THEORETICAL PEARL

Type-safe cast does no harm: Syntactic parametricity for F_ω and beyond

DIMITRIOS VYTINIOTIS
Microsoft Research

STEPHANIE WEIRICH
University of Pennsylvania

Abstract

Generic functions can specialize their behavior 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 paper, we give a rigorous road map through the proof of parametricity for a calculus with higher-order polymorphism and type representations. We then use parametricity to derive the correctness of *type-safe cast*.

1 Introduction

Generic programming refers to the ability to specialize the behavior 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 (Baars & Swierstra, 2002; Cheney & Hinze, 2002; Hinze, 2002; Clarke *etal.*, 2001; Lämmel & Peyton Jones, 2003; Weirich, 2006b; Weirich, 2006a). 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 generic equality functions, marshalers, reductions and maps, to application-specific traversals and queries (Lämmel & Peyton Jones, 2003), user interface generators (Achten *etal.*, 2004), XML-inspired transformations (Lämmel, 2007), and compilers (Cheney, 2005).

Representation types (Crary *etal.*, 2002) are an attractive mechanism for generic programming. The key idea is simple: because the behavior of parametrically polymorphic functions cannot be influenced by the types at which they are instantiated, generic functions dispatch on term arguments that *represent* types. Representation types were originally proposed in the context of type-preserving compilation, but they may be encoded in Haskell in several ways (Cheney & Hinze, 2002; Weirich, 2006b; Weirich, 2006a). The most natural implementation uses *generalized alge-*

braic datatypes (GADTs) (Cheney & Hinze, 2003; Sheard & Pasalic, 2004), a recent extension to the Glasgow Haskell Compiler (GHC).
For example:

```
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 type `Int`, hence its type is `R Int`. Likewise `Runit` represents `()` and has type `R ()`. The constructors `Rprod` and `Rsum` represent products and sums (called `Either` in Haskell). They take as inputs a representation for `a`, a representation for `b`, and return representations for `(a,b)` and `Either a b` respectively. The important property of datatype `R t` is that the type parameter `t` is determined by the data constructor. In contrast, in an ordinary datatype, all data constructors must return the same type.

A simple example of a generic function is `add`, shown below, which adds together all 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
```

Note that in the definition of `add`, the argument `x` is treated as integer, product or sum depending on the clause. This behavior 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 generic *type-safe* cast, 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 (2004) defined two different versions of type-safe cast, `cast` and `gcast`, shown in Figures 1 and 2. Our implementations differ slightly from Weirich's—namely they use Haskell's `Maybe` type to account for potential failure, instead of an `error` primitive—but the essential structure is the same.

```
cast :: R a -> R b -> Maybe (a -> b)
cast Rint Rint   = Just (\x -> x)
cast Runit Runit = Just (\x -> x)
cast (Rprod (ra0  :: R a0)  (rb0  :: R b0))
     (Rprod (ra0' :: R a0') (rb0' :: R b0'))
 = do g :: ra0 -> ra0'
      g <- cast ra0 ra0'
      h :: rb0 -> rb0'
      h <- cast rb0 rb0'
      Just (\(a,b) -> (g a, h b))
cast (Rsum (ra0  :: R a0) (rb0  :: R b0))
     (Rsum (ra0' :: R a0')(rb0' :: R b0'))
 = do g :: ra0 -> ra0'
      g <- cast ra0 ra0'
      h :: rb0 -> rb0'
      h <- cast rb0 rb0'
      Just (\x -> case x of
                 Left a -> Left (g a)
                 Right b -> Right (h b))
cast _ _ = Nothing
```

Fig. 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')
cast (Rsum (ra0  :: R a0) (rb0  :: R b0))
     (Rsum (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 _ _ = Nothing
```

Fig. 2: `gcast`

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 first clause, both representations are `Rint`, so the type checker knows that `a=b=Int`, and so the identity function may be returned. Similar reasoning holds for `Runit`. 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 coercions of the subcomponents. To show how this works, the cases for products and sums have been decorated with type annotations.

Alternatively, `gcast` produces a coercion function that never needs to decompose (or even evaluate) its argument. The key ingredient is the use of the higher-kinded type argument `c`, that allows `gcast` to return a coercion from `c a` to `c b`. As Baars and Swierstra (2002), and Cheney and Hinze (2002) point out, `gcast` corresponds to *Leibniz equality*. From an implementation point of view, the type constructor `c` 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. The case for sums is identical.

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 suggests that both implementations do so. 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)
```

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.

Contributions. In this paper, we make the above arguments precise and rigorous. In particular, we show using a *free theorem* (Reynolds, 1983; Wadler, 1989) 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 and higher-order polymorphism, called R_ω (Crary *etal.*, 2002) (Section 2.1). We then extend Reynolds's abstraction theorem (Reynolds, 1983) to this language (Section 2.2). Reynolds's abstraction theorem, also referred to as the "parametricity theorem" (Wadler, 1989), asserts that every well-typed expression of the polymorphic λ-calculus (System F) (Girard, 1972) satisfies a particular property directly derivable from its type. After proving a version of the abstraction theorem

Kinds	κ	$::=$	$\star \mid \kappa_1 \to \kappa_2$
Types	σ, τ	$::=$	$a \mid \mathcal{K} \mid \sigma_1\ \sigma_2 \mid \lambda a{:}\kappa\,.\,\sigma$
Type constants	$\mathcal{K}$	$::=$	$\mathtt{R} \mid \mathtt{()} \mid \mathtt{int} \mid \to \mid \times \mid + \mid \forall_\kappa$
Expressions	e	$::=$	$\mathtt{R}_{\mathtt{int}} \mid \mathtt{R}_{()} \mid \mathtt{R}_{\times}\ e_1\ e_2 \mid \mathtt{R}_{+}\ e_1\ e_2$
		$\mid$	$\mathtt{typerec}\ e\ \mathtt{of}\ \{e_{\mathtt{int}}\ ;\ e_{()}\ ;\ e_{\times}\ ;\ e_{+}\}$
		$\mid$	$\mathtt{fst}\ e \mid \mathtt{snd}\ e \mid (e_1, e_2) \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e$
		$\mid$	$\mathtt{case}\ e\ \mathtt{of}\ \{x.e_l\ ;\ x.e_r\}$
		$\mid$	$\mathtt{()} \mid i \mid x \mid \lambda x\,.\,e \mid e_1\ e_2$
Typing contexts	Γ	$::=$	$\cdot \mid \Gamma, a{:}\kappa \mid \Gamma, x{:}\tau$

Fig. 3: Syntax of System R_ω

for R_ω, 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 6. 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 (Vytiniotis & Weirich, 2007), which was limited to the case of second-order polymorphism, we had difficulty finding free theorems for generic functions that were not trivial. This paper 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 work 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 often used 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 are folklore, there are few explicit proofs of parametricity for F_ω available 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_ω by Gallier (1990), but we do our reasoning in a typed meta-logic. Therefore, we expect our development to be particularly well-suited for mechanical verification in proof assistants based on Type Theory, such as Coq (`http://coq.inria.fr`).

2 Parametricity for R_ω

2.1 The R_ω calculus.

We begin with a formal description of the R_ω calculus, an extension of a Curry-style variant of F_ω (Girard, 1972). The syntax of this language appears in Figure 3, and

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

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

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

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

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

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \tau \equiv \tau : \kappa}\ \text{REFL} \qquad \frac{\Gamma \vdash \tau_2 \equiv \tau_1 : \kappa}{\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa}\ \text{SYM}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa \quad \Gamma \vdash \tau_2 \equiv \tau_3 : \kappa}{\Gamma \vdash \tau_1 \equiv \tau_3 : \kappa}\ \text{TRANS}$$

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

$$\frac{\Gamma, a{:}\kappa_1 \vdash \tau_1 : \kappa_2 \quad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash (\lambda a{:}\kappa_1 . \tau_1)\ \tau_2 \equiv \tau_1\{\tau_2/a\} : \kappa_2}\ \text{BETA}$$

$$\frac{\Gamma \vdash \tau : \kappa_1 \to \kappa_2 \quad a \notin fv(\tau)}{\Gamma \vdash (\lambda a{:}\kappa_1 . \tau\ a) \equiv \tau : \kappa_1 \to \kappa_2}\ \text{ETA}$$

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

Fig. 4: Type well-formedness and equivalence

the static semantics appears in Figures 4 and 5. Kinds κ include the base kind, $\star$, which classifies the types of expressions, and constructor kinds, $\kappa_1 \to \kappa_2$. The type syntax, σ, includes type variables, type constants, type-level applications, and type functions. Although type-level λ-abstractions complicate the formal development of the parametricity theorem, they simplify programming—for example, in Figure 2 we had to introduce the constructors `CL` and `CR` only because Haskell does not include type-level λ-abstractions.

Type constructor constants, $\mathcal{K}$, include standard operators, plus representation types R. In the following, we write $\to$, $\times$, and $+$ using infix notation and associate applications of $\to$ to the right. We treat impredicative polymorphism with

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

$$\frac{}{\Gamma \vdash i : \mathtt{int}}\ \text{INT} \qquad \frac{}{\Gamma \vdash \mathtt{()} : \mathtt{unit}}\ \text{UNIT} \qquad \frac{\Gamma, (x{:}\tau_1) \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : \star}{\Gamma \vdash \lambda x\,.\,e : \tau_1 \to \tau_2}\ \text{ABS}$$

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \text{VAR} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{APP}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma \times \tau}\ \text{PROD} \qquad \frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \mathtt{fst}\ \ e : \sigma}\ \text{FST} \qquad \frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \mathtt{snd}\ \ e : \tau}\ \text{SND}$$

$$\frac{\Gamma \vdash e : \sigma_1 + \sigma_2 \quad \Gamma, x : \sigma_1 \vdash e_l : \tau \quad \Gamma, x : \sigma_2 \vdash e_r : \tau}{\Gamma \vdash \mathtt{case}\ \ e\ \ \mathtt{of}\ \ \{x\ .\ e_l\ ;\ x\ .\ e_r\} : \tau}\ \text{CASE}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathtt{inl}\ \ e : \sigma + \tau}\ \text{INL} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathtt{inr}\ \ e : \sigma + \tau}\ \text{INR}$$

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

$$\frac{\Gamma \vdash e : \forall_\kappa \sigma \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e : \sigma\ \tau}\ \text{INST} \qquad \frac{\Gamma, (a{:}\kappa) \vdash e : \sigma\ a \quad a \# \Gamma}{\Gamma \vdash e : \forall_\kappa \sigma}\ \text{GEN}$$

$$\frac{}{\Gamma \vdash \mathtt{R}_{\mathtt{int}} : \mathtt{R}\ \mathtt{int}}\ \text{RINT} \qquad \frac{}{\Gamma \vdash \mathtt{R}_{\mathtt{()}} : \mathtt{R}\ \mathtt{()}}\ \text{RUNIT}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{R}\,\sigma_1 \quad \Gamma \vdash e_2 : \mathtt{R}\,\sigma_2}{\Gamma \vdash \mathtt{R}_\times\ e_1\ e_2 : \mathtt{R}\ (\sigma_1, \sigma_2)}\ \text{RPROD}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{R}\,\sigma_1 \quad \Gamma \vdash e_2 : \mathtt{R}\,\sigma_2}{\Gamma \vdash \mathtt{R}_+\ e_1\ e_2 : \mathtt{R}\ (\sigma_1 + \sigma_2)}\ \text{RSUM}$$

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

Fig. 5: Typing relation for R_ω

an infinite family of universal type constructors $\forall_\kappa$ indexed by kinds. We write $\forall (a_1{:}\kappa_1) \ldots (a_n{:}\kappa_n)\,.\,\sigma$ to abbreviate

$$\forall_{\kappa_1}(\lambda a_1{:}\kappa_1\,.\,\ldots \forall_{\kappa_n}(\lambda a_n{:}\kappa_n\,.\,\sigma)\ldots)\ .$$

R_ω expressions e include abstractions, products, sums, integers and unit. For simplicity, type abstractions and type applications are implicit. R_ω includes type representations $\mathtt{R}_{\mathtt{int}}$, $\mathtt{R}_{\mathtt{()}}$, $\mathtt{R}_\times$ and $\mathtt{R}_+$, which must be fully applied to their arguments.

```
1  cast :: ∀a : ⋆.∀b : ⋆.R a → R b → () + (a → b)
2  cast = λx.typerec x of {
3   λy.typerec y of {inr λz.z ; inl () ; inl () ; inl ()};
4   λy.typerec y of {inl () ; inr λz.z ; inl () ; inl ()};
5   λra₁.λf₁.λra₂.λf₂.λy.typerec y of {
6     inl ();
7     inl ();
8     λrb₁.λg₁.λrb₂.λg₂.
9       case f₁ rb₁ of {h.inl () ; h₁.
10      case f₂ rb₂ of {h.inl () ; h₂.
11       inr λz.(h₁ (fst z), h₂ (snd z))
12      }};
13    λrb₁.λg₁.λrb₂.λg₂.inl ()}
14  λra₁.λf₁.λra₂.λf₂.λy.typerec y of {
15    inl ();
16    inl ();
17    λrb₁.λg₁.λrb₂.λg₂.inl ();
18    λrb₁.λg₁.λrb₂.λg₂.
19      case f₁ rb₁ of {h.inl () ; h₁.
20      case f₂ rb₂ of {h.inl () ; h₂.
21       inr (λz.case z of {z₁.h₁ z₁ ; z₂.h₂ z₂})
22      }}}}
```

Fig. 6: Definition of *cast* in R_ω. The definition of *gcast* may be obtained from this one by replacing both lines 11 and 21 with $\mathtt{inr}\ (\lambda z.h_2\ (h_1\ z))$

We do not include representations for function or polymorphic types in R_ω as neither are that useful for generic programming. The former can be added in a straightforward manner, but the latter significantly changes the semantics of the language, as we discuss in Section 4.2. The language is terminating, but includes a term `typerec` that can perform primitive recursion on type representations, and includes branches for each possible representation.

For completeness, we give the R_ω implementations of *cast* and *gcast* in Figure 6. Thanks to implicit types, almost the same code defines both functions.

The dynamic semantics of R_ω is a standard large-step non-strict operational semantics, presented in Figure 7. Essentially `typerec` performs a fold over its type representation argument. We use u, v, w for R_ω values, the syntax of which is also given in Figure 7.

The static semantics of R_ω contains judgments for kinding, type equivalence, and typing. Each of these judgments uses a unified environment, Γ, containing bindings for type variables $(a{:}\kappa)$ and 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 kinding judgment $\Gamma \vdash \tau : \kappa$ (in Figure 4) 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.

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 $\cdot \vdash \tau : \kappa$.

$$\begin{array}{llll} \text{Values} & v, w, u & ::= & \mathtt{R_{int}} \mid \mathtt{R_{()}} \mid \mathtt{R_{\times}}\ e_1\ e_2 \mid \mathtt{R_{+}}\ e_1\ e_2 \\ & & \mid & (e_1, e_2) \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e \\ & & \mid & () \mid i \mid \lambda x . e \end{array}$$

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

$$\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow \lambda x . e' \quad e'\{e_2/x\} \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{e \Downarrow (e_1, e_2) \quad e_1 \Downarrow v}{\mathtt{fst}\ e \Downarrow v} \qquad \frac{e \Downarrow (e_1, e_2) \quad e_2 \Downarrow v}{\mathtt{snd}\ e \Downarrow v}$$

$$\frac{e \Downarrow \mathtt{inl}\ e_1 \quad e_l\{e_1/x\} \Downarrow v}{\mathtt{case}\ e\ \mathtt{of}\ \{x . e_l \,;\, x . e_r\} \Downarrow v} \qquad \frac{e \Downarrow \mathtt{inr}\ e_2 \quad e_r\{e_2/x\} \Downarrow v}{\mathtt{case}\ e\ \mathtt{of}\ \{x . e_l \,;\, x . e_r\} \Downarrow v}$$

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

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

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

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

Fig. 7: Operational rules

The typing judgment has the form $\Gamma \vdash e : \tau$ and appears in Figure 5. The interesting typing rules are the introduction and elimination forms for type representations. The rest of this typing relation is standard. 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 judgment $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$ defines type equivalence as a congruence relation that includes $\beta\eta$-conversion for types. (In rule BETA, we write $\tau\{\sigma/a\}$ for the capture avoiding substitution of a for σ inside τ.) In addition, we implicitly identify α-equivalent types, and treat them as syntactically equal in the rest of the paper. We give its definition in Figure 4. The presence of the rule T-EQ is important for R_ω because it allows expressions to be typed with any member of an equivalence classes of types. This behavior fits our intuition, but complicates the formalization of parametricity; a significant part of this paper is devoted to complications introduced by type equivalence.

2.2 The abstraction theorem.

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

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

Free theorems result from unfolding the definition of the interpretation of types (which appears in Figure 9, using Definition 2.5). However, before we can present that definition, we must first explain a number of auxiliary concepts.

First, we define a (meta-logical) type, $\mathtt{GRel}^{\kappa}$, to describe the interpretation of types of arbitrary kind. Only types of kind $\star$ are interpreted as term relations—types of higher kind are interpreted as sets of morphisms. (To distinguish between R_ω and meta-logical functions, we use the term *morphism* for the latter.) For example, the interpretation of a type of kind $\star \to \star$, a type level function from types to types, is the set of morphisms that take term relations to appropriate term relations.

2.2 Definition [(Typed-)Generalized Relations]:

$$\begin{array}{llcl} r,s \in & \mathtt{GRel}^{\star} & \triangleq & \mathcal{P}(\mathtt{term} \times \mathtt{term}) \\ & \mathtt{GRel}^{\kappa_1 \to \kappa_2} & \triangleq & \mathtt{TyGRel}^{\kappa_1} \supset \mathtt{GRel}^{\kappa_2} \\ \\ \rho,\pi \in & \mathtt{TyGRel}^{\kappa} & \triangleq & \mathtt{ty}(\kappa) \times \mathtt{ty}(\kappa) \times \mathtt{GRel}^{\kappa} \end{array}$$

The notation $\mathcal{P}(\mathtt{term} \times \mathtt{term})$ stands for the space of binary relations on terms 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 relations are mutually defined with Typed-Generalized Relations, $\mathtt{TyGRel}^{\kappa}$, which are triples of generalized relations and types of the appropriate kind. Elements of $\mathtt{GRel}^{\kappa_1 \to \kappa_2}$ accept one of these triples. These extra $\mathtt{ty}(\kappa)$ arguments allow the morphisms to dispatch control depending on types as well as relational arguments. This flexibility is important for the free theorems about R_ω programs, as we demonstrate in Example 2.13.

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 (Paulin-Mohring, 1993) or ZF set theory, such a definition is indeed well-formed, as there exists a type containing both spaces (for example `Type` in CIC[1], or pure ZF sets in ZF set theory). In contrast, in HOL it is not clear how to build a common type "hosting" the interpretations at all kinds.

[1] One can find a Coq definition of `GRel` and other relevant definitions in Appendix A.

$$r \in \texttt{VRel}(\tau_1, \tau_2) \triangleq \forall (e_1, e_2) \in r,\ e_1 \text{ and } e_2 \text{ are values } \wedge (\cdot \vdash e_1 : \tau_1) \wedge (\cdot \vdash e_2 : \tau_2)$$

$$(\tau_1, \tau_2, r) \in \texttt{wfGRel}^{\star} \triangleq r \in \texttt{VRel}(\tau_1, \tau_2)$$

$$\begin{array}{l}(\tau_1, \tau_2, r) \in \texttt{wfGRel}^{\kappa_1 \to \kappa_2} \triangleq \\ \quad \text{for all } \rho \in \texttt{wfGRel}^{\kappa_1}, (\tau_1\ \rho^1, \tau_2\ \rho^2, r\ \rho) \in \texttt{wfGRel}^{\kappa_2} \wedge \\ \quad \text{for all } \pi \in \texttt{wfGRel}^{\kappa_1}, \rho \equiv \pi \Longrightarrow r\ \rho \equiv_{\kappa_2} r\ \pi\end{array}$$

$$\begin{array}{lll} r \equiv_{\star} s & \triangleq & \text{for all } e_1\ e_2,\ (e_1, e_2) \in r \Longleftrightarrow (e_1, e_2) \in s \\ r \equiv_{\kappa_1 \to \kappa_2} s & \triangleq & \text{for all } \rho \in \texttt{wfGRel}^{\kappa_1},\ (r\ \rho) \equiv_{\kappa_2} (s\ \rho) \end{array}$$

$$\rho \equiv \pi \triangleq (\cdot \vdash \rho^1 \equiv \pi^1 : \kappa) \wedge (\cdot \vdash \rho^2 \equiv \pi^2 : \kappa) \wedge \hat{\rho} \equiv_{\kappa} \hat{\pi}$$

Fig. 8: Well-formed generalized relations and equality

Unfortunately, not all objects of $\texttt{GRel}^{\kappa}$ are suitable for the interpretation of types. In Figure 8, we define *well-formed generalized relations*, $\texttt{wfGRel}^{\kappa}$, a predicate on objects in $\texttt{TyGRel}^{\kappa}$. We define this predicate mutually with extensional equality on generalized relations ($\equiv_{\kappa}$) and on typed-generalized relations ($\equiv$). Because our $\texttt{wfGRel}^{\kappa}$ conditions depend on equality for type $\texttt{GRel}^{\kappa}$, we cannot include those conditions in the definition of $\texttt{GRel}^{\kappa}$ itself.

At kind $\star$, $(\tau_1, \tau_2, r) \in \texttt{wfGRel}^{\star}$ checks that r is not just any relation between terms, but a relation between values of types τ_1 and τ_2. (We use $\Longrightarrow$ and $\wedge$ for meta-logical implication and conjunction, respectively.) At kind $\kappa_1 \to \kappa_2$ we require two conditions. First, if r is applied to a well-formed $\texttt{TyGRel}^{\kappa_1}$, then the result must also be well-formed. (We project the three components of ρ with the notations ρ^1, ρ^2 and $\hat{\rho}$ respectively.) Second, for any pair of equivalent triples, ρ and π, the results $r\ \rho$ and $r\ \pi$ must also be equal. This condition asserts that morphisms that satisfy $\texttt{wfGRel}^{\kappa}$ *respect* the type equivalence classes of their type arguments.

Equality on generalized relations is also indexed by kinds; for any two $r, s \in \texttt{GRel}^{\kappa}$, the proposition $r \equiv_{\kappa} s$ asserts that the two generalized relations are extensionally equal. Extensional equality between generalized relations asserts that at kind $\star$ the two relation arguments denote the same set,[2], whereas at higher kinds it asserts that the relation arguments return equal results, when given the same argument ρ which must satisfy the $\texttt{wfGRel}^{\kappa_1}$ predicate.[3] Dropping the requirement that ρ be well-formed is not possible, as we discuss in the proof of Coherence, Theorem 2.11.

[2] We use extensional equivalence for relations in this case instead of the simpler intensional equivalence ($r = s$) to again reduce the requirements of the meta-logic. Stating it in the simpler form would require the logic to include propositional extensionality. Propositional extensionality is consistent with but independent of the Calculus of Inductive Constructions. (see `http://coq.inria.fr/V8.1/faq.html`)

[3] Equivalence at higher-kind may equivalently be defined relationally (i.e. r and s are equivalent if they take equivalent arguments to equivalent results) instead of point-wise. This version is slightly simpler, but no less expressive. See lemma 2.10.

$$
\begin{array}{lcl}
[\![\Gamma \vdash \tau : \kappa]\!] & \in & \texttt{Subst}_\Gamma \supset \texttt{GRel}^\kappa \\
\\
[\![\Gamma \vdash a : \kappa]\!]_\delta & \triangleq & \hat{\delta}(a) \\
[\![\Gamma \vdash \mathcal{K} : \kappa]\!]_\delta & \triangleq & [\![\mathcal{K}]\!] \\
[\![\Gamma \vdash \tau_1\ \tau_2 : \kappa]\!]_\delta & \triangleq & \\
\quad [\![\Gamma \vdash \tau_1 : \kappa_1 \to \kappa]\!]_\delta\ (\delta^1\tau_2,\ \delta^2\tau_2,\ [\![\Gamma \vdash \tau_2 : \kappa_1]\!]_\delta) & & \\
\quad \text{when } \Gamma \vdash \tau_1 : \kappa_1 \to \kappa \text{ and } \Gamma \vdash \tau_2 : \kappa_1 & & \\
[\![\Gamma \vdash \lambda a{:}\kappa_1 . \tau : \kappa_1 \to \kappa_2]\!]_\delta & \triangleq & \\
\quad \lambda\rho \in \texttt{TyGRel}^{\kappa_1} \mapsto [\![\Gamma, a{:}\kappa_1 \vdash \tau : \kappa_2]\!]_{\delta, a\mapsto\rho} & & \\
\quad \text{where } a \# \Gamma & &
\end{array}
$$

Fig. 9: Relational interpretation of R_ω

Equality for typed-generalized relations, $\rho \equiv \pi$, is defined point-wise. Generalized relation equality is reflexive, symmetric, and transitive, and hence is an equivalence relation. All properties follow from simple induction on the kind κ.

Importantly, the $\texttt{wfGRel}^\kappa$ predicate respects this equivalence.

2.3 Lemma: For all $\rho \equiv \pi$, if $\rho \in \texttt{wfGRel}^\kappa$ then $\pi \in \texttt{wfGRel}^\kappa$.

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 typed-generalized relations. 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 $\delta, a \mapsto \rho$ for the extension of δ that maps a to ρ and require that $a \notin dom(\delta)$. If $\delta(a) = (\tau_1, \tau_2, r)$, we define the notations $\delta^1(a) = \tau_1$, $\delta^2(a) = \tau_2$, and $\hat{\delta}(a) = r$. We also define $\delta^1\tau$ and $\delta^2\tau$ to be the extension of the domain of the substitutions δ^1 and δ^2 to include full types τ.

2.4 Definition [Substitution kind checks in environment]: We say that a substitution δ *kind checks in an environment* Γ, and write $\delta \in \texttt{Subst}_\Gamma$, when $dom(\delta) = dom(\Gamma)$ and for every $(a{:}\kappa) \in \Gamma$, we have $\delta(a) \in \texttt{TyGRel}^\kappa$.

The interpretation of R_ω types is shown in Figure 9 and is defined inductively over kinding derivations for types. The interpretation function $[\![\cdot]\!]_\cdot$ accepts a derivation $\Gamma \vdash \tau : \kappa$, and a substitution $\delta \in \texttt{Subst}_\Gamma$ and returns a generalized relation at kind κ, hence, the meta-logical type, $\texttt{Subst}_\Gamma \supset \texttt{GRel}^\kappa$. We write the δ argument as a subscript to $[\![\Gamma \vdash \tau : \kappa]\!]$.

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 $[\![\mathcal{K}]\!]$, shown in Figure 10. For an application, $\tau_1\ \tau_2$, we apply the interpretation of τ_1 to appropriate type arguments and the interpretation of τ_2. Type-level λ-abstractions are interpreted as abstractions in the meta-logic. We use λ and $\mapsto$ for meta-logic abstractions. Confirming that $[\![\Gamma \vdash \tau : \kappa]\!]_\delta \in \texttt{GRel}^\kappa$ is straightforward using the fact that $\delta \in \texttt{Subst}_\Gamma$.

$$
\begin{array}{lcl}
[\![\mathcal{K}]\!] & \in & \texttt{GRel}^{kind(\mathcal{K})} \\
\\
[\![\texttt{int}]\!] & \triangleq & \{(i,i) \mid \text{ for all } i\} \\
[\![\texttt{()}]\!] & \triangleq & \{(\texttt{()},\texttt{()})\} \\
[\![\to]\!] & \triangleq & \lambda\rho,\pi \in \texttt{TyGRel}^\star \mapsto \\
 & & \quad \{(v_1,v_2) \mid \quad (\cdot \vdash v_1 : \rho^1 \to \pi^1) \wedge \\
 & & \qquad\qquad\qquad (\cdot \vdash v_2 : \rho^2 \to \pi^2) \wedge \\
 & & \qquad\qquad\qquad \text{for all } (e_1', e_2') \in \mathcal{C}(\hat{\rho}), \\
 & & \qquad\qquad\qquad (v_1\ e_1', v_2\ e_2') \in \mathcal{C}(\hat{\pi})\ \} \\
[\![\times]\!] & \triangleq & \lambda\rho,\pi \in \texttt{TyGRel}^\star \mapsto \\
 & & \{(v_1,v_2) \mid (\texttt{fst}\ \ v_1, \texttt{fst}\ \ v_2) \in \mathcal{C}(\hat{\rho})\} \cap \\
 & & \{(v_1,v_2) \mid (\texttt{snd}\ \ v_1, \texttt{snd}\ \ v_2) \in \mathcal{C}(\hat{\pi})\} \\
[\![+]\!] & \triangleq & \lambda\rho,\pi \in \texttt{TyGRel}^\star \mapsto \\
 & & \{(\texttt{inl}\ e_1, \texttt{inl}\ e_2) \mid (e_1,e_2) \in \mathcal{C}(\hat{\rho})\} \cup \\
 & & \{(\texttt{inr}\ e_1, \texttt{inr}\ e_2) \mid (e_1,e_2) \in \mathcal{C}(\hat{\pi})\} \\
[\![\forall_\kappa]\!] & \triangleq & \lambda\rho \in \texttt{TyGRel}^{\kappa\to\star} \mapsto \\
 & & \quad \{(v_1,v_2) \mid \quad (\cdot \vdash v_1 : \forall_\kappa\ \rho^1) \wedge (\cdot \vdash v_2 : \forall_\kappa\ \rho^2) \wedge \\
 & & \qquad\qquad\qquad \text{for all } \pi \in \texttt{wfGRel}^\kappa, (v_1,v_2) \in (\hat{\rho}\ \pi)\} \\
[\![\texttt{R}]\!] & \triangleq & \mathcal{R} \\
\\
\\
\mathcal{R} & \triangleq & \lambda(\tau,\sigma,r) \in \texttt{TyGRel}^\star \mapsto \\
 & & \{(\texttt{R}_{\texttt{int}}, \texttt{R}_{\texttt{int}}) \mid (\tau,\sigma,r) \equiv (\texttt{int},\texttt{int},[\![\texttt{int}]\!]\} \\
 & \cup & \{(\texttt{R}_{\texttt{()}}, \texttt{R}_{\texttt{()}}) \mid (\tau,\sigma,r) \equiv (\texttt{()},\texttt{()},[\![\texttt{()}]\!])\} \\
 & \cup & \{(\texttt{R}_\times\ e_a^1\ e_b^1, \texttt{R}_\times\ e_a^2\ e_b^2) \ \mid \\
 & & \quad \exists \rho_a, \rho_b \in \texttt{wfGRel}^\star \wedge \\
 & & \quad \cdot \vdash \tau \equiv \rho_a^1 \times \rho_b^1 : \star \wedge \cdot \vdash \sigma \equiv \rho_a^2 \times \rho_b^2 : \star \wedge \\
 & & \quad r \equiv_\star [\![\times]\!]\ \rho_a\ \rho_b \wedge \\
 & & \quad (e_a^1, e_a^2) \in \mathcal{C}(\mathcal{R}\ \rho_a) \wedge (e_b^1, e_b^2) \in \mathcal{C}(\mathcal{R}\ \rho_b)\ \} \\
 & \cup & \{(\texttt{R}_+\ e_a^1\ e_b^1, \texttt{R}_+\ e_a^2\ e_b^2) \ \mid \\
 & & \quad \exists \rho_a, \rho_b \in \texttt{wfGRel}^\star \wedge \\
 & & \quad \cdot \vdash \tau \equiv \rho_a^1 + \rho_b^1 : \star \wedge \cdot \vdash \sigma \equiv \rho_a^2 + \rho_b^2 : \star \\
 & & \quad \wedge r \equiv_\star [\![+]\!]\ \rho_a\ \rho_b \wedge \\
 & & \quad (e_a^1, e_a^2) \in \mathcal{C}(\mathcal{R}\ \rho_a) \wedge (e_b^1, e_b^2) \in \mathcal{C}(\mathcal{R}\ \rho_b)\ \}
\end{array}
$$

Fig. 10: Operations of type constructors on relations

The interpretation $[\![\mathcal{K}]\!]$ gives the relation that corresponds to constructor $\mathcal{K}$. This relation depends on the following definition, which extends a value relation to a relation between arbitrary well-typed terms.

2.5 Definition [Computational lifting]: The *computational lifting* of a relation $r \in \texttt{VRel}(\tau_1, \tau_2)$, 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 $(v_1, v_2) \in r$.

For integer and unit types, $[\![\texttt{int}]\!]$ and $[\![\texttt{()}]\!]$ give the identity value relations respectively on `int` and `()`. The operation $[\![\to]\!]$ lifts ρ and π to a new relation between functions that send related arguments in $\hat{\rho}$ to related results in $\hat{\pi}$. The operation $[\![\times]\!]$ lifts ρ and π to a relation between products such that the first components of the products belong in $\hat{\rho}$, and the second in $\hat{\pi}$. The operation $[\![+]\!]$ on ρ and π consists of all the pairs of left injections between elements of $\hat{\rho}$ and right injections between elements of $\hat{\pi}$. Because sums and products are call-by-name, their subcomponents must come from the computational lifting of the value relations. For the $\forall_\kappa$ constructor, since its kind is $(\kappa \to \star) \to \star$ we define $[\![\forall_\kappa]\!]$ to be a morphism that, given a $\mathtt{TyGRel}^{\kappa\to\star}$ argument ρ, returns the intersection over all well-formed π of the applications of $\hat{\rho}$ to π. The requirement that $\pi \in \mathtt{wfGRel}^{\kappa}$ is necessary to show that the interpretation of the $\forall_\kappa$ constructor is itself well-formed (Lemma 2.6).

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. 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 $\mathcal{R}\ (\tau, \sigma,\ r)$, the case for integer representations, r is required to be equal to $[\![\mathtt{int}]\!]$. In the case for unit representations, r is required to be equal 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. Note that the definition $\mathcal{R}$ is all that is required to extend the parametricity proof of F_ω to R_ω—representation types are a fairly isolated addition to this development.

Importantly, the interpretation of any constructor $\mathcal{K}$, including $\mathcal{R}$, is well-formed.

2.6 Lemma [Constructor interpretation is well-formed]: For all $\mathcal{K}$, $(\mathcal{K}, \mathcal{K}, [\![\mathcal{K}]\!]) \in \mathtt{wfGRel}^{kind(\mathcal{K})}$.

Proof

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

$$(\forall_\kappa, \forall_\kappa, [\![\forall_\kappa]\!]) \in \mathtt{wfGRel}^{(\kappa\to\star)\to\star}$$

Let us fix $\tau_1,\ \tau_2 \in \mathtt{ty}(\kappa \to \star)$, and a generalized relation $g_\tau \in \mathtt{GRel}^{\kappa\to\star}$, with $(\tau_1, \tau_2, g_\tau) \in \mathtt{wfGRel}^{\kappa\to\star}$, Then we know that:

$$\begin{array}{rcl} [\![\forall_\kappa]\!]\ (\tau_1, \tau_2, g_\tau) & = & \{(v_1, v_2) \mid \\ & & \quad \cdot \vdash v_1 : \forall_\kappa\ \tau_1\ \wedge\ \cdot \vdash v_2 : \forall_\kappa\ \tau_2 \wedge \\ & & \quad \text{for all } \rho \in \mathtt{TyGRel}^{\kappa} \\ & & \quad \rho \in \mathtt{wfGRel}^{\kappa} \Longrightarrow (v_1, v_2) \in (g_\tau\ \rho)\} \end{array}$$

which belongs in $\mathtt{wfGRel}^{\star}$ 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 $\sigma_1, \sigma_2 \in \mathtt{ty}(\kappa \to \star)$, and $g_\sigma \in \mathtt{GRel}^{\kappa\to\star}$, with $(\sigma_1, \sigma_2, g_\sigma) \in \mathtt{wfGRel}^{\kappa\to\star}$. Assume that $\cdot \vdash \tau_1 \equiv \sigma_1 : \kappa \to \star$, $\cdot \vdash \tau_2 \equiv \sigma_2 : \kappa \to \star$, and $g_\tau \equiv_{\kappa\to\star} g_\sigma$. Then we know that:

$$\begin{array}{rcl} [\![\forall_\kappa]\!]\ (\sigma_1, \sigma_2, g_\sigma) & = & \{(v_1, v_2) \mid \\ & & \quad \cdot \vdash v_1 : \forall_\kappa\ \sigma_1 \wedge\ \vdash v_2 : \forall_\kappa\ \sigma_2 \wedge \\ & & \quad \text{for all } \rho \in \mathtt{TyGRel}^{\kappa}, \\ & & \quad \rho \in \mathtt{wfGRel}^{\kappa} \Longrightarrow (v_1, v_2) \in (g_\sigma\ \rho)\} \end{array}$$

We need to show that

$$[\![\forall_\kappa]\!]\ (\tau_1, \tau_2, g_\tau) \equiv_\star [\![\forall_\kappa]\!]\ (\sigma_1, \sigma_2, g_\sigma)$$

To finish the case, using rule T-EQ to take care of the typing requirements, it is enough to show that, for any $\rho \in \texttt{TyGRel}^{\kappa}$, with $\rho \in \texttt{wfGRel}^{\kappa}$, we have $g_\tau\ \rho \equiv_\star g_\sigma\ \rho$. But this follows from reflexivity of $\equiv_\kappa$, and the fact that g_τ and g_σ are well-formed. □

We next show that the interpretation of types is well-formed. We must prove this result simultaneously with the fact that the interpretation of types gives equivalent results when given equal substitutions. We define equivalence for substitutions, $\delta_1 \equiv \delta_2$, pointwise. This result only holds for substitutions that map type variables to *well-formed* generalized relations.

2.7 Definition [Environment-respecting substitution]: We write $\delta \vDash \Gamma$ iff $\delta \in \texttt{Subst}_\Gamma$ and for every $a \in dom(\delta)$, it is the case that $\delta(a) \in \texttt{wfGRel}^{\kappa}$.

With this definition we can now state the lemma.

2.8 Lemma [Type interpretation is well-formed]: If $\Gamma \vdash \tau : \kappa$ then

1. for all $\delta \vDash \Gamma$, $(\delta^1\tau, \delta^2\tau, [\![\Gamma \vdash \tau : \kappa]\!]_\delta) \in \texttt{wfGRel}^{\kappa}$.
2. for all $\delta \vDash \Gamma$, $\delta' \vDash \Gamma$ such that $\delta \equiv \delta'$, it is the case that $[\![\Gamma \vdash \tau : \kappa]\!]_\delta \equiv_\kappa [\![\Gamma \vdash \tau : \kappa]\!]_{\delta'}$.

Proof
Straightforward induction over the type well-formedness derivations, appealing to Lemma 2.6. The only interesting case is the case for type abstractions, which follows from Lemma 2.3. □

Furthermore, the interpretation of types is compositional, in the sense that the interpretation of a type depends on the interpretation of its sub-terms. The proof of this lemma depends on the fact that type interpretations are well-formed.

2.9 Lemma [Compositionality]: Given an environment-respecting substitution, $\delta \vDash \Gamma$, a well-formed type with a free variable, $\Gamma, a{:}\kappa_a \vdash \tau : \kappa$, a type to substitute, $\Gamma \vdash \tau_a : \kappa_a$, and its interpretation, $r_a = [\![\Gamma \vdash \tau_a : \kappa_a]\!]_\delta$, it is the case that

$$[\![\Gamma, a{:}\kappa_a \vdash \tau : \kappa]\!]_{\delta, a \mapsto (\delta^1\tau_a, \delta^2\tau_a, r_a)} \equiv_\kappa [\![\Gamma \vdash \tau\{\tau_a/a\} : \kappa]\!]_\delta$$

Furthermore, our extensional definition of equality for Generalized relations means that it also preserves η-equivalence.

2.10 Lemma [Extensionality]: Given an environment-respecting $\delta \vDash \Gamma$, a well-formed type $\Gamma \vdash \tau : \kappa_1 \to \kappa_2$, and a fresh variable $a \# fv(\tau), \Gamma$, it is the case that

$$[\![\Gamma \vdash \lambda a{:}\kappa_1 . \tau\ a : \kappa_1 \to \kappa_2]\!]_\delta \equiv_{\kappa_1 \to \kappa_2} [\![\Gamma \vdash \tau : \kappa_1 \to \kappa_2]\!]_\delta$$

Proof
Unfolding the definitions we get that the left-hand side is the morphism

$$\lambda\rho \in \texttt{TyGRel}^{\kappa_1} \mapsto [\![\Gamma, a{:}\kappa_1 \vdash \tau : \kappa_2]\!]_{\delta, a \mapsto \rho}$$

Pick $\rho \in \texttt{wfGRel}^{\kappa_1}$. To finish the case we have to show that

$$[\![\Gamma, a{:}\kappa_1 \vdash \tau\ a : \kappa_2]\!]_{\delta, a \mapsto \rho} \equiv_{\kappa_2} [\![\Gamma \vdash \tau : \kappa_1 \to \kappa_2]\!]_\delta\ \rho$$

The left-hand side becomes

$$[\![\Gamma, a{:}\kappa_1 \vdash \tau : \kappa_1 \to \kappa_2]\!]_{\delta,a\mapsto\rho}\ (\rho^1, \rho^2, [\![\Gamma, a{:}\kappa_1 \vdash a : \kappa_1]\!]_{\delta,a\mapsto\rho})$$

which is equal to

$$[\![\Gamma, a{:}\kappa_1 \vdash \tau : \kappa_1 \to \kappa_2]\!]_{\delta,a\mapsto\rho}\ \rho$$

By a straightforward weakening property, this is equal (not just equivalent) to $[\![\Gamma \vdash \tau : \kappa_1 \to \kappa_2]\!]_\delta\ \rho$. Reflexivity of $\equiv_{\kappa_2}$ finishes the case. $\square$

Finally, we show that the interpretation of types respects the equivalence classes of types.

2.11 Theorem [Coherence]: If $\Gamma \vdash \tau_1 : \kappa$, $\delta \vDash \Gamma$, and $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$, then $[\![\Gamma \vdash \tau_1 : \kappa]\!]_\delta \equiv_\kappa [\![\Gamma \vdash \tau_2 : \kappa]\!]_\delta$.

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 2.9, the case for rule ETA follows from Lemma 2.10, 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

$$[\![\Gamma \vdash \tau_1\ \tau_3 : \kappa_2]\!]_\delta \equiv_{\kappa_2} [\![\Gamma \vdash \tau_2\ \tau_4 : \kappa_2]\!]_\delta$$

  Let

$$\begin{array}{rcl} r_1 & = & [\![\Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2]\!]_\delta \\ r_2 & = & [\![\Gamma \vdash \tau_2 : \kappa_1]\!]_\delta \\ r_3 & = & [\![\Gamma \vdash \tau_3 : \kappa_1 \to \kappa_2]\!]_\delta \\ r_4 & = & [\![\Gamma \vdash \tau_4 : \kappa_1]\!]_\delta \end{array}$$

  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 2.8, we have that:

$$\begin{array}{l} (\delta^1\tau_1, \delta^2\tau_1, r_1) \in \mathtt{wfGRel}^{\kappa_1 \to \kappa_2} \\ (\delta^1\tau_2, \delta^2\tau_2, r_2) \in \mathtt{wfGRel}^{\kappa_1} \\ (\delta^1\tau_3, \delta^2\tau_3, r_3) \in \mathtt{wfGRel}^{\kappa_1 \to \kappa_2} \\ (\delta^1\tau_4, \delta^2\tau_4, r_4) \in \mathtt{wfGRel}^{\kappa_1} \end{array}$$

  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

$$\Gamma \vdash \lambda a{:}\kappa_1 . \tau_1 \equiv \lambda a{:}\kappa_1 . \tau_2 : \kappa_1 \to \kappa_2$$

  given that $\Gamma, a{:}\kappa_1 \vdash \tau_1 \equiv \tau_2 : \kappa_2$. To show the required result let us pick $\rho \in$

$\texttt{TyGRel}^{\kappa_1}$ with $\rho \in \texttt{wfGRel}^{\kappa_1}$. Then for $\delta_a = \delta, a \mapsto \rho$, we have $\delta_a \vDash \Gamma, (a{:}\kappa_1)$, and hence by induction hypothesis we get:

$$[\![\Gamma, a{:}\kappa_1 \vdash \tau_1 : \kappa_2]\!]_{\delta_a} \equiv_{\kappa_2} [\![\Gamma, a{:}\kappa_1 \vdash \tau_2 : \kappa_2]\!]_{\delta_a}$$

and the case is finished. As a side note, the important condition that $\rho \in \texttt{wfGRel}^{\kappa_1}$ 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* $\rho \in \texttt{TyGRel}^{\kappa_1}$ 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.

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

To account for open terms, the theorem must be generalized in the standard manner.

If Γ is well-formed, and $\gamma \vDash \Gamma$ and $\Gamma \vdash e : \tau$ then $(\gamma^1 e, \gamma^2 e) \in \mathcal{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$.

Above, we extend the definition of substitutions to include also mappings of term variables to pairs of closed expressions.

$$\gamma, \delta := \cdot \mid \delta, (\tau \mapsto (\tau_1, \tau_2, r)) \mid \delta, (x \mapsto (e_1, e_2))$$

The definition of $\texttt{Subst}_\Gamma$ remains the same, but we add one more clause to $\gamma \vDash \Gamma$: for all x such that $\gamma(x) = (e_1, e_2)$, it is the case that $(e_1, e_2) \in \mathcal{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$ where $(x{:}\tau) \in \Gamma$. We write $\gamma^1(x)$, $\gamma^2(x)$ for the left and write projections of $\gamma(x)$, and extend this notation to arbitrary terms. For example, if $\gamma(x) = (e_1, e_2)$ then the term $\gamma^1((\lambda z\,.\,\lambda y\,.\,z)\ x\ x)$ is $(\lambda z\,.\,\lambda y\,.\,z)\ e_1\ e_1$ and $\gamma^2((\lambda z\,.\,\lambda y\,.\,z)\ x\ x)$ is $(\lambda z\,.\,\lambda y\,.\,z)\ e_2\ e_2$. A well-formed environment is one with disjoint domain of term and type variables, and where for all $(x{:}\tau) \in \Gamma$, $\Gamma \vdash \tau : \star$, so the above definition makes sense for well-formed environments.

We give a detailed sketch below of the proof of the abstraction theorem.

Proof
The proof proceeds by induction on the typing derivation, $\Gamma \vdash e : \tau$ with an inner induction for the case of `typerec` expressions. It crucially relies on Coherence (Theorem 2.11) for the case of rule T-EQ.

- Case INT. Straightforward.
- Case VAR. The result follows immediately from the fact that the environment is well-formed and the definition of $\gamma \vDash \Gamma$.
- Case ABS. In this case we have that $\Gamma \vdash \lambda x\,.\,e : \tau_1 \to \tau_2$ given that $\Gamma, (x{:}\tau_1) \vdash e : \tau_2$, and where we assume w.l.o.g that $x \# \Gamma, fv(\gamma)$. It suffices to show that $(\lambda x\,.\,\gamma^1 e, \lambda x\,.\,\gamma^2 e) \in [\![\Gamma \vdash \tau_1 \to \tau_2 : \star]\!]_\gamma$. To show this, let us pick $(e_1, e_2) \in [\![\Gamma \vdash \tau_1 : \star]\!]_\gamma$, it is then enough to show that

$$((\lambda x\,.\,\gamma^1 e)\ e_1, (\lambda x\,.\,\gamma^2 e)\ e_2) \in \mathcal{C}\,[\![\Gamma \vdash \tau_2 : \star]\!]_\gamma \tag{1}$$

But we can take $\gamma_0 = \gamma, (x \mapsto (e_1, e_2))$, which certainly satisfies $\gamma_0 \vDash \Gamma, (x{:}\tau_1)$ and by induction hypothesis: $(\gamma_0^1 e, \gamma_0^2 e) \in \mathcal{C}\,[\![\Gamma, (x{:}\tau_1) \vdash \tau_2 : \star]\!]_{\gamma_0}$. By an easy weakening lemma for term variables in the type interpretation we have that $(\gamma_0^1 e, \gamma_0^2 e) \in \mathcal{C}\,[\![\Gamma \vdash \tau_2 : \star]\!]_{\gamma}$ and by unfolding the definitions, equation (1) follows.

- Case APP. In this case we have that $\Gamma \vdash e_1\ e_2 : \tau$ given that $\Gamma \vdash e_1 : \sigma \to \tau$ and $\Gamma \vdash e_2 : \sigma$. By induction hypothesis,

$$\begin{aligned}(\gamma^1 e_1, \gamma^2 e_1) &\in \mathcal{C}\,[\![\Gamma \vdash \sigma \to \tau : \star]\!]_{\gamma} \qquad (2)\\ (\gamma^1 e_2, \gamma^2 e_2) &\in \mathcal{C}\,[\![\Gamma \vdash \sigma : \star]\!]_{\gamma} \qquad (3)\end{aligned}$$

From (2) we get that $\gamma^1 e_1 \Downarrow w_1$ and $\gamma^2 e_1 \Downarrow w_2$ such that $(w_1\ (\gamma^1 e_2), w_2\ (\gamma^2 e_2)) \in \mathcal{C}\,[\![\Gamma \vdash \tau : \star]\!]_{\gamma}$, where we made use of equation (3) and unfolded definitions. Hence, by the operational semantics for applications, we also have that: $((\gamma^1 e_1)\ (\gamma^1 e_2), (\gamma^2 e_1)\ (\gamma^2 e_2)) \in \mathcal{C}\,[\![\Gamma \vdash \tau : \star]\!]_{\gamma}$, as required.
- Case T-EQ. The case follows directly from appealing to the Coherence theorem 2.11.
- Case INST. In this case we have that $\Gamma \vdash e : \sigma\ \tau$, given that $\Gamma \vdash e : \forall_\kappa \sigma$ and $\Gamma \vdash \tau : \kappa$. By induction hypothesis we get that $(\gamma^1 e, \gamma^2 e) \in \mathcal{C}([\![\forall_\kappa]\!]\ \ (\gamma^1\sigma, \gamma^2\sigma, [\![\Gamma \vdash \sigma : \kappa \to \star]\!]_{\gamma}))$; hence by the definition of $[\![\forall_\kappa]\!]$ and by making use of the fact that $(\gamma^1\tau, \gamma^2\tau, [\![\Gamma \vdash \tau : \kappa]\!]_{\gamma}) \in \mathtt{wfGRel}^{\kappa}$ (by Lemma 2.8), we get that $\gamma^1 e \Downarrow v_1$ and $\gamma^2 e \Downarrow v_2$ such that

$$(v_1, v_2) \in [\![\Gamma \vdash \sigma : \kappa \to \star]\!]_{\gamma}\ \ (\gamma^1\tau, \gamma^2\tau, [\![\Gamma \vdash \tau : \kappa]\!]_{\gamma})$$

hence, $(v_1, v_2) \in [\![\Gamma \vdash \sigma\ \tau : \star]\!]_{\gamma}$ as required.
- Case GEN. We have that $\Gamma \vdash e : \forall_\kappa \sigma$, given that $\Gamma, (a{:}\kappa) \vdash e : \sigma\ a$ where $a \# \Gamma$, and we assume w.l.o.g. that $a \# ftv(\gamma)$ as well. We need to show that $(\gamma^1 e, \gamma^2 e) \in \mathcal{C}([\![\forall_\kappa]\!]\ \ (\gamma^1\sigma, \gamma^2\sigma, [\![\sigma]\!]_{\gamma})$. Hence we can fix $\rho \in \mathtt{TyGRel}^{\kappa}$ such that $\rho \in \mathtt{wfGRel}^{\kappa}$. We can form the substitution $\gamma_0 = \gamma, (a \mapsto \rho)$, for which it is easy to show that $\gamma_0 \vDash \Gamma, (a{:}\kappa)$. Then, by induction hypothesis $(\gamma_0^1 e, \gamma_0^2 e) \in \mathcal{C}\,[\![\Gamma, (a{:}\kappa) \vdash \sigma\ a : \star]\!]_{\gamma_0}$ which means $(\gamma_0^1 e, \gamma_0^2 e) \in \mathcal{C}\,[\![\Gamma, (a{:}\kappa) \vdash \sigma : \kappa \to \star]\!]_{\gamma_0}\ \ \rho$. By an easy weakening lemma this implies $(\gamma_0^1 e, \gamma_0^2 e) \in \mathcal{C}\,[\![\Gamma \vdash \sigma : \kappa \to \star]\!]_{\gamma}\ \ \rho$ and moreover since terms do not contain types $\gamma_0^i e = \gamma^i e$ and the case is finished.
- Case RINT. We have that $\Gamma \vdash \mathtt{R}_{\mathtt{int}} : \mathtt{R}\ \mathtt{int}$, hence $(\mathtt{R}_{\mathtt{int}}, \mathtt{R}_{\mathtt{int}}) \in \mathcal{R}\ (\mathtt{int}, \mathtt{int}, [\![\mathtt{int}]\!])$ by unfolding definitions.
- Case RUNIT. Similar to the case for RINT.
- Case RPROD. We have that $\Gamma \vdash \mathtt{R}_{\times}\ e_1\ e_2 : \mathtt{R}\ (\sigma_1 \times \sigma_2)$, given that $\Gamma \vdash e_1 : \mathtt{R}\ \sigma_1$ and $\Gamma \vdash e_2 : \mathtt{R}\ \sigma_2$. It suffices to show that $(\mathtt{R}_{\times}\ \gamma^1 e_1\ \gamma^1 e_2, \mathtt{R}_{\times}\ \gamma^2 e_1\ \gamma^2 e_2) \in \mathcal{R}\ (\gamma^1(\sigma_1 \times \sigma_2), \gamma^2(\sigma_1 \times \sigma_2), [\![\Gamma \vdash \sigma_1 \times \sigma_2 : \star]\!]_{\gamma})$. The result follows by taking as $\rho_a = (\gamma^1\sigma_1, \gamma^2\sigma_1, [\![\Gamma \vdash \sigma_1 : \star]\!]_{\gamma})$, $\rho_b = (\gamma^1\sigma_2, \gamma^2\sigma_2, [\![\Gamma \vdash \sigma_2 : \star]\!]_{\gamma}$. By Lemma 2.8, regularity and inversion on the kinding relation, one can show that ρ_a and ρ_b are well-formed and hence to finish the case we only need to show that $(\gamma^1 e_1, \gamma^2 e_1) \in \mathcal{C}(\mathcal{R}\ \rho_a)$ and $(\gamma^1 e_2, \gamma^2 e_2) \in \mathcal{C}(\mathcal{R}\ \rho_b)$, which follow by induction hypotheses for the typing of e_1 and e_2.

- Case RSUM. Similar to the case for RPROD.
- Case TREC. This is really the only interesting case. After we decompose the premises and get the induction hypotheses, we proceed with an inner induction on the type of the scrutinee. In this case we have that:

$$\Gamma \vdash \texttt{typerec}\ e\ \texttt{of}\ \{e_{\texttt{int}}\ ;\ e_{()}\ ;\ e_{\times}\ ;\ e_{+}\} : \sigma\ \tau$$

Let us introduce some abbreviations:

$$\begin{array}{rcl} u[e] & = & \texttt{typerec}\ e\ \texttt{of}\ \{e_{\texttt{int}}\ ;\ e_{()}\ ;\ e_{\times}\ ;\ e_{+}\} \\ \sigma_{\times} & = & \forall(a{:}\star)(b{:}\star)\texttt{.R}\ a \to \sigma\ a \to \\ & & \qquad \texttt{R}\ b \to \sigma\ b \to \sigma\ (a \times b) \\ \sigma_{+} & = & \forall(a{:}\star)(b{:}\star)\texttt{.R}\ a \to \sigma\ a \to \\ & & \qquad \texttt{R}\ b \to \sigma\ b \to \sigma\ (a + b) \end{array}$$

By the premises of the rule we have:

$$\Gamma \vdash \sigma : \star \to \star \tag{4}$$
$$\Gamma \vdash e : \texttt{R}\ \tau \tag{5}$$
$$\Gamma \vdash e_{\texttt{int}} : \sigma\ \texttt{int} \tag{6}$$
$$\Gamma \vdash e_{()} : \sigma\ \texttt{()} \tag{7}$$
$$\Gamma \vdash e_{\times} : \sigma_{\times} \tag{8}$$
$$\Gamma \vdash e_{+} : \sigma_{+} \tag{9}$$

We also know the corresponding induction hypotheses for (6),(7),(8), (9). We now show that:

$$\begin{array}{l} \forall e_1\ e_2\ \rho \in \texttt{TyGRel}^{\star}, \tau_1 \in \texttt{ty}(\star)\ \tau_2 \in \texttt{ty}(\star)\ r, \\ \rho \in \texttt{wfGRel}^{\star} \wedge (e_1, e_2) \in \mathcal{C}(\mathcal{R}\ \rho) \\ \Longrightarrow (\gamma^1 u[e_1], \gamma^2 u[e_2]) \in \mathcal{C}([\![\Gamma \vdash \sigma : \star \to \star]\!]_{\gamma}\ \rho) \end{array}$$

by introducing our assumptions, and performing inner induction on the size of the normal form of τ_1. Let us call this property for fixed e_1, e_2, ρ, $\mathsf{INNER}(e_1, e_2, \rho)$. We have that $(e_1, e_2) \in \mathcal{C}(\mathcal{R}\ \rho)$ and hence we know that $e_1 \Downarrow w_1$ and $e_2 \Downarrow w_2$, such that:

$$(w_1, w_2) \in \mathcal{R}\ \rho$$

We then have the following cases to consider by the definition of $\mathcal{R}$:

— $w_1 = w_2 = \texttt{R}_{\texttt{int}}$ and $\rho \equiv (\texttt{int}, \texttt{int}, [\![\texttt{int}]\!])$. In this case, $\gamma^1 u \Downarrow w_1$ such that $\gamma^1 e_{\texttt{int}} \Downarrow w_1$ and similarly $\gamma^2 u \Downarrow w_2$ such that $\gamma^2 e_{\texttt{int}} \Downarrow w_2$, and hence it is enough to show that: $(\gamma^1 e_{\texttt{int}}, \gamma^2 e_{\texttt{int}}) \in \mathcal{C}([\![\Gamma \vdash \sigma : \star \to \star]\!]_{\gamma}\ \rho)$. From the outer induction hypothesis for (6) we get that: $(\gamma^1 e_{\texttt{int}}, \gamma^2 e_{\texttt{int}}) \in \mathcal{C}\ [\![\Gamma \vdash \sigma\ \texttt{int} : \star]\!]_{\gamma}$ And we have that:

$$\begin{array}{rl} [\![\Gamma \vdash \sigma\ \texttt{int} : \star]\!]_{\gamma} & = \\ [\![\Gamma \vdash \sigma : \star \to \star]\!]_{\gamma}\ (\texttt{int}, \texttt{int}, [\![int]\!]) & \equiv_{\star} \\ [\![\Gamma \vdash \sigma : \star \to \star]\!]_{\gamma}\ \rho & \end{array}$$

where we have made use of the properties of well-formed generalized relations to substitute equivalent types and relations in the middle step.

— $w_1 = w_2 = ()$ and $[\![\Gamma \vdash \tau : \star]\!]_\gamma \equiv_\star [\![\texttt{()}]\!]$. The case is similar to the previous case.

— $w_1 = \mathtt{R}_\times\ e_a^1\ e_a^2$ and $w_2 = \mathtt{R}_\times\ e_b^1\ e_b^2$, such that there exist ρ_a^1 and ρ_a^2, well-formed, such that

$$\rho \equiv_\star ((\rho_a^1 \times \rho_b^1), (\rho_a^2 \times \rho_b^2), [\![\times]\!]\ \ \rho_a\ \rho_b \tag{10}$$
$$(e_a^1, e_a^2) \in \mathcal{C}(\mathcal{R}\ \rho_a) \tag{11}$$
$$(e_b^1, e_b^2) \in \mathcal{C}(\mathcal{R}\ \rho_b) \tag{12}$$

In this case we know that $\gamma^1 u[e_1] \Downarrow w_i$ and $\gamma^2 u[e_2] \Downarrow w_2$ where

$$(\gamma^1 e_\times)\ e_a^1\ (\gamma^1 u[e_a^1])\ e_b^1\ (\gamma^1 u[e_b^1]) \Downarrow w_1$$
$$(\gamma^2 e_\times)\ e_a^2\ (\gamma^2 u[e_a^2])\ e_b^2\ (\gamma^2 u[e_b^2]) \Downarrow w_2$$

By the outer induction hypothesis for (8) we will be done, as before, if we instantiate with relations r_a and r_b for the quantified variables a and b, respectively. But we need to show that, for $\gamma_0 = \gamma, (a \mapsto \rho_a), (b \mapsto \rho_b)$, $\Gamma_0 = \Gamma, (a{:}\star), (b{:}\star)$, we have:

$$(\gamma^1 u[e_a^1], \gamma^2 u[e_a^2]) \in \mathcal{C}\ [\![\Gamma_0 \vdash \sigma\ a : \star]\!]_{\gamma_0} \tag{13}$$
$$(\gamma^1 u[e_b^1], \gamma^2 u[e_b^2]) \in \mathcal{C}\ [\![\Gamma_0 \vdash \sigma\ b : \star]\!]_{\gamma_0} \tag{14}$$

But notice that the size of the normal form of τ_a^1 must be less than the size of the normal form of τ_1, and similarly for τ_b^1 and τ_b, and hence we can apply the (inner) induction hypothesis for (11) and (12). From these, compositionality, and an easy weakening‘ lemma, we have that (13) and (14) follow. By the outer induction hypothesis for (8) we then finally have that:

$$(w_1, w_2) \in [\![\Gamma, (a{:}\star), (b{:}\star) \vdash \sigma\ (a \times b) : \star]\!]_{\gamma_0}$$

which gives us the desired $(w_1, w_2) \in [\![\Gamma \vdash \sigma : \star \to \star]\!]_\gamma\ \ \rho$ by appealing to the properties of well-formed generalized relations.

We now have by the induction hypothesis for (5), that $(\gamma^1 e, \gamma^2 e) \in \mathcal{C}(\mathcal{R}\ (\gamma^1\tau, \gamma^2\tau, [\![\Gamma \vdash \tau : \star]\!]_\gamma))$, and hence we can get

$$\mathsf{INNER}(\gamma^1 e, \gamma^2 e, (\gamma^1\tau, \gamma^2\tau, [\![\Gamma \vdash \tau : \star]\!]_\gamma)),$$

which gives us that:

$$(\gamma^1 u[e], \gamma^2 u[e]) \in \mathcal{C}([\![\Gamma \vdash \sigma : \star \to \star]\!]_\gamma\ \ (\gamma^1\tau, \gamma^2\tau, [\![\Gamma \vdash \tau : \star]\!]_\gamma)),$$

or $(\gamma^1 u[e], \gamma^2 u[e]) \in \mathcal{C}([\![\Gamma \vdash \sigma\ \tau : \star]\!]_\gamma)$, as required.

□

Incidentally, this statement of the abstraction theorem shows that all well-typed expressions of R_ω 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.

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_ω type.

2.13 Example [Theorem for $\forall c{:}\star \to \star.c\ () \to c\ ()$]: Any e with type $\forall c{:}\star \to \star.c\ () \to c\ ()$ 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 $\cdot \vdash u : \tau_c\ ()$, $e\ u \Downarrow u$.

Proof

Assume that $\cdot \vdash e : \forall c{:}\star \to \star.c\ () \to c\ ()$. Then by Theorem 2.12 we have: $(e, e) \in \mathcal{C}\,[\![\cdot \vdash \forall c{:}\star \to \star.c\ () \to c\ () : \star]\!]$. By expanding the definition of the interpretation, for any $\rho_c \in \mathtt{wfGRel}^{\star\to\star}$, and $(e_1, e_2) \in \mathcal{C}\,[\![c{:}\star \to \star \vdash c\ () : \star]\!]_{c\mapsto\rho_c}$, it is the case that:

$$(e\ e_1, e\ e_2) \in \mathcal{C}\,[\![c{:}\star \to \star \vdash c\ () : \star]\!]_{c\mapsto\rho_c} \tag{15}$$

We can now pick $\rho_c = (\tau_c, \tau_c, f_c)$ where:

$$f_c\ (\tau, \sigma, \_) \quad \triangleq \quad \begin{array}{l} \text{if } (\cdot \vdash \tau \equiv () : \star \wedge \cdot \vdash \sigma \equiv () : \star) \\ \text{then } \{(v, u) \mid \cdot \vdash v : \tau_c\ ()\} \text{ else } \emptyset \end{array}$$

Intuitively, the morphism f_c returns the graph of a constant function that always returns u when called with type arguments equivalent to $()$, and the empty relation otherwise. It is straightforward to see that $(\tau_c, \tau_c, f_c) \in \mathtt{wfGRel}^{\star\to\star}$. Therefore

$$[\![c{:}\star \to \star \vdash c\ () : \star]\!]_{c\mapsto(\tau_c,\tau_c,f_c)} = \{(v, u) \mid \cdot \vdash v : \tau_c\ ()\}$$

Because (u, u) is in this set, we can pick e_1 and e_2 both to be u and use (15) to show that $e\ e_2 \Downarrow u$, hence $e\ u \Downarrow u$ as required. $\square$

As a side-remark, notice that our choice for the morphism f_c is not unique. Another proof of the same theorem could simply use the singleton relation $\{(u, u)\}$ instead of the graph of the constant function that always returns u.

We observe that to derive our result we had to instantiate a generalized relation to be a morphism that is itself not representable in F_ω. In particular, this morphism is not parametric: it behaves differently at type $()$ than at other types. Hence, despite the fact that we are discussing a theorem for an F_ω type, we needed morphisms at higher kinds to accept *both types and morphisms* as arguments. 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 correctness of generic cast. The R_ω type for generic cast is:

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

The abstraction theorem for this type follows. Assume that, $\rho_a \in \mathtt{wfGRel}^*$, $\rho_b \in \mathtt{wfGRel}^*$, and $\rho_c \in \mathtt{wfGRel}^{*\to *}$. Moreover, assume that:

$$\begin{array}{rcl} \Gamma & = & (a{:}\star), (b{:}\star), (c{:}\star \to \star) \\ \delta & = & a \mapsto \rho_a, b \mapsto \rho_b, c \mapsto \rho_c \\ (e^1_{ra}, e^2_{ra}) & \in & \mathcal{C}\,[\![\Gamma \vdash \mathtt{R}\; a : \star]\!]_\delta \\ (e^1_{rb}, e^2_{rb}) & \in & \mathcal{C}\,[\![\Gamma \vdash \mathtt{R}\; b : \star]\!]_\delta \end{array}$$

Then, either the cast fails and

$$\begin{array}{l} gcast\; e^1_{ra}\; e^1_{rb} \Downarrow \mathtt{inl}\; e'_1 \wedge \\ gcast\; e^2_{ra}\; e^2_{rb} \Downarrow \mathtt{inl}\; e'_2 \wedge e'_1 \Downarrow () \wedge e'_2 \Downarrow () \end{array}$$

or the cast succeeds and

$$\begin{array}{l} gcast\; e^1_{ra}\; e^1_{rb} \Downarrow \mathtt{inr}\; e'_1 \wedge gcast\; e^2_{ra}\; e^2_{rb} \Downarrow \mathtt{inr}\; e'_2 \wedge \\ \quad \text{for all } (e_1, e_2) \in \mathcal{C}(\hat{\rho}_c\; \rho_a),\ (e'_1\; e_1, e'_2\; e_2) \in \mathcal{C}(\hat{\rho}_c\; \rho_b) \end{array}$$

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* $\tau_c \in \mathtt{ty}(\star \to \star)$, it is the case that $\cdot \vdash gcast\; e_{ra}\; e_{rb} : \mathtt{()}{+}(\tau_c\; \tau_a \to \tau_c\; \tau_b)$. Assume by contradiction now that $\cdot \not\vdash \tau_a \equiv \tau_b : \star$. Then we instantiate the abstraction theorem with $e^1_{ra} = e^2_{ra} = e_{ra}$, $e^1_{rb} = e^2_{rb} = e_{rb}$, $\rho_a = (\tau_a, \tau_a, [\![\cdot \vdash \tau_a : \star]\!]_\cdot)$, $\rho_b = (\tau_b, \tau_b, [\![\cdot \vdash \tau_b : \star]\!]_\cdot)$ and $\rho_c = (\lambda a{:}\star\,.\,\mathtt{()}, \lambda a{:}\star\,.\,\mathtt{()}, f_c)$ where

$$\begin{array}{rcl} f_c\; (\tau,\ \sigma,\ r) & = & \text{if } (\cdot \vdash \tau \equiv \tau_a : \star \wedge \cdot \vdash \sigma \equiv \tau_a : \star) \\ & & \text{then } [\![\cdot \vdash (\lambda a{:}\star\,.\,\mathtt{()})\; \tau_a : \star]\!]_\cdot \text{ else } \emptyset \end{array}$$

One can confirm that $\rho_c \in \mathtt{wfGRel}^{\star\to\star}$ Moreover $(e_{ra}, e_{ra}) \in \mathcal{C}(\mathcal{R}\; \rho_a)$ by the abstraction theorem, and similarly $(e_{rb}, e_{rb}) \in \mathcal{C}(\mathcal{R}\; \rho_b)$. Then by the free theorem for *gcast* above we know that, since $((), ()) \in \mathcal{C}(f_c\; \rho_a)$, we have $(e\; (), e\; ()) \in \mathcal{C}(f_c\; \rho_b)$ (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 $\mathcal{C}(f_c\; \rho_b) = \emptyset$, a contradiction. $\Box$

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—it may well return (). An implementation of *gcast* may always fail for any pair of arguments and still be well typed.

3.2 Lemma [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 $\rho_c = (\tau_c, \tau_c, f_c)$ where f_c is:

$$\begin{array}{lcl} f_c\ (\tau,\ \sigma,\ r) & = & \text{if } (\cdot \vdash \tau \equiv \tau_a : \star \wedge \cdot \vdash \sigma \equiv \tau_a : \star) \\ & & \text{then } \{(v, w) \mid \cdot \vdash v : \tau_c\ \tau_a\} \text{ else } \emptyset \end{array}$$

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:

$$gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e_1' \tag{16}$$

$$gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e_2' \tag{17}$$

$$\forall (e_1, e_2) \in \mathcal{C}(f_c\ \rho_a), (e_1'\ e_1, e_2'\ e_2) \in \mathcal{C}(f_c\ \rho_b) \tag{18}$$

Because of the particular choice for f_c we know that $(e_a, e_a) \in \mathcal{C}(f_c\ \rho_a)$. From determinacy of evaluation and equations (16) and (17) we get that $e_1' = e_2' = e$. Then, from (18) we get that $(e\ e_a, e\ e_a) \in \mathcal{C}(f_c\ \rho_b)$, 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:

$$\begin{array}{lcl} f\ (\mathtt{()},\ \mathtt{()},\ \_) & = & \{(v, v) \mid \cdot \vdash v : \tau_c\ \mathtt{()}\} \\ f\ (\_,\ \_,\ \_) & = & \emptyset \end{array}$$

Let $\tau_c = \lambda c{:}\star\,.\,c$. We instantiate the type of *gcast* as follows: we instantiate c with $\rho_c = (\tau_c, \tau_c, f)$, a with $\rho_a = (\mathtt{()}, \mathtt{()}, [\![\mathtt{()}]\!])$, and b with $\rho_b = ((\lambda d{:}\star\,.\,d)\ \mathtt{()}, \mathtt{()}, [\![\mathtt{()}]\!])$. The important detail is that although f can take any relation r such that $\mathtt{wfGRel}^{\star}(\alpha_1, \alpha_2, r)$ to a relation s that satisfies $\mathtt{wfGRel}^{\star}(\tau_c\ \alpha_1, \tau_c\ \alpha_2, s)$, it can return different results for *equivalent but syntactically different type arguments*. In particular, the instantiation of b involves a type not syntactically equal to `()`. 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, e_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 (i.e. a subset of 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 (i.e. contains contextual equivalence). A potential solution to this problem would start by modifying the clauses of the definition that correspond to sums (such as the $[\![+]\!]$ and $\mathcal{R}$ operations) by $\top\top$-closing them as Pitts suggests (**?**; **?**). 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 $\star$ (but this is the only part of `wfGRel` that needs to be modified). Although we conjecture that this approach achieves completeness with respect to contextual equivalence, adding $\top\top$-closures is typically a heavy technical undertaking (but probably not hiding surprises, if one follows Pitt's roadmap) and we have not yet carried out the experiment.

4.2 Parametricity, representations, 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, consider what would happen if we added the representation $\mathtt{R_{id}}$ of type $\mathtt{R}\ \mathit{Rid}$ to R_ω, and extended `typerec` and *gcast* accordingly, where *Rid* abbreviates the type $(\forall (a{:}\star)\,\mathtt{.R}\ a \to a \to a)$. Then we could encode an infinite loop in R_ω, based on an example by Harper and Mitchell (1999) which in turn uses Girard's J operator. This example begins by using *gcast* to enable a self-application term with a concise type.

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

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

$$\mathit{delta}\ \mathtt{R_{id}}\ \mathit{delta} \approx (\lambda x\,.\,x\ \mathtt{R_{id}}\ x)\ \mathit{delta} \approx \mathit{delta}\ \mathtt{R_{id}}\ \mathit{delta}$$

This example demonstrates that we cannot extend the relational interpretation to $\mathtt{R_{id}}$ and the proof of the abstraction theorem in a straightforward manner as our proof implies termination. That does not mean that we cannot give any relational interpretation to $\mathtt{R_{id}}$, only that our proof would have to change significantly. Recent work by Neic *et al.* (**?**) gives a way to reconcile Girard's J operator and parametricity.

Our current proof breaks in the definition of the morphism $\mathcal{R}$ in Figure 10. The application $\mathcal{R}\ (\tau,\ \sigma,\ r)$ 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 $\mathtt{R_{id}}$, we must restrict r in a similar way. To do so, it is tempting to add:

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

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

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 behavior 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 behavior cannot happen when we only include representations of types such as $\mathtt{R}(\mathtt{R\ int})$, $\forall a.a \rightarrow a$, $\forall a.a \rightarrow \mathtt{R\ int} \rightarrow a$, or even $\forall a.a$. We can still give a definition of $\mathcal{R}$ that calls recursively the main interpretation function, but the definition must be shown well-founded using a more elaborate metric on types.

4.3 ***Encoding*** R_ω

Did we really need to go to R_ω to get this result? Weirich (2001) previously showed how to encode a simplified version of representation type in F_ω. In fact, the result of this current paper does not even rely on the the free theorem for representation types at all. However, extending the F_ω proof to R_ω only requires local changes. Furthermore, this proof is more general than the encoding. Weirich's encoding of representation types is limited: it permits only iteration as the elimination operation instead of primitive recursion (Spławski & Urzyczyn, 1999) and does not extend to the inclusion of self-representation (i.e. a representation $\mathtt{R\_R}$ of type $\forall (a : \star)\ .\ \mathtt{R}a \rightarrow \mathtt{R}(\mathtt{R}a)$.) As the discussion above demonstrates, our definitions here separate the issues of encoding representations from their interpretations.

4.4 Injectivity of type equalities

Higher-order types may encode type equalities – the type $\forall c.\, c\ \tau_1 \to c\ \tau_2$ is inhabited iff $\tau_1 \equiv \tau_2$. However, not all properties of type equalities seem to be expressible as R_ω or F_ω terms. For instance the term *inj* below could witness the injectivity of products:

$$inj : \forall ab.\, (\forall c.\, c\ (a \times \mathtt{Int}) \to c\ (b \times \mathtt{Int})) \to (\forall c.\, c\ a \to c\ b)$$

However, it it not easy to construct such a term in F_ω or R_ω. On the other hand, proving that such a type is uninhabited (using the relational semantics in this paper) is not straightforward either. The typical way one would prove this would be by assuming the existence of a term *inj* and deriving that $(inj, inj) \in \emptyset$ by using the fundamental theorem for *inj*. This approach however can't work since we would have to apply *inj* to arguments that are in the interpretation of $\forall c.\, c\ (a \times \mathtt{Int}) \to c\ (b \times \mathtt{Int}))$ – and such arguments exist *only* if a and b are instantiated to the same type and use the same relations. In this case we can show that the term is inhabited. But in the case where a and b are instantiated to different types, the fundamental theorem is of no use. It seems that, although the types $\forall c.\, c\ (\tau_1 \times \mathtt{Int}) \to c\ (\tau_2 \times \mathtt{Int})$ where τ_1 is not equivalent to τ_2, and $\forall c.\, c\ \tau_1 \to c\ \tau_2$ are both interpreted as the empty set, it is not the case that we can construct coercions from the first to the second. We conjecture that this is in contrast to System F, where any two uninhabited types *can* be coerced to each other *inside* System F. We leave it as future work to address the aforementioned issues.

5 Related work.

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 (Meijer & Hutton, 1995), it can be found in few sources in the literature.

Kŭcan (1997) 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 different: he shows the correspondence between free theorems obtained directly from algebraic datatype signatures and those derived from Church encodings.

Gallier gives a detailed formalization (Gallier, 1990) 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. Our work was developed in CIC instead of untyped set theory, but there are similarities. In particular, our inductive definition of $\mathtt{GRel}^\kappa$, corresponds to his definition of (generalized) candidate sets. The important requirement that the generalized morphisms respect equivalence classes of types ($\mathtt{wfGRel}^\kappa$) is also present in his formalization (Definition 16.2, Condition (4)). However, because Gallier is working in set theory, he includes no explicit account of what equality means, and omits the extra complication that it must be given simultaneously with the definition of $\texttt{wfGRel}^{\kappa}$.

A logic for reasoning about parametricity, that extends the Abadi-Plotkin logic (Plotkin & Abadi, 1993) to the λ-cube has been proposed in a manuscript by Takeuti (Takeuti, 2001). Crole presents in his book (Crole, 1994) a categorical interpretation of higher-order polymorphic types, which could presumably be instantiated to the concrete syntactic relations used here.

Concerning the interpretation of representation types, this paper extends the ideas developed in previous work by the authors (Vytiniotis & Weirich, 2007) 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 (2002). 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 behavior 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 (Washburn & Weirich, 2005), 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 (Harper & Pfenning, 2005)) 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.

6 Future work and conclusions

In order for the technique in this paper to evolve to a reasoning technique for Haskell, several limitations need to be addressed. If we wished to use these results to reason about Haskell implementations of `gcast`, we must extend our model to include more—in particular, general recursion and recursive types (Melliès & Vouillon, 2005; Johann & Voigtländer, 2004; Appel & McAllester, 2001; Ahmed, 2006; Crary & Harper, 2007). We believe that the techniques developed here are independent of those for advanced language features.

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 behavior 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 (Lämmel & Peyton Jones, 2005; Holdermans *etal.*, 2006; Weirich, 2006a). However, techniques that allow this behavior cannot define `gcast`. As a result, generic programming libraries that depend on `gcast` (Lämmel & Peyton Jones, 2003) implement it as a language extension, not directly in Haskell.

Conclusions. 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 correctness of generic cast from its type. In conclusion, this paper demonstrates that parametric reasoning is possible in the representation-based approach to generic programming.

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

References

Achten, Peter, van Eekelen, Marko C. J. D., & Plasmeijer, Marinus J. (2004). Compositional model-views with generic graphical user interfaces. *Pages 39–55 of: Practical aspects of declarative languages, 6th international symposium, padl 2004, dallas, tx, usa, june 18-19, 2004, proceedings.*

Ahmed, Amal J. (2006). Step-indexed syntactic logical relations for recursive and quantified types. *Pages 69–83 of:* Sestoft, Peter (ed), *Esop.* Lecture Notes in Computer Science, vol. 3924. Springer.

Appel, Andrew W., & McAllester, David. (2001). An indexed model of recursive types for foundational proof-carrying code. *Acm trans. program. lang. syst.*, **23**(5), 657–683.

Baars, Arthur I., & Swierstra, S. Doaitse. (2002). Typing dynamic typing. *Pages 157–166 of: Icfp '02: Proceedings of the seventh acm sigplan international conference on functional programming.* New York, NY, USA: ACM Press.

Cheney, James. (2005). Scrap your nameplate: (functional pearl). *Pages 180–191 of: Icfp '05: Proceedings of the tenth acm sigplan international conference on functional programming.* New York, NY, USA: ACM Press.

Cheney, James, & Hinze, Ralf. (2002). A lightweight implementation of generics and dynamics. *Pages 90–104 of: Haskell '02: Proceedings of the 2002 acm sigplan workshop on haskell.* New York, NY, USA: ACM Press.

Cheney, James, & Hinze, Ralf. (2003). *First-class phantom types.* CUCIS TR2003-1901. Cornell University.

Clarke, Dave, Hinze, Ralf, Jeuring, Johan, Löh, Andres, & de Wit, Jan. (2001). *The Generic Haskell user's guide.* Tech. rept. UU-CS-2001-26. Utrecht University.

Crary, Karl, & Harper, Robert. (2007). Syntactic logical relations for polymorphic and recursive types. *Electronic notes in theoretical computer science.* (To appear.).

Crary, Karl, Weirich, Stephanie, & Morrisett, Greg. (2002). Intensional polymorphism in type erasure semantics. *Journal of functional programming*, **12**(6), 567–600.

Crole, Roy. (1994). *Categories for types.* Cambridge University Press.

Gallier, Jean H. (1990). On Girard's "Candidats de Reductibilité". *Pages 123–203 of:*

Odifreddi, P. (ed), *Logic and computer science*. The APIC Series, vol. 31. Academic Press.

Girard, Jean-Yves. (1972). *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII.

Harper, Robert, & Mitchell, John C. (1999). Parametricity and variants of Girard's J operator. *Inf. process. lett.*, **70**(1), 1–5.

Harper, Robert, & Pfenning, Frank. (2005). On equivalence and canonical forms in the LF type theory. *Acm trans. comput. logic*, **6**(1), 61–101.

Hinze, Ralf. (2002). Polytypic values possess polykinded types. *Science of computer programming*, **43**(2–3), 129–159. MPC Special Issue.

Holdermans, Stefan, Jeuring, Johan, Löh, Andres, & Rodriguez, Alexey. (2006). Generic views on data types. *Mathematics of program construction, 8th international conference, mpc 2006, kuressaare, estonia, july 3-5, 2006, proceedings*. Lecture Notes in Computer Science, vol. 4014. Springer.

Johann, Patricia, & Voigtländer, Janis. (2004). Free theorems in the presence of seq. *Sigplan not.*, **39**(1), 99–110.

Kučan, Jacov. 1997 (February). *Metatheorems about convertibility in typed lambda calculi: Applications to cps transform and free theorems*. Ph.D. thesis, Massachusetts Institute of Technology.

Lämmel, Ralf. (2007). Scrap your boilerplate with XPath-like combinators. *Pages 137–142 of: Popl '07: Proceedings of the 34th annual acm sigplan-sigact symposium on principles of programming languages*. New York, NY, USA: ACM Press.

Lämmel, Ralf, & Peyton Jones, Simon. (2003). Scrap your boilerplate: a practical design pattern for generic programming. *Proc. of the acm sigplan workshop on types in language design and implementation (tldi 2003)*.

Lämmel, Ralf, & Peyton Jones, Simon. (2005). Scrap your boilerplate with class: extensible generic functions. *Pages 204–215 of: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*. ACM Press.

Meijer, Erik, & Hutton, Graham. 1995 (June). Bananas in space: Extending fold and unfold to exponential types. *Pages 324–333 of: Fpca95: Conference on functional programming languages and computer architecture*.

Melliès, Paul-André, & Vouillon, Jérôme. (2005). Recursive polymorphic types and parametricity in an operational framework. *Pages 82–91 of: Lics '05: Proceedings of the 20th annual ieee symposium on logic in computer science (lics' 05)*. Washington, DC, USA: IEEE Computer Society.

Paulin-Mohring, Christine. (1993). Inductive definitions in the system Coq: Rules and properties. *Pages 328–345 of: International conference on typed lambda calculi and applications, tlca '93*. Lecture Notes in Computer Science, vol. 664. Springer.

Plotkin, Gordon, & Abadi, Martín. (1993). A logic for parametric polymorphism. *Pages 361–375 of: International conference on typed lambda calculi and applications*.

Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of: Information processing '83*. North-Holland. Proceedings of the IFIP 9th World Computer Congress.

Sheard, Tim, & Pasalic, Emir. 2004 (July). Meta-programming with built-in type equality. *Pages 106–124 of: Proc 4th international workshop on logical frameworks and meta-languages (lfm'04), cork*.

Spławski, Zdzisław, & Urzyczyn, Paweł. 1999 (Sept.). Type fixpoints: Iteration vs. recursion. *Pages 102–113 of: Fourth ACM SIGPLAN international conference on functional programming*.

Takeuti, Izumi. 2001 (June). *The theory of parametricity in lambda cube (towards new interaction between category theory and proof theory).* Tech. rept. Kyoto University Research Information Repository.

Vytiniotis, Dimitrios, & Weirich, Stephanie. (2007). Free theorems and runtime type representations. *Electron. notes theor. comput. sci.*, **173**, 357–373.

Wadler, Philip. 1989 (Sept.). Theorems for free! *Pages 347–359 of: Fpca89: Conference on functional programming languages and computer architecture.*

Washburn, Geoffrey, & Weirich, Stephanie. (2005). Generalizing parametricity using information flow. *Pages 62–71 of: The twentieth annual ieee symposium on logic in computer science (lics 2005).* Chicago, IL: IEEE Computer Society Press, for IEEE Computer Society.

Weirich, Stephanie. 2001 (Apr.). Encoding intensional type analysis. *Pages 92–106 of:* Sands, D. (ed), *10th european symposium on programming.*

Weirich, Stephanie. (2004). Type-safe cast. *Journal of functional programming*, **14**(6), 681–695.

Weirich, Stephanie. 2006a (Sept.). RepLib: A library for derivable type classes. *Pages 1–12 of: Haskell workshop.*

Weirich, Stephanie. (2006b). Type-safe run-time polytypic programming. *J. funct. program.*, **16**(6), 681–710.

A Generalized relations, in Coq

A Coq definition of `GRel`, `wfGRel`, and `eqGRel` ($\equiv_\kappa$), follows.[4] First, we assume datatypes that encode R_ω syntax, such as `kind`, `term`, `type`, and `env`. Moreover we assume constants such as `ty_app` (for type applications) and `empty` (for empty environments).

```
(* R-omega kinds (Fig. 3) *)
Inductive kind : Set :=
| KStar : kind
| KFun  : kind -> kind -> kind.

(* R-omega types and a constant for type applications *)
Parameter type : Set.
Parameter term : Set.

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

(* R-omega judgments *)
Parameter kinding : env -> type -> kind -> Prop.
Parameter typing  : env -> term -> type -> Prop.
Parameter teq     : env -> type -> type -> kind -> Prop.
Parameter value : term -> Prop.

(* Definition and operations on closed types *)
Definition ty (k: kind) : Set := { t : type & kinding empty t k }.
Parameter ty_app : forall k1 k2, ty (KFun k1 k2) -> ty k1 -> ty k2.
Parameter ty_eq  : forall k, ty k -> ty k -> Prop.

(* closed terms *)
Parameter tm : (ty KStar) -> term -> Prop.
Parameter typing_eq : forall (t1 t2 : ty KStar) e, ty_eq t1 t2 -> tm t1 e -> tm t2 e.
```

Term relations are represented with the datatype `rel`, for which we give an equality predicate `eq_rel`. The definition `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`.

```
(* Relations over terms *)
Definition rel : Type := term -> term -> Prop.
Definition eq_rel (r1 : rel) (r2 : rel) :=
```

[4] These definitions are valid in Coq 8.1 with implicit arguments set.

```
    forall e1 e2, r1 e1 e2 <-> r2 e1 e2.

(* Value relations as a predicate on relations *)
Definition vrel : (ty KStar * ty KStar * rel) -> Prop :=
  fun x =>
    match x with
    | ((t1, t2), r) =>
      forall e1 e2,
        r e1 e2 ->
        value e1 /\ value e2 /\ tm t1 e1 /\ tm t2 e2
    end.

(* (Typed-)Generalized relations: Definition 2.2 *)
Fixpoint GRel (k : kind) : Type :=
  match k with
    | KStar => rel
    | KFun k1 k2 => (ty k1 * ty k1 * GRel k1) -> GRel k2
  end.

Notation "'TyGRel' k" := (ty k * ty k * GRel k)%type (at level 67).
Notation "x ^1" := (fst (fst x)) (at level 2).
Notation "x ^2" := (snd (fst x)) (at level 2).
Notation "x ^3 " := (snd x) (at level 2).


(** Well-formed gen. relations and equality (Fig. 9) *)
Fixpoint wfGRel (k:kind) : TyGRel k -> Prop :=
  match k as k' return TyGRel k' -> Prop with
    | KStar => vrel
    | KFun k1 k2 => fun (c : TyGRel (KFun k1 k2)) =>
      (forall (a : TyGRel k1),
        wfGRel a ->
        (wfGRel (ty_app c^1  a^1, ty_app c^2 a^2, c^3 a)) /\
        (forall b, wfGRel b ->
          ty_eq a^1 b^1 ->
          ty_eq a^2 b^2 -> eqGRel k1 a^3 b^3 ->
          eqGRel k2 (c^3 a) (c^3 b)))
   end
with eqGRel (k:kind) : GRel k -> GRel k -> Prop :=
  match k as k' return GRel k' -> GRel k' -> Prop with
    | KStar => eq_rel
    | KFun k1 k2 => fun r1 r2 =>
        (forall a, wfGRel a -> eqGRel k2 (r1 a) (r2 a))
  end.
```

```
(* Equivalence between typed generalized relations *)
Definition eqTyGRel k (rho : TyGRel k) (pi : TyGRel k) :=
  ty_eq rho^1 pi^1 /\
  ty_eq rho^2 pi^2 /\
  eqGRel k rho^3 pi^3
```