1/x_1\} \ldots \{\sigma\, b_j/x_j\}\ c.$$

Now apply TCCcongruence.

TCCinjdom, TCCinjrng, TCCiinjdom, TCCiinjrng,TCCinjeq Direct by IH.

□

***Lemma* 101** (Symmetry of context equivalence)**.**
If $\vDash \sigma : \Gamma = \Gamma'$, then there exists ρ such that $\vDash \rho : \Gamma' = \Gamma$.

Proof. By induction on the judgement $\vDash \sigma : \Gamma = \Gamma'$. The EEsame case is trivial.

In the EEcons case we are given

$$\frac{\begin{array}{l}\vDash \sigma : \Gamma = \Gamma' \\ \Gamma \vdash A : \mathsf{Type} \\ \Gamma' \vDash A' = \sigma\, A \\ \Gamma' \vdash v : A' = \sigma\, A\end{array}}{\vDash \sigma\,\{x_{\triangleright v}/x\} : \Gamma, x : A = \Gamma', x : A'}\ \text{EEcons}$$

By IH we have $\vDash \rho : \Gamma' = \Gamma$.

Using that ρ to apply context conversion (lemma 100) to the premise $\Gamma' \vDash A' = \sigma\, A$, we get $\Gamma \vDash \rho\, A' = \rho\,(\sigma\, A)$.

By regularity of the context equivalence relation (lemma 95) we know $\Gamma \vdash A : \mathsf{Type}$, and since context equivalence preserves erasure (lemma 97) we know $|\rho\,(\sigma\, A)| = |A|$. So by TCCErasure, we have $\Gamma \vDash \rho\, A' = A$. By TCCSym we get $\Gamma \vDash A = \rho\, A'$.

Furthermore, by lemma 72 this equation is witnessed by some value $\Gamma \vdash v : A = \rho\, A'$. Now pick $\rho\,\{x_{\triangleright v}/x\}$ as the witnessing substitution. □

***Lemma* 102** (Context equivalence symmetry is an inverse)**.**
If $\vDash \sigma : \Gamma = \Gamma'$ and $\vDash \rho : \Gamma' = \Gamma$ and $\Gamma \vdash a : A$, then $\Gamma \vDash \rho\,\sigma\, a = a$.

Proof. Since the substitutions only change erased parts of the term (lemma 97), $|\rho\,\sigma\, a| = |a|$. And by applying context conversion (lemma 99) twice we have $\Gamma \vdash \rho\,\sigma\, a : \rho\,\sigma\, A$. So by TCCErasure, $\Gamma \vDash \rho\,\sigma\, a = a$. □

***Lemma* 103** (Contexts are equivalent if they are equal up to erasure)**.**
If $\vdash \Gamma$ and $\vdash \Gamma'$ and $|\Gamma| = |\Gamma'|$, then there exists σ such that $\vDash \sigma : \Gamma = \Gamma'$.

Proof. Induction of the length of the contexts. (We know that Γ and Γ' have the same length since they erase to the same thing).

- Two empty contexts are trivially equivalent.
- Suppose the contexts are $\Gamma, x : A$ and $\Gamma', x : A'$. By inversion on $\vdash \Gamma, x : A$ we get $\vdash \Gamma$ and $\Gamma \vdash A : \mathsf{Type}$, and similarly we get $\vdash \Gamma'$ and $\Gamma' \vdash A' : \mathsf{Type}$. And we know $|\Gamma| = |\Gamma'|$ and $|A| = |A'|$.

By the IH we know that there exists some σ such that $\vDash \sigma : \Gamma = \Gamma'$. By context conversion (lemma 99) we have $\Gamma' \vdash \sigma\, A : \mathsf{Type}$. And since context equivalences do not affect erasure, the $|\sigma\, A| = |A|$. Thus, A' and $\sigma\, A$ are two well-typed terms which are equal up to erasure, so by TCCERASURE we have $\Gamma' \vDash A' = \sigma\, A$.

Finally, picking the term $v = \mathsf{join}_{\leadsto_{\mathsf{cbv}} 00 : A' = \sigma\, A}$, we have $\Gamma' \vdash v : A' = \sigma\, A$.

So applying EECONS we have

$$\vDash \sigma\, \{x_{\triangleright v}/x\} : \Gamma, x : A = \Gamma', x : A'$$

as we wanted to prove.

□

***Lemma* 104** (Context conversion for injrng)**.**
If $\Gamma \vDash \mathsf{injrng}\, A\, \mathsf{for}\, v$ and $\vDash \sigma : \Gamma = \Gamma'$, then $\Gamma' \vDash \mathsf{injrng}\, \sigma\, A\, \mathsf{for}\, \sigma\, v$.

Proof. We only show the case when A is $(x : A_1) \rightarrow\; A_2$; the case when A is $\bullet(x : A_1) \rightarrow\; A_2$ is similar.

We are given that for all B_1, B_2, if $\Gamma \vDash ((x : A_1) \rightarrow\; A_2) = ((x : B_1) \rightarrow\; B_2)$ and $\Gamma \vdash v_0 : A_1 = B_1$ is the corresponding proof term, then $\Gamma \vDash \{v/x\}\, A_2 = \{v_{\triangleright v_0}/x\}\, B_2$. We must show that for all B_1', B_2', if $\Gamma' \vDash ((x : \sigma\, A_1) \rightarrow\; \sigma\, A_2) = ((x : B_1') \rightarrow\; B_2')$ and $\Gamma' \vdash v_0' : \sigma\, A_1 = B_1'$, then $\Gamma \vDash \{\sigma\, v/x\}\, \sigma\, A_2 = \{(\sigma\, v)_{\triangleright v_0'}/x\}\, B_2$.

So consider some B_1', B_2', v_0' satisfying the hypothesis.

Let ρ be such that $\vDash \rho : \Gamma' = \Gamma$ (using lemma 101).

Then by context conversion (lemma 100) we have $\Gamma \vDash ((x : \rho\, \sigma\, A_1) \rightarrow\; \rho\, \sigma\, A_2) = ((x : \rho\, B_1') \rightarrow\; \rho\, B_2')$. By lemma 102 and transitivity this equation is equivalent to $\Gamma \vDash ((x : A_1) \rightarrow\; A_2) = (x : \rho\, B_1') \rightarrow\; \rho\, B_2'$. Suppose the proof term for this equation is $\Gamma \vdash v_{00} : A_1 = \rho\, B_1'$. By assumption we have $\Gamma \vDash \{v/x\}\, A_2 = \{(\rho\, v)_{\triangleright v_{00}}/x\}\, \rho\, B_2'$. Now by context conversion again, $\Gamma' \vDash \sigma\, \{v/x\}\, A_2 = \sigma\, \{v_{\triangleright v_{00}}/x\}\, \rho\, B_2'$.

Since x was a bound variable, we can pick it so it is not in the domain of σ or ρ, so the above equation is equivalent to $\Gamma' \vDash \{\sigma\, v/x\}\, \sigma\, A_2 = \{\sigma\, ((\rho\, v)_{\triangleright v_{00}})/x\}\, \sigma\, \rho\, B_2'$. Since σ and ρ cancel (lemma 102), this equation is equivalent to $\Gamma' \vDash \{\sigma\, v/x\}\, \sigma\, A_2 = \{(\sigma\, v)_{\triangleright \sigma\, v_{00}}/x\}\, B_2'$.

By inversion on $\Gamma \vDash ((x : A_1) \rightarrow\; A_2) = ((x : B_1') \rightarrow\; B_2')$ (lemmas 58 and 49) we know $\Gamma, x : B_1' \vdash B_2' : \mathsf{Type}$, so by substitution (lemma 50) we have $\Gamma \vdash \{(\sigma\, v)_{\triangleright v_0'}/x\}\, B_2 : \mathsf{Type}$. Then since $|\{(\sigma\, v)_{\triangleright \sigma\, v_{00}}/x\}\, B_2'| = |\{(\sigma\, v)_{\triangleright v_0'}/x\}\, B_2'|$, by TCCERASURE and TCCTRANS we have $\Gamma' \vDash \{\sigma\, v/x\}\, \sigma\, A_2 = \{(\sigma\, v)_{\triangleright v_0'}/x\}\, B_2'$ as required. □

A.5 Properties of injrng

***Lemma* 105** (injrng respects CC)**.**
If $\Gamma \vDash \mathsf{injrng}\, A\, \mathsf{for}\, v$ and $\Gamma \vDash A = B$, then $\Gamma \vDash \mathsf{injrng}\, B\, \mathsf{for}\, v$.

Proof. By transitivity, any type which is equal to B is also equal to A. □

***Lemma* 106** (injrng up to erasure of the value)**.** If $\Gamma \vDash \mathsf{injrng}\,(x\!:\!A) \to\ B\, \mathsf{for}\, v$ and $\Gamma \vdash v' : A$ and $|v'| = |v|$, then $\Gamma \vDash \mathsf{injrng}\,(x\!:\!A) \to\ B\, \mathsf{for}\, v'$

Proof. Let $A1$, $B1$ such that $\Gamma \vDash (x : A) \to\ B = (x : A_1) \to\ B_1$ with the proof term $\Gamma \vdash v_0 : ((x\!:\!A) \to\ B) = ((x\!:\!A_1) \to\ B_1)$. We need to show $\Gamma \vDash \{v'/x\}\, B = \{v'_{\triangleright v_0}/x\}\, B_1$.

By the injrng assumption we have $\Gamma \vDash \{v/x\}\, B = \{v_{\triangleright v_0}/x\}\, B_1$. So by regularity (lemma 58) we have $\Gamma \vdash \{v/x\}\, B : \mathsf{Type}$ and $\Gamma \vdash \{v_{\triangleright v_0}/x\}\, B_1 : \mathsf{Type}$.

Also, by inversion on $\Gamma \vDash ((x\!:\!A_1) \to\ A_2) = ((x\!:\!B_1') \to\ B_2')$ (lemmas 58 and 49) we know $\Gamma, x : A_1 \vdash A_2 : \mathsf{Type}$ and $\Gamma, x : B_1' \vdash B_2' : \mathsf{Type}$, so by substitution (lemma 50) we have $\Gamma \vdash \{v'/x\}\, A_2 : \mathsf{Type}$ $\Gamma \vdash \{v'_{\triangleright v_0}/x\}\, B_2' : \mathsf{Type}$. So by TCCErasure $\Gamma \vDash \{v/x\}\, A_2 = \{v'/x\}\, A_2$ and $\{v'_{\triangleright v_0}/x\}\, B_2' = \{v'/x\}\, B_2'$. Conclude by TCCTrans. □

***Lemma* 107** (Instantiating injrng with a different value on the right)**.** If $\Gamma \vDash \mathsf{injrng}\,(x\!:\!A) \to\ B\, \mathsf{for}\, v$ and $\Gamma \vDash (x\!:\!A) \to\ B = (x\!:\!A') \to\ B'$ and $\Gamma \vdash v' : A'$ and $|v'| = |v|$, then $\Gamma \vDash \{v/x\}\, B = \{v'/x\}\, B'$.

Proof. By the assumption $\Gamma \vDash \mathsf{injrng}\,(x\!:\!A) \to\ B\, \mathsf{for}\, v$ we know that $\Gamma \vDash \{v/x\}\, B = \{v_{\triangleright v_0}/x\}\, B'$.

By inversion on $\Gamma \vDash ((x\!:\!A) \to\ B) = ((x\!:\!A') \to\ B')$ (lemmas 58 and 49) we know $\Gamma, x : A' \vdash B' : \mathsf{Type}$, so by substitution (lemma 50) we have $\Gamma \vdash \{v'/x\}\, B' : \mathsf{Type}$. We also know $|\{v_{\triangleright v_0}/x\}\, B'| = |\{v'/x\}\, B'|$, so by TCCErasure we have $\Gamma \vDash \{v_{\triangleright v_0}/x\}\, B' = \{v'/x\}\, B'$.

Then by TCCTrans, $\Gamma \vDash \{v/x\}\, B = \{v'/x\}\, B'$ as required. □

A.6 Proofs about elaboration

In general, in the following we will use primed metavariables for fully-elaborated core language environments and terms.

This lemma states that the elaboration algorithm produces output that type checks according to the core language and differs from the input only in the erasable parts of the term.

***Lemma* 108** (Soundness w.r.t. fully annotated typing)**.**

1. If $\vdash \Gamma'$ and $\Gamma' \mapsto a \Rightarrow a' : A'$, then $\Gamma' \vdash a' : A'$ and $|a| = |a'|$.
2. If $\vdash \Gamma'$ and $\Gamma' \vdash A' : \mathsf{Type}$ and $\Gamma' \mapsto a \Leftarrow A' \leadsto a'$, then $\Gamma' \vdash a' : A'$ and $|a| = |a'|$.
3. If $\mapsto \Gamma \leadsto \Gamma'$, then $\vdash \Gamma'$ and $|\Gamma| = |\Gamma'|$

Proof. Induction on the assumed typing derivations. The cases for $\Gamma \mapsto b \Rightarrow b' : B$ are:

EItype Trivial.

EIvar Trivial.

EIpi By ih. $\Gamma' \vdash A' : \mathsf{Type}$ and $|A| = |A'|$. $\Gamma', x : A' \vdash B' : \mathsf{Type}$ and $|B| = |B'|$. Thus $\Gamma' \vdash (x\!:\!A') \to B' : \mathsf{Type}$ and $|(x\!:\!A) \to B| = |(x\!:\!A') \to B'|$.

EIipi Similar to EIIPI.

EIdapp

$$\frac{\begin{array}{l}\Gamma \mapsto a \Rightarrow a' : A_1 \\ \Gamma \mapsto A_1 =^{?} (x : A) \to B \leadsto v_1 \\ \Gamma \mapsto v \Leftarrow A \leadsto v' \\ \Gamma \vDash \mathsf{injrng}\,(x\!:\!A) \to B\ \mathsf{for}\ v'\end{array}}{\Gamma \mapsto a\ v \Rightarrow a'_{\triangleright v_1}\ v' : \{v'/x\}\,B}\ \textsc{EIdapp}$$

By ih we have $\Gamma' \vdash a' : A_1$ where $|a| = |a'|$. By assumption 53 we have $\Gamma' \vdash v_1 : A_1 = ((x\!:\!A) \to B)$. By several inversions (lemma 49) of this judgement, we can conclude $\Gamma' \vdash (x\!:\!A) \to B : \mathsf{Type}$ and $\Gamma' \vdash A : \mathsf{Type}$. Therefore by casting, $\Gamma' \vdash a'_{\triangleright v_1} : (x\!:\!A) \to B$. Also by induction we have $\Gamma' \vdash v' : A$ and $|v| = |v'|$. Therefore $\Gamma' \vdash a'_{\triangleright v_1}\ v' : \{v'/x\}\,B$ and $|a'_{\triangleright v_1}\ v'| = |a\ v|$.

EIapp and EIdiapp Similar to EIDAPP.

EIeq Directly by induction.

EIjjoinc By induction $|a = b| = |a' = b'|$ and $\Gamma' \vdash a' = b' : \mathsf{Type}$. Therefore, we know that the terms have the same erasure (i.e. $|a| = |a'|$ and $|b| = |b'|$) so the same premises can used in rule TJOINC.

EIjoinp Similar to EIJOINC.

EIannot By induction.

The cases for $\Gamma \mapsto a \Leftarrow A \leadsto a'$ are:

ECrec By assumption 53 we have $\Gamma' \vdash v_1 : A = ((x\!:\!A_1) \to A_2)$. By inversions of this judgement (lemma 49), $\Gamma' \vdash (x\!:\!A_1) \to A_2 : \mathsf{Type}$ and $\Gamma', x : A_1 \vdash A_2 : \mathsf{Type}$. By core language weakening $\Gamma', f : (x : A_1) \to A_2, x : A_1 \vdash A_2 : \mathsf{Type}$, so the induction hypothesis applies. Therefore $\Gamma', f : (x\!:\!A_1) \to A_2, x : A_1 \vdash a' : A_2$

and $|a| = |a'|$. By TREC, we have $\Gamma' \vdash \mathsf{rec}\, f_{(xA_1)\to A}\ x.a' : (x{:}A_1) \to A$, and by TCAST, we have $\Gamma' \vdash (\mathsf{rec}\, f_{(xA_1)\to A_2}\ x.a')_{\triangleright \mathsf{symm}\, v_1} : A$. Furthermore the erasures are equal.

ECirec Similar to ECREC.

ECrefl By assumption (analogous to 53) we have $\Gamma' \vdash v_1 : A = (a = b)$. By inversion, $\Gamma' \vdash a = b : \mathsf{Type}$. By assumption 54, we also have $\Gamma' \vdash v : a = b$, and that $|v| = \mathsf{join}$. Therefore by TCAST we conclude that $\Gamma' \vdash v_{\triangleright \mathsf{symm}\, v_1} : A$ and that $|v_{\triangleright \mathsf{symm}\, v_1}| = |\mathsf{join}_\bullet|$.

ECinf We know that $\Gamma' \vdash B : \mathsf{Type}$. By induction we have that $\Gamma' \vdash a' : A$ where $|a| = |a'|$. That means $|a'_{\triangleright v_1}| = |a|$ By assumption 54, we have $\Gamma' \vdash v_1 : A = B$, therefore we can use TCAST to conclude $\Gamma' \vdash a'_{\triangleright v_1} : B$.

The cases for $\vdash \Gamma \leadsto \Gamma'$ are:

EGnil Trivial.

EGvar By the IH we know $\vdash \Gamma'$. So by the mutual IH for $\Gamma' \vdash A \Leftarrow \mathsf{Type} \leadsto A'$ we know $\Gamma' \vdash A' : \mathsf{Type}$, and therefore $\vdash \Gamma', x : A'$. Similarly, $|\Gamma, x : A| = |\Gamma', x : A'|$.

□

A.6.1 Checking is closed under CC

This next lemma says that the input type of the elaboration judgement can be replaced with an equivalent type (according to congruence closure) and elaboration will still succeed, producing a result that differs only in typing annotations.

Lemma* 109** (Admissibility of CCAST in elaboration).***
If $\Gamma' \vdash a \Leftarrow A' \leadsto a'$ and $\Gamma' \models A' = B'$, then $\Gamma' \vdash a \Leftarrow B' \leadsto a''$ for some a'' such that $|a''| = |a'|$.

Proof. Case analysis on $\Gamma' \vdash a \Leftarrow A' \leadsto a'$. Cases ECREC, ECIREC, ECREFL, ECSUBST, ECDCON, and ECCASE are all very similar, so we show just ECREC in detail.

Here, the assumed typing derivation looks like

$$\frac{\begin{array}{l}\Gamma \vdash A =^? (x : A_1) \to A_2 \leadsto v_1 \\ \Gamma, f : (x{:}A_1) \to A_2, x : A_1 \vdash a \Leftarrow A_2 \leadsto a' \\ \Gamma, f : (x{:}A_1) \to A_2, x : A_1 \models \mathsf{injrng}\, (x{:}A_1) \to A_2\, \mathsf{for}\, x \\ \Gamma, f : (x{:}A_1) \to A_2 \vdash (x{:}A_1) \to A_2 \Leftarrow \mathsf{Type} \leadsto A_0\end{array}}{\Gamma \vdash \mathsf{rec}\, f\ x.a \Leftarrow A \leadsto (\mathsf{rec}\, f_{(xA_1)\to A_2}\ x.a')_{\triangleright \mathsf{symm}\, v_1}}\textsc{ECrec}$$

By assumption 52 we have $\Gamma \mathrel{\vDash\!\!\!\!\!\vdash} B =^{?} (x\!:\!A_1) \to A_2$. Then apply ECrec again. The elaborated term only differs in the proof used by the cast, $\mathsf{symm}\; v_1$, and this difference gets erased.

The rule ECinf instead relies on transitivity of $\vDash$. We have $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} A \overset{?}{=} B \rightsquigarrow v_1$ as a premise of the rule and $\Gamma' \vDash A' = A$ as an assumption, so $\Gamma' \vDash A' = B$, and hence $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} A' \overset{?}{=} B \rightsquigarrow v_2$ for some v_2. Then apply ECinf again; again the elaborated term only differs by the proof of the cast. □

A.6.2 Context conversion for elaboration

***Lemma* 110** (Context conversion for elaboration)**.** Suppose $\vDash \sigma : \Gamma = \Gamma'$. Then,

1. $\Gamma \mathrel{\vDash\!\!\!\!\!\vdash} a \Rightarrow a' : A$ implies $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} a \Rightarrow a'' : A'$ for some A' such that $\Gamma' \vDash A' = \sigma\, A$ and some a'' such that $|a''| = |a'|$.
2. $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma \mathrel{\vDash\!\!\!\!\!\vdash} a \Leftarrow A \rightsquigarrow a'$ implies $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} a \Leftarrow \sigma\, A \rightsquigarrow a''$ for some a'' such that $|a''| = |a'|$.

Proof. Induction on the assumed derivations. The cases for $\Gamma \mathrel{\vDash\!\!\!\!\!\vdash} b \Rightarrow b' : B$ are:

EItype Pick $A' := \mathsf{Type}$.

EIvar By the variable lookup lemma (lemma 96) we have $x : A' \in \Gamma$ with $\Gamma' \vDash A' = \sigma\, A$, as required. The elaborated term is still x, so it is equal up to erasure as required.

EIpi By the mutual IHs we have $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} A \Leftarrow \mathsf{Type} \rightsquigarrow A''$ and $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} B \Leftarrow \mathsf{Type} \rightsquigarrow B''$. Then re-apply EIpi. By IH the subterms of the elaborated term are equal up to erasure, so the entire elaborated term is also equal up to erasure.

EIipi Similar to EIipi.

EIapp By the IH for the first premise we know $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} a \Rightarrow a'' : A_1'$ for some type A_1' such that $\Gamma' \vDash A_1' = \sigma\, A_1$.

From the premise $\Gamma \mathrel{\vDash\!\!\!\!\!\vdash} A_1 =^{?} (x : A) \to B \rightsquigarrow v_1$ and context conversion (lemma 100) we get $\Gamma' \vDash \sigma\, A_1 = (x\!:\!\sigma\, A) \to \sigma\, B$. So by transitivity, $\Gamma' \vDash A_1' = (x\!:\!\sigma\, A) \to \sigma\, B$. So the search $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} A_1' =^{?} (x : A') \to B' \rightsquigarrow v_1'$ will succeed for some arrow type $(x\!:\!A') \to B'$ and proof v_1', since these exists at least one such arrow type.

Now note that by TCCtrans and TCCinjdom, we have $\Gamma' \vDash A' = \sigma\, A$. From the IH for b we know $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} b \Leftarrow \sigma\, A \rightsquigarrow b''$. So by casting the return type (lemma 109) we get $\Gamma' \mathrel{\vDash\!\!\!\!\!\vdash} b \Leftarrow A' \rightsquigarrow b'''$.

Now apply EIApp to get $\Gamma' \vdash a\ b \Rightarrow a''\ b''' : B'$. By TCCInjRng we have $\Gamma' \vDash B' = \sigma\, B$ as required.

EIdapp By the IH for the first premise, we know $\Gamma' \vdash a \Rightarrow a'' : A_1'$ for some type A_1' such that $\Gamma' \vDash A_1' = \sigma\, A_1$.

From the premise $\Gamma \vdash A_1 =^? (x : A) \to B \leadsto v_1$ and context conversion (lemma 100) we get $\Gamma' \vDash \sigma\, A_1 = (x\!:\!\sigma\, A) \to\ \sigma\, B$. So by transitivity, $\Gamma' \vDash A_1' = (x\!:\!\sigma\, A) \to\ \sigma\, B$. So the search $\Gamma' \vdash A_1' =^? (x : A') \to B' \leadsto v_1'$ will succeed for some arrow type $(x\!:\!A') \to\ B'$ and proof v_1', since these exists at least one such arrow type.

Now note that by TCCTrans and TCCInjDom, we have $\Gamma' \vDash A' = \sigma\, A$. From the IH for v we know $\Gamma' \vdash v \Leftarrow \sigma\, A \leadsto v''$. So by casting the return type (lemma 109) we get $\Gamma' \vdash v \Leftarrow A' \leadsto v'''$.

By context conversion for injrng (lemma 104) we get $\Gamma' \vDash \mathsf{injrng}\,(x : \sigma\, A) \to \sigma\, B\, \mathsf{for}\, \sigma\, v'$. Now by correctness of elaboration (lemma 108) we know $\Gamma' \vdash v''' : A'$ and also $|v| = |v'''|$. The latter also implies $|v'''| = |\sigma\, v'|$, so since injrng respects type equality and erasure (lemmas 105, 106) we then have $\Gamma' \vDash \mathsf{injrng}\,(x\!:\!A') \to\ B'\,\mathsf{for}\, v'''$

Then apply EIDApp again, to get $\Gamma' \vdash a\ v \Rightarrow a'_{\triangleright v_1'}\ v''' : \{v'''/x\}\, B'$.

From $\Gamma' \vDash\ \mathsf{injrng}\,(x : A') \to\ B'\,\mathsf{for}\, v'''$ we get $\Gamma' \vDash \{v'''/x\}\, B' = \{\sigma\, v'/x\}\, \sigma\, B$. Since we can pick the bound variable so that $x\ \notin \mathsf{FV}\,(B)$, that is the same as $\Gamma' \vDash \{v'''/x\}\, B' = \sigma\, \{v'/x\}\, B$, as required. Also as required, $|a'\ v'| = |a''\ v'''|$ since the subterms are equal up to erasure.

ottdrulenameEIdiapp Similar to EIDApp.

EIeq By the IHs we get $\Gamma' \vdash a \Rightarrow a'' : A_0$ and $\Gamma' \vdash b \Rightarrow b'' : B_0$, then apply EIEq again.

EIjjoinc By the mutual IH we get $\Gamma' \vdash a = b \Leftarrow \mathsf{Type} \leadsto a'' = b''$. Since a' and a'' erase to the same thing we know $|a| \leadsto^{i}_{\mathsf{cbv}}\ c$ (and similarly for b''), so applying EIJoinc again we get $\Gamma' \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a = b} \Rightarrow \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a'' = b''} : a'' = b''$.

By soundness (lemma 108) and regularity (lemma 51) we know $a'' = b''$ is well-typed, so by TCCErasure we have $\Gamma' \vDash (a' = b') = (a'' = b'')$ as required.

EIjoinp Similar to EIJoinc.

EIannot By the mutual IH we get $\Gamma' \vdash A \Leftarrow \mathsf{Type} \leadsto A''$ and $\Gamma' \vDash A'' = \sigma\, A'$. Again by mutual IH we have $\Gamma' \vdash a \Leftarrow \sigma\, A' \leadsto a''$. So by casting (lemma 109) we have $\Gamma \vdash a \Leftarrow A'' \leadsto a'''$.

Then apply EIAnnot again, to get $\Gamma \vdash a_A \Rightarrow a''' : A''$. We have $|a'''| = |a''| = |a'|$ as required.

The cases for $\Gamma \vdash a \Leftarrow A \leadsto a'$ are:

ECrec By context conversion for CC (lemma 100) we know $\Gamma' \vDash \sigma\, A = (x\!:\!\sigma\, A_1) \to \sigma\, A_2$. So the search $\Gamma' \vdash A =^? (x : A_1') \to A_2' \leadsto v_1'$ will succeed for some arrow type $(x\!:\!A_1') \to A_2'$ and proof v_1', since there exists at least one such arrow type.

By regularity of CC (lemma 58) and inversion for type well-formedness we know $\Gamma, x : A_1 \vdash A_2 : \mathsf{Type}$, and so by weakening (lemma 47) $\Gamma, f : (x\!:\!A_1) \to A_2, x : A_1 \vdash A_2 : \mathsf{Type}$. So the induction hypothesis for the a premise is available.

By TCCTrans and TCCInjDom, we have $\Gamma' \vDash (x\!:\!A_1') \to A_2' = (x\!:\!\sigma\, A_1) \to \sigma\, A_2$ and $\Gamma' \vDash A_1' = \sigma\, A_1$. So $\vDash \sigma' : \Gamma, f : (x : A_1) \to A_2, x : A_1 = \Gamma', f : (x : A_1') \to A_2', x : A_1'$, where σ' is the substitution σ suitably extended. So by IH, $\Gamma', f : (x\!:\!A_1') \to A_2, x : A_1' \vdash a \Leftarrow \sigma'\, A_2 \leadsto a''$.

Because injrng respects context conversion (Lemma 104) we have $\Gamma', f : (x : \sigma\, A_1) \to \sigma\, A_2, x : \sigma\, A_1 \vDash \mathsf{injrng}\, (x\!:\!\sigma\, A_1) \to \sigma\, A_2\, \mathsf{for}\, \sigma\, x$. Since it respects CC (lemma 105) that implies $\Gamma', f : (x\!:\!\sigma\, A_1) \to \sigma\, A_2, x : \sigma\, A_1 \vDash \mathsf{injrng}\, (x\!:\!A_1') \to A_2'\, \mathsf{for}\, \sigma\, x$. Also, using the CC judgements we proved above, we can construct a ρ such that $\vDash \rho : \Gamma', f : (x\!:\!\sigma\, A_1) \to \sigma\, A_2, x : \sigma\, A_1 = \Gamma', f : (x\!:\!A_1') \to A_2', x : A_1'$. So by lemma 104 again, we have $\Gamma', f : (x\!:\!A_1') \to A_2', x : A_1' \vDash \mathsf{injrng}\, (x\!:\!\rho\, A_1') \to \rho\, A_2'\, \mathsf{for}\, \rho\, \sigma\, x$. The variables f and x were bound, so we can pick them to not appear in the arrow type, so this is the same as $\Gamma', f : (x\!:\!A_1') \to A_2', x : A_1' \vDash \mathsf{injrng}\, (x\!:\!A_1') \to A_2'\, \mathsf{for}\, \rho\, \sigma\, x$. Finally, since injrng respects erasure (lemma 106) we can conclude that $\Gamma', f : (x\!:\!A_1') \to A_2', x : A_1' \vDash \mathsf{injrng}\, (x\!:\!A_1') \to A_2'\, \mathsf{for}\, x$.

By weakening of CC (lemma 57) we have $\Gamma', f : (x : A_1') \to A_2', x : A_1' \vDash (x : A_1') \to A_2' = (x : \sigma\, A_1) \to \sigma\, A_2$. So by the injrng assumption we know that $\Gamma', f : (x\!:\!A_1') \to A_2', x : A_1' \vDash A_2' = \sigma\, A_2$.

So by casting (lemma 109) we have $\Gamma', f : (x\!:\!A_1') \to A_2', x : A_1' \vdash a \Leftarrow A_2' \leadsto a'''$.

Now apply ECRec to get $\Gamma' \vdash \mathsf{rec}\ f\ \ x.a \Leftarrow \sigma\, A \leadsto (\mathsf{rec}\, f\ x.a''')_{\triangleright \mathsf{symm}\, v_1'}$ as required.

ECirec Similar to ECRec.

ECrefl By context conversion for CC (lemma 100) we know $\Gamma' \vDash \sigma\, A = \sigma\, (a = b)$. Therefore, $\Gamma' \vdash \sigma\, A =^? (a_1 = b_1) \leadsto v_1'$ will succeed for some $a_1 = b_1$ such that $\Gamma' \vDash \sigma\, (a = b) = (a_1 = b_1)$. By TCCInjEq, that implies $\Gamma' \vDash \sigma\, a = a_1$ and $\Gamma' \vDash \sigma\, b = b_1$.

We know $\Gamma' \vDash (\sigma\, a) = (\sigma\, b)$ by context conversion for CC.

So by transitivity (TCCTrans) we have $\Gamma' \vDash a_1 = b_1$.So $\Gamma' \vdash a_1 \overset{?}{=} b_1 \leadsto v'$ will also succeed.

Then apply ECRefl again. By assumption 52 we know $|v_{\triangleright \mathsf{symm}\, v_1}| = |v'_{\triangleright \mathsf{symm}\, v_1'}| =$

join, so the elaborated terms are equal up to erasure as required.

ECinf By the mutual IH we have $\Gamma' \vdash a \Rightarrow a'' : A'$ with $\Gamma' \vDash \sigma\, A = A'$. And by context conversion for CC (lemma 100) we have $\Gamma' \vDash \sigma\, A = \sigma\, B$. By transitivity, $\Gamma' \vDash A' = \sigma\, B$, so $\Gamma' \vdash A' \overset{?}{=} \sigma\, B \rightsquigarrow v_1'$ succeeds for some v_1'. Then apply ECINF again.

□

A.6.3 Completeness of elaboration

Note: in the following lemma statement and proof we use the convention that metavariables with primes $(A', B' \dots)$ are expressions in the fully annotated language, and metavariables without primes are in the surface language.

The first completeness lemma says that if the surface language CC judgement is derivable, then the target CC judgement is also derivable after elaborating the context and terms.

***Lemma* 111** (Completeness of CC)**.** If $\Gamma \vDash^{\exists} a = b$ and $\vdash \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \vdash a \Rightarrow a' : A'$ and $\Gamma' \vdash b \Rightarrow b' : B'$ then $\Gamma' \vDash a' = b'$

Proof. The proof follows from the fact that typing annotations don't matter to congruence closure (Lemma 78). By inversion of $\Gamma \vDash^{\exists} a = b$ we have some Γ_1', a_1' and b_1' such that $\Gamma_1' \vDash a' = b'$ and $|\Gamma_1'| = |\Gamma|$, $|a_1'| = |a|$, and $|b_1'| = |b|$. By translation soundness (Lemma 108), we also have $|\Gamma'| = |\Gamma|$, $|a'| = |a|$, and $|b'| = |b|$, with $\Gamma' \vdash a' : A'$ and $\Gamma' \vdash b' : B'$. This is all that we need to use the lemma. □

Likewise, we need to know that the surface language injrng judgement also describes when the corresponding fully annotated version is derivable.

***Lemma* 112** (Completeness of injrng)**.**
If $\Gamma \vDash^{\exists} \mathsf{injrng}\, (x : A) \to B\, \mathsf{for}\, v$ and $\vdash \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \vdash (x : A) \to B \Leftarrow \mathsf{Type} \rightsquigarrow (x : A') \to B'$ and $\Gamma' \vdash v \Leftarrow A' \rightsquigarrow v'$ then $\Gamma' \vDash \mathsf{injrng}\, (x : A') \to B'\, \mathsf{for}\, v'$.

Proof. Consider A_1, B_1 such that $\Gamma' \vDash (x : A') \to B' = (x : A_1) \to B_2$ with the proof term $\Gamma' \vdash v_0 : ((x : A') \to B') = ((x : A_1) \to B_2)$. We must show $\Gamma' \vDash \{v'/x\}\, B' = \{v'_{\triangleright v_0}/x\}\, B_1$.

By inversion and substitution, we know that $\Gamma' \vdash \{v'/x\}\, B' : \mathsf{Type}$ and $\Gamma' \vdash \{v'_{\triangleright v_0}/x\}\, B_1 : \mathsf{Type}$.

Now instantiation the assumption $\Gamma \vDash^{\exists} \mathsf{injrng}\, (x : A) \to B\, \mathsf{for}\, v$ with A_1 and B_1. We have $\Gamma \vDash^{\exists} \{v_A/x\}\, B = \{v_{A_1}/x\}\, B_1$. That is, there are some Γ'', a'' and b'' such that $|\Gamma''| = |\Gamma|$ and $|a''| = |\{v_A/x\}\, B|$ and $|b''| = |\{v_{A_1}/x\}\, B_1|$ and $\Gamma'' \vDash a'' = b''$.

Since elaboration produced terms which are equal up to erasure, we also have $|\Gamma''| = |\Gamma'|$ and $|a''| = |\{v/x\}\, B'|$ and $|b''| = |\{v_{\triangleright v_0}/x\}\, B_1|$. So since CC doesn't care about annotations (lemma 78) we have $\Gamma' \vDash \{v'/x\}\, B' = \{v'_{\triangleright v_0}/x\}\, B_1$ as required. □

We next prove the completeness of the entire system using mutual induction on the three judgements of the surface language. For convenience, we use an alternative ("regularized") version of the typing rules, written $\Gamma \vdash_{\text{reg}} a \Rightarrow A$, that adds additional regularity assumptions to the typing judgement. For example, in the RIdapp rule we add the premise $\Gamma \vdash (x : A) \to B \Leftarrow \mathsf{Type}$. The typing rules for that system are shown in Figures A.6 and A.7.

To justify the addition of these premises, we show the following regularity lemma about the inference judgement.

***Lemma* 113.** If $\Gamma \vdash a \Rightarrow A$ then $\Gamma \vdash A \Leftarrow \mathsf{Type}$.

Proof. Proof is by case analysis of $\Gamma \vdash a \Rightarrow A$.

Itype Holds by Itype and Cinf.

Ivar Holds by premise of the rule.

Ipi Holds by Itype and Cinf.

Idapp Holds by premise of the rule.

Iidapp Holds by premise of the rule.

Iapp Holds by premise of the rule.

Ieq Holds by Itype and Cinf.

Ijoinc Holds by premise of the rule.

Ijoinp Holds by premise of the rule.

Iannot Holds by premise of the rule.

Icast Holds by premise of the rule.

□

***Lemma* 114** (Completeness, with strengthened invariants)**.**

1. If $\vdash_{\text{reg}} \Gamma \Leftarrow$ then $\mapsto \Gamma \rightsquigarrow \Gamma'$.
2. If $\Gamma \vdash_{\text{reg}} a \Rightarrow A$ and $\vdash_{\text{reg}} \Gamma \Leftarrow$ and $\mapsto \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \rightsquigarrow A'$, then $\Gamma' \mapsto a \Rightarrow a' : A''$ and $\Gamma' \vDash A' = A''$
3. If $\Gamma \vdash_{\text{reg}} a \Leftarrow A$ and $\vdash_{\text{reg}} \Gamma \Leftarrow$ and $\mapsto \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \rightsquigarrow A'$, then $\Gamma' \mapsto a \Leftarrow A' \rightsquigarrow a'$.

$$\boxed{\Gamma \vdash_{\mathrm{reg}} a \Rightarrow A}$$

$$\frac{}{\Gamma \vdash_{\mathrm{reg}} \mathsf{Type} \Rightarrow \mathsf{Type}}\textsc{RItype} \qquad \frac{\vdash_{\mathrm{reg}} \Gamma \Leftarrow \quad x : A \in \Gamma \quad \Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type}}{\Gamma \vdash_{\mathrm{reg}} x \Rightarrow A}\textsc{RIvar}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type}\\ \Gamma, x : A \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} (x\!:\!A) \to\ B \Rightarrow \mathsf{Type}}\textsc{RIpi} \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} (x\!:\!A) \to\ B \Leftarrow \mathsf{Type}\\ \Gamma \vdash_{\mathrm{reg}} a \Rightarrow (x\!:\!A) \to\ B\\ \Gamma \vdash_{\mathrm{reg}} v \Leftarrow A\\ \Gamma \vDash^{\exists}\ \mathsf{injrng}\,(x\!:\!A) \to\ B\,\mathsf{for}\ v\\ \Gamma \vdash_{\mathrm{reg}} \{v_A/x\}\, B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} a\ v \Rightarrow \{v_A/x\}\, B}\textsc{RIdapp}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\\ \Gamma \vdash_{\mathrm{reg}} A \to B \Leftarrow \mathsf{Type}\\ \Gamma \vdash_{\mathrm{reg}} a \Rightarrow A \to B\\ \Gamma \vdash_{\mathrm{reg}} b \Leftarrow A\end{array}}{\Gamma \vdash_{\mathrm{reg}} a\ b \Rightarrow B}\textsc{RIapp} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type}\\ \Gamma, x : A \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} \bullet(x\!:\!A) \to\ B \Rightarrow \mathsf{Type}}\textsc{RIipi}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} \bullet(x\!:\!A) \to\ B \Leftarrow \mathsf{Type}\\ \Gamma \vdash_{\mathrm{reg}} a \Rightarrow \bullet(x\!:\!A) \to\ B\\ \Gamma \vdash_{\mathrm{reg}} v \Leftarrow A\\ \Gamma \vDash^{\exists}\ \mathsf{injrng}\ \bullet\ (x\!:\!A) \to\ B\,\mathsf{for}\ v\\ \Gamma \vdash_{\mathrm{reg}} \{v_A/x\}\, B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} a\ \bullet_v \Rightarrow \{v_A/x\}\, B}\textsc{RIidapp}$$

$$\boxed{\Gamma \vdash_{\mathrm{reg}} a \Leftarrow A}$$

$$\frac{\begin{array}{l}\Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash_{\mathrm{reg}} a \Leftarrow A_2\\ \Gamma, f : (x\!:\!A_1) \to\ A_2 \vdash_{\mathrm{reg}} A_1 \Leftarrow \mathsf{Type}\\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash_{\mathrm{reg}} A_2 \Leftarrow \mathsf{Type}\\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vDash^{\exists}\ \mathsf{injrng}\,(x\!:\!A_1) \to\ A_2\,\mathsf{for}\ x\end{array}}{\Gamma \vdash_{\mathrm{reg}} \mathsf{rec}\, f\ x.a \Leftarrow (x\!:\!A_1) \to\ A_2}\textsc{RCrec}$$

$$\frac{\begin{array}{l}\Gamma, f : \bullet(x\!:\!A_1) \to\ A_2, x : A_1 \vdash_{\mathrm{reg}} a \Leftarrow A_2\\ \Gamma, f : (x\!:\!A_1) \to\ A_2 \vdash_{\mathrm{reg}} A_1 \Leftarrow \mathsf{Type}\\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash_{\mathrm{reg}} A_2 \Leftarrow \mathsf{Type}\\ x\ \notin \mathsf{FV}\,(|a|)\\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vDash^{\exists}\ \mathsf{injrng}\,(x\!:\!A_1) \to\ A_2\,\mathsf{for}\ x\end{array}}{\Gamma \vdash_{\mathrm{reg}} \mathsf{rec}\, f\ \bullet.a \Leftarrow \bullet(x\!:\!A_1) \to\ A_2}\textsc{RCirec}$$

Figure A.6: Typing rules for surface language, with added extra regularity premises: functions and variables

$$\boxed{\Gamma \vdash_{\mathrm{reg}} a \Rightarrow A} \qquad \boxed{\Gamma \vdash_{\mathrm{reg}} a \Leftarrow A}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} a \Rightarrow A \\ \Gamma \vdash_{\mathrm{reg}} b \Rightarrow B\end{array}}{\Gamma \vdash_{\mathrm{reg}} a = b \Rightarrow \mathsf{Type}}\textsc{RIeq}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} a \Leftarrow A\end{array}}{\Gamma \vdash_{\mathrm{reg}} a_A \Rightarrow A}\textsc{RIannot} \qquad \frac{\Gamma \vdash_{\mathrm{reg}} a \Rightarrow A}{\Gamma \vdash_{\mathrm{reg}} a \Leftarrow A}\textsc{RCinf}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{reg}} a_1 = a_2 \Leftarrow \mathsf{Type} \\ |a_1| \leadsto^{i}_{\mathsf{cbv}} b \quad |a_2| \leadsto^{j}_{\mathsf{cbv}} b\end{array}}{\Gamma \vdash_{\mathrm{reg}} \mathsf{join}_{\leadsto_{\mathsf{cbv}} i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\textsc{RIjoinc} \qquad \frac{\Gamma \vDash^{\exists} a = b}{\Gamma \vdash_{\mathrm{reg}} \mathsf{join} \Leftarrow a = b}\textsc{RCrefl}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{reg}} a_1 = a_2 \Leftarrow \mathsf{Type} \\ |a_1| \leadsto^{i}_{\mathsf{p}} b \quad |a_2| \leadsto^{j}_{\mathsf{p}} b\end{array}}{\Gamma \vdash_{\mathrm{reg}} \mathsf{join}_{\leadsto_{\mathsf{p}} i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\textsc{RIjoinp}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} a \Rightarrow A \quad \Gamma \vDash^{\exists} A = B \\ \Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} a \Rightarrow B}\textsc{RIcast} \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} a \Leftarrow A \\ \Gamma \vDash A = B \\ \Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} a \Leftarrow B}\textsc{RCcast}$$

Figure A.7: Typing rules for surface language, with added extra regularity premises: equality

Proof. Mutual induction on the derivations. The cases for $\Gamma \vdash_{\text{reg}} a \Rightarrow A$ are:

Itype Pick $A' := \mathsf{Type}$.

Ivar By soundness of elaboration (lemma 108) applied to the assumption $\mapsto \Gamma \leadsto \Gamma'$, there is some $x : A'' \in \Gamma'$ with $|A''| = |A|$ and $\Gamma' \vdash A'' : \mathsf{Type}$. By soundness of elaboration applied to the assumption $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \leadsto A'$, we know $\Gamma' \vdash A' : \mathsf{Type}$.

Now by EIvar we have $\Gamma' \mapsto x \Rightarrow x : A''$, and by TCCerasure $\Gamma' \vDash A' = A''$ as required.

Ipi We know $\Gamma' \mapsto \mathsf{Type} \Leftarrow \mathsf{Type} \leadsto \mathsf{Type}$. So by the mutual IH for the A premise, $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \leadsto A'$.

Then by Gvar we have $\vdash \Gamma, x : A \Leftarrow$, and by GFvar we have $\mapsto \Gamma, x : A \leadsto \Gamma', x : A'$. So by the mutual IH for the B premise, $\Gamma', x : A' \mapsto B \Leftarrow \mathsf{Type} \leadsto B'$.

Now apply EIpi to get $\Gamma' \mapsto (x\!:\!A) \to B \Rightarrow (x\!:\!A') \to B' : \mathsf{Type}$.

Iipi Similar to Ipi.

Idapp The given typing derivation looks like

$$\dfrac{\begin{array}{l}\Gamma \vdash_{\text{reg}} (x\!:\!A) \to B \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\text{reg}} a \Rightarrow (x\!:\!A) \to B \\ \Gamma \vdash_{\text{reg}} v \Leftarrow A \\ \Gamma \vDash^{\exists} \mathsf{injrng}\,(x\!:\!A) \to B\,\mathsf{for}\,v \\ \Gamma \vdash_{\text{reg}} \{v_A/x\}\,B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\text{reg}} a\ v \Rightarrow \{v_A/x\}\,B}\ \textsc{RIdapp}$$

In the regularized type system, we have $\Gamma \vdash (x\!:\!A) \to B \Leftarrow \mathsf{Type}$ as a premise to the given rule. So by IH, $\Gamma' \mapsto (x\!:\!A) \to B \Leftarrow \mathsf{Type} \leadsto B'_1$ for some type B'_1, where $\Gamma' \vDash (x : A) \to B = B'_1$. In fact there is only one rule for elaborating arrow types, so by inversion of that judgement, we get $\Gamma' \mapsto (x : A) \to B \Leftarrow \mathsf{Type} \leadsto (x\!:\!A') \to B'$, where B'_1 is $(x\!:\!A') \to B'$ and $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \leadsto A'$ and $\Gamma', x : A' \mapsto B \Leftarrow \mathsf{Type} \leadsto B'$. By soundness, this also means that $|(x\!:\!A) \to B| = |(x\!:\!A') \to B'|$.

From the IH for the a premise we know $\Gamma' \mapsto a \Rightarrow a' : A'_0$ with $\Gamma' \vDash A'_0 = (x : A') \to B'$.

So, by Assumption 55 the search $\Gamma' \mapsto A'_0 =^{?} (x : A'') \to B'' \leadsto v_1$ through the equivalence class of A'_0 will terminate successfully with some arrow type $(x\!:\!A'') \to B''$ and proof v_1, since there exists at least one such arrow type, and by Assumption 53 we know that $\Gamma' \vdash v_1 : (A'_0 = ((x\!:\!A'') \to B''))$.

As a result, we have $\Gamma' \vDash (x : A') \to B' = (x : A'') \to B''$, By TCCInjdom we know $\Gamma' \vDash A' = A''$.

Now by the IH for the v premise, we get $\Gamma' \mapsto v \Leftarrow A' \leadsto v'$ and, by lemma 108, that $|v| = |v'|$. By casting (lemma 109) this implies $\Gamma' \mapsto v \Leftarrow A'' \leadsto v''$. Again by soundness (lemma 108), we have $\Gamma' \vdash v'' : A''$ and $|v| = |v''|$.

The algorithmic injrng premise of EIdapp, namely $\Gamma' \vDash \mathsf{injrng}\,(x : A'') \to B''\,\mathsf{for}\,v''$ is satisfied by Lemma 112.

Now apply EIdapp, to get $\Gamma' \mapsto a\ v \Rightarrow a'\ v'' : \{v''/x\}\,B''$.

We know by assumption that $\Gamma' \mapsto \{v_A/x\}\,B \Leftarrow \mathsf{Type} \leadsto B_0$. The lemma also requires showing $\Gamma' \vDash B_0 = \{v''/x\}\,B''$. By instantiating the injrng premise at v' (lemma 107), it suffices to show that $\Gamma' \vDash B_0 = \{v'/x\}\,B'$. We derive this equality via TCCErasure, as $\Gamma' \vdash B_0 : \mathsf{Type}$ (via soundness), $\Gamma' \vdash \{v'/x\}\,B' : \mathsf{Type}$ (via substation for annotated language), and $|B_0| = |\{v'/x\}\,B'|$. This last equality holds because, by $|B| = |B'|$ and $|v_A| = |v'|$ and the fact that substitution commutes with erasure we know that $|\{v_A/x\}\,B| = |\{v'/x\}\,B'|$. Furthermore by soundness, we have $|\{v_A/x\}\,B| = |B_0|$.

Iiapp, Iapp Similar to the previous case.

Ieq By the IHs for the (added) premises $\Gamma \vdash_{\text{reg}} A \Leftarrow \mathsf{Type}$ and $\Gamma \vdash_{\text{reg}} B \Leftarrow \mathsf{Type}$, we know $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \leadsto A'$ and $\Gamma' \mapsto B \Leftarrow \mathsf{Type} \leadsto B'$.

Then by the IHs for the premises for a and b we know $\Gamma' \mapsto a \Rightarrow a' : A''$ and $\Gamma' \mapsto b \Rightarrow b' : B''$. Now apply EIeq to get $\Gamma' \mapsto a = b \Rightarrow a' = b' : \mathsf{Type}$.

Ijoinc,Ijoinp By the IH for the premise $\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type}$ we know $\Gamma' \mapsto a_1 = a_2 \Leftarrow \mathsf{Type} \leadsto A_0$. There is only one rule for elaborating equality types, so by inversion on that judgement we in fact have $\Gamma' \mapsto a_1 = a_2 \Leftarrow \mathsf{Type} \leadsto a_1' = a_2'$ and $\Gamma' \mapsto a_1 \Rightarrow a_1' : A_1'$ and $\Gamma' \mapsto a_2 \Rightarrow a_2' : A_2'$.

By soundness of elaboration 108 we know $|a_i| = |a_i'|$, so the the reduction behavior is the same. So apply EIjoinc to get $\Gamma' \mapsto \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a_1 = a_2} \Rightarrow \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a_1' = a_2'} : a_1' = a_2'$. Also by soundness of elaboration we know $a_1' = a_2'$ is well-typed, so by TCCRefl $\Gamma' \vDash (a_1' = a_2') = (a_1' = a_2')$ as required.

Iannot By the IH for the premise $\Gamma \vdash A \Leftarrow \mathsf{Type}$ we get $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \leadsto A'$. Then by the IH for $\Gamma \vdash a \Leftarrow A$, we get $\Gamma' \mapsto a \Leftarrow A' \leadsto a'$. Now by EIannot, $\Gamma' \mapsto a_A \Rightarrow a' : A'$.

By soundness of elaboration 108 we know A' is well-typed, so $\Gamma' \vDash A' = A'$ as required.

Icast By the IH for the (added) premise $\Gamma \vdash_{\text{reg}} A \Leftarrow \mathsf{Type}$, we have $\Gamma' \mapsto A \Leftarrow \mathsf{Type} \leadsto A'$ (and $|A| = |A'|$ by soundness). Then by the IH for $\Gamma \vdash a \Rightarrow A$, we know

$\Gamma' \vdash a \Rightarrow a' : A''$ with $\Gamma' \vDash A' = A''$. Likewise, by the IH for the (added) premise $\Gamma \vdash_{\text{reg}} B \Leftarrow \mathsf{Type}$, we have $\Gamma' \vdash B \Leftarrow \mathsf{Type} \rightsquigarrow B'$ (and $|B| = |B'|$ by soundness).

By the definition of $\vDash^{\exists}$ we know there are $\Gamma_1,\ A_1, B_1$ such that $\Gamma_1 \vDash A_1 = B_1$ where $|\Gamma_1| = |\Gamma|$,$|A_1| = |A|$ and $|B_1| = |B|$, so by lemma 78 we have $\Gamma' \vDash A' = B'$. So by TCCTRANS $\Gamma' \vDash A'' = B'$, as required.

The cases for $\Gamma \vdash a \Leftarrow A$ are:

Crec The rule is

$$\dfrac{\begin{array}{l}\Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash_{\text{reg}} a \Leftarrow A_2\\ \Gamma, f : (x\!:\!A_1) \to\ A_2 \vdash_{\text{reg}} A_1 \Leftarrow \mathsf{Type}\\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash_{\text{reg}} A_2 \Leftarrow \mathsf{Type}\\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vDash^{\exists}\ \mathsf{injrng}\,(x\!:\!A_1) \to\ A_2\,\mathsf{for}\,x\end{array}}{\Gamma \vdash_{\text{reg}} \mathsf{rec}\,f\ x.a \Leftarrow (x\!:\!A_1) \to\ A_2}\ \text{RCREC}$$

To apply the induction hypothesis to the first premise, we need to know how the type A_2 elaborates in both the context Γ' and in that context extended with f.

There is only one rule for elaborating arrow types. So by inversion on the hypothesis $\Gamma' \vdash (x : A_1) \to\ A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A$, we in fact have $\Gamma' \vdash (x : A_1) \to\ A_2 \Leftarrow \mathsf{Type} \rightsquigarrow (x : A_1') \to\ A_2'$ and $\Gamma' \vdash A_1 \Leftarrow \mathsf{Type} \rightsquigarrow A_1'$ and $\Gamma', x : A_1' \vdash A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A_2'$ for some A_1' and A_2' such that $\Gamma' \vdash A_1 \Leftarrow \mathsf{Type} \rightsquigarrow A_1'$ and $\Gamma', x : A_1' \vdash A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A_2'$.

Using the fact that $(x : A_1) \to\ A_2$ elaborates to $(x : A_1') \to\ A_2'$, we know $\vdash \Gamma, f : (x : A_1) \to\ A_2 \rightsquigarrow \Gamma', f : (x : A_1') \to\ A_2'$. So by the IH for the first regularity premise we have $\Gamma, f : (x\!:\!A_1') \to\ A_2' \vdash A_1 \Leftarrow \mathsf{Type} \rightsquigarrow A_1''$ for some A_1''.

Similarly, using that A_1 elaborates to A_1'' we know $\vdash \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \rightsquigarrow \Gamma', f : (x : A_1') \to\ A_2', x : A_1''$. So by the IH for the second regularity premise we have $\Gamma, f : (x\!:\!A_1') \to\ A_2', x : A_1'' \vdash A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A_2''$ for some A_2''.

Now by the IH for the first premise of the rule, we get $\Gamma, f : (x\!:\!A_1') \to\ A_2', x : A_1'' \vdash a \Leftarrow A_2'' \rightsquigarrow a'$ for some a'.

By soundness of elaboration we know $(x\!:\!A_1') \to\ A_2'$ is a well-formed type, so $\Gamma' \vDash (x\!:\!A_1') \to\ A_2' = (x\!:\!A_1') \to\ A_2'$. So the search $\Gamma' \vdash (x\!:\!A_1') \to\ A_2' =^{?} (x : A_1''') \to A_2''' \rightsquigarrow v_1$ will succeed for some arrow type $(x\!:\!A_1''') \to\ A_2'''$, since there exists at least one.

By TCCINJDOM, we also know $\Gamma' \vDash A_1' = A_1'''$. Furthermore, by soundness of elaboration (lemma 108) we know that both A_1' and A_1'' are well-formed types in the context $\Gamma', f : (x : A_1') \to A_2'$, and that they erase to the same thing. So we have $\Gamma', f : (x : A_1') \to A_2' \vDash A_1' = A_1''$. By symmetry and transitivity, $\Gamma', f : (x : A_1') \to A_2' \vDash A_1''' = A_1''$. Let v_2 be a proof of that fact. Then using these two proofs, we can produce a proof of equivalence of the contexts.

$$\vDash \{f_{\triangleright v_1}/f\}\,\{x_{\triangleright v_2}/x\} : \Gamma', f : (x\!:\!A_1') \to A_2', x : A_1'' = \Gamma', f : (x\!:\!A_1''') \to A_2''', x : A_1'''.$$

So by context conversion (lemma 110) we know $\Gamma', f : (x\!:\!A_1''') \to A_2''', x : A_1''' \mapsto a \Leftarrow \{f_{\triangleright v_1}/f\}\,\{x_{\triangleright v_2}/x\}\,A_2' \rightsquigarrow a''$. By soundness of elaboration $\Gamma', x : A_1' \vdash A_2' : \mathsf{Type}$, so $f \ \notin \mathsf{FV}\,(A_2')$ and the above statement simplifies to $\Gamma', f : (x\!:\!A_1''') \to A_2''', x : A_1''' \mapsto a \Leftarrow \{x_{\triangleright v_2}/x\}\,A_2' \rightsquigarrow a''$.

By completeness of injrng (lemma 112), we know that $\Gamma', f : (x\!:\!A_1') \to A_2', x : A_1' \vDash \mathsf{injrng}\,(x : A_1') \to A_2'\,\mathsf{for}\,x$. By weakening the judgement $\Gamma' \vDash (x : A_1') \to A_2' = (x\!:\!A_1''') \to A_2'''$ (lemma 57), and because injrng respects CC (lemma 105), we get $\Gamma', f : (x : A_1') \to A_2', x : A_1' \vDash \mathsf{injrng}\,(x : A_1''') \to A_2'''\,\mathsf{for}\,x$. By the CC-equivalences proved above we can find a context equivalence $\vDash \rho : \Gamma', f : (x\!:\!A_1') \to A_2', x : A_1' = \Gamma', f : (x\!:\!A_1''') \to A_2''', x : A_1'''$. So since injrng respects context conversion (lemma 104) we have $\Gamma', f : (x : A_1''') \to A_2''', x : A_1''' \vDash \mathsf{injrng}\,\rho\,(x\!:\!A_1''') \to A_2'''\,\mathsf{for}\,\rho\,x$. We can pick the bound variables f and x to not be free in $(x\!:\!A_1') \to A_2'$, so this is the same as $\Gamma', f : (x\!:\!A_1''') \to A_2''', x : A_1''' \vDash \mathsf{injrng}\,(x\!:\!A_1''') \to A_2'''\,\mathsf{for}\,\rho\,x$. And because injrng respects erasure (lemma 106), we have $\Gamma', f : (x : A_1''') \to A_2''', x : A_1''' \vDash \mathsf{injrng}\,(x : A_1''') \to A_2'''\,\mathsf{for}\,x$, which is what we need as a premise to ECREC.

By TCCERASURE we know $\Gamma', x : A_1''' \vDash \{x_{\triangleright v_2}/x\}\,A_2' = A_2'''$. By weakening (lemma 57) thus $\Gamma', f : (x : A_1''') \to A_2''', x : A_1''' \vDash \{x_{\triangleright v_2}/x\}\,A_2' = A_2'''$. So by casting (lemma 109), we have $\Gamma', f : (x\!:\!A_1'') \to A_2'', x : A_1'' \mapsto a \Leftarrow A_2'' \rightsquigarrow a'''$.

Now apply ECREC to get
$\Gamma \mapsto \mathsf{rec}\,f\ x.a \Leftarrow (x\!:\!A_1) \to A_2 \rightsquigarrow (\mathsf{rec}\,f_{(xA_1'')\to A_2''}\ x.a''')_{\triangleright \mathsf{symm}\,v_1}$.

Cirec Similar to CREC.

Crefl There is only one rule for elaborating equality types, so by inversion on the hypothesis $\Gamma' \mapsto a = b \Leftarrow \mathsf{Type} \rightsquigarrow A'$ we know that in fact $\Gamma' \mapsto a = b \Leftarrow \mathsf{Type} \rightsquigarrow a' = b'$ and $\Gamma' \mapsto a \Rightarrow a' : a_0'$ and $\Gamma' \mapsto b \Rightarrow b' : b_0'$. So by soundness of elaboration (lemma 108) we know a' and b' are well-typed terms, and therefore by lemma 78 and the premise $\Gamma \vDash^{\exists} a = b$, we have $\Gamma' \vDash a' = b'$.

So the search $\Gamma \mapsto (a' = b') =^{?} (a'' = b'') \rightsquigarrow v_1$ will terminate successfully with some equality type $a'' = b''$ such that $\Gamma' \mapsto a'' \overset{?}{=} b'' \rightsquigarrow v$, since there exists at least one such type.

Then apply ECREFL to get $\Gamma \mapsto \mathsf{join} \Leftarrow a' = b' \rightsquigarrow v_{\triangleright \mathsf{symm}\,v_1}$ as required.

Cinf By the mutual IH we have $\Gamma \mapsto a \Rightarrow a' : A'$ for some A' such that $\Gamma \vDash A' = A$.

By transitivity, $\Gamma \vDash A' = B$. Now apply ECinf.

$\square$

Bibliography

[1] Andreas Abel. foetus – termination checker for simple functional programs, 1998.

[2] Andreas Abel. Semi-continuous sized types and termination. *Logical Methods in Computer Science*, 4(2), 2008.

[3] Andreas Abel. MiniAgda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Workshop on Partiality And Recursion in Interative Theorem Provers (PAR 2010), Satellite Workshop of ITP'10 at FLoC 2010*, 2010.

[4] Andreas Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *14th international conference on Foundations of Software Science and Computational Structures (FOSSACS 2011)*, volume 6604 of *LNCS*, pages 57–71. Springer, 2011. doi: 10.1007/978-3-642-19805-2_5.

[5] Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for Type:Type. In *Workshop on Mathematically Structured Functional Programming, MSFP 2008*, 2008.

[6] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[7] Thorsten Altenkirch. The case for smart case: How to implement conditional convertibility? Presentation at NII Shonan seminar 007, Japan, September 2011. Slides available at `http://www.cs.nott.ac.uk/~txa/talks/`.

[8] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming Languages meets Program Verification*, pages 57–68. ACM, 2007.

[9] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. ΠΣ: Dependent types without the sugar. *Functional and Logic Programming*, pages 40–55, 2010. doi: 10.1007/978-3-642-12251-4_5.

[10] Lennart Augustsson. Cayenne – a language with dependent types. In *ICFP '98: International Conference on Functional Programming*, pages 239–250. ACM, 1998.

[11] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In David McAllester, editor, *Automated Deduction — CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 64–78. Springer-Verlag, jun 2000.

[12] Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. Completion without failure. In Aït H. Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 1–30. Academic Press, 1989.

[13] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo De'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, June 1995.

[14] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[15] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *11th international conference on Foundations of Software Science and Computational Structures (FOSSACS 2008)*, volume 4962 of *LNCS*, pages 365–379. Springer, 2008.

[16] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.

[17] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In *Proceedings of 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, volume 3945 of *LNCS*, pages 114–129. Springer-Verlag, 2006.

[18] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Cicˆ: Type-based termination of recursive definitions in the calculus of inductive constructions. In *LPAR*, pages 257–271, 2006.

[19] Gérard Becher and Uwe Petermann. Rigid unification by completion and rigid paramodulation. In Bernhard Nebel and Leonie Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence*, volume 861 of *LNCS*, pages 319–330. Springer Berlin Heidelberg, 1994.

[20] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[21] Yves Bertot and Vladimir Komendantsky. Fixed point semantics and partial recursion in Coq. In *PPDP '08: Principles and practice of declarative programming*, pages 89–96. ACM, 2008.

[22] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. In *ICFP '10: International Conference on Functional Programming*, pages 105–116, 2010.

[23] Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *15th International Conference on Rewriting Techniques and Applications - RTA'04*, 2004.

[24] Max Bolingbroke. Contraint kinds for GHC. Blog post `http://blog.omega-prime.co.uk/?p=127`, September 2011.

[25] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programmming*, 23 (5):552–593, 2013.

[26] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *LNCS*, pages 115–129. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-24849-1_8.

[27] Edwin C. Brady. Idris—systems programming meets full dependent types. In *PLPV'11: Programming languages meets program verification*, pages 43–54. ACM, 2011.

[28] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[29] Luca Cardelli. A polymorphic lambda-calculus with Type:Type. Technical report, DEC SRC, 130 Lytton Avenue, Palo Alto, CA 94301. May. SRC Research Report, 1986.

[30] Chris Casinghino. *Combining Proofs and Programs.* PhD thesis, University of Pennsylvania, 2014.

[31] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *POPL '14: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[32] Adam Chlipala. Certified programming with dependent types, 2011. URL `http://adam.chlipala.net/cpdt`.

[33] Adam Chlipala, J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP'09*, pages 79–90. ACM, 2009.

[34] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, 2007.

[35] Robert Constable and the PRL group. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[36] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *Logic in Computer Science (LICS'87)*, pages 183–193. IEEE, 1987.

[37] Thierry Coquand. An analysis of Girard's paradox. In *In Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, 1986.

[38] Pierre Corbineau. *Démonstration automatique en Théorie des Types.* PhD thesis, University Paris 11, September 2005.

[39] Pierre Corbineau. Deciding equality in the constructor theory. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, volume 4502 of *LNCS*, pages 78–92. Springer Berlin Heidelberg, 2007.

[40] Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages.* PhD thesis, Cornell University, 1998.

[41] Nils Anders Danielsson. Operational semantics using the partiality monad. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 127–138. ACM, 2012. doi: 10.1145/2364527.2364546.

[42] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960. doi: 10.1145/321033.321034.

[43] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962. doi: 10.1145/368273.368557.

[44] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[45] Leonardo de Moura, Harald Rueß, and Natarajan Shankar. Justifying equality. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 125(3):69–85, July 2005.

[46] Anatoli Degtyarev and Andrei Voronkov. The undecidability of simultaneous rigid E-unification. *Theoretical Computer Science*, 166(1&2):291–300, 1996.

[47] Anatoli Degtyarev and Andrei Voronkov. What you always wanted to know about rigid E-unification. In *Logics in Artificial Intelligence*, volume 1126 of *LNCS*, pages 50–69. Springer Berlin Heidelberg, 1996.

[48] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 143–155. ACM, 2011. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034796.

[49] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[50] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130. ACM, 2012. doi: 10.1145/2364506.2364522.

[51] Martín Escardó. Ordinals in Goedel's system T and in Martin-Loef type theory, 2011. Unpublished note, `http://www.cs.bham.ac.uk/~mhe/papers/ordinals/ordinals.html`.

[52] Michael Franssen. Implementing rigid E-unification, 2009.

[53] Jean Gallier, Wayne Snyder, Paliath Narendran, and David Plaisted. Rigid E-unification is NP-complete. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88)*, pages 218–227, 1988.

[54] Jean Gallier, Paliath Narendran, David A. Plaisted, and Wayne Snyder. Rigid E-unification: NP-completeness and applications to equational matings. *Information and Computation*, 87(1-2):129–195, July 1990.

[55] Jean H. Gallier and Tomás Isakowitz. Order-sorted rigid E-unification. Technical Report STERN IS-91-40, Information Systems Department, Leonard N. Stern School of Business, New York University, December 1991.

[56] Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *LNCS*, pages 166–181. Springer Berlin Heidelberg, 2001.

[57] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 28–38. ACM, 1986.

[58] Eduarde Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, volume 996 of *LNCS*, pages 39–59. Springer Berlin Heidelberg, 1995.

[59] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[60] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, 1989.

[61] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 163–175. ACM, 2011. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034798.

[62] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013. doi: 10.1007/978-3-642-39634-2_14.

[63] Jean Goubault. A rule-based algorithm for rigid E-unification. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Computational Logic and Proof Theory*, volume 713 of *LNCS*, pages 202–210. Springer Berlin Heidelberg, 1993.

[64] Gunter Grieser. An implementation of rigid E-unification using completion and rigid paramodulation. Research Report FITL–96–4, FIT Leipzig, June 1996.

[65] Jason Gross, Adam Chlipala, and David I. Spivak. Experience implementing a performant category-theory library in coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, volume 8558 of *LNCS*, pages 275–291. Springer International Publishing, 2014.

[66] Robert Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14(1):71 – 84, 1992.

[67] Fritz Henglein. *Polymorphic Type Inference and Semi-Unification.* PhD thesis, Rutgers, the State University of New Jersey, 1989.

[68] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming*, volume 2793 of *LNCS*, pages 1–56. Springer Berlin Heidelberg, 2003. doi: 10.1007/978-3-540-45191-4_1.

[69] Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). In *ESOP'04: European Symposium on Programming*, volume 2986 of *LNCS*, pages 219–233. Springer, 2004.

[70] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. AURA: A programming language for authorization and audit. In *ICFP '08: International Conference on Functional Programming)*, pages 27–38, 2008.

[71] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 275–286, 2010. doi: 10.1145/1706299.1706333.

[72] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[73] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 468–476, New York, NY, USA, 1990. ACM. doi: 10.1145/100216.100279.

[74] Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *PLPV '12: Proceedings of the sixth workshop on Programming languages meets program verification*, 2012. doi: 10.1145/2103776.2103780.

[75] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37. ACM, 2003. doi: 10.1145/604174.604179.

[76] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 348–370. Springer-Verlag, 2010.

[77] Jordi Levy. Decidable and undecidable second-order unification problems. In *In Proceedings of the 9th Int. Conf. on Rewriting Techniques and Applications (RTA98), volume 1379 of LNCS*, pages 47–60, 1998.

[78] Daniel R. Licata and Robert Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.

[79] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed Haskell programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 81–92. ACM, 2013. doi: 10.1145/2503778.2503786.

[80] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 237–248. ACM, 2010. doi: 10.1145/1706299.1706329.

[81] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[82] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[83] Conor McBride. First-order unification by structural recursion, 2001.

[84] Conor McBride. Elimination with a Motive. In *Types for Proofs and Programs: International Workshop (TYPES 2000)*, volume 2277 of *LNCS*, pages 197–216. Springer, 2002. doi: 10.1007/3-540-45842-5_13.

[85] Conor McBride. Lets see how things unfold: Reconciling the infinite with the intensional (extended abstract). In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science*, volume 5728 of *LNCS*, pages 113–126. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-03741-2_9.

[86] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004. doi: 10.1017/S0956796803004829.

[87] Adam Megacz. A coinductive monad for prop-bounded recursion. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, pages 11–20. ACM, 2007. doi: 10.1145/1292597.1292601.

[88] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[89] Alexandre Miquel. The implicit calculus of constructions - extending pure type systems with an intersection type binder and subtyping. In *TLCA '01: Proceeding of 5th international conference on Typed Lambda Calculi and Applications*, volume 2044 of *LNCS*, pages 344–359. Springer, 2001. doi: 10.1007/3-540-45413-6_27.

[90] Nathan Mishra-Linger. *Irrelevance, Polymorphism, and Erasure in Type Theory*. PhD thesis, Portland State University, 2008.

[91] Nathan Mishra-Linger and Tim Sheard. Erasure and Polymorphism in Pure Type Systems. In *11th international conference on Foundations of Software Science and Computational Structures (FOSSACS 2008)*, volume 4962 of *LNCS*, pages 350–364. Springer, 2008. doi: 10.1007/978-3-540-78499-9_25.

[92] Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 169–180. ACM, 2010. doi: 10.1145/1863543.1863569.

[93] Michał Moskal. Programming with triggers. In *SMT '09: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 20–29, 2009.

[94] Tom Murphy VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007*, 2007.

[95] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *In ACM SIGPLAN International Conference on Functional Programming*, 2008.

[96] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[97] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, April 2007.

[98] Tobias Nipkow. Functional unification of higher-order patterns. In *LICS '93: Proceedings of Eighth Annual IEEE Symposium onLogic in Computer Science*, pages 64–74, 1993.

[99] Ulf Norell and James Chapman. Dependently typed programming in Agda. `http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`.

[100] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. versat: A verified modern sat solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *LNCS*, pages 363–378. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-27940-9_24.

[101] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360. ACM, 2010. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869489.

[102] Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *PLPV '12: Proceedings of the sixth workshop on Programming languages meets program verification*, 2012. doi: 10.1145/2103776.2103779.

[103] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004. doi: 10.1007/1-4020-8141-3_34.

[104] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, 1985. doi: 10.1016/0167-6423(85)90009-7.

[105] Adam Petcher and Aaron Stump. Deciding joinability modulo ground equations in Operational Type Theory. In S. Lengrand and D. Miller, editors, *Proof Search in Type Theories (PSTT)*, 2009.

[106] Simon Peyton-Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: International Conference on Functional Programming*, pages 50–61. ACM, 2006.

[107] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings of the 16th IEEE Symposium on Logic in Computer Science (LICS)*, pages 221–230, 2001. doi: 10.1109/LICS.2001.932499.

[108] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, August 2001. doi: 10.1017/S0960129501003322.

[109] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

[110] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[111] Jason Reed. Proof irrelevance and strict definitions in a logical framework. Technical report, Carnegie-Mellon University, 2002. Senior Thesis, published as technical report CMU-CS-02-153.

[112] Aleksy Schubert. Second-order unification and type inference for Church-style polymorphism. In *POPL '98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 279–288. ACM Press, 1998.

[113] Anton Setzer. Interactive programs in Agda, 2009. URL `http://www.cs.swan.ac.uk/~csetzer/slides/`. Talk presented at Agda Intensive Meeting AIMX, slides available.

[114] Anton Setzer and Peter Hancock. Interactive programs and weakly final coalgebras (extended version). In *Dependently typed programming*, number 04381 in Dagstuhl Seminar Proceedings, 2004.

[115] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 1–12, New York, NY, USA, 2007. ACM. doi: 10.1145/1291151.1291155.

[116] Tim Sheard and Nathan Linger. Programming in Ωmega. In Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók, editors, *2nd Central European Functional Programming School (CEFP)*, volume 5161 of *LNCS*, pages 158–227. Springer, 2007.

[117] Tim Sheard, Aaron Stump, and Stephanie Weirich. Language-based verification will change the world. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 343–348. ACM, 2010. doi: 10.1145/1882362.1882432.

[118] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[119] Vilhelm Sjöberg and Aaron Stump. Equality, quasi-implicit products, and large eliminations. In *ITRS 2010: Proceedings of the 5th workshop on Intersection Types and Related Systems*, 2010. doi: 10.4204/EPTCS.45.7.

[120] Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type

systems. In James Chapman and Paul Blain Levy, editors, *MSFP '12: Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *EPTCS*, pages 112–162. Open Publishing Association, 2012.

[121] Vilhem Sjöberg and Stephanie Weirich. Programming up to congruence. In *POPL '15: 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015. doi: 10.1145/2676726.2676974.

[122] Scott Smith. A computational induction principle, 1991. Unpublished note.

[123] Scott Fraser Smith. *Partial Objects in Type Theory.* PhD thesis, Cornell University, 1988.

[124] Antonis Stampoulis and Zhong Shao. Static and user-extensible proof checking. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 273–284. ACM, 2012.

[125] Thomas Streicher. Investigations into intensional type theory, 1993. Habilitation Thesis, Ludwig Maximilian Universität.

[126] Pierre-Yves Strub. Coq modulo theory. In *CSL*, pages 529–543, 2010.

[127] Aaron Stump and Li-yang Tan. The algebra of equality proofs. In *16th International Conference on Rewriting Techniques and Applications (RTA'05)*, pages 469–483. Springer, 2005.

[128] Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified programming in Guru. In *PLPV '09: Proceedings of the 3rd workshop on Programming Languages meets Program Verification*, pages 49–58, 2009. doi: 10.1145/1481848.1481856.

[129] Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. Termination casts: A flexible approach to termination with general recursion. In *PAR '10: Proceedings of the Workshop on Partiality and Recursion in Interactive Theorem Provers*, 2010. doi: 10.4204/EPTCS.43.6.

[130] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013. doi: 10.1007/s10703-012-0163-3.

[131] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton-Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI 07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in Languages Design and Implementation*, pages 53–66. ACM, 2007.

[132] Kasper Svendsen, Lars Birkedal, and Aleksandar Nanevski. Partiality, state and dependent types. In Luke Ong, editor, *Typed Lambda Calculi and Applications*, volume 6690 of *LNCS*, pages 198–212. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-21691-6_17.

[133] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *ICFP '11: International Conference on Functional Programming*, pages 285–296. ACM, 2011.

[134] William W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967.

[135] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013. URL `http://arxiv.org/abs/1308.0729`.

[136] Ashish Tiwari, Leo Bachmair, and Harald Ruess. Rigid E-unification revisited. In David McAllester, editor, *Automated Deduction - CADE-17*, volume 1831 of *LNCS*, pages 220–234. Springer Berlin Heidelberg, 2000.

[137] Margus Veanes. The relation between second-order unification and simultaneous rigid E-unification. Research Report MPI-I-98-2-005, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, February 1998. Extended version of a paper accepted for LICS'98.

[138] Wendy Verbruggen. *Formal Polytypic Programs and Proofs*. PhD thesis, Trinity College Dublin, 2010.

[139] Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 341–352. ACM, 2012. doi: 10.1145/2364527.2364554.

[140] Philip Wadler. Propositions as types, 2014. Unpublished draft.

[141] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, 2003.

[142] Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In *PLPV '10: Proceedings of the 4th Workshop on Programming Languages Meets Program Verification*, 2010.

[143] Benjamin Werner. Sets in types, types in sets. In *Proceedings of TACS'97*, pages 530–546. Springer-Verlag, 1997.

[144] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[145] Hongwei Xi. Dependent types for program termination verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, pages 231–242, Boston, June 2001.

[146] Hongwei Xi. Applied type system. In *Types for Proofs and Programs: International Workshop (TYPES 2003)*, volume 3085 of *LNCS*, pages 394–408. Springer, 2004. doi: 10.1007/978-3-540-24849-1_25.

[147] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 214–227, 1999. doi: 10.1145/292540.292560.

[148] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285. IEEE Press, 2001.