Type-safe cast does no harm

Theoretical Pearl

Dimitrios Vytiniotis   Stephanie Weirich

University of Pennsylvania
{dimitriv,sweirich}@cis.upenn.edu

Abstract

Generic functions can specialize their behaviour depending on the types of their arguments, and can even recurse over the structure of the types of their arguments. Such functions can be programmed using *type representations*. Generic functions programmed this way possess certain parametricity properties, which become interesting in the presence of higher-order polymorphism. In this Theoretical Pearl, we give a rigorous roadmap through the proof of parametricity for a calculus with higher-order polymorphism and type representations. We then use parametricity to derive the partial correctness of *type-safe cast*.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types, Polymorphism; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages–Operational Semantics

General Terms Languages, Theory

Keywords Parametricity, higher-order polymorphism, generalized algebraic datatypes, type representations, generic programming, type-safe cast, free theorems

1. Generic programming via type representations

Generic programming refers to the ability to specialize the behaviour of functions based on the *types* of their arguments. There are many tools, libraries, and language extensions that support generic programming, particularly for the Haskell programming language [5, 7, 17, 9, 22, 38, 37]. Although the theory that underlies these mechanisms differs considerably, the common goal of these mechanisms is to eliminate boilerplate code. Examples of generic programs range from very generic equality functions, marshallers, reductions and maps, to application-specific traversals and queries [22], user interface generators [1], XML-inspired transformations [21], and compilers [6].

Representation types [11] is an attractive mechanism for generic programming. The key idea is simple: because polymorphic functions are parametric in Haskell (their behaviour cannot be influenced by the types at which they are instantiated), generic functions dispatch on term arguments that *represent* types.

July 17, 2007

Representation types were originally proposed in the context of type-preserving compilation, but they may be encoded in Haskell in several ways [7, 38, 37]. The most natural implementation of representation types is with *generalized algebraic datatypes* (GADTs) [4, 8, 31, 32], a recent extension to the Glasgow Haskell Compiler (GHC) compiler. [1]

For example, in GHC one can define a GADT for representation types as follows:

```
data R a where
    Rint  :: R Int
    Runit :: R ()
    Rprod :: R a -> R b -> R (a,b)
    Rsum  :: R a -> R b -> R (Either a b)
```

The datatype `R` includes four data constructors: The constructor `Rint` provides a representation for `Int`, hence its type is `R Int`. The constructor `Runit` provides a representation for `()` and has type `R ()`. The constructors `Rprod` and `Rsum` represent products and sums (the latter expressed by Haskell's `Either` datatype). They take as inputs a representation for `a` (of type `R a`), a representation for `b` (of type `R b`), and return representations for `(a,b)` and `Either a b` respectively. The important property of this datatype is that the return type of the constructors is not uniform—`Rint` has type `R Int` whereas `Runit` has type `R ()`. In fact, the type parameter is determined by the data constructor. In contrast, in an ordinary algebraic datatype, all data constructors must return the same type.

A simple example of a generic function is `add`, shown below, that adds together all of the integers that appear in a data structure.

```
add :: R c -> c -> Int
add (Rint)  x = x
add (Runit) x = 0
add (Rprod ra rb) x
  = add ra (fst x) + add rb (snd x)
add (Rsum ra rb) (Left x)  = add ra x
add (Rsum ra rb) (Right x) = add rb x
```

The `add` function may be applied to any argument composed of integers, products, unit, and sums.

```
*> add (Rprod Rint Rint) (1,3)
4
*> add (Rprod Rint (Rprod Runit Rint)) (2, ((), 3))
5
```

Note that in the definition of `add`, the argument `x` is treated as integer, product or sum depending on the clause of the definition. This

[1] `http://www.haskell.org/ghc`

```
cast :: R a -> R b -> Maybe (a -> b)
cast Rint Rint   = Just (\x -> x)
cast Runit Runit = Just (\x -> x)
cast (Rprod ra0 rb0) (Rprod ra0' rb0') =
  do g <- cast ra0 ra0'
     h <- cast rb0 rb0'
     Just (\(a,b) -> (g a, h b))
cast (Rsum ra0 rb0) (Rsum ra0' rb0') =
  do g <- cast ra0 ra0'
     h <- cast rb0 rb0'
     Just (\x -> case x of
                   Left a -> Left (g a)
                   Right b -> Right (h b))
cast _ _ = Nothing
```

Figure 1: `cast`

```
newtype CL f c a d = CL (c (f d a))
unCL (CL e) = e
newtype CR f c a d = CR (c (f a d))
unCR (CR e) = e

gcast :: forall a b c.
         R a -> R b -> Maybe (c a -> c b)
gcast Rint Rint   = Just (\x -> x)
gcast Runit Runit = Just (\x -> x)
gcast (Rprod (ra0 :: R a0)  (rb0  :: R b0))
      (Rprod (ra0':: R a0') (rb0' :: R b0'))
 = do g <- gcast ra0 ra0'
      h <- gcast rb0 rb0'
      let g' :: c (a0, b0)  -> c (a0', b0)
          g' = unCL . g . CL
          h' :: c (a0', b0) -> c (a0', b0')
          h' = unCR . h . CR
      Just (h' . g')
gcast (Rsum ra0 rb0) (Rsum ra0' rb0')
 = do g <- gcast ra0 ra0'
      h <- gcast rb0 rb0'
      Just (unCR . h . CR . unCL . g . CL)
gcast _ _ = Nothing
```

Figure 2: `gcast`

behaviour of the type checker is sound because pattern matching on the representation argument reveals information about the type of x. For example, in the third clause of the definition, the type variable `c` is *refined* to be equal to some `(a,b)` such that `ra :: R a` and `rb :: R b`.

In this paper, we focus on the generic *type-safe* cast function, which compares two different type representations and, if they match, produces a coercion function from one type to the other. Type-safe cast can be used to test, at runtime, whether a value of a given representable type can safely be viewed as a value of a second representable type—even when the two types cannot be shown equal at compile-time. Previously, Weirich [36] defined two different versions of type-safe cast, shown in Figures 1 and 2.[2] To distinguish between these two versions, we call them `cast` and `gcast` respectively.

The first version, `cast`, works by comparing the two representations and then producing a coercion function that takes its argument apart, coerces the subcomponents individually, and then puts it back together. In the case for products and sums, Haskell's monadic syntax for `Maybe` ensures that `cast` returns `Nothing` when one of the recursive calls returns `Nothing`; otherwise `g` and `h` are bound to the underlying coercions.

Alternatively, `gcast` produces a coercion function that never needs to decompose (or even evaluate) its argument—it merely changes its type. The key inspiration is the use of the higher-kinded type argument `c`. This type constructor allows the recursive calls to `gcast` to create a coercion that changes the type of a *part* of its argument. In a recursive call, the instantiation of `c` hides the parts of the type that remain unchanged. To show how this works, the case for products has been decorated with type annotations. In this case we know that the argument has type `c (a0,b0)`. We first produce `g` and `h`, with types `c1 a0 -> c1 a0'` and `c2 b0 -> c2 b0'` respectively for some `c1` and `c2`, by recursively calling `gast` for the sub-representations. We are interested in the particular cases where `c1 a0` can act as `c (a0,b0)`, and `c1 a0'` can act as `c (a0',b0)`. Since Haskell does not support type-level abstractions, we introduce the newtype `CL` (for "cast left"). In particular we let the type checker implicitly unify `c1` with `CL (,) c b0` where `(,)` is the pair constructor. This means that an element of `c1 a0` is the application of the `CL` data constructor to an element of type `c (a0,b0)`. Then `g` returns an element of `CL (,) c b0 a0'`, which is actually an application `CL` to an element of type `c (a0',b0)`. Hence we create `g'` that first wraps an element of `c (a0,b0)` with the `CL` constructor, calls `g` on it, and finally un-wraps the returned `CL (,) c b0 a0'` as the required `c (a0',b0)`, by calling `unCL`. The net effect is that `g'` is a coercion of type `c (a0,b0) -> c (a0',b0)`. For the instantiation of `c2` we introduce the newtype `CR` (for "cast right") and make sure `c2` can be instantiated to `CR (,) c a0'`. The net effect is that `h'` is a coercion of type `c (a0',b0) -> c (a0',b0')`. Composing them is the required conversion. The case for sums is similar but we omit the intermediate type annotations and compose all the intermediate functions directly.

An important difference between the two versions has to do with correctness. When the type comparison succeeds, type-safe cast should behave like an identity function. Informal inspection reveals that both implementations have this property. However in the case of `cast`, it is possible to mess up. In particular, it is type sound to replace the clause for `Rint` with:

```
cast Rint Rint = Just (\x -> 21)
```

However, the type of `gcast` more strongly constrains its implementation. We could not replace the first clause with

```
gcast Rint Rint = Just (\x -> 21)
```

because the type of the returned coercion must be `c Int -> c Int`, not `Int -> Int`. Informally, we can argue that the only coercion function that could be returned *must* be an identity function as `c` is abstract. The only way to produce a result of type `c Int` (discounting divergence) is to use exactly the one that was supplied.[3]

1.1 Contributions

In this pearl, we make the above arguments precise and rigorous. In particular, we show using a *free theorem* [34] that, if `gcast` returns a coercion function then that function must be an identity function. In fact, because we use a free theorem, any function with the type of `gcast` must behave in this manner. To do so, we start with a formalization of the λ-calculus with representation types

[2] These implementations differ slightly from Weirich's pearl, but the essential structure remains the same.

[3] Baars and Sweirstra [5] originally made this observation about the differences between these versions, and concurrently with Cheney and Hinze [7] point out that `gcast` corresponds to *Leibniz equality*.

and higher-order polymorphism, called R$_\omega$ [11] (Section 2.1). We then extend Reynolds's abstraction theorem [30] to this language (Section 2.2). Reynolds's abstraction theorem, also referred to as the "parametricity theorem" [34], asserts that every well-typed expression of the second-order polymorphic λ-calculus (System F) [13, 14] satisfies a particular property directly derivable from its type. After proving a version of the abstraction theorem for R$_\omega$, we show how to apply it to the type of `gcast` to get the desired results (Section 3).

Our broader goal is not just to prove the correctness of `gcast`—there are certainly simpler ways to do so, and there are some limitations in our approach, as we describe in Section 4.4. Instead, our intention is to demonstrate that it is possible to use parametricity and free theorems to reason about generic functions written with representation types. In previous work [33], which was limited to the case of second-order polymorphism, we had difficulty finding free theorems for generic functions that were not trivial. This pearl demonstrates a fruitful example of such reasoning when higher-order polymorphism is present, and encourages the use of variations of this method to reason about other generic functions.

A second goal of this pearl is to explore free theorems for higher-order polymorphism. Our use of these theorems exhibits an intriguing behaviour. Free theorems for types with second-order polymorphism quantify over arbitrary relations but are typically used only with relations that happen to be expressible as functions in the polymorphic λ-calculus. In contrast, we must instantiate free theorems with *non-parametric* functions to get the desired result.

Finally, although the ideas that we use to define parametricity for F$_\omega$ are folklore, they appear in very few sources in the literature. Therefore, an additional contribution of this work is an accessible roadmap to the proof of parametricity for higher-order polymorphism using the technique of syntactic logical relations. Our development is most closely related to the proof of strong normalization of F$_\omega$ by Jean Gallier [12], but we are more explicit about the requirements from the meta-logic and the well-formedness of our definitions. Therefore, we expect our development to be particularly well-suited for mechanical verification in proof assistants, such as Coq[4].

2. Parametricity formalized

In the following, we assume familiarity with higher-order polymorphic λ-calculi, such as the language F$_\omega$ [13]. Our version of F$_\omega$ resembles that of Pierce [27, Ch.30], although there are several differences that we discuss below.

2.1 The R$_\omega$ calculus

We begin with a formal description of the R$_\omega$ calculus. The syntax appears in Figure 3. Kinds include the kind, $\star$, which classifies the types of expressions, and constructor kinds, $\kappa \to \kappa$. The type syntax includes type variables, type constants, type-level applications, and type functions. We treat impredicative polymorphism by introducing an infinite family of universal type constructors $\forall_\kappa$ indexed by kinds. Standard F$_\omega$ polymorphic types can be viewed as applications of some $\forall_\kappa$ constructor to some type-level abstraction. In the rest of the paper we use the following abbreviations:

$$\begin{array}{lcl}
\forall a{:}\kappa\,.\,\tau & \triangleq & \forall_\kappa\;(\lambda a{:}\kappa\,.\,\tau) \\
\sigma_1 \to \sigma_2 & \triangleq & (\to)\;\sigma_1\;\sigma_2 \\
\sigma_1 + \sigma_2 & \triangleq & (+)\;\sigma_1\;\sigma_2 \\
\sigma_1 \times \sigma_2 & \triangleq & (\times)\;\sigma_1\;\sigma_2
\end{array}$$

and associate infix applications of → to the right; for instance $\sigma_1 \to \sigma_2 \to \sigma_3$ means $\sigma_1 \to (\sigma_2 \to \sigma_3)$. Although our syntax does not include constructs for binding lists of type variables, we further write $\forall(a_1{:}\kappa_1)\ldots(a_n{:}\kappa_n)\,.\,\sigma$ to abbreviate $\forall a_1{:}\kappa_1\,.\,\ldots\forall a_n{:}\kappa_n\,.\,\sigma$.

$$\begin{array}{llcl}
\text{Kinds} & \kappa & ::= & \star \mid \kappa \to \kappa \\
\text{Type constants} & \mathcal{K} & ::= & \mathtt{R} \mid \mathtt{()} \mid \mathtt{int} \mid \to \mid \times \mid + \mid \forall_\kappa \\
\text{Types} & \sigma, \tau & ::= & a \mid \mathcal{K} \mid \sigma_1\;\sigma_2 \mid \lambda a{:}\kappa\,.\,\sigma \\
\text{Expressions} & e & ::= & \mathtt{R_{int}} \mid \mathtt{R_{()}} \mid \mathtt{R_\times}\;e_1\;e_2 \mid \mathtt{R_+}\;e_1\;e_2 \\
 & & \mid & \mathtt{typerec}\;e\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\} \\
 & & \mid & \mathtt{fst}\;e \mid \mathtt{snd}\;e \mid (e_1, e_2) \\
 & & \mid & \mathtt{inl}\;e \mid \mathtt{inr}\;e \\
 & & \mid & \mathtt{case}\;e\;\mathtt{of}\;\{x.e_l\;;\;x.e_r\} \\
 & & \mid & \mathtt{()} \mid i \mid x \mid \lambda x\,.\,e \mid e_1\;e_2 \\
\text{Values} & v, w & ::= & \mathtt{R_{int}} \mid \mathtt{R_{()}} \mid \mathtt{R_\times}\;e_1\;e_2 \mid \mathtt{R_+}\;e_1\;e_2 \\
 & & \mid & (e_1, e_2) \mid \mathtt{inl}\;e \mid \mathtt{inr}\;e \\
 & & \mid & \mathtt{()} \mid i \mid \lambda x\,.\,e \\
\text{Environments} & \Gamma & ::= & \cdot \mid \Gamma, a{:}\kappa \mid \Gamma, x{:}\tau
\end{array}$$

Figure 3: Syntax of System R$_\omega$

$$\boxed{e \Downarrow v}$$

$$\frac{e \Downarrow \mathtt{R_{int}} \quad e_{\mathtt{int}} \Downarrow v}{\mathtt{typerec}\;e\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\} \Downarrow v}$$

$$\frac{e \Downarrow \mathtt{R_{()}} \quad e_{()} \Downarrow v}{\mathtt{typerec}\;e\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\} \Downarrow v}$$

$$\frac{\begin{array}{c} e \Downarrow \mathtt{R_\times}\;e_1\;e_2 \\ e_\times\;e_1\;(\mathtt{typerec}\;e_1\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\}) \\ \quad e_2\;(\mathtt{typerec}\;e_2\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\}) \Downarrow v \end{array}}{\mathtt{typerec}\;e\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\} \Downarrow v}$$

$$\frac{\begin{array}{c} e \Downarrow \mathtt{R_+}\;e_1\;e_2 \\ e_+\;e_1\;(\mathtt{typerec}\;e_1\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\}) \\ \quad e_2\;(\mathtt{typerec}\;e_2\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\}) \Downarrow v \end{array}}{\mathtt{typerec}\;e\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\} \Downarrow v}$$

Figure 4: Operational rules for type recursion

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{}{\Gamma \vdash \mathtt{R_{int}} : \mathtt{R}\;\mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{R_{()}} : \mathtt{R}\;\mathtt{()}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{R}\;\sigma_1 \quad \Gamma \vdash e_2 : \mathtt{R}\;\sigma_2}{\Gamma \vdash \mathtt{R_\times}\;e_1\;e_2 : \mathtt{R}\;(\sigma_1, \sigma_2)} \qquad \frac{\Gamma \vdash e_1 : \mathtt{R}\;\sigma_1 \quad \Gamma \vdash e_2 : \mathtt{R}\;\sigma_2}{\Gamma \vdash \mathtt{R_+}\;e_1\;e_2 : \mathtt{R}\;(\sigma_1 + \sigma_2)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \sigma_c : \star \to \star \quad \Gamma \vdash e : \mathtt{R}\;\sigma \\ \Gamma \vdash e_{\mathtt{int}} : \sigma_c\;\mathtt{int} \quad \Gamma \vdash e_{()} : \sigma_c\;\mathtt{()} \\ \Gamma \vdash e_\times : \forall(a{:}\star)(b{:}\star)\,.\,\mathtt{R}\;\;a \to \sigma_c\;a \to \mathtt{R}\;\;b \to \sigma_c\;b \to \sigma_c\;(a \times b) \\ \Gamma \vdash e_+ : \forall(a{:}\star)(b{:}\star)\,.\,\mathtt{R}\;\;a \to \sigma_c\;a \to \mathtt{R}\;\;b \to \sigma_c\;b \to \sigma_c\;(a + b) \end{array}}{\Gamma \vdash \mathtt{typerec}\;e\;\mathtt{of}\;\{e_{\mathtt{int}}\;;\;e_{()}\;;\;e_\times\;;\;e_+\} : \sigma_c\;\sigma}$$

Figure 5: Typing relation—R$_\omega$ specifics

Expressions of the language include the standards of many typed λ-calculi: abstractions, products, sums, integers and unit. To simplify our discussion, we treat type abstractions and type applications implicitly—this omission makes no difference for the metatheory discussed here.

[4] `http://coq.inria.fr`

$$\boxed{\Gamma \vdash \tau : \kappa}$$

$$\frac{(a{:}\kappa) \in \Gamma}{\Gamma \vdash a : \kappa} \qquad \frac{\mathit{kind}(\mathcal{K}) = \kappa}{\Gamma \vdash \mathcal{K} : \kappa}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \to \kappa \quad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \tau_1\ \tau_2 : \kappa} \qquad \frac{\Gamma, a{:}\kappa_1 \vdash \tau : \kappa_2 \quad a\#\Gamma}{\Gamma \vdash \lambda a{:}\kappa_1 . \tau : \kappa_1 \to \kappa_2}$$

$$\begin{array}{lcl|lcl} \mathit{kind}(\to) & = & \star \to \star \to \star & \mathit{kind}(\mathtt{int}) & = & \star \\ \mathit{kind}(\times) & = & \star \to \star \to \star & \mathit{kind}(\mathtt{()}) & = & \star \\ \mathit{kind}(+) & = & \star \to \star \to \star & \mathit{kind}(\mathtt{R}) & = & \star \to \star \\ \mathit{kind}(\forall_\kappa) & = & (\kappa \to \star) \to \star & & & \end{array}$$

Figure 6: Well-formed types

R$_\omega$ includes the type representations $\mathtt{R}_{\mathtt{int}}$, $\mathtt{R}_{()}$, $\mathtt{R}_\times$ and $\mathtt{R}_+$, which must be fully applied to their arguments. The language is terminating, but includes a term `typerec` that can perform primitive recursion on type representations, and includes branches for each possible representation. Programming in this calculus with this primitive recursion operator (and without the syntactic sugar of pattern matching) is somewhat tedious. For completeness, we give the R$_\omega$ implementations of *cast* and *gcast* in Appendix A.

We do not include representations for function or polymorphic types in R$_\omega$. Neither are that useful for generic programming, and the latter significantly changes the semantics of the language: we return to this point in Section 4.2. Another omission from this language is a *uniform* representation, which represents *any* type without specifying exactly what type that is (see our previous work for an example of such a representation [33]).

The operational semantics of the language is standard, so we only present the rules for `typerec` in Figure 4. Essentially `typerec` performs a fold over its type representation argument. We use a big-step formalization for simplicity and a call-by name semantics to maintain a connection to the semantics of Haskell. The syntax of R$_\omega$ values is also shown in Figure 3.

Environments, Γ, contain bindings for type variables $(a{:}\kappa)$ and bindings for term variables $(x{:}\tau)$. We use $\cdot$ for the empty environment, and write $a\#\Gamma$ to mean that a does not appear anywhere in Γ. The judgement $\Gamma \vdash \tau : \kappa$ in Figure 6 states that τ is a well-formed type of kind κ and ensures that all the free type variables of the type τ appear in the environment Γ with correct kinds. The following rule, which is standard in treatments of F$_\omega$, is derivable in our system:

$$\frac{\Gamma, a{:}\kappa \vdash \tau : \star \quad a\#\Gamma}{\Gamma \vdash \forall a{:}\kappa . \tau : \star}$$

The main typing judgement of R$_\omega$ has the form $\Gamma \vdash e : \tau$. The interesting typing rules are the introduction and elimination forms for type representations. These rules appear in Figure 5. The rest of the definition of this typing relation is standard, except that our language is implicitly typed. This means that the standard rule for type abstraction is replaced with a *generalization rule* and the rule for type applications is replaced with an *instantiation rule*, neither of which is syntax-directed.

$$\frac{\Gamma, a{:}\kappa \vdash e : \tau \quad a\#\Gamma}{\Gamma \vdash e : \forall a{:}\kappa . \tau} \qquad \frac{\Gamma \vdash e : \forall a{:}\kappa . \tau \quad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash e : \tau\{\sigma/a\}}$$

$$\boxed{\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa}$$

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \tau \equiv \tau : \kappa}\ \textsc{Refl} \qquad \frac{\Gamma \vdash \tau_2 \equiv \tau_1 : \kappa}{\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa}\ \textsc{Sym}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa \quad \Gamma \vdash \tau_2 \equiv \tau_3 : \kappa}{\Gamma \vdash \tau_1 \equiv \tau_3 : \kappa}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_3 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash \tau_2 \equiv \tau_4 : \kappa_1}{\Gamma \vdash \tau_1\ \tau_2 \equiv \tau_3\ \tau_4 : \kappa_2}\ \textsc{App}$$

$$\frac{\Gamma, a{:}\kappa_1 \vdash \tau_1 \equiv \sigma_1 : \kappa_2 \quad \Gamma \vdash \tau_2 \equiv \sigma_2 : \kappa_2}{\Gamma \vdash (\lambda a{:}\kappa_1 . \tau_1)\ \tau_2 \equiv \sigma_2\{\sigma_1/a\} : \kappa_2}\ \textsc{Beta}$$

$$\frac{\Gamma, a{:}\kappa_1 \vdash \tau_1 \equiv \tau_2 \quad a\#\Gamma}{\Gamma \vdash \lambda a{:}\kappa_1 . \tau_1 \equiv \lambda a{:}\kappa_1 . \tau_2 : \kappa_1 \to \kappa_2}\ \textsc{Abs}$$

Figure 7: Type equivalence

We write $\tau\{\sigma/a\}$ for the capture avoiding substitution of a for σ inside τ. Notably, our typing relation includes the standard conversion rule:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 \equiv \tau_2 : \star}{\Gamma \vdash e : \tau_2}\ \textsc{T-Eq}$$

The judgement $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$ defines type equivalence, as a congruence relation that includes β conversion for types. For completeness we give its definition in Figure 7.

Our type equivalence does not include η conversion, but this is not significant for the rest of the development. Additionally, we implicitly identify α-equivalent types, and treat them as syntactically equal in the rest of the paper.

The presence of the rule T-EQ is important for R$_\omega$, but complicates significantly the formalization of parametricity; a significant part of this paper is devoted to taking care of complications introduced by type equivalence.

2.2 The abstraction theorem

Deriving free theorems relies on first defining an appropriate interpretation of types that classify terms as binary relations between terms and showing that these relations are reflexive. This result is the core of Reynolds's abstraction theorem:

$$\text{If } \cdot \vdash e : \tau \text{ then } (e, e) \in \mathcal{C}\,[\![\cdot \vdash \tau : \star]\!].$$

The definition of the interpretation of types appears in Figure 9, but before we can describe that Figure (and the notation used in the statement of the abstraction theorem), we must define a number of auxiliary concepts.

First, we refer to arbitrary *closed* types of a particular kind with the following predicate:

2.1 Definition [Closed types]: We write $\tau \in \mathtt{ty}(\kappa)$ iff $\cdot \vdash \tau : \kappa$.

Only types of kind $\star$ will be interpreted as term relations. Types of higher kind are interpreted as sets of functions in the meta-logic. To distinguish between R$_\omega$ and meta-logical functions, we use the term *morphism* for the latter. For example, the interpretation of a type of kind $\star \to \star$ should be a set of morphisms taking term relations to appropriate term relations. Additionally, we use greek letters (such as α, β) to represent meta-logical parameters that stand for arbitrary types, in contrast to the latin letters (such as a, b) that we use for R$_\omega$ type variables.

To uniformly classify the interpretation of types of any kind, we define the predicate $\mathtt{GRel}^\kappa$ by induction on the kind κ. This

$$
\begin{array}{llll}
\mathtt{wfGRel}^{\star}(\tau_1,\tau_2) & (r \in \mathtt{GRel}^{\star}) & = & r \in \mathtt{VRel}(\tau_1,\tau_2) \\
\mathtt{wfGRel}^{\kappa_1\to\kappa_2}(\tau_1,\tau_2) & (f \in \mathtt{GRel}^{\kappa_1\to\kappa_2}) & = & \text{for all } \alpha_1,\alpha_2 \in \mathtt{ty}(\kappa_1), \\
 & & & \text{for all } g_\alpha \in \mathtt{GRel}^{\kappa_1}, \mathtt{wfGRel}^{\kappa_1}(\alpha_1,\alpha_2)(g_\alpha) \Longrightarrow \\
 & & & \quad \mathtt{wfGRel}^{\kappa_2}(\tau_1\ \alpha_1, \tau_2\ \alpha_2)(f\ \alpha_1\ \alpha_2\ g_\alpha) \wedge \\
 & & & \quad (\text{for all } \beta_1,\beta_2 \in \mathtt{ty}(\kappa_1), g_\beta \in \mathtt{GRel}^{\kappa_1}, \mathtt{wfGRel}^{\kappa_1}(\beta_1,\beta_2)(g_b) \Longrightarrow \\
 & & & \quad\quad \cdot \vdash \alpha_1 \equiv \beta_1 : \kappa_1 \wedge \cdot \vdash \alpha_2 \equiv \beta_2 : \kappa_1 \Longrightarrow \\
 & & & \quad\quad\quad g_\alpha \equiv_{\kappa_1} g_\beta \Longrightarrow f\ \alpha_1\ \alpha_2\ g_\alpha \equiv_{\kappa_2} f\ \beta_1\ \beta_2\ g_\beta)
\end{array}
$$

$$
\begin{array}{lllll}
(r_\alpha \in \mathtt{GRel}^{\star}) & \equiv_\star & (r_\beta \in \mathtt{GRel}^{\star}) & = & \text{for all } e_1, e_2, (e_1,e_2) \in r_\alpha \Longleftrightarrow (e_1,e_2) \in r_\beta \\
(r_\alpha \in \mathtt{GRel}^{\kappa_1\to\kappa_2}) & \equiv_{\kappa_1\to\kappa_2} & (r_\beta \in \mathtt{GRel}^{\kappa_1\to\kappa_2}) & = & \text{for all } \gamma_1,\gamma_2 \in \mathtt{ty}(\kappa_1), g \in \mathtt{GRel}^{\kappa_1}, \\
 & & & & \quad \mathtt{wfGRel}^{\kappa_1}(\gamma_1,\gamma_2)(g) \Longrightarrow (r_\alpha\ \gamma_1\ \gamma_2\ g)\ \equiv_{\kappa_2}\ (r_\beta\ \gamma_1\ \gamma_2\ g)
\end{array}
$$

Figure 8: Well-formed generalized relations and equality

predicate determines when a particular set is the interpretation of some type of kind κ. In the base case, the elements of GRel are binary term relations, whereas in the higher-kind case, the elements of GRel are morphisms.

2.2 Definition [Generalized relations]: We extend term relations to higher kinds by induction on the kind index:

$$
\begin{array}{lll}
\mathtt{GRel}^{\star} & = & \mathcal{P}(\mathtt{term} \times \mathtt{term}) \\
\mathtt{GRel}^{\kappa_1\to\kappa_2} & = & \mathtt{type} \supset \mathtt{type} \supset \mathtt{GRel}^{\kappa_1} \supset \mathtt{GRel}^{\kappa_2}
\end{array}
$$

The notation $\mathcal{P}(\mathtt{term} \times \mathtt{term})$ stands for the space of binary relations on terms of R_ω, and we use type for the syntactic domain of the types of R_ω. We use $\supset$ for the function space constructor of our meta-logic, to avoid confusion with the $\to$ constructor of R_ω.

Generalized morphisms at higher kinds accept two type arguments that are intended to index the input relation of type $\mathtt{GRel}^{\kappa_1}$. These extra arguments allow elements of $\mathtt{GRel}^{\kappa_1\to\kappa_2}$ to dispatch control depending on types as well as on relational arguments. This flexibility is important for the free theorems about R_ω programs.

At first glance, Definition 2.2 seems strange because it returns the term relation space at kind $\star$, while at higher kinds it returns a particular function space of the meta-logic. These two do not necessarily "type check" with a common type. However, in an expressive enough meta-logic (such as CIC [26] or ZF set theory), such a definition is indeed well-formed, as there exists a type containing both spaces (for example Type in CIC [5], or pure ZF sets in ZF set theory).

The objects of $\mathtt{GRel}^{\kappa}$ are either arbitrary term relations or functions. However, not all such objects are suitable for the interpretation of types, so we refine our definition to pick out particular GRel objects. For this refinement, we impose certain conditions on GRel, which are summarized below:

- First, the relations that are the interpretation of types of kind $\star$ must be between *well-typed closed values*. The types of these values need not be identical.
- Second, morphisms that are the interpretation of types of higher kinds must respect type equivalence classes, that is, although objects of $\mathtt{GRel}^{\kappa_1\to\kappa_2}$ may dispatch control based on the equivalence classes of their type arguments, they must not be able to distinguish different syntactic forms within an equivalence class. We explain this requirement in more detail below.

Before precisely stating these conditions, we first stratify term relations into *value relations* and *computation relations*. This distinction is not theoretically strictly necessary but is common in the literature and exposes the connection between our definitions and the operational semantics of R_ω.

2.3 Definition [Type-indexed value relations]: Assume that τ_1, $\tau_2 \in \mathtt{ty}(\star)$. Then $r \in \mathcal{P}(\mathtt{term} \times \mathtt{term})$ is a *type-indexed value relation*, written $r \in \mathtt{VRel}(\tau_1,\tau_2)$, iff for every e_1, e_2 with $(e_1, e_2) \in r$, e_1 and e_2 are values, $\cdot \vdash e_1 : \tau_1$ and $\cdot \vdash e_2 : \tau_2$.

2.4 Definition [Type-indexed computation relations]: The *computation lifting* of a relation $r \in \mathtt{VRel}(\tau_1,\tau_2)$, written as $\mathcal{C}(r)$, is the set of all (e_1, e_2) such that $\cdot \vdash e_1 : \tau_1$, $\cdot \vdash e_2 : \tau_2$ and $e_1 \Downarrow v_1$, $e_2 \Downarrow v_2$, and $(v_1, v_2) \in r$.

Note that because of rule T-EQ, we can view value and computation relations as being indexed by equivalence classes of types: if $\cdot \vdash \tau_1 \equiv \tau_1' : \star$ and $\cdot \vdash \tau_2 \equiv \tau_2' : \star$, then $r \in \mathtt{VRel}(\tau_1,\tau_2)$ iff $r \in \mathtt{VRel}(\tau_1', \tau_2')$.

Now we may state the conditions on $\mathtt{GRel}^{\kappa}$ objects that make them appropriate for use as type interpretations. In Figure 8 we define *well-formed generalized relations*, a type-indexed predicate on GRel. The motivation behind this definition of the wfGRel predicate in Figure 8 is the proof of a theorem which states that the interpretation of types respects type equivalence (Coherence, Theorem 2.16). This predicate, written $\mathtt{wfGRel}^{\kappa}(\tau_1,\tau_2)(\cdot)$, is defined for objects of $\mathtt{GRel}^{\kappa}$ by induction on the kind κ. We define this predicate mutually with equality on generalized relations. Equality on generalized relations is also indexed by kinds; for any two $r_1, r_2 \in \mathtt{GRel}^{\kappa}$, the proposition $r_1 \equiv_\kappa r_2$ asserts that the two generalized relations are extensionally equal. We use $\Longrightarrow$ and $\wedge$ for meta-logical implication and conjunction, respectively.

At kind $\star$, $\mathtt{wfGRel}^{\star}(\tau_1,\tau_2)(r)$ checks that r is a value relation indexed by types τ_1 and τ_2. At the higher kind $\kappa_1 \to \kappa_2$ we require a few conditions on f. First, if f is applied to two type arguments and an appropriate well-formed GRel indexed by these types, then the result must also be well-formed. Second, for any equivalent types $\cdot \vdash \beta_1 \equiv \alpha_1 : \kappa_1$ and $\cdot \vdash \beta_2 \equiv \alpha_2 : \kappa_1$ and equivalent well-formed relations g_α and g_β indexed by these types, the results $f\ \ \alpha_1\ \ \alpha_2\ \ g_\alpha$ and $f\ \ \beta_1\ \ \beta_2\ \ g_\beta$ must also be equal. This condition asserts that objects that satisfy wfGRel at higher kinds *respect* the type equivalence classes of their type arguments.

Extensional equality between generalized relations asserts that at kind $\star$ the two relation arguments denote the same set, whereas at higher kinds it asserts that the relation arguments return equal results, when given the same argument g which must satisfy the wfGRel predicate. Note that the only dependency of $\equiv_\kappa$ on wfGRel is exactly this applicative test. Dropping the requirement that g be a wfGRel produces a definition that is not suitable for our purposes, as we discuss in the proof of Coherence, Theorem 2.16.

Generalized relation equality is reflexive, symmetric, and transitive, and hence is an equivalence relation. All properties follow from simple induction on the kind κ, and we state the reflexivity property, which will be used later.

[5] One can find a Coq definition of GRel and other relevant definitions in Appendix B.

2.5 Lemma [Reflexivity of $\equiv_\kappa$]: Let $r \in \mathtt{GRel}^\kappa$. Then $r \equiv_\kappa r$.

Additionally, the $\mathtt{wfGRel}$ predicate is indexed by equivalence classes of types.

2.6 Lemma [$\mathtt{wfGRel}$ respects type equivalence classes]: Assume that $\cdot \vdash \tau_1 \equiv \tau_2 : \kappa$, $\cdot \vdash \sigma_1 \equiv \sigma_2 : \kappa$, and $r \in \mathtt{GRel}^\kappa$. If $\mathtt{wfGRel}^\kappa(\tau_1,\tau_2)(r)$ then $\mathtt{wfGRel}^\kappa(\sigma_1,\sigma_2)(r)$.

We turn now to the key to the abstraction theorem, the interpretation of R_ω types as relations between closed terms. This interpretation makes use of a *substitution* δ from type variables to triples: pairs of types and a value relation.

2.7 Definition [Substitution kind checks in environment]: We say that a substitution δ *kind checks in an environment* Γ, and write $\delta \in \mathtt{Subst}_\Gamma$, when for every $(a{:}\kappa) \in \Gamma$, it is $\delta(a) = (\tau_1, \tau_2, r)$ with $r \in \mathtt{GRel}^\kappa$. And conversely, for every $a \in dom(\delta)$, $(a{:}\kappa) \in \Gamma$ for some κ.

We project the individual mappings from an environment with the notations $\delta^1(a) = \tau_1$, $\delta^2(a) = \tau_2$, and $\delta[a] = r$. We define $\delta^1\tau$ and $\delta^2\tau$ to be the extension of δ^1 and δ^2 to types applied to the type τ. We write $dom(\delta)$ for the domain of the substitution, that is, the subset of all type variables on which δ is not the identity. We use $\cdot$ for the identity-everywhere substitution, and write $\delta, a \mapsto (\tau_1, \tau_2, r)$ for the extension of δ that maps a to (τ_1, τ_2, r) and require that $a \notin dom(\delta)$.

The interpretation of R_ω types is shown in Figure 9 and is defined inductively over the structure of well-formedness derivations for types. The interpretation function $\llbracket\cdot\rrbracket_\cdot$ accepts a derivation $\Gamma \vdash \tau : \kappa$, and a substitution $\delta \in \mathtt{Subst}_\Gamma$ and returns a generalized relation at kind κ. Hence the meta-logical type of $\llbracket\Gamma \vdash \tau : \kappa\rrbracket$ is $\mathtt{Subst}_\Gamma \supset \mathtt{GRel}^\kappa$. We write the δ argument as a subscript to $\llbracket\Gamma \vdash \tau : \kappa\rrbracket$. When τ is a type variable a we project the relation component out of $\delta(a)$. In the case where τ is a constructor $\mathcal{K}$ we call the auxiliary function $\llbracket\mathcal{K}\rrbracket$, to which we return shortly. For an application $\tau_1\ \tau_2$ we apply the interpretation of τ_1 to appropriate type arguments and the interpretation of τ_2. In the definition we assume that $\Gamma \vdash \tau_1 : \kappa_1 \to \kappa$ is the immediate subderivation of $\Gamma \vdash \tau_1\ \tau_2 : \kappa$. Since $\llbracket\Gamma \vdash \tau_1 : \kappa_1 \to \kappa\rrbracket_\delta$ is a generalized morphism of higher kind, according to Definition 2.2, we must apply it to $\delta^1\tau_2$ and $\delta^2\tau_2$ before applying it to $\llbracket\Gamma \vdash \tau_2 : \kappa_1\rrbracket_\delta$. Type-level λ-abstractions are interpreted as abstractions in the meta-logic. We use λ and $\mapsto$ for meta-logic abstractions. Interpretations of type abstractions first abstract two types, α and β, a generalized relation r, and interpret the body of the function in the extended substitution map $\delta, a \mapsto (\alpha, \beta, r)$ that maps the previously bound variable a to the new triple. Finally, confirming that $\llbracket\Gamma \vdash \tau : \kappa\rrbracket_\delta \in \mathtt{GRel}^\kappa$ is straightforward using the fact that $\delta \in \mathtt{Subst}_\Gamma$.

The interpretation $\llbracket\mathcal{K}\rrbracket$ gives the relation that corresponds to constructor $\mathcal{K}$. For integer and unit types, $\llbracket\mathtt{int}\rrbracket$ and $\llbracket\mathtt{()}\rrbracket$ give the identity value relations respectively on $\mathtt{int}$ and $\mathtt{()}$. The operation $\llbracket\to\rrbracket$ lifts two relations r_1 and r_2 to a new relation between functions that send related arguments in r_1 to related results in r_2. The operation $\llbracket\times\rrbracket$ lifts two relations r_1 and r_2 to a relation between products such that the first components of the products belong in r_1, and the second in r_2. The operation $\llbracket+\rrbracket$ on relations r_1 and r_2 consists of all the pairs of left injections between elements of r_1 and right injections between elements of r_2. Because sums and products are call-by-name, their subcomponents must come from the computation lifting of the value relations. For the $\forall_\kappa$ constructor, since its type is $(\kappa \to \star) \to \star$ we define $\llbracket\forall_\kappa\rrbracket$ to be a morphism that, given a $\mathtt{GRel}^{\kappa\to\star}$ argument f, returns the intersection over all r that are well-formed generalized relations (hence the requirement $\mathtt{wfGRel}^\kappa(\beta_1,\beta_2)(r)$) of the applications of f to r. The requirement that $\mathtt{wfGRel}^\kappa(\beta_1,\beta_2)(r)$ and $\beta_1, \beta_2 \in \mathtt{ty}(\kappa)$ is necessary in order to show that the interpretation of the $\forall_\kappa$ constructor is indeed a well-formed generalized relation (Lemma 2.9).

$$
\begin{array}{lcl}
\mathcal{R} & \in & \mathtt{GRel}^{\star\to\star} \\
\\
\mathcal{R} & = & \lambda\alpha, \beta, r \in \mathtt{GRel}^\star \mapsto \\
 & & \{(\mathtt{R}_{\mathtt{int}}, \mathtt{R}_{\mathtt{int}}) \mid r \equiv_\star \llbracket\mathtt{int}\rrbracket \wedge \cdot \vdash \alpha \equiv \beta \equiv \mathtt{int} : \star\} \\
 & \cup & \{(\mathtt{R}_{()}, \mathtt{R}_{()}) \mid r \equiv_\star \llbracket\mathtt{()}\rrbracket \wedge \cdot \vdash \alpha \equiv \beta \equiv \mathtt{()} : \star\} \\
 & \cup & \{(\mathtt{R}_\times\ e_a^1\ e_b^1, \mathtt{R}_\times\ e_a^2\ e_b^2)\ \mid \\
 & & \quad \exists\tau_a^1, \tau_a^2 \in \mathtt{ty}(\star), r_a \in \mathtt{GRel}^\star, \mathtt{wfGRel}^\star(\tau_a^1, \tau_a^2)(r_a)\ \wedge \\
 & & \quad \exists\tau_b^1, \tau_b^2 \in \mathtt{ty}(\star), r_b \in \mathtt{GRel}^\star, \mathtt{wfGRel}^\star(\tau_b^1, \tau_b^2)(r_b)\ \wedge \\
 & & \qquad r \equiv_\star \llbracket\times\rrbracket\ \tau_a^1\ \tau_a^2\ r_a\ \tau_b^1\ \tau_b^2\ r_b\ \wedge \\
 & & \qquad \cdot \vdash \alpha \equiv \tau_a^1 \times \tau_b^1 : \star \wedge \cdot \vdash \beta \equiv \tau_a^2 \times \tau_b^2 : \star\ \wedge \\
 & & \qquad (e_a^1, e_a^2) \in \mathcal{C}(\mathcal{R}\ \tau_a^1\ \tau_a^2\ r_a) \wedge (e_b^1, e_b^2) \in \mathcal{C}(\mathcal{R}\ \tau_b^1\ \tau_b^2\ r_b)\} \\
 & \cup & \{(\mathtt{R}_+\ e_a^1\ e_b^1, \mathtt{R}_+\ e_a^2\ e_b^2)\ \mid \\
 & & \quad \exists\tau_a^1, \tau_a^2 \in \mathtt{ty}(\star), r_a \in \mathtt{GRel}^\star, \mathtt{wfGRel}^\star(\tau_a^1, \tau_a^2)(r_a)\ \wedge \\
 & & \quad \exists\tau_b^1, \tau_b^2 \in \mathtt{ty}(\star), r_b \in \mathtt{GRel}^\star, \mathtt{wfGRel}^\star(\tau_b^1, \tau_b^2)(r_b)\ \wedge \\
 & & \qquad r \equiv_\star \llbracket+\rrbracket\ \tau_a^1\ \tau_a^2\ r_a\ \tau_b^1\ \tau_b^2\ r_b\ \wedge \\
 & & \qquad \cdot \vdash \alpha \equiv \tau_a^1 + \tau_b^1 : \star \wedge \cdot \vdash \beta \equiv \tau_a^2 + \tau_b^2 : \star\ \wedge \\
 & & \qquad (e_a^1, e_a^2) \in \mathcal{C}(\mathcal{R}\ \tau_a^1\ \tau_a^2\ r_a) \wedge (e_b^1, e_b^2) \in \mathcal{C}(\mathcal{R}\ \tau_b^1\ \tau_b^2\ r_b)\}
\end{array}
$$

Figure 11: Representation type interpretation

For the case of representation types $\mathtt{R}$, the definition relies on an auxiliary morphism $\mathcal{R}$, defined by induction on the size of the β-normal form of its type arguments, and shown in Figure 11. The interesting property about this definition is that it imposes requirements on the relational argument r in every case of the definition. For example, in the first clause of the definition of $\mathcal{R}\ \tau_1\ \tau_2\ r$, the case for integer representations, r is required to be equal to $\llbracket\mathtt{int}\rrbracket$, and consequently τ_1 and τ_2 must be equivalent to $\mathtt{int}$. In the case for unit representations, r is required to be equal to $\llbracket\mathtt{()}\rrbracket$ and τ_1, τ_2 equivalent to $\mathtt{()}$. In the case for products, r is required to be some product of relations, and in the case for sums, r is required to be some sum of relations. In general, r must be equal to the interpretation of the argument types τ_1 and τ_2 in the empty environment, which themselves *must be equivalent* to each other.

2.8 Lemma: Assume that $\tau_1, \tau_2 \in \mathtt{ty}(\star)$, and $\mathtt{wfGRel}^\star(\tau_1, \tau_2)(r)$. If $\mathcal{R}\ \tau_1\ \tau_2\ r \neq \emptyset$ then $\cdot \vdash \tau_1 \equiv \tau_2 : \star$ and $r \equiv_\star \llbracket\cdot \vdash \tau_1 : \star\rrbracket_\cdot$.

Importantly, the interpretation of any constructor $\mathcal{K}$, including $\mathcal{R}$, not only is an element of $\mathtt{GRel}^{kind(\mathcal{K})}$, but satisfies the conditions of well-formed generalized relations.

2.9 Lemma [Constructor interpretation is well-formed]:

$$\mathtt{wfGRel}^{kind(\mathcal{K})}(\mathcal{K}, \mathcal{K})(\llbracket\mathcal{K}\rrbracket)$$

PROOF. The only interesting case is the one for $\forall_\kappa$, which we show below. We need to show that

$$\mathtt{wfGRel}^{(\kappa\to\star)\to\star}(\forall_\kappa, \forall_\kappa)(\llbracket\forall_\kappa\rrbracket)$$

Let us fix α_1, $\alpha_2 \in \mathtt{ty}(\kappa \to \star)$, and a generalized relation $g_\alpha \in \mathtt{GRel}^{\kappa\to\star}$, with $\mathtt{wfGRel}^{\kappa\to\star}(\alpha_1, \alpha_2)(g_\alpha)$, Then we know that

$$
\begin{array}{rcl}
\llbracket\forall_\kappa\rrbracket\ \ \alpha_1\ \alpha_2\ g_\alpha & = & \{(v_1, v_2) \mid \cdot \vdash v_{1,2} : \forall_\kappa\ \alpha_{1,2}\ \wedge \\
 & & \quad \text{for all } \gamma_1, \gamma_2 \in \mathtt{ty}(\kappa), r \in \mathtt{GRel}^\kappa, \\
 & & \quad \mathtt{wfGRel}^\kappa(\gamma_1, \gamma_2)(r) \Longrightarrow \\
 & & \qquad (v_1, v_2) \in (g_\alpha\ \gamma_1\ \gamma_2\ r)\}
\end{array}
$$

which belongs in $\mathtt{wfGRel}^\star(\forall_\kappa\ \alpha_1, \forall_\kappa\ \alpha_2)$ since it is a relation between values of the correct types. Additionally, we need to show that $\forall_\kappa$ can only distinguish between equivalence classes of its type arguments. For this fix β_1, β_2 in $\mathtt{ty}(\kappa \to \star)$, and $g_\beta \in \mathtt{GRel}^{\kappa\to\star}$, with $\mathtt{wfGRel}^{\kappa\to\star}(\alpha_1, \alpha_2)(g_\beta)$. Assume that $\cdot \vdash \alpha_1 \equiv \beta_1 : \kappa \to \star$,

$$
\begin{array}{lcl}
\llbracket \Gamma \vdash \tau : \kappa \rrbracket & \in & \mathtt{Subst}_\Gamma \supset \mathtt{GRel}^\kappa \\
\\
\llbracket \Gamma \vdash a : \kappa \rrbracket_\delta & = & \delta[a] \\
\llbracket \Gamma \vdash \mathcal{K} : \kappa \rrbracket_\delta & = & \llbracket \mathcal{K} \rrbracket \\
\llbracket \Gamma \vdash \tau_1\ \tau_2 : \kappa \rrbracket_\delta & = & \llbracket \Gamma \vdash \tau_1 : \kappa_1 \to \kappa \rrbracket_\delta\ \delta^1\tau_2\ \delta^2\tau_2\ \llbracket \Gamma \vdash \tau_2 : \kappa_1 \rrbracket_\delta \\
 & & \text{for the unique } \kappa_1 \text{ such that } \Gamma \vdash \tau_1 : \kappa_1 \to \kappa \text{ and } \Gamma \vdash \tau_2 : \kappa_1 \\
\llbracket \Gamma \vdash \lambda a{:}\kappa_1 . \tau : \kappa_1 \to \kappa_2 \rrbracket_\delta & = & \lambda \alpha, \beta, r \in \mathtt{GRel}^{\kappa_1} \mapsto \llbracket \Gamma, a{:}\kappa_1 \vdash \tau : \kappa_2 \rrbracket_{\delta, a \mapsto (\alpha,\beta,r)} \\
 & & \text{where } a \# \Gamma
\end{array}
$$

Figure 9: Relational interpretation of R_ω

$$
\begin{array}{lcl}
\llbracket \mathcal{K} \rrbracket & \in & \mathtt{GRel}^{kind(\mathcal{K})} \\
\\
\llbracket \mathtt{int} \rrbracket & = & \{(i,i) \mid \text{ for all } i\} \\
\llbracket \mathtt{()} \rrbracket & = & \{(\mathtt{()},\mathtt{()})\} \\
\llbracket \to \rrbracket & = & \lambda \alpha_1,\ \alpha_2,\ r_1 \in \mathtt{GRel}^\star, \lambda \beta_1,\ \beta_2,\ r_2 \in \mathtt{GRel}^\star \mapsto \\
 & & \quad \{(v_1,v_2) \mid (\cdot \vdash v_1 : \alpha_1 \to \beta_1) \wedge (\cdot \vdash v_2 : \alpha_2 \to \beta_2) \wedge \text{ for all } (e_1',e_2') \in \mathcal{C}(r_1),\ (v_1\ e_1', v_2\ e_2') \in \mathcal{C}(r_2)\} \\
\llbracket \times \rrbracket & = & \lambda \alpha_1,\ \alpha_2,\ r_1 \in \mathtt{GRel}^\star, \lambda \beta_1,\ \beta_2,\ r_2 \in \mathtt{GRel}^\star \mapsto \\
 & & \quad \{(v_1,v_2) \mid (\mathtt{fst}\ \ v_1, \mathtt{fst}\ \ v_2) \in \mathcal{C}(r_1)\} \cap \{(v_1,v_2) \mid (\mathtt{snd}\ \ v_1, \mathtt{snd}\ \ v_2) \in \mathcal{C}(r_2)\} \\
\llbracket + \rrbracket & = & \lambda \alpha_1,\ \alpha_2,\ r_1 \in \mathtt{GRel}^\star, \lambda \beta_1,\ \beta_2,\ r_2 \in \mathtt{GRel}^\star \mapsto \\
 & & \quad \{(\mathtt{inl}\ e_1, \mathtt{inl}\ e_2) \mid (e_1,e_2) \in \mathcal{C}(r_1)\} \cup \{(\mathtt{inr}\ e_1, \mathtt{inr}\ e_2) \mid (e_1,e_2) \in \mathcal{C}(r_2)\} \\
\llbracket \forall_\kappa \rrbracket & = & \lambda \alpha_1,\ \alpha_2,\ f \in \mathtt{GRel}^{\kappa \to \star} \mapsto \\
 & & \quad \{(v_1,v_2) \mid (\cdot \vdash v_1 : \forall_\kappa\ \alpha_1) \wedge (\cdot \vdash v_2 : \forall_\kappa\ \alpha_2) \wedge \\
 & & \qquad \text{for all } \beta_1, \beta_2 \in \mathtt{ty}(\kappa), r \in \mathtt{GRel}^\kappa, \mathtt{wfGRel}^\kappa(\beta_1,\beta_2)(r) \Longrightarrow (v_1,v_2) \in (f\ \beta_1\ \beta_2\ r)\} \\
\llbracket \mathtt{R} \rrbracket & = & \mathcal{R}\ \ \text{(see Figure 11)}
\end{array}
$$

Figure 10: Operations of type constructors on relations

$\cdot \vdash \alpha_2 \equiv \beta_2 : \kappa \to \star$, and $g_\alpha \equiv_{\kappa \to \star} g_\beta$. Then we know that:

$$
\begin{array}{lcl}
\llbracket \forall_\kappa \rrbracket\ \beta_1\ \beta_2\ g_\beta & = & \{(v_1,v_2) \mid \cdot \vdash v_{1,2} : \forall_\kappa\ \beta_{1,2} \wedge \\
 & & \quad \text{for all } \gamma_1, \gamma_2 \in \mathtt{ty}(\kappa), r \in \mathtt{GRel}^\kappa, \\
 & & \quad \mathtt{wfGRel}^\kappa(\gamma_1,\gamma_2)(r) \Longrightarrow \\
 & & \qquad (v_1,v_2) \in (g_\beta\ \gamma_1\ \gamma_2\ r)\}
\end{array}
$$

We need to show that

$$\llbracket \forall_\kappa \rrbracket\ \alpha_1\ \alpha_2\ g_\alpha \equiv_\star \llbracket \forall_\kappa \rrbracket\ \beta_1\ \beta_2\ g_\beta$$

To finish the case, using rule T-EQ to take care of the typing requirements, it is enough to show that, for any γ_1, γ_2 in $\mathtt{ty}(\kappa)$, any r with $\mathtt{wfGRel}^\kappa(\gamma_1,\gamma_2)(r)$, it is:

$$g_\alpha\ \gamma_1\ \gamma_2\ r \equiv_\star g_\beta\ \gamma_1\ \gamma_2\ r$$

But this follows from reflexivity of $\equiv_\kappa$, Lemma 2.5, and the fact that g_α and g_β are well-formed. □

Generalizing Lemma 2.9, we wish to show that the interpretation of any type is a well-formed generalized relation (see Lemma 2.13 below). To show this we need to strengthen the condition $\delta \in \mathtt{Subst}_\Gamma$ to force δ to map type variables to *well-formed* generalized relations.

2.10 Definition [Environment respecting substitution]: We write $\delta \vDash \Gamma$ iff $\delta \in \mathtt{Subst}_\Gamma$ and moreover, for every $a \mapsto (\tau_1, \tau_2, r)$, such that $(a : \kappa) \in \Gamma$ it is the case that $\cdot \vdash \tau_1 : \kappa$, $\cdot \vdash \tau_2 : \kappa$ and $\mathtt{wfGRel}^\kappa(\tau_1,\tau_2)(r)$.

Given equal substitutions, the interpretation of types gives equivalent results.

2.11 Definition [Equal substitutions]: Assume that $\delta_a \vDash \Gamma$, $\delta_b \vDash \Gamma$. Then we write $\delta_a \equiv \delta_b$ iff for every $(a{:}\kappa) \in \Gamma$, it is the case that $a \mapsto (\tau_1, \tau_2, r) \in \delta_a$, $a \mapsto (\sigma_1, \sigma_2, s) \in \delta_b$ and $\cdot \vdash \tau_1 \equiv \sigma_1 : \kappa$, $\cdot \vdash \tau_2 \equiv \sigma_2 : \kappa$ and $r \equiv_\kappa s$.

2.12 Lemma: If $\Gamma \vdash \tau : \kappa$ and $\delta_a \vDash \Gamma$, $\delta_b \vDash \Gamma$ and $\delta_a \equiv \delta_b$, it is the case that

$$\llbracket \Gamma \vdash \tau : \kappa \rrbracket_{\delta_a} \equiv_\kappa \llbracket \Gamma \vdash \tau : \kappa \rrbracket_{\delta_b}$$

2.13 Lemma [Type interpretation is well-formed]: Assume that $\Gamma \vdash \tau : \kappa$ and $\delta \vDash \Gamma$. Then:

$$\mathtt{wfGRel}^\kappa(\delta^1\tau, \delta^2\tau)(\llbracket \Gamma \vdash \tau : \kappa \rrbracket_\delta)$$

PROOF. Straightforward induction over the type well-formedness derivations, appealing to Lemma 2.9. The only interesting case is the case for type abstractions, which follows from Lemma 2.12 and Lemma 2.6. □

The interpretation of types supports weakening:

2.14 Lemma [Weakening]: Assume that $\Gamma \vdash \tau : \kappa$, $\delta \vDash \Gamma$, $a \# \Gamma$, $\tau_1, \tau_2 \in \mathtt{ty}(\kappa_a)$, and $\mathtt{wfGRel}^{\kappa_a}(\tau_1,\tau_2)(r)$. Then:

$$\llbracket \Gamma, a{:}\kappa_a \vdash \tau : \kappa \rrbracket_{\delta, a \mapsto (\tau_1,\tau_2,r)} \equiv_\kappa \llbracket \Gamma \vdash \tau : \kappa \rrbracket_\delta$$

Furthermore, the interpretation of types is compositional, in the sense that the interpretation of a type depends on the interpretation of its sub-terms.

2.15 Lemma [Compositionality]: If $\delta \vDash \Gamma$, $\Gamma, a{:}\kappa_a \vdash \tau : \kappa$, $\Gamma \vdash \tau_a : \kappa_a$, and $r_a = \llbracket \Gamma \vdash \tau_a : \kappa_a \rrbracket_\delta$ then

$$\llbracket \Gamma, a{:}\kappa_a \vdash \tau : \kappa \rrbracket_{\delta, a \mapsto (\delta^1\tau_a, \delta^2\tau_a, r_a)} \equiv_\kappa \llbracket \Gamma \vdash \tau\{\tau_a/a\} : \kappa \rrbracket_\delta$$

The proof of compositionality depends on the fact that type interpretations are well formed relations (Lemma 2.13). Finally, the interpretation of types respects the equivalence classes of types.

2.16 Theorem [Coherence]: If $\Gamma \vdash \tau_1 : \kappa$, $\delta \vDash \Gamma$, and $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$, then

$$\llbracket \Gamma \vdash \tau_1 : \kappa \rrbracket_\delta \equiv_\kappa \llbracket \Gamma \vdash \tau_2 : \kappa \rrbracket_\delta$$

PROOF. The proof can proceed by induction on derivations of $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$. The case for rule BETA follows by appealing to Lemma 2.15, and the cases for rules APP and ABS we give below. The rest of the cases are straightforward.

- Case APP. In this case we have that $\Gamma \vdash \tau_1\ \tau_2 \equiv \tau_3\ \tau_4 : \kappa_2$ given that $\Gamma \vdash \tau_1 \equiv \tau_3 : \kappa_1 \to \kappa_2$ and $\Gamma \vdash \tau_2 \equiv \tau_4 : \kappa_1$. It is easy to show as well that $\Gamma \vdash \tau_{1,3} : \kappa_1 \to \kappa_2$ and $\Gamma \vdash \tau_{2,4} : \kappa_1$. We need to show that
$$\llbracket \Gamma \vdash \tau_1\ \tau_3 : \kappa_2 \rrbracket_\delta \equiv_{\kappa_2} \llbracket \Gamma \vdash \tau_2\ \tau_4 : \kappa_2 \rrbracket_\delta$$
Let
$$\begin{array}{rclrcl} r_1 & = & \llbracket \Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2 \rrbracket_\delta & r_2 & = & \llbracket \Gamma \vdash \tau_2 : \kappa_1 \rrbracket_\delta \\ r_3 & = & \llbracket \Gamma \vdash \tau_3 : \kappa_1 \to \kappa_2 \rrbracket_\delta & r_4 & = & \llbracket \Gamma \vdash \tau_4 : \kappa_1 \rrbracket_\delta \end{array}$$
We know by induction hypothesis that $r_1 \equiv_{\kappa_1 \to \kappa_2} r_3$ and $r_2 \equiv_{\kappa_1} r_4$. By Lemma 2.13, we have that:
$$\begin{array}{ll} \mathtt{wfGRel}^{\kappa_1 \to \kappa_2}(\delta^1\tau_1, \delta^2\tau_1)(r_1) & \mathtt{wfGRel}^{\kappa_1}(\delta^1\tau_2, \delta^2\tau_2)(r_2) \\ \mathtt{wfGRel}^{\kappa_1 \to \kappa_2}(\delta^1\tau_3, \delta^2\tau_3)(r_3) & \mathtt{wfGRel}^{\kappa_1}(\delta^1\tau_4, \delta^2\tau_4)(r_4) \end{array}$$
Finally it is not hard to show that $\cdot \vdash \delta^1\tau_2 \equiv \delta^1\tau_4 : \kappa_1$ and $\cdot \vdash \delta^2\tau_2 \equiv \delta^2\tau_4 : \kappa_1$. Hence, by the properties of well-formed relations, and our definition of equivalence, we can show that
$$r_1\ \delta^1\tau_2\ \delta^2\tau_2\ r_2 \equiv_{\kappa_2} r_3\ \delta^1\tau_4\ \delta^2\tau_4\ r_4$$
which finishes the case.
- Case ABS. Here we have that
$$\Gamma \vdash \lambda a{:}\kappa_1 . \tau_1 \equiv \lambda a{:}\kappa_2 . \tau_2 : \kappa_1 \to \kappa_2$$
given that $\Gamma, a{:}\kappa_1 \vdash \tau_1 \equiv \tau_2 : \kappa_2$. To show the required result let us pick σ_1, σ_2 in $\mathtt{ty}(\kappa_1)$, $g \in \mathtt{GRel}^{\kappa_1}$, with $\mathtt{wfGRel}^{\kappa_1}(\sigma_1, \sigma_2)(g)$. Then for $\delta_a = \delta, a \mapsto (\sigma_1, \sigma_2, g)$, it is $\delta_a \vDash \Gamma, (a{:}\kappa_1)$, and hence by induction hypothesis we get:
$$\llbracket \Gamma, a{:}\kappa_1 \vdash \tau_1 : \kappa_2 \rrbracket_{\delta_a} \equiv_{\kappa_2} \llbracket \Gamma, a{:}\kappa_1 \vdash \tau_2 : \kappa_2 \rrbracket_{\delta_a}$$
and the case is finished. As a side note, the important condition that $\mathtt{wfGRel}^{\kappa_1}(\sigma_1, \sigma_2)(g)$ allows us to show that $\delta_a \vDash \Gamma, (a{:}\kappa_1)$ and therefore enables the use of the induction hypothesis. If $\equiv_{\kappa_1 \to \kappa_2}$ tested against *any possible* $g \in \mathtt{GRel}^{\kappa_1}$ that would no longer be true, and hence the case could not be proved.

□

With the above definitions and properties, we may now state the abstraction theorem.

2.17 Theorem [Abstraction theorem for R$_\omega$]: Assume $\cdot \vdash e : \tau$. Then $(e, e) \in \mathcal{C}\llbracket \cdot \vdash \tau : \star \rrbracket_\cdot$.

To account for open terms, the theorem must be generalized slightly, in the standard manner. The proof then proceeds by induction on the typing derivation, with an inner induction for the case of `typerec` expressions. It relies on Coherence (Theorem 2.16) for the case of rule T-EQ, and on Compositionality (Lemma 2.15) for the case of the instantiation rule.

Incidentally, this statement of the abstraction theorem shows that all well-typed expressions of R$_\omega$ terminate. All such expressions belong in computation relations, which include only terms that reduce to values. Moreover, since these values are well-typed, the abstraction theorem also proves type soundness.

As a warm-up exercise, we next show how we can use the abstraction theorem to reason about programs using their types. The following is a free theorem about an F$_\omega$ type.

2.18 Lemma [Free theorem for $\forall c{:}\star \to \star . c\ \mathtt{()} \to c\ \mathtt{()}$]: Any expression e with type $\forall c{:}\star \to \star . c\ \mathtt{()} \to c\ \mathtt{()}$ may only be inhabited by the identity function. In other words, for every $\tau_c \in \mathtt{ty}(\star \to \star)$ and value u with $\cdot \vdash u : \tau_c\ \mathtt{()}$, $e\ u \Downarrow u$.

PROOF. Assume that $\cdot \vdash e : \forall c{:}\star \to \star . c\ \mathtt{()} \to c\ \mathtt{()}$. Then by Theorem 2.17 we have:
$$(e, e) \in \mathcal{C}\llbracket \cdot \vdash \forall c{:}\star \to \star . c\ \mathtt{()} \to c\ \mathtt{()} : \star \rrbracket$$
By expanding definition of the interpretation, for any
$$\begin{array}{rcl} \tau_c^1 & \in & \mathtt{ty}(\star \to \star) \\ \tau_c^2 & \in & \mathtt{ty}(\star \to \star) \\ f_c & \in & \mathtt{GRel}^{\star \to \star} \text{ with } \mathtt{wfGRel}^{\star \to \star}(\tau_c^1, \tau_c^2)(f_c) \\ (e_1, e_2) & \in & \mathcal{C}\llbracket c{:}\star \to \star \vdash c\ \mathtt{()} : \star \rrbracket_{c \mapsto (\tau_c^1, \tau_c^2, f_c)} \end{array}$$
it is the case that:
$$(e\ e_1, e\ e_2) \in \mathcal{C}\llbracket c{:}\star \to \star \vdash c\ \mathtt{()} : \star \rrbracket_{c \mapsto (\tau_c^1, \tau_c^2, f_c)} \qquad (1)$$
We can now pick $\tau_c^1 = \tau_c^2 = \tau_c$ and an appropriate f_c:
$$\begin{array}{rcl} f_c & \in & \mathtt{type} \supset \mathtt{type} \supset \mathtt{GRel}^\star \supset \mathtt{GRel}^\star \\ f_c\ \alpha\ \beta\ \_ & = & \text{if } (\cdot \vdash \alpha \equiv \mathtt{()} : \star \wedge \cdot \vdash \beta \equiv \mathtt{()} : \star) \\ & & \text{then } \{(v, u) \mid \cdot \vdash v : \tau_c\ \mathtt{()}\} \text{ else } \emptyset \end{array}$$

Intuitively, the morphism f_c returns the graph of a constant function that always returns u when called with type arguments equivalent to $\mathtt{()}$, and the empty relation otherwise. It is straightforward to see that $\mathtt{wfGRel}^{\star \to \star}(\tau_c, \tau_c)(f_c)$. Therefore
$$\llbracket c{:}\star \to \star \vdash c\ \mathtt{()} : \star \rrbracket_{c \mapsto (\tau_c, \tau_c, f_c)} = \{(v, u) \mid \cdot \vdash v : \tau_c\ \mathtt{()}\}$$
Because (u, u) is in this set, we can pick e_1 and e_2 both to be u and use (1) to show that that $e\ e_2 \Downarrow u$, hence $e\ u \Downarrow u$ as required. □

Note a departure from the approach to free theorems for System F. For System F, useful theorems are derived by instantiating relations to be graphs of functions expressible in System F. Here, we instantiated a generalized relation to be a morphism in our metalogic that is itself not representable in F$_\omega$. In particular, this morphism is not parametric: it behaves differently at type $\mathtt{()}$ than at other types. This same idea will be used with a free theorem for the *gcast* function in the next Section.

3. Free theorem for generic cast

We are now ready to move on to showing the (partial) correctness of generic cast. The R$_\omega$ type for generic cast is:

$$gcast : \forall (a{:}\star)(b{:}\star)(c{:}\star \to \star) . \mathtt{R}\ a \to \mathtt{R}\ b \to (() + (c\ a \to c\ b))$$

The abstraction theorem for this type follows. Assume that:
$$\begin{array}{rcl} \tau_a^1, \tau_a^2, \tau_b^1, \tau_b^2 & \in & \mathtt{ty}(\star) \\ \tau_c^1, \tau_c^2 & \in & \mathtt{ty}(\star \to \star) \\ \Gamma & = & (a{:}\star), (b{:}\star), (c{:}\star \to \star) \\ r_a & \in & \mathtt{GRel}^\star \text{ with } \mathtt{wfGRel}^\star(\tau_a^1, \tau_a^2)(r_a) \\ r_b & \in & \mathtt{GRel}^\star \text{ with } \mathtt{wfGRel}^\star(\tau_b^1, \tau_b^2)(r_b) \\ f_c & \in & \mathtt{GRel}^{\star \to \star} \text{ with } \mathtt{wfGRel}^{\star \to \star}(\tau_c^1, \tau_c^2)(f_c) \\ \delta & = & a \mapsto (\tau_a^1, \tau_a^2, r_a), b \mapsto (\tau_b^1, \tau_b^2, r_b), \\ & & c \mapsto (\tau_c^1, \tau_c^2, f_c) \\ (e_{ra}^1, e_{ra}^2) & \in & \mathcal{C}\llbracket \Gamma \vdash \mathtt{R}\ a : \star \rrbracket_\delta \\ (e_{rb}^1, e_{rb}^2) & \in & \mathcal{C}\llbracket \Gamma \vdash \mathtt{R}\ b : \star \rrbracket_\delta \end{array}$$
Then, either the cast fails and
$$gcast\ e_{ra}^1\ e_{rb}^1 \Downarrow \mathtt{inl}\ e_1' \wedge gcast\ e_{ra}^2\ e_{rb}^2 \Downarrow \mathtt{inl}\ e_2' \wedge e_{1,2}' \Downarrow ()$$
or the cast succeeds and
$$\begin{array}{l} gcast\ e_{ra}^1\ e_{rb}^1 \Downarrow \mathtt{inr}\ e_1' \wedge gcast\ e_{ra}^2\ e_{rb}^2 \Downarrow \mathtt{inr}\ e_2' \wedge \\ \quad \text{for all } (e_1, e_2) \in \mathcal{C}(f_c\ \tau_a^1\ \tau_a^2\ r_a), \\ \qquad (e_1'\ e_1, e_2'\ e_2) \in \mathcal{C}(f_c\ \tau_b^1\ \tau_b^2\ r_b) \end{array}$$

We can use this theorem to derive properties about *any* implementation of $gcast$. The first property that we can show (which is only auxiliary to the proof of the main theorem about $gcast$) is that if $gcast$ returns positively then the two types must be equivalent.

3.1 Lemma: If $\cdot \vdash e_{ra} : \mathtt{R}\ \tau_a$, $\cdot \vdash e_{rb} : \mathtt{R}\ \tau_b$, and $gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e$ then it follows that $\cdot \vdash \tau_a \equiv \tau_b : \star$.

PROOF. From the assumptions we get that *for any* $\tau_c \in \mathtt{ty}(\star \to \star)$:

$$\cdot \vdash gcast\ e_{ra}\ e_{rb} : \mathtt{()} + (\tau_c\ \tau_a \to \tau_c\ \tau_b)$$

Assume by contradiction now that $\cdot \not\vdash \tau_a \equiv \tau_b : \star$. Then we instantiate the abstraction theorem with $\tau_c^{1,2} = \lambda a{:}\star\,.\mathtt{()}$, $\tau_a^{1,2} = \tau_a$, $\tau_b^{1,2} = \tau_b$, $r_a = \llbracket \cdot \vdash \tau_a : \star \rrbracket_{\cdot}$, $r_b = \llbracket \cdot \vdash \tau_b : \star \rrbracket_{\cdot}$, $e_{ra}^{1,2} = e_{ra}$, $e_{rb}^{1,2} = e_{rb}$. We additionally take

$$\begin{array}{lcl} f_c & \in & \mathtt{type} \supset \mathtt{type} \supset \mathtt{GRel}^{\star} \supset \mathtt{GRel}^{\star} \\ f_c\ \alpha\ \beta\ r & = & \text{if } (\cdot \vdash \alpha \equiv \tau_a : \star \wedge \cdot \vdash \beta \equiv \tau_a : \star) \\ & & \text{then } \llbracket \cdot \vdash (\lambda a{:}\star\,.\mathtt{()})\ \tau_a : \star \rrbracket_{\cdot} \text{ else } \emptyset \end{array}$$

One can confirm that $\mathtt{wfGRel}^{\star\to\star}(\lambda a{:}\star\,.\mathtt{()}, \lambda a{:}\star\,.\mathtt{()})(f_c)$. Moreover $(e_{ra}, e_{ra}) \in \mathcal{C}(\mathcal{R}\ \tau_a\ \tau_a\ r_a)$ by the abstraction theorem, and similarly $(e_{rb}, e_{rb}) \in \mathcal{C}(\mathcal{R}\ \tau_b\ \tau_b\ r_b)$. Then by the free theorem for $gcast$ above we know that since $((), ()) \in \mathcal{C}(f_c\ \tau_a\ \tau_a\ r_a)$ it is: $(e\ (), e\ ()) \in \mathcal{C}(f_c\ \tau_b\ \tau_b\ r_b)$ (e is equal to both e_1' and e_2' in the theorem for $gcast$). But, if $\cdot \not\vdash \tau_a \equiv \tau_b$ then $\mathcal{C}(f_c\ \tau_b\ \tau_b\ r_b) = \emptyset$, a contradiction. □

We can now show our important result about $gcast$: if $gcast$ succeeds and returns a conversion function, then that function *must* behave as the identity. Note that if the type representations agree, we cannot conclude that $gcast$ will succeed. An implementation of $gcast$ may always fail for any pair of arguments and still be well typed.

3.2 Lemma [Partial correctness of $gcast$]: If $\cdot \vdash e_{ra} : \mathtt{R}\ \tau_a$, $\cdot \vdash e_{rb} : \mathtt{R}\ \tau_b$, $gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e$, and e_a is such that $\cdot \vdash e_a : \tau_c\ \tau_a$, with $e_a \Downarrow w$, then $e\ e_a \Downarrow w$.

PROOF. First, by Lemma 3.1 we get that $\cdot \vdash \tau_a \equiv \tau_b : \star$. We may then instantiate the free theorem for the type of $gcast$ as in Lemma 3.1. and pick the same instantiation for types and relations except for the instantiation of c. We choose c to be instantiated with (τ_c, τ_c, f_c) where f_c is:

$$\begin{array}{lcl} f_c & \in & \mathtt{type} \supset \mathtt{type} \supset \mathtt{GRel}^{\star} \supset \mathtt{GRel}^{\star} \\ f_c\ \alpha\ \beta\ r & = & \text{if } (\cdot \vdash \alpha \equiv \tau_a : \star \wedge \cdot \vdash \beta \equiv \tau_a : \star) \\ & & \text{then } \{(v, w) \mid \cdot \vdash v : \tau_c\ \tau_a\} \text{ else } \emptyset \end{array}$$

and τ_c can be any type in $\mathtt{ty}(\star \to \star)$. It is easy to see that $\mathtt{wfGRel}^{\star\to\star}(\tau_c, \tau_c)(f_c)$. Then, using the abstraction theorem we get that:

$$gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e_1 \qquad (2)$$

$$gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e_2 \qquad (3)$$

$$\forall (e_1', e_2') \in \mathcal{C}(f_c\ \tau_a\ \tau_a\ r_a), (e_1\ e_1', e_2\ e_2') \in \mathcal{C}(f_c\ \tau_b\ \tau_b\ r_b) \qquad (4)$$

Because of the particular choice for f_c we know that $(e_a, e_a) \in \mathcal{C}(f_c\ \tau_a\ \tau_a\ r_a)$. From determinacy of evaluation and equations (2) and (3) we get that $e_1 = e_2 = e$. Then, from (4) we get that $(e\ e_a, e\ e_a) \in \mathcal{C}(f_c\ \tau_b\ \tau_b\ r_b)$, hence $e\ e_a \Downarrow w$ as required. □

3.3 Remark: A similar theorem as the above would be true for any term of type $\forall (a{:}\star)(b{:}\star)(c{:}\star \to \star)\,.\,\mathtt{()} + (c\ a \to c\ b)$, if such a term could be constructed that would return a right injection. What is important in R_ω is that the extra $\mathtt{R}\ a$ and $\mathtt{R}\ b$ arguments and `typerec` make the programming of such a function possible! While the theorem is true in F_ω, we cannot really use it because there are no terms of that type that can return right injections.

The condition that the function f_c has to operate uniformly for equivalence classes of type α and β, which is imposed in the definition of `wfGRel`, is not to be taken lightly. If this condition is violated, the coherence theorem breaks. The abstraction theorem then can no longer be true. By contradiction, if the abstraction theorem remained true if this condition was violated, we could derive a false statement about $gcast$. Assume that we had picked a function f which does not satisfy this property:

$$\begin{array}{lcl} f & \in & \mathtt{type} \supset \mathtt{type} \supset \mathtt{GRel}^{\star} \supset \mathtt{GRel}^{\star} \\ f\ \mathtt{()}\ \mathtt{()}\ \_ & = & \{(v, v) \mid \cdot \vdash v : \tau_c\ \mathtt{()}\} \\ f\ \_\ \_\ \_ & = & \emptyset \end{array}$$

Let $\tau_c = \lambda c{:}\star\,.c$. We instantiate the type of $gcast$ as follows: we instantiate c with (τ_c, τ_c, f), a with $(\mathtt{()}, \mathtt{()}, \llbracket \mathtt{()} \rrbracket)$, and b with $((\lambda d{:}\star\,.d)\ \mathtt{()}, \mathtt{()}, \llbracket \mathtt{()} \rrbracket)$. The important detail is that although f can take any relation that satisfies $\mathtt{wfGRel}^{\star}(\alpha_1, \alpha_2)$ to a relation that satisfies $\mathtt{wfGRel}^{\star}(\tau_c\ \alpha_1, \tau_c\ \alpha_2)$, it can return different results for *equivalent but syntactically different type arguments*. In particular, the instantiation of b involves types not syntactically equal to $\mathtt{()}$. Then, if $gcast\ \mathtt{R}_{()}\ \mathtt{R}_{()}$ returns $\mathtt{inr}\ e$, it has to be the case that $(e\ (), e\ ()) \in \emptyset$, a contradiction! Hence the abstraction theorem must break when generalized morphisms at higher kinds do not respect type equivalence classes of their type arguments.

4. Discussion

4.1 Relational interpretation and contextual equivalence

How does the relational interpretation of types given here compare to contextual equivalence? We write $e_1 \equiv_{ctx} e_2 : \tau$, and read e_1 is contextually equivalent to e_2 at type τ, for $e_{1,2}$ closed expressions of type τ whenever the following condition holds: For any program context that returns `int` and has a hole of type τ, plugging e_1 and e_2 in that context returns the same integer value. It can be shown that the relational interpretation of R_ω is sound with respect to contextual equivalence, and hence can be used as a *proof method* for establishing contextual equivalence between expressions.

On the other hand it is known that in the presence of sums and polymorphism the interpretation of types is not complete with respect to contextual equivalence. There exists a standard fix to this problem which involves modifying the clauses of the definition that correspond to sums (such as the $\llbracket + \rrbracket$ and $\mathcal{R}$ operations) by $\top\top$-closing them [29, 28]. The $\top\top$-closure of a value relation can be defined by taking the set of pairs of program contexts under which related elements are indistinguishable, and taking again the set of pairs of values that are indistinguishable under related program contexts. In the presence of polymorphism, $\top\top$-closure is additionally required in the interpretation of type variables of kind $\star$, or as an extra condition on the definition of `wfGRel` at kind $\star$.

4.2 Parametricity, polymorphism, and non-termination

R_ω does not include representations of all types for a good reason. Some type representations complicate the relational interpretation of types and even change the fundamental properties of the language.

To demonstrate these complications, what would happen if we added the following representation to R_ω?

$$\mathtt{R_{id}} :: \mathtt{R}\ (\forall a{:}\star\,.\mathtt{R}\ a \to a \to a)$$

Suppose we extend `typerec` with a branch for this representation, and extended $gcast$ accordingly. To simplify the presentation, below we abbreviate the type $(\forall a{:}\star\,.\mathtt{R}\ a \to a \to a)$ as Rid.

Then, we could encode an infinite loop in R_ω, based on an example by Mitchell and Harper [15]. This example begins by using *gcast* to enable a self-application term with a concise type.

$$\begin{array}{l}
\mathit{delta} :: \forall a{:}\star\,.\mathtt{R}\ a \rightarrow a \rightarrow a \\
\mathit{delta}\ \mathit{ra} = \ \mathtt{case}\ (\mathit{gcast}\ \mathtt{R}_{\mathtt{id}}\ \mathit{ra})\ \mathtt{of}\ \{\ \mathtt{inr}\ y.y\ (\lambda x\,.\,x\ \mathtt{R}_{\mathtt{id}}\ x); \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{inl}\ z.(\lambda x\,.\,x)\ \}
\end{array}$$

Above, if the cast succeeds, then y has type $\forall c{:}\star \rightarrow \star.\,c\ \mathit{Rid} \rightarrow c\ a$, and we can then instantiate y to $(\mathit{Rid} \rightarrow \mathit{Rid}) \rightarrow (a \rightarrow a)$. We can now add another self-application to get an infinite loop:

$$\begin{array}{l}
\mathit{omega} :: \forall a{:}\star\,.\mathtt{R}\ a \rightarrow a \rightarrow a \\
\mathit{omega}\ = \ \mathit{delta}\ \mathtt{R}_{\mathtt{id}}\ \mathit{delta}
\end{array}$$

Unfolding the definitions shows that *omega* is divergent:

$$\begin{array}{rcl}
\mathit{omega} & \cong & \mathit{delta}\ \mathtt{R}_{\mathtt{id}}\ \mathit{delta} \\
 & \cong & (\lambda x\,.\,x\ \mathtt{R}_{\mathtt{id}}\ x)\ \mathit{delta} \\
 & \cong & \mathit{delta}\ \mathtt{R}_{\mathtt{id}}\ \mathit{delta}
\end{array}$$

What this example demonstrates is that we cannot extend the relational interpretation to $\mathtt{R}_{\mathtt{id}}$ and the proof of the abstraction theorem in a straightforward manner. Recall the definition of the morphism $\mathcal{R}$ in Figure 11. The application $\mathcal{R}\ \ \alpha\ \ \beta\ \ r$ depends on whether r can be constructed as an application of morphisms $[\![\mathtt{int}]\!]$, $[\![\mathtt{()}]\!]$, $[\![\times]\!]$, and $[\![+]\!]$. If we are to add a new representation constructor $\mathtt{R}_{\mathtt{id}}$, we must restrict r in a similar way. To do so, it is tempting to add:

$$\begin{array}{rcl}
\mathcal{R} & = & \ldots\text{as before}\ldots \\
 & \cup & \{(\mathtt{R}_{\mathtt{id}}, \mathtt{R}_{\mathtt{id}}) \mid \cdot \vdash \alpha \equiv \mathit{Rid} : \star \wedge \cdot \vdash \beta \equiv \mathit{Rid} : \star \wedge \\
 & & \qquad\qquad r \equiv_{\star} [\![\cdot \vdash \mathit{Rid} : \star]\!]_{\cdot}\}
\end{array}$$

And recall that

$$[\![\Gamma \vdash \mathtt{R}\ \tau : \star]\!]_{\delta} \quad = \quad \mathcal{R}\ \delta^{1}\tau\ \delta^{2}\tau\ \ [\![\Gamma \vdash \tau : \star]\!]_{\delta}$$

However, this definition is not well-founded. In particular, $\mathcal{R}$ recursively calls the main interpretation function on the type *Rid*, which is not necessarily smaller than τ.

However, this example does not mean that we cannot give any relational interpretation to $\mathtt{R}_{\mathtt{id}}$. One strategy might be based on contextual equivalence:

$$\begin{array}{rcl}
\mathcal{R} & = & \ldots\text{as before}\ldots \\
 & \cup & \{(\mathtt{R}_{\mathtt{id}}, \mathtt{R}_{\mathtt{id}}) \mid \cdot \vdash \alpha \equiv \mathit{Rid} : \star \wedge \cdot \vdash \beta \equiv \mathit{Rid} : \star \wedge \\
 & & \qquad\qquad r \equiv_{\star} (\cdot \equiv_{ctx} \cdot : \mathit{Rid})^{val}\}
\end{array}$$

where $(\cdot\ \equiv_{ctx}\ \cdot\ :\ \mathit{Rid})^{val}$ is the restriction of contextual equivalence on type *Rid* on values. Although this is a plausible extension, quite a bit of our infrastructure would have to change. Importantly, the computation lifting of value relations would have to take into account the non-termination, and for the proof of the abstraction theorem (case for `typerec`) we would have to show that $[\![\cdot \vdash \mathit{Rid} : \star]\!]_{\cdot}$ coincides with $(\cdot \equiv_{ctx} \cdot : \mathit{Rid})^{val}$, a change that requires even further modifications in other clauses of the definition of the relational interpretation of types (as outlined in the previous section). We have not carried out this experiment.

A different question is: what class of polymorphic types *can* we represent with our current methodology (i.e. without breaking strong normalization)? The answer is that we can represent polymorphic types as long as those types contain only representations of *closed* types. For example, the problematic behaviour above was caused because the type $\forall a.\mathtt{R}\ a \rightarrow a \rightarrow a$ includes $\mathtt{R}\ a$, the representation of a quantified type. Such behaviour cannot happen when we only include representations of types such as $\forall a\,.\,a \rightarrow a$, $\forall a\,.\,a \rightarrow \mathtt{R}\ \mathtt{int} \rightarrow a$, or even $\forall a\,.\,a$. We can still give a definition of $\mathcal{R}$ that calls recursively the main interpretation function, but the definition can be shown well-formed using a more elaborate metric on types that takes into account the return types of the representation constructors. One can come up with various such ad-hoc restrictions but it is not clear whether these restrictions are useful to programmers or theoreticians.

4.3 Related work

Surprisingly, although the interpretation of higher-kinded types as morphisms in the meta-logic between syntactic term relations seems to be folklore in the programming languages theory [24], it can be found in very few sources in the literature.

Kučan [20] interprets the higher-order polymorphic λ-calculus within a second-order logic in a way similar to ours. However, the type arguments (which are important for our examples) are missing from the higher-order interpretations, and it is not clear that the particular second-order logic that Kučan employs is expressive enough to host the large type of generalized relations. On the other hand, Kučan's motivation is rather different from ours: he shows the correspondence between free theorems obtained directly from algebraic datatype signatures, and free theorems derived from Church encodings.

Gallier gives a detailed formalization [12] closer to ours, although his motivation is a strong normalization proof for F_{ω}, based on Girard's reducibility candidates method, and not free-theorem reasoning about F_{ω} programs. Therefore the interpretation that he gives is a unary instead of binary relation. Our inductive definition of `GRel`, corresponds to his definition of (generalized) candidate sets. The important requirement that the generalized morphisms respect equivalence classes of types (`wfGRel`) is also present in this formalization (Definition 16.2, Condition (4)). Nevertheless there is no explicit account of what equality means, and what assumptions are made about the meta-logic. In contrast, we explicitly define extensional equality for `GRel`s with the extra complication that this must be given simultaneously with the definition of `wfGRel`.

Concerning the interpretation of representation types, this paper extends the ideas developed in previous work by the authors [33] to a calculus with higher-order polymorphism.

A similar (but more general) approach of performing recursion over the type structure of the arguments for generic programming has been employed in Generic Haskell. Free theorems about generic functions written in Generic Haskell have been explored by Hinze [17]. Hinze derives equations about generic functions by generalizing the usual equations for base kinds using an appropriate logical relation at the type level, assuming a cpo model, assuming the main property for the logical relation, and assuming a polytypic fixpoint induction scheme. Our approach relies on no extra assumptions, and our goal is slightly different: While Hinze aims to generalize behaviour of Generic Haskell functions from base kind to higher kinds, we are more interested in investigating the abstraction properties that higher-order types carry. Representation types simply make programming interesting generic functions possible.

Finally, Washburn and Weirich give a relational interpretation for a language with non-trivial type equivalence [35], but without quantification over higher-kinded types. To deal with the complications of type equivalence that we explain in this paper, Washburn and Weirich use canonical forms of types (β-normal η-long forms of types [16]) as canonical representatives of equivalence classes. Though perhaps more complicated, our analysis (especially outlining the necessary `wfGRel` conditions) provides better insight on the role of type equivalence in the interpretation of higher-order polymorphism.

4.4 Future work

There are some limitations of this work to be addressed before it can move from being a theoretical pearl to a practical reasoning technique. In the first place, the language R_{ω}, is not full Haskell. If we wished to use these results to reason about Haskell implementations of `gcast`, we must extend our model to include more of Haskell—in particular, general recursion and recursive types [25, 19, 3, 2, 10]. We believe that the techniques developed here are independent of those for advanced language features, so did not include them in this pearl. However, including arbitrary recursive types is probably all that is required to extend this work to free theorems for all GADTs.

Another Haskell feature lacking from R_ω is support for generative types. In Haskell, these are newtypes and datatype definitions where each declaration creates a new type that is structurally isomorphic to existing types, but not equal. Dealing with these datatypes in generic programming is tricky—the desired behaviour is that generic functions should automatically extend to new type definitions based on its isomorphic structure, optionally allowing "after-the-fact" specialization for specific types [23, 18, 37]. However, techniques that allow this behavior cannot define `gcast`. As a result, generic programming libraries that depend on `gcast` [22] implement it as a language extension, not directly in Haskell.

Free theorems for programs with higher-order polymorphic types were derived in this pearl using morphisms that ignore their relational arguments. An interesting question to address, especially when recursive features are added in the language, is whether more expressive functions, such as recursive morphisms on the type arguments, or morphisms that manipulate their relational arguments, lead to more interesting free theorems. Is there a classification of the equations that we can derive about programs according to the expressive power of the interpretation morphisms? Can we show for example that the free theorem for generic cast cannot be proved using morphisms that treat their type arguments parametrically?

4.5 Conclusion

We have given a rigorous roadmap through the proof of the abstraction theorem for a language with higher-order polymorphism and representation types, by interpreting types of higher kind directly into the meta-logic. We have shown how parametricity can be used to derive the partial correctness of generic cast from its type. In conclusion, this pearl demonstrates that parametric reasoning is possible in the representation-based approach to generic programming.

Acknowledgments Thanks to Aaron Bohannon, Jeff Vaughan, and Steve Zdancewic for their comments. Janis Voigtländer brought Kučan's dissertation to our attention.

References

[1] Peter Achten, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Compositional model-views with generic graphical user interfaces. In *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, pages 39–55, 2004.

[2] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[3] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[4] Lennart Augustsson and Kent Petersson. Silly type families. Available from `http://www.cs.pdx.edu/~sheard/papers/silly.pdf`, September 1994.

[5] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166, New York, NY, USA, 2002. ACM Press.

[6] James Cheney. Scrap your nameplate: (functional pearl). In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 180–191, New York, NY, USA, 2005. ACM Press.

[7] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104, New York, NY, USA, 2002. ACM Press.

[8] James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

[9] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[10] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electronic Notes in Theoretical Computer Science*, 2007. (To appear.).

[11] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

[12] Jean H. Gallier. On Girard's "Candidats de Reductibilité". In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *The APIC Series*, pages 123–203. Academic Press, 1990.

[13] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[14] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1988.

[15] Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Inf. Process. Lett.*, 70(1):1–5, 1999.

[16] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Logic*, 6(1):61–101, 2005.

[17] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002. MPC Special Issue.

[18] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*. Springer, 2006.

[19] Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. *SIGPLAN Not.*, 39(1):99–110, 2004.

[20] Jacov Kučan. *Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS Transform and Free Theorems*. PhD thesis, Massachusetts Institute of Technology, February 1997.

[21] Ralf Lämmel. Scrap your boilerplate with XPath-like combinators. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 137–142, New York, NY, USA, 2007. ACM Press.

[22] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, 2003.

[23] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.

[24] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, La Jolla, CA, June 1995.

[25] Paul-André Melliès and Jérôme Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 82–91, Washington, DC, USA, 2005. IEEE Computer Society.

[26] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In *International Conference on Typed Lambda Calculi and Applications, TLCA '93*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[27] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[28] A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.

[29] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–

359, 2000.

[30] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[31] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proc 4th International Workshop on Logical Frameworks and Meta-languages (LFM'04), Cork*, pages 106–124, July 2004.

[32] Vincent Simonet and Franois Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1), January 2007.

[33] Dimitrios Vytiniotis and Stephanie Weirich. Free theorems and runtime type representations. *Electron. Notes Theor. Comput. Sci.*, 173:357–373, 2007.

[34] Philip Wadler. Theorems for free! In *FPCA89: Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, London, September 1989.

[35] Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information flow. In *The Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 62–71, Chicago, IL, June 2005. IEEE Computer Society, IEEE Computer Society Press.

[36] Stephanie Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6):681–695, November 2004.

[37] Stephanie Weirich. RepLib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, September 2006.

[38] Stephanie Weirich. Type-safe run-time polytypic programming. *J. Funct. Program.*, 16(6):681–710, 2006.

A. Definition of *cast* in R_ω

The R_ω definition of *cast* appears below.

```
1   $cast :: \forall a : \star . \forall b : \star . \mathtt{R}\ a \to \mathtt{R}\ b \to \mathtt{()} + (a \to b)$
2   $cast = \lambda x\,.\mathtt{typerec}\ x\ \mathtt{of}\ \{$
3    $\lambda y\,.\mathtt{typerec}\ y\ \mathtt{of}\ \{\mathtt{inr}\ \lambda z.z\ ; \mathtt{inl}\ \mathtt{()}\ ; \mathtt{inl}\ \mathtt{()}\ ; \mathtt{inl}\ \mathtt{()}\};$
4    $\lambda y\,.\mathtt{typerec}\ y\ \mathtt{of}\ \{\mathtt{inl}\ \mathtt{()}\ ; \mathtt{inr}\ \lambda z.z\ ; \mathtt{inl}\ \mathtt{()}\ ; \mathtt{inl}\ \mathtt{()}\};$
5    $\lambda ra_1 . \lambda f_1 . \lambda ra_2 . \lambda f_2 . \lambda y\,.\mathtt{typerec}\ y\ \mathtt{of}\ \{$
6      $\mathtt{inl}\ \mathtt{()};$
7      $\mathtt{inl}\ \mathtt{()};$
8      $\lambda rb_1 . \lambda g_1 . \lambda rb_2 . \lambda g_2 .$
9        $\mathtt{case}\ f_1\ rb_1\ \mathtt{of}\ \{h.\mathtt{inl}\ \mathtt{()}\ ; h_1 .$
10       $\mathtt{case}\ f_2\ rb_2\ \mathtt{of}\ \{h.\mathtt{inl}\ \mathtt{()}\ ; h_2 .$
11        $\mathtt{inr}\ \lambda z\,.(h_1\ (\mathtt{fst}\ z), h_2\ (\mathtt{snd}\ z))$
12       $\}\};$
13     $\lambda rb_1 . \lambda g_1 . \lambda rb_2 . \lambda g_2 . \mathtt{inl}\ \mathtt{()}\}$
14   $\lambda ra_1 . \lambda f_1 . \lambda ra_2 . \lambda f_2 . \lambda y\,.\mathtt{typerec}\ y\ \mathtt{of}\ \{$
15     $\mathtt{inl}\ \mathtt{()};$
16     $\mathtt{inl}\ \mathtt{()};$
17     $\lambda rb_1 . \lambda g_1 . \lambda rb_2 . \lambda g_2 . \mathtt{inl}\ \mathtt{()};$
18     $\lambda rb_1 . \lambda g_1 . \lambda rb_2 . \lambda g_2 .$
19       $\mathtt{case}\ f_1\ rb_1\ \mathtt{of}\ \{h.\mathtt{inl}\ \mathtt{()}\ ; h_1 .$
20       $\mathtt{case}\ f_2\ rb_2\ \mathtt{of}\ \{h.\mathtt{inl}\ \mathtt{()}\ ; h_2 .$
21        $\mathtt{inr}\ (\lambda z\,.\mathtt{case}\ z\ \mathtt{of}\ \{z_1.h_1\ z_1\ ; z_2.h_2\ z_2\})$
22       $\}\}\}\}$
```

Thanks to implicit types, the definition of *gcast* may be obtained from this one by replacing lines 11 and 21 with `inr` $(\lambda z.h_2\ (h_1\ z))$.

B. Generalized relations, in Coq

A Coq definition of `GRel`, `wfGRel`, and `eqGRel` ($\equiv_\kappa$), follows.

```
Inductive kind : Set :=
| KStar : kind
| KFun  : kind -> kind -> kind.

(* types and a constant for type applications *)
Parameter type : Set.
Parameter TyApp : type -> type -> type.
```

```
Parameter term : Set.

(* environments and constant for empty envs   *)
Parameter env  : Set.
Parameter empty : env.

Parameter teq  : env ->
                 type -> type -> kind -> Prop.

Definition rel : Type := term -> term -> Prop.
Definition eq_rel (r1 : rel) (r2 : rel) :=
    forall e1 e2, r1 e1 e2 <-> r2 e1 e2.

(* value relations as a predicate on relations *)
Parameter vrel : type -> type -> rel -> Prop.

Fixpoint GRel (k : kind) : Type :=
  match k with
    | KStar => rel
    | KFun k1 k2 => type -> type -> GRel k1 -> GRel k2
  end.

Fixpoint wfGRel (k:kind) : type -> type ->
                              GRel k -> Prop :=
  match k as kr
  return type -> type -> GRel kr -> Prop with
    | KStar => fun t1 t2 r => vrel t1 t2 r
    | KFun k1 k2 => fun c1 c2 r =>
      (forall a1 a2 ga b1 b2 gb,
        wfGRel k1 a1 a2 ga ->
           wfGRel k2 (TyApp c1 a1)
                     (TyApp c2 a2) (r a1 a2 ga) /\
           wfGRel k1 b1 b2 gb ->
            teq empty a1 b1 k1 ->
            teq empty a2 b2 k1 -> eqGRel k1 ga gb ->
               eqGRel k2 (r a1 a2 ga) (r b1 b2 gb))
   end
with eqGRel (k:kind) : GRel k -> GRel k -> Prop :=
  match k as kr return GRel kr -> GRel kr -> Prop with
    | KStar => fun r1 r2 => eq_rel r1 r2
    | KFun k1 k2 => fun r1 r2 =>
        (forall a1 a2 g,
         wfGRel k1 a1 a2 g ->
           eqGRel k2 (r1 a1 a2 g) (r2 a1 a2 g))
  end.
```

We assume datatypes that encode R_ω syntax, such as `kind`, `term`, `type`, and `env`. Moreovere we assume constants such as `TyApp` (for type applications) and `empty` (for empty environments). Term relations are represented with the datatype `rel`, for which we give an equality predicate `eq_rel`. `rel` contains functions that return objects of type `Prop`. `Prop` is Coq's universe for propositions, therefore `rel` itself lives in Coq's `Type` universe. Then the definitions of `wfGRel` and `eqGRel` follow the paper definitions. Importantly, since `rel` lives in `Type`, the whole definition of `GRel` is a well-typed inhabitant of `Type`.