TinkerType: A Language for Playing with Formal Systems

Michael Y. Levin
Department of CIS
University of Pennsylvania
milevin@cis.upenn.edu

Benjamin C. Pierce
Department of CIS
University of Pennsylvania
bcpierce@cis.upenn.edu

October 23, 2000

Abstract

TinkerType is a pragmatic framework for compact and modular description of formal systems (type systems, operational semantics, logics, etc.). A family of related systems is broken down into a set of *clauses*—individual inference rules—and a set of *features* controlling the inclusion of clauses in particular systems. Simple static checks are used to help maintain consistency of the generated systems.

We present TinkerType and its implementation, and describe its application to two substantial repositories of typed lambda-calculi. The first repository covers a broad range of typing features, including subtyping, polymorphism, type operators and kinding, computational effects, and dependent types. It describes both declarative and algorithmic aspects of the systems, and can be used with our tool, the *TinkerType Assembler*, to generate calculi either in the form of typeset collections of inference rules or as executable ML typecheckers. The second repository addresses a smaller collection of systems, and provides modularized *proofs* of basic safety properties.

1 Introduction

The quest for modular presentations of families of programming language features has a long history in the programming language community. Language designers since Landin [Lan65, Lan66] have understood how to view a multitude of high-level constructs through the unifying lens of the lambda-calculus. Further work has led to more structured approaches such as categorical semantics (e.g. [Gun92, Mit96, Jac99]), action semantics [Mos92], and monadic frameworks [Mog89]. Using these tools, it is now possible to synthesize a variety of interpreters [Ste94, LHJ95, Esp95, etc.] and compilers [LH96, HK98, etc.] from common blueprints or interchangeable building blocks.

For the type systems that accompany these languages, progress on unifying formalisms has been slower, though there have been some significant achievements in restricted domains, including Barendregt's Pure Type Systems [Bar92] and Sulzmann, Odersky, and Wehr's generic treatment of type inference for systems of constrained types [SOW97].[1] In these proposals, the idea is to define a single "parameterized" system from which many particular systems can be obtained by instantiation. This method supports once-and-for-all proofs of properties like subject reduction and decidability that apply automatically to all instances. However, to give a single, parametric description of a collection of formal systems, we must first understand *all* the possible interactions among their features. If—as is common with type systems—some combinations of features are not well understood, then this completely parametric method is not possible; a less structured, more flexible approach is required.

Our goal is to develop a framework that facilitates compact and modular description of very diverse collections of formal systems, taking typed lambda-calculi as our driving example. We adopt a *feature-based*

[1] A related result outside the domain of programming languages is Basin, Matthews, and Viganò's modular presentation of modal logics in Isabelle [BMV95].

approach, breaking down a family of formal systems into a set of *clauses* annotated with a set of *features* chosen by the user to reflect the structure of the domain. (In the domain of typed lambda-calculi, the clauses are individual inference rules, and features correspond to the presence of particular type constructors or structures such as subtyping or kinding in a given calculus.) A clause may have multiple variants, each annotated with a set of relevant features (drawn from some set of atomic feature names) that control its inclusion in particular systems. A complete system is specified by a set of features.

Several things can go wrong in the process of maintaining a repository of features and clauses and extracting systems from it. A change in a clause may introduce inconsistencies with other variants of the same clause; a set of features identifying an extracted system may be nonsensical; the clauses of a system may turn out to be incompatible with each other. In our open-ended setting (the clauses themselves are uninterpreted strings as far as TinkerType is concerned), ensuring the "reasonableness" of generated systems is difficult in general. We have, however, identified several common sources of error in practice and introduced static consistency checks to help prevent them.

The contributions described in this paper are twofold. First, we present the TinkerType framework and describe its implementation. Second, we use it to classify a number of familiar typed lambda-calculi, including systems with subtyping, polymorphism, type operators and kinding, computational effects, and dependent types. Our first repository of typed lambda-calculi can be used to extract both inference-rule presentations of systems (as latex documents) and ML sources that can be compiled to produce running typecheckers and interpreters. Our second repository can be used to generate proofs of basic metatheoretic properties. (Strictly speaking, what we generate are proof *scripts*, whose correctness must be externally verified. This point is discussed in Section 6.) These experiments represent substantial experience with using the TinkerType framework in practice.

The remainder of the paper proceeds as follows. In Sections 2 and 3, we give precise definitions of the fundamental concepts underlying TinkerType: clauses, features, and the process of composing systems and checking their consistency. Section 4 describes our implementation. Sections 5 and 6 present our two repositories for typed lambda-calculi. Sections 7 and 8 describe related and future work.

The TinkerType implementation, user manual, and examples are available from our web site, `http://www.cis.upenn.edu/~milevin/tinkertype`.

2 Assembling Systems from Features and Clauses

A formal system can be described as a set of *judgements*, each consisting of a set of *clauses*. The simply typed lambda-calculus ($\lambda^{\rightarrow}$), for example, is a formal system with two judgements: typing and evaluation. The typing judgement contains clauses like

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \rightarrow \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \qquad \text{T-APP}$$

while the evaluation relation has clauses like the beta-reduction rule.[2]

This is obviously a rather syntactic view of formal systems. More abstractly, we might say that the simply typed lambda-calculus is a pair of sets of derivation trees: one set of trees with conclusions like $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and one with conclusions like $\mathtt{t} \rightarrow \mathtt{t}'$. More abstractly yet, we might view $\lambda^{\rightarrow}$ as a pair of *relations* obtained from these sets of trees. Or again, $\lambda^{\rightarrow}$ can be represented by a pair of functions in, say, ML. Since we are interested in *all* of these views, we avoid committing to a particular one by taking clauses as primary and dealing with them as uninterpreted atoms in our formalism.

A given clause may appear in many different systems. For example, both pure $\lambda^{\rightarrow}$ and $\lambda^{\rightarrow}$ with booleans contain the application rule shown above. On the other hand, in other systems, the same clause may take different forms. In (an algorithmic presentation of) $\lambda^{\rightarrow}$ with subtyping, the application rule refines the rule above by adding an extra subtyping premise:

[2] Strictly speaking, there are also "judgements" defining the syntax of types, terms, and contexts. For example, the term syntax judgement contains clauses like "if $\mathtt{T}_1$ and $\mathtt{T}_2$ are types, then so is $\mathtt{T}_1 \rightarrow \mathtt{T}_2$." Our discussion elides these syntactic judgements, for brevity.

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_2}{\to}\mathtt{T_1} \qquad \Gamma \vdash \mathtt{t_2} : \boxed{\mathtt{U}} \qquad \boxed{\mathtt{U} <: \mathtt{T_2}}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_1}} \qquad \text{T-APP}$$

The difference between these rules is that the latter is relevant to systems with subtyping, while the former belongs in simple systems without subtyping. We formalize this intention by annotating each rule with a set of *features*:

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_2}{\to}\mathtt{T_1} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_1}} \qquad \text{T-APP } [\mathit{arrow}, \mathit{typing}]$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_2}{\to}\mathtt{T_1} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{U} \qquad \mathtt{U} <: \mathtt{T_2}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_1}} \qquad \text{T-APP } [\mathit{arrow}, \mathit{sub}]$$

The choice of features is determined by the set of systems we intend to describe. The *arrow* feature is present in any system with function types, while *sub* characterizes systems with subtyping. The *typing* feature is a "base case," present in any system with a typing relation.

One might argue that the *typing* feature should appear in the annotations on the second rule as well, since this rule is obviously "relevant to" typing. In general, every lambda-calculus with subtyping will also have a typing relation. To account for facts like this, it is convenient to introduce the notion of *dependencies* between features. The *sub* feature depends on *typing*, and we omit *typing* from the annotations on the second variant of the T-APP rule. (Indeed, since we are talking only about typed calculi in this paper, the *typing* feature will always be elided in feature annotations from now on.)

To put a little more flesh on these bones, let us examine a small collection of features and clauses for type systems with booleans, arrow types and subtyping (a fragment of the repository of typed lambda-calculi described in Section 5). For the sake of brevity, we present just the clauses for the algorithmic typing and subtyping judgements.

Figure 1 shows the features used in this example and the dependencies between them. In addition to the features we have already mentioned, the figure introduces *bool*, which classifies systems with booleans and conditional expressions, *top*, which corresponds to `Top` type and the subtyping rule for it, and *calcjoin*, which introduces the functionality of calculating joins (and meets) of types, used in the typing rule for conditionals in the presence of subtyping. The feature *tmvar* signals the presence of term variable binding in a system.

Figure 2 defines the set of clauses for our example. By specifying lists of features, we can build the following systems from this repository:

[*bool*]	first-order boolean expressions
[*arrow*]	$\lambda^{\to}$
[*arrow*, *bool*]	$\lambda^{\to}$ with boolean expressions
[*arrow*, *sub*]	$\lambda^{\to}$ with subtyping
[*arrow*, *bool*, *calcjoin*]	$\lambda^{\to}$ with subtyping and booleans

(We can also use lists of features to specify systems that don't make sense: for example, [*arrow*, *bool*, *sub*] without *calcjoin*. Our tool provides some support for detecting and rejecting such specifications, as described in Section 3.2.) The last system in the list is the most extensive system one can build from the given repository. It contains every clause except T-APP [*arrow*] and T-IF [*bool*], which are superseded by T-APP [*arrow*, *sub*] and T-IF [*bool*, *calcjoin*].

We can formalize the above intuitions as follows. First, fix a set **Names** of *clause names* and a set **Cnt** of *clause contents*. (Both of these sets are uninterpreted here. In the implementation, they are strings.) A *repository* is a tuple $\langle \mathbf{Fts}, \mathbf{Dep}, \mathbf{Cls} \rangle$, where

$\mathbf{Fts}$		is a set of *features*,
$\mathbf{Dep}$	$\subseteq \mathcal{P}(\mathbf{Fts}) \times \mathcal{P}(\mathbf{Fts})$	is a *feature dependency* relation, and
$\mathbf{Cls}$	$\subseteq \mathbf{Names} \times \mathcal{P}(\mathbf{Fts}) \times \mathbf{Cnt}$	is a set of *clauses*.

Given a set of features F, we define $\mathit{closure}(F)$ to be the least superset F' of F that is closed under the dependency relation—i.e., such that if $F_1 \subseteq F'$ and $(F_1, F_2) \in \mathbf{Dep}$, then $F_2 \subseteq F'$. We say that a set of features F_1 *dominates* another set F_2 if $\mathit{closure}(F_2) \subseteq \mathit{closure}(F_1)$.

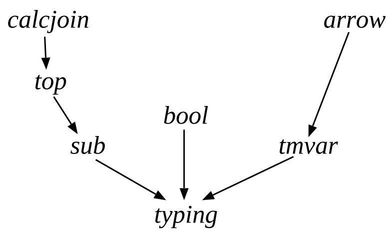

Figure 1: Features for arrows, booleans, and subtyping

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{T-VAR } [\textit{tmvar}]$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_2 \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1} \qquad \text{T-LAM } [\textit{arrow}]$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \qquad \text{T-APP } [\textit{arrow}]$$

$$\Gamma \vdash \mathtt{false} : \mathtt{Bool} \qquad \text{T-FALSE } [\textit{bool}]$$

$$\Gamma \vdash \mathtt{true} : \mathtt{Bool} \qquad \text{T-TRUE } [\textit{bool}]$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{T-IF } [\textit{bool}]$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \qquad \text{T-APP } [\textit{arrow}, \textit{sub}]$$

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{S-ARROW } [\textit{arrow}, \textit{sub}]$$

$$\mathtt{S} <: \mathtt{Top} \qquad \text{S-TOP } [\textit{top}]$$

$$\mathtt{Bool} <: \mathtt{Bool} \qquad \text{S-BOOL } [\textit{bool}, \textit{sub}]$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathtt{T}_1 <: \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}_3 \qquad \mathtt{T} = \mathtt{T}_2 \vee \mathtt{T}_3}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{T-IF } [\textit{bool}, \textit{calcjoin}]$$

Figure 2: Clauses for arrows, booleans, and subtyping

A *clause* cl is a triple $\langle n, F, c\rangle$, where $n \in \mathbf{Names}$ is a label identifying the clause, $F \subseteq \mathbf{Fts}$ is a set of features that governs inclusion of the clause in particular systems, and $c \in \mathbf{Cnt}$ is the actual content of the clause. We say that cl is *relevant* to the set of features F. Finally, we say that a clause $cl_1 = \langle n_1, F_1, c_1\rangle$ dominates $cl_2 = \langle n_2, F_2, c_2\rangle$ if $n_1 = n_2$ and F_1 dominates F_2.

Now we have the tools to specify how a system is assembled, given a set of features F. First, we extract from the repository all the clauses whose sets of features are dominated by F. Then we partition these clauses into sets of clauses with identical labels. We verify that each partition has a single maximal clause according to the dominance relation—or, if there are multiple maximal clauses in a partition, that all of them have the same contents (this bit of flexibility turns out to be quite convenient in practice). We select a maximal clause from each partition. The contents of these clauses form the system.

3 Consistency Checking

The basic framework presented in the previous section is very flexible, but it needs a little more structure before we can use it to develop large repositories. In this section, we introduce some simple static consistency checks that help ensure both coherence of the repository and consistency of generated systems. While these checks provide no absolute guarantees of correctness, we have found them to be extremely useful in practice.

To make the checks precise, we begin by adding some elements to the definitions of the previous section. A repository is now a tuple $\langle \mathbf{Fts}, \mathbf{Dep}, \mathbf{Cls}, \mathbf{Rfn}, \mathbf{Con}, \mathbf{Csig}\rangle$, where

$\mathbf{Fts}$, $\mathbf{Dep}$, $\mathbf{Cls}$	are as before,
$\mathbf{Rfn} \subseteq \mathbf{Cnt} \times \mathbf{Cnt}$	is a *refinement relation* on clause contents,
$\mathbf{Con}$	is a set of *feature constraint formulas* (propositional formulas over $\mathbf{Fts}$), and
$\mathbf{Csig}$	is a *clause signature* relation.

The new elements are discussed in the subsections that follow.

3.1 Clause Refinement

The refinement relation $\mathbf{Rfn}$ verifies that the dominance relation between clauses is reflected in their contents. For example, in the previous section we saw two variants of the application clause, T-App $[\mathit{arrow}, \mathit{typing}]$ and T-App $[\mathit{arrow}, \mathit{sub}]$, with the latter dominating the former because of the dependency of *sub* on *typing*. It is natural to expect that the contents of these clauses should be similar, indeed that the dominating clause should be "more specific" in the sense that it performs the same function but takes into account the presence of subtyping. If this were not the case, it would almost surely indicate some confusion in the repository (an accidental name clash between clauses, incorrect feature dependency, mistaken feature annotation, etc.). To prevent such inconsistencies, we demand that, whenever a clause $\langle n, F_1, c_1\rangle$ dominates a clause $\langle n, F_2, c_2\rangle$, we also have $c_1\ \mathbf{Rfn}\ c_2$.

We need to take $\mathbf{Rfn}$ as a part of the repository—i.e., we must assume that it is provided externally, not calculated by the framework—because the clause contents themselves are uninterpreted by the framework. Forcing the user to think explicitly about the refinement relation between different versions of a clause introduces a useful cross-check between the activities of deriving new clauses from simpler variants and annotating them with features. Although this check works completely at the level of strings, it is surprisingly effective in catching "version control" errors during maintenance of large rule repositories.

3.2 Feature Consistency

In some families of formal systems, there are combinations of features that do not make sense. For example, it has been shown [Ghe90] that the subtype relation of the "full" variant of System $\mathrm{F}_{<:}$ is not closed under joins. Other systems, like $\lambda^{\rightarrow}$ with booleans and subtyping, rely on the existence of joins (signaled by the *calcjoin* feature) to calculate minimal types. Thus, a system like $[\mathit{ffsub}, \mathit{bool}, \mathit{calcjoin}]$, where *ffsub* is the feature selecting the full variant of $\mathrm{F}_{<:}$, will be defective (in particular, the typing algorithm will be incomplete with respect to a declarative presentation of the system).

To prevent the extraction of such systems, we include in the repository a set **Con** of feature constraint formulas—propositional formulas over the set of features—and say that a system identified by features F is *consistent* if $closure(F)$ satisfies every formula in **Con**.

3.3 Judgement Signatures

Our last form of consistency checking is more speculative (and as yet unimplemented). From our experience using TinkerType, we feel that something of this kind is needed, but we are less confident about the details of the design: it seems difficult to strike the right balance between helping and getting in the way.

A given judgement may have different "shapes" when it appears in different formal systems. For example, the subtyping judgement in the simply typed lambda-calculus is a two-place relation on types, $\mathtt{S} <: \mathtt{T}$. In $F_{<:}$, it is a three-place relation between contexts and pairs of types, $\Gamma \vdash \mathtt{S} <: \mathtt{T}$. To track these variations in shape and prevent mixing rules of different shapes, we can introduce a notion of *judgement signatures*. For example, in a type system with kinds, the signature of the typing judgement (and all of its clauses) would be:

$$\mathit{Typing}(\Gamma, \mathtt{t}, \mathtt{T}) \times \mathit{Kinding}(\Gamma, \mathtt{T}, \mathtt{K}) \to \mathit{Typing}(\Gamma, \mathtt{t}, \mathtt{T})$$

That is, typing in this system is a three-place relation on contexts, terms, and types, and it depends both on itself and on the kinding relation (i.e., both typing and kinding assertions may occur as major premises of typing rules). Formally, judgement signatures are added by extending the definition of a repository with new sets **Syn** of *syntactic categories* and **Jdg** of *judgement names*. A *statement signature* like $\mathit{Typing}(\Gamma, \mathtt{t}, \mathtt{T})$ consists of a judgement name (*Typing*) and a sequence of syntactic categories. A *judgement signature* has the form $S_1 \times \cdots S_n \to S$, where each S_i and S are statement signatures.

Judgement signatures enable a consistency check that prevents clauses intended for different versions of a judgement from ending up in the same system. This is accomplished by requiring that all the clauses defining a particular judgement in a given system have exactly the same signature. This check alerts the user when a clause that should have been overridden to take a new feature into account is "improperly inherited" verbatim from a simpler system. For example, it will prevent the inclusion of the clause

$$\frac{\Gamma, \mathtt{x}\mathtt{:}\mathtt{T}_2 \vdash \mathtt{t}_1 \mathrel{:} \mathtt{T}_1}{\Gamma \vdash \lambda \mathtt{x}\mathtt{:}\mathtt{T}_2\mathtt{.}\mathtt{t}_1 \mathrel{:} \mathtt{T}_2 \to \mathtt{T}_1} \qquad \text{T-L{\scriptsize AM}}\ [\mathit{arrow}]$$

in a system with a kinding relation, because its signature is $\mathit{Typing}(\Gamma, \mathtt{t}, \mathtt{T}) \to \mathit{Typing}(\Gamma, \mathtt{t}, \mathtt{T})$, rather than $\mathit{Typing}(\Gamma, \mathtt{t}, \mathtt{T}) \times \mathit{Kinding}(\Gamma, \mathtt{T}, \mathtt{K}) \to \mathit{Typing}(\Gamma, \mathtt{t}, \mathtt{T})$. This helps prevent the generation of unsound or nonsensical systems—for example, when the user forgets to override the T-Lam clause with its kind-checking variant.

Our implementation does not support judgement signature checking yet: some substantial design issues remain to be addressed before the idea can be tried out in practice. In particular, since clause contents are uninterpreted (e.g., just strings), TinkerType cannot infer judgement signatures automatically; like the refinement relation on clauses, the relation mapping clauses to signatures must be specified by the user as part of the repository. Accomplishing this smoothly, without imposing an undue burden on the user, requires a mechanism for indicating judgement signatures for large collections of rules at once. Also, a straightforward realization of the consistency check prevents some legitimate uses of inheritance. Consider again the two type systems with and without kinding. Even though their typing judgements have different signatures, the application clause T-App is identical in both systems (unlike T-Lam). We believe that *silent* inheritance in cases like this should be prohibited—the user should be forced to look at the simple clause and verify that it will actually work unchanged in a system with kinding, but we need to make it easy for the user to record the fact that this check has indeed been performed. In the meantime, the implementation provides a simpler consistency check that detects some of the same problems.

4 The TinkerType Assembler

Based on the above ideas, we have designed a small language for describing repositories and implemented a tool that assembles systems and checks their consistency.

We use arbitrary strings for the contents of clauses. In our repositories of typed lambda-calculi, some of these strings are bits of ML code; others are bits of TeX source; others are bits of proofs. For example, here is the ML clause for typechecking conditional expressions:

```
T-If
  {#TmIf(fi,s1,s2,s3) →
      if tyeqv ctx (typeof ctx s1) TyBool then
        let tyS = typeof ctx s2 in
        if tyeqv ctx tyS (typeof ctx s3) then tyS
        else error fi "arms of conditional have different types"
      else error fi "guard of conditional not a boolean"#}
```

`T-If` is the name of the clause, and the content appears between the brackets `{#` and `#}`. This clause forms part of the definition of the `typeof` function in a generated typechecker. (It can be paraphrased as follows: In the case where we are typechecking a `TmIf` abstract syntax node, we first check whether the type of the guard `s1` (in the current context `ctx`) is equivalent to `TyBool`. If so, we calculate the type of the `then` part `s2` and call this type `tyS`. We calculate the type of the `else` part `s3` and check that it is equivalent to `tyS`. If it is, then `tyS` is the type of the `TmIf` node. If either test fails, we generate an appropriate error.) Individual clauses are not annotated with their relevant features. Instead, we introduce a coarser-grained structuring mechanism called a *component*, which gives a single annotation for several clauses relevant to the same set of features. Besides reducing clutter, components are useful units of grouping in the repositories.

Within a component, clauses are further grouped into nested sections. At the top level, we have one section for all the TeX rules and another for ML code. Within the ML section, there are subsections for abstract syntax, for lexing and parsing, for printing, and for core typechecking functions. The latter contains subsections for individual functions (typing, subtyping, kinding, etc.), and they, in turn, contain the actual clauses. The following fragment is a part of the [*bool*, *typing*] component:

```
core {
  tyeqv {
    Eqv-Bool {# ... #}
  }
  typeof {
    T-False {# ... #}
    T-True {# ... #}
    T-If {# ... #}
  }
}
```

Each section may define special clauses called `header`, `footer`, and `separator`. For example, the `typeof` section contains the following header and separator:

```
header    {#let rec typeof ctx t = match t with#}
separator {#| #}
```

(In the actual repositories, these definitions also contain information about line-breaking and indentation.) When the tool prints a section, it outputs the header, the section's subitems separated by the separator, and then the footer. For example, the generated `typeof` function looks like this:

```
let rec typeof ctx t = match t with
  ...
| TmIf(fi,s1,s2,s3) →
    if tyeqv ctx (typeof ctx s1) TyBool then
      let tyS = typeof ctx s2 in
      if tyeqv ctx tyS (typeof ctx s3) then tyS
      else error fi "arms of conditional have different types"
    else error fi "guard of conditional not a boolean"
| ...
```

The refinement relation is generated from user annotations in clause contents. To indicate that one version of a clause refines another, we enclose the new or changed parts of the refined clause in `[[` and `]]` brackets. The assembler notes that the latter refines the former if the unbracketed segments of the refined clause appear verbatim in the original. For example, the variant of `T-If` for systems with subtyping is annotated like this:

```
T-If
  {#TmIf(fi,s1,s2,s3) →
      if [[subtype]] ctx (typeof ctx s1) TyBool then
        [[join ctx (typeof ctx s2) (typeof ctx s3)]]
      else error fi "guard of conditional not a boolean"#}
```

To build a system, we specify a set of features plus a file or directory, called a *template*, containing a skeleton for the generated system. For ML systems, the template is a directory containing a makefile, boilerplate for lexing and parsing, a top-level command loop, and skeleton modules (containing holes marked with the names of sections to be inserted at each point) for the major components of the typechecker.

The TinkerType implementation also includes fairly sophisticated facilities for prettyprinting, controlling the ordering of clauses and sections, automatic highlighting of differences between a generated system and its ancestors, macro substitution, debugging support for generated systems, and analysis of feature conflicts and rule inconsistencies. Details can be found in the user manual (available through `http://www.cis.upenn.edu/~milevin/tinkertype`).

5 The Next 700 Type Systems

We have carried out two substantial experiments with using TinkerType. In the first, the goal was to encode a very broad range of typing features, including subtyping, polymorphism, type operators and kinding, computational effects, and dependent types, and to develop both printable TeX presentations and executable ML typecheckers for the systems in parallel. In the second, we addressed a smaller collection of features, concentrating instead on modularizing the proofs of their basic metatheory. This section describes the first experiment; the second is discussed in Section 6.

In the following subsections, we introduce several groups of related features and show how to combine them to obtain several familiar type systems, including the systems of Barendregt's lambda cube and various calculi with subtyping. We concentrate on the algorithmic variants of the systems (i.e., the running ML systems, rather than their more abstract presentations in TeX), since they involve a more interesting "feature skeleton" than their declarative counterparts. Each of the following subsections shows one "slice" through our repository, introducing several related features and discussing their use.

5.1 Features for Variables and Binders

Some of the intricacies in the repository arise from the need to deal carefully with (term and type) variables and substitution. In particular, we would like to make each generated system as simple as possible, avoiding generating unnecessary functionality like type-substitution operations in systems whose types do not contain variables. We must therefore take into account whether type and term variables are present in each system, whether type and term variables can appear in terms and types (respectively), and which substitution operations (terms in terms, types in terms, types in types, etc.) are needed. (Remember that we are generating running ML typecheckers here; the standard "We assume the usual conventions about alpha-conversion and capture-avoiding substitution..." does not suffice! We could, of course, alternatively adopt a "higher-order abstract syntax" treatment of binders and substitution, as in LF [HHP92] and some concrete compiler implementations [PE88]. Our choice of an explicit treatment of names is a matter of taste, not principle: TinkerType could be used equally well to formulate the HOAS variants of our systems.)

Figure 3 shows the hierarchy of features related to variables and binders. The two "trunks" in the diagram control the core functionality of variables and substitution. We use the features *tmtmsubst*, *tmtysubst*, *tytmsubst*, and *tytysubst* to control the generation of functions for substitution of terms inside terms, terms inside types, types inside terms, and types inside types respectively. The "leaves" in the hierarchies represent specific binding operators whose definitions make use of various kinds of substitution. The features *arrow* and *family* in the term-variable hierarchy (on the left) stand for abstraction of terms over terms (ordinary lambda-abstraction) and types over terms (families of types indexed by terms). They correspond to the *operator* and *all* features in the type variable hierarchy, which characterize abstraction of types over types (type operators) and terms over types (polymorphic functions), respectively. The feature *let* stands for local definitions, *some* for existential types, *tmrec* and *tyrec* for recursive terms and iso-recursive types, *variant* for variants and a case construct, and *tmabbrev* and *tyabbrev* for top-level abbreviations of terms and types.

Note that "technical features" like *tytmsubst* do not need to be mentioned in user-level descriptions of systems, since mentioning a higher-level feature like *all* will cause it to be included automatically. Also, note that *tytmsubst* implies *tytysubst*, since substituting a type into a term might involve substituting through a type embedded in the term, and similarly for *tmtysubst* and *tmtmsubst*.

The variable related features contain the cornerstones for the systems of Barendregt's lambda cube: *arrow*, *all*, *operator*, and *family*. We will return to the cube later on.

5.2 Simple Features

Another way to classify features is based on the kinds of judgements they support. The feature *typing* enables the typing judgement. A large number of systems can be built based on this judgement. We call such systems *simple*, and Figure 4 shows the hierarchy of features used to build them.

The simple feature hierarchy includes almost all of the variable operation hierarchy and defines several new "content" features. The feature *nat* represents natural numbers and the operation of iteration on them. Because one argument of such iterations is a higher-order function, *nat* implies *arrow*. The feature *pattern* combines the capabilities of local definition (*let*) and record projection (*record*) to enable pattern matching syntax for record values. The features *bool* and *unit* represent booleans and a unit type; *basety* introduces atomic base types; *coerce* provides explicit typecasts.

5.3 Evaluation and Normalization Features

Any system built from the simple features must contain either a normal-order or a call-by-value reduction relation on terms. We prohibit inclusion of both evaluation and reduction in the same system by introducing features *normalize* and *eval* (shown in Figure 5) and defining a feature constraint *normalize* $\oplus$ *eval* (where $\oplus$ is exclusive or). For systems with computational effects, on the other hand, call-by-value reduction must be selected. The feature *eff* controls inclusion of the necessary infrastructure to implement effects, and *ref* is built on top of it to support reference cells.

5.4 Kinding Features

Systems with non-trivial well-formedness conditions on types (for example, systems with type operators or dependent type families) introduce a kinding judgement. The feature hierarchy shown in Figure 6 introduces *kinding* as a base feature for kinding support; *dep* signals mutual dependency between the typing and kinding relations and introduces dependent functions; *operator* and *family* add type operators and type families respectively.

5.5 Type Conversion Features

Syntactically distinct types must be checked for "convertibility" in systems with either type abbreviations (*tyabbrev*) or beta-reduction on types (*operator* or *family*). The feature *tyconvert* (shown in Figure 7) signals the presence of any one of the above three features and triggers the use of conversion testing at many points in the typing rules instead of simple type equality.

We have now defined all the necessary features for building the systems of Barendregt's lambda cube:

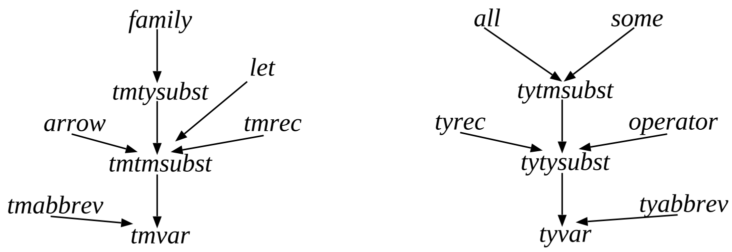

Figure 3: Variable and binding features

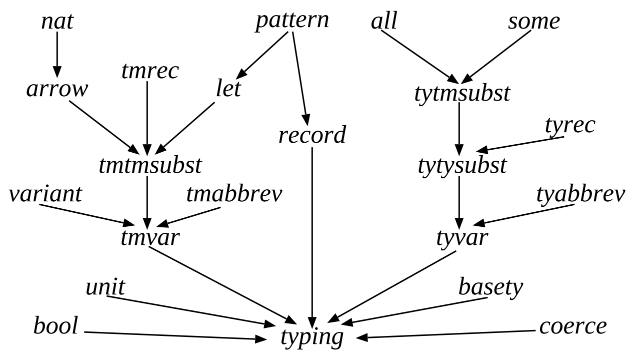

Figure 4: Simple features

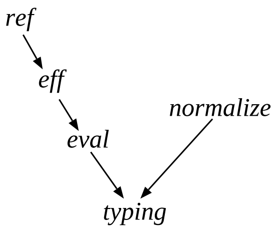

Figure 5: Evaluation and normalization features

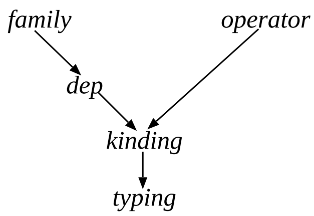

Figure 6: Kinding features

$[\mathit{arrow}]$	$\lambda^{\to}$
$[\mathit{arrow}, \mathit{all}]$	System F
$[\mathit{arrow}, \mathit{family}]$	types dependent on terms
$[\mathit{arrow}, \mathit{all}, \mathit{operator}]$	System F^{ω}
$[\mathit{arrow}, \mathit{all}, \mathit{operator}, \mathit{family}]$	calculus of constructions

The features *all*, *operator*, and *family* define the three dimensions of the cube, while *arrow* marks the origin point.

5.6 Subtyping Features

A difficulty in the design of algorithmic subtyping systems arises from the nonexistence of joins in the full variant of System $F_{<:}$. We want to employ the algorithm for calculating joins in systems where it makes sense, but systems that are not closed under joins must provide some other mechanism of obtaining a common supertype (for example, they might annotate "multi-armed" expressions like `if` and `case` with their intended result types). We introduce two mutually exclusive features, *calcjoin* and *anotjoin*, one of which must be specified for any system that contains a multi-armed expression. While *calcjoin* signals that a system contains the algorithm for calculating joins, *anotjoin* represents systems where multi-armed expressions must use annotated forms.

We also introduce a mutually exclusive pair of features *kfsub* and *ffsub*, which characterize the "kernel" and "full" versions of $F_{<:}$ (see [Pie94] for details). Systems including *ffsub* will not be closed under joins and thus cannot take advantage of the algorithm for calculating joins.

The above intuitions are formalized by the following feature constraints:

$$\mathit{sub} \wedge (\mathit{bool} \vee \mathit{variant}) \Rightarrow \mathit{calcjoin} \oplus \mathit{anotjoin}$$
$$\neg(\mathit{kfsub} \wedge \mathit{ffsub})$$
$$\neg(\mathit{ffsub} \wedge \mathit{calcjoin})$$

Figure 8 presents the subtyping hierarchy (*boundedtyvar* represents bounded type variable declarations, which are required by both variants of $F_{<:}$). We can build the following familiar subtyping systems from the presented features:

$[\mathit{arrow}, \mathit{bool}, \mathit{calcjoin}]$	$\lambda^{\to}$ with subtyping, booleans and calculated joins
$[\mathit{arrow}, \mathit{all}, \mathit{kfsub}]$	kernel $F_{<:}$
$[\mathit{arrow}, \mathit{all}, \mathit{ffsub}]$	full $F_{<:}$
$[\mathit{arrow}, \mathit{all}, \mathit{operator}, \mathit{kfsub}]$	kernel $F^{\omega}_{<:}$
$[\mathit{arrow}, \mathit{all}, \mathit{operator}, \mathit{ffsub}]$	full $F^{\omega}_{<:}$

6 Modular Metatheory

We now describe our second experiment with modularization of typed lambda-calculi. Here, our focus was not on implementations, but rather on modularized proofs of basic metatheoretic properties, leading up to subject reduction and progress theorems.

In modularizing proofs, we followed exactly the same lines as the modular presentations of the systems themselves. Just as systems are cut up into clauses annotated with features, so proofs are cut up into individual cases and annotated with these same features. Just as we assemble the various judgements of a system by collecting the appropriate inference rules and printing them out with headers like "the typing relation is the least relation closed under the following rules," we likewise assemble proofs by collecting the appropriate clauses and printing them with headers like "by induction on typing derivations."

Of course, just as it is possible to generate nonsensical type systems because of errors in the repository, the "proofs" that are assembled in this way are not guaranteed to be well formed. Strictly speaking, they are *proof scripts*, whose validity must be checked after the fact. In our present repository, proofs are presented in standard mathematical English, and their validity must be checked by hand. Ultimately, of course, one would like to present them in the form of proof scripts for some automated proof checker. We see no obstacle—beyond the usual tedium associated with formalizing metatheory of type systems—to taking this

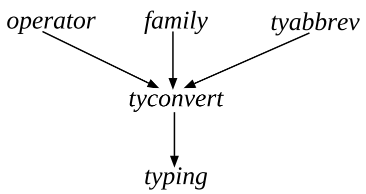

Figure 7: Type conversion features

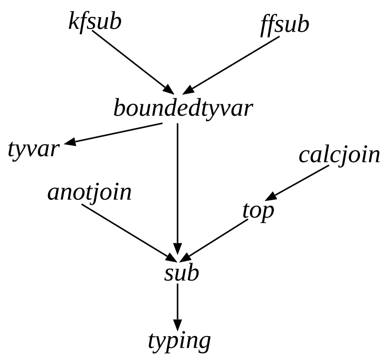

Figure 8: Subtyping features

step. The repository would certainly become larger, but we believe its modular structure would not need to change significantly.

This repository a smaller collection of features than the repository described in Section 5. It includes numbers, booleans, functions, subtyping, polymorphism, and mutable references, plus one significant extension to the systems that we have seen up to now: equi-recursive types. It can be used to generate proofs of subject reduction and progress theorems, plus all required lemmas, for any sensible combination of these features.

We begin by discussing systems without recursive types, then address recursive types in Section 6.3.

6.1 Judgement Forms

The systems encoded in this repository may include four judgements: evaluation, typing, subtyping, and type exposure.

The evaluation judgement appears in all systems. In its simplest form it relates terms to terms: $\mathtt{t} \longrightarrow \mathtt{t}$. In systems with effects (identified by the feature *store*), the evaluation relation is defined in store-passing style and has the form: $\mathtt{t};\mu \longrightarrow \mathtt{t};\mu$.

The typing judgement is also present in every system. In its simplest form, it relates a term to a type: $\mathtt{t} : \mathtt{T}$. In systems with term variables (feature *tmvar*) the typing judgement also carries a context: $\Gamma \vdash \mathtt{t} : \mathtt{T}$. Orthogonally, in systems with mutable references (feature *store*), the typing relation includes a store typing: $\Sigma \vdash \mathtt{t} : \mathtt{T}$. In systems with both *tmvar* and *store*, the typing judgement has the form $\Gamma;\Sigma \vdash \mathtt{t} : \mathtt{T}$.

The subtyping judgement appears only in systems including the feature *sub*. In simple subtyping systems, the subtyping judgement relates pairs of types: $\mathtt{T}\mathtt{<:}\mathtt{T}$. In systems with bounded quantification (feature *fsub*), we must also track bounds of type variables by extending contexts with subtyping assumptions and annotating the subtyping relation with a context: $\Gamma \vdash \mathtt{T}\mathtt{<:}\mathtt{T}$.

Systems with bounded quantification also require a type exposure judgement, whose goal is to reveal the least concrete supertype of a type variable: $\Gamma \vdash \mathtt{T} \Uparrow \mathtt{T}$. Unlike the other judgements, this one is not mentioned in the inference rules defining the other judgements. Rather, it is necessary for the statements of the subtyping inversion lemmas discussed below.

6.2 Proofs

Figure 9 shows the dependencies between the two main theorems we have encoded—type preservation and progress—and their associated lemmas. An edge from one node (lemma) to another means that the proof of the former invokes the statement of the latter. The lemmas fall into several groups: various forms of substitution (SUBS1–SUBS4), covering substitution of both types and terms into typing and subtyping judgements, permutation (PERM1–PERM4), weakening (T-WEAK and S-WEAK), inversion properties of the typing judgement (TABS-INV through ABS-INV) and subtyping judgement (ALL-INV through NAT-INV), and canonical forms properties (TABS-CANON through NAT-CANON). COMM-SUBS states a commutation property of substitutions; E-EXP is preservation of typing in systems with references under extensions of the store.

Each node is tagged with the features characterizing the systems in which it is relevant. For instance, the system of arithmetic expressions built from features *nat* and *bool* needs the successor inversion lemma (SUCC-INV), and the canonical forms lemmas for booleans and numeric values (BOOL-CANON and NAT-CANON) in addition to the two main theorems.

6.3 Recursive Types

The proofs we have described can all be encoded in TinkerType quite compactly. (Roughly, the overhead introduced by modularization is about half again as long as an ordinary, monolithic proof for a system combining all possible features.) In essence, this is because the systems involved are fundamentally fairly similar.

The introduction of equi-recursive types, on the other hand, has more serious consequences. Firstly, we must confront the issue of type equivalence. Whenever in non-recursive systems we indicated the syntactic equivalence of two types by using the same metavariable for both of them, in systems with recursive types, we

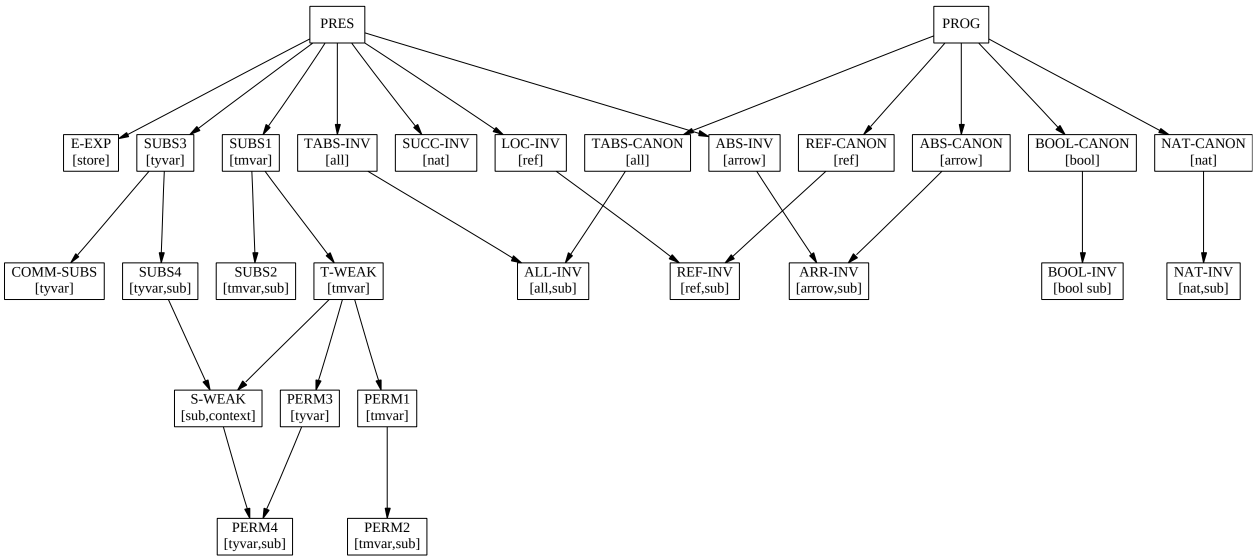

Figure 9: Preservation and Progress theorems and their lemmas

must instead allow the two types to be syntactically distinct but equivalent "modulo unfolding" of recursive types. One example of a clause affected by the new notion of equivalence is the universal type subtyping rule in system kernel $F_{<:}$.

$$\frac{\Gamma, \mathtt{X}\texttt{<:}\mathtt{T} \vdash \mathtt{T}_1 \texttt{<:} \mathtt{T}_2}{\Gamma \vdash \forall \mathtt{X}\texttt{<}\mathtt{T}.\mathtt{T}_1 \texttt{<:} \forall \mathtt{X}\texttt{<}\mathtt{T}.\mathtt{T}_2} \qquad \text{S-ALL } [\mathit{kfsub}]$$

$$\frac{\Gamma, \mathtt{X}\texttt{<:}\mathtt{T} \vdash \mathtt{T}_1 \texttt{<:} \mathtt{T}_2 \qquad \mathtt{T} \equiv \mathtt{T}'}{\Gamma \vdash \forall \mathtt{X}\texttt{<}\mathtt{T}.\mathtt{T}_1 \texttt{<:} \forall \mathtt{X}\texttt{<}\mathtt{T}'.\mathtt{T}_2} \qquad \text{S-ALL } [\mathit{kfsub}, \mathit{ref}]$$

The equivalence relation is defined coinductively as the greatest fixed point of the equivalence rules. Similarly, we define the subtyping relation coinductively. This allows us to inherit most of the subtyping rules from the earlier systems while changing the interpretation of the rules from inductive to coinductive.

To encode these mechanisms, we create a number of new features. Feature *rec* identifies systems with equi-recursive types. The equivalence relation, identified by *eq*, is needed in the kernel $F_{<:}$system to define the universal type subtyping rule. Similarly, we need it in systems with reference cells to define the invariant subtyping rule for type `Ref T`. In the full $F_{<:}$system, on the other hand, the equivalence relation is unnecessary, because it is subsumed by the subtyping relation. These relations are encoded in the feature constraint $rec \Rightarrow (\mathit{ffsub} \wedge \neg ref) \vee eq$.

The features *indsub* and *coindsub* offer a choice between two different versions of subtyping relations. In the presence of recursive types, we must use coinductive subtyping—this is captured by the constraint $rec \wedge sub \Rightarrow coindsub$. Also, any subtyping system must choose between the the inductive or coinductive view: $sub \Rightarrow indsub \vee coindsub$. Finally, these two views are mutually exclusive: $\neg(indsub \wedge coindsub)$.

The inclusion of recursive types results in a significant increase in the size of the repository. First, the introduction of the equivalence judgement necessitates creation of new equivalence rules and various new properties and their proofs. Similarly, new properties are required for the new subtyping relation. Transitivity is one of them: while in inductive subtyping systems, we can explicitly stipulate that subtyping is transitive by including the S-TRANS rule, we may not do so in a coinductive system. (This rule renders a coinductively defined subtyping relation total—cf. ??.)

More severely, the new repository must maintain two parallel, but totally unrelated, hierarchies of proof cases. Any theorem or lemma that used to be proved by induction on subtyping derivations, must basically be re-proved from scratch in the coinductive framework. (Fortunately, the properties of the typing relation remain largely unchanged by the addition of recursive types.)

7 Related Work

The initial inspiration for our work came from the *type system fragments* used by Abadi and Cardelli in their book, *A Theory of Objects* [AC96]. There, the repository consists of a collection of named "fragments" analogous to our components. A system is specified by naming a collection of fragments whose contents are to be concatenated. This is a degenerate instance of our framework, where each component is labeled with a single, distinct feature and where there is no dependency between features, and no static consistency checks are performed. Their book presents a substantial collection of fragments, covering (declarative formulations of) approximately the same range of type systems as the ones described here in Section 5.

Another close relative of our work is Prehofer's *feature-oriented programming* [Pre97]. Like our approach, it includes features and dependencies between them, components with multiple variants, and an assembly process that combines appropriate variants based on a set of requested features. The main difference is the application domain. Our approach focuses on formal systems, and the basic unit of composition is an individual inference rule. Feature-oriented programming is used to assemble objects; its basic unit of composition is a group of related methods. Prehofer introduces an extension of Java with feature support and describes two approaches for compiling it into Java.

The *Hyperspace* project [OT99, TOHS99] proposes a general theory of multi-dimensional separation of concerns. In this work, *units* are atomic entities similar to our clauses. A unit can be related to several *concerns*, which correspond to our features. Concerns are partitioned into orthogonal *dimensions*. *Hyperslices* are composed of units and resemble our components. They can be merged to form *hypermodules* that are similar to systems in our work. This approach is somewhat more abstract than ours (for example, the algorithm for merging hyperslices is taken as a parameter).

Earlier work in the same group promoted a technology called *subject-oriented programming* [HO93]. One of its principal goals was to allow parallel development of classes and provide a composition mechanism to obtain a final system. In this view, classes resemble our components, and their merging is analogous to system assembly. No mechanism corresponding to features is provided.

Aspect-oriented programming [KLM$^+$97, Kic96] starts from the observation that it is sometimes difficult to address certain issues in a program without obscuring its main functionality. These issues, called *aspects*, "cross-cut" the natural decomposition of the main functionality, resulting in small bits of related code strewn across the system. To simplify designing programs with these properties, AOP proposes using conventional *component languages* to implement basic functionality, and special purpose *aspect languages* to deal with the cross-cutting issues. A special process called *weaving* merges programs written in these languages to produce the resulting system. To some extent, we can view our language of features, clauses, and components as a particular aspect language; the component language is whatever language is used to express the contents of clauses.

Another area of related work is monadic techniques for structuring interpreters and compilers [Ste94, LHJ95, Esp95, LH96, HK98, etc.]. The focus here is on modular definition and combination of different aspects of computation (state, exceptions, concurrency, etc.). It is a highly structured approach, using the type system of the metalanguage to control the composition process and focusing on constraints arising from interaction between features. It does not appear easy to extend the monadic approach to typing features in the spirit of the present work. On the other hand, we believe that a monadic style could be used to structure the presentation of the operational semantics of our typed lambda-calculi.

8 Conclusions and Future Work

The TinkerType formalism, its implementation, and our repository of typed lambda-calculi have evolved in parallel over many months. At present, our larger repository contains about 12,000 lines of TinkerType sources, of which roughly 20% is TeX sources in the bodies of clauses, 60% is ML clauses, and 20% is TinkerType proper. From this, we routinely generate about 70 different typecheckers, totaling over 85,000 lines of ML. Maintaining all these checkers by hand would be next to impossible.

Our focus in this paper has been on using TinkerType to define typed lambda-calculi. Of course, its range of potential applications is broader: any family of programs or mathematical definitions that are naturally structured as collections of rules is a candidate. This includes other programming calculi (e.g., Abadi and Cardelli's Object Calculi, process calculi, etc.), compilers, and a wide variety of logics, as well as programs from other domains such as expert systems.

One of the main drawbacks of our approach to formal systems (compared, say, to Pure Type Systems) is that it provides no direct support for "once-and-for-all" proofs of common properties like subject reduction for whole families of formal systems. However, our second repository shows that it may be possible to achieve similar results by using the facilities of our framework to build modular proofs following exactly the same lines as our modular presentations of the systems themselves. Of course, the TinkerType system cannot provide any guarantee that the generated "proofs" are well formed, but there is no obstacle, in principle, to generating them in machine-checkable form and running them through a mechanical proof checker.

It would also be interesting to investigate more structured approaches retaining our basic mechanisms of clause selection by features—finding some happy medium between the flexibility and generality of our framework and the structure and specificity of Pure Type Systems. Recent work by Fiore, Plotkin, and Turi on a general treatment of abstract syntax and variable binding [FPT] offers a crucial first step. Burstall and Goguen's older notion of *institutions* [GB84] may also provide useful guidance.

Acknowledgments

This work was supported by the University of Pennsylvania and by NSF grant CCR-9701826, *Principled Foundations for Programming with Objects.*

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Bar92] Henk Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[BMV95] David Basin, Seán Matthews, and Luca Viganò. A modular presentation of modal logics in a logical framework. In *Proceedings of the Tbilisi Symposium on Language, Logic and Computation*, October 1995.

[Esp95] David Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[FPT] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. Preprint, available through `http://www.dcs.ed.ac.uk/~dt/abstractsyn.ps`.

[GB84] J. A. Goguen and R. M. Burstall. Introducing institutions. In Edmund Clarke and Dexter Kozen, editors, *Logics of Programs: Workshop, Carnegie Mellon University, June 1983*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer-Verlag, New York, NY, 1984.

[Ghe90] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, March 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, MA, 1992.

[HHP92] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS'87.

[HK98] William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the IEEE International Conference on Computer Languages*, 1998.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In Andreas Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, number 28(10) in ACM SIGPLAN Notices, pages 411–428. ACM Press, October 1993.

[Jac99] Bart Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier, 1999.

[Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154, December 1996.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science. Springer-Verlag, June 1997.

[Lan65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101, 158–165, February and March 1965.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[LH96] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, Cambridge, MA, 1996.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science (Asilomar, CA)*, pages 14–23. IEEE Computer Society Press, June 1989.

[Mos92] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[OT99] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, IBM T.J. Watson Research Center, 1999.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988. Available as Ergo Report 88–036, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). Preliminary version in POPL '92.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Jyväskylä, Finland, 9–13 June 1997. Springer.

[SOW97] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, January 1997. Full version in *Theory and Practice of Object Systems, 1998.*

[Ste94] Guy L. Steele, Jr. Building interpreters by composing monads. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press.

[TOHS99] Peri Tarr, Harold L. Ossher, William H. Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimentional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.