JFP **20** (2): 175–210, 2010. © Cambridge University Press 2010
doi:10.1017/S0956796810000079

Parametricity, type equality, and higher-order polymorphism

DIMITRIOS VYTINIOTIS
Microsoft Research Cambridge, 7 JJ Thomson Avenue, Cambridge CB3 0FB UK
(*e-mail:* dimitris@microsoft.com)

STEPHANIE WEIRICH
University of Pennsylvania, Department of Computer and Information Science, Levine Hall, 3330 Walnut St., Philadelphia PA 19104–6389, USA
(*e-mail:* sweirich@cis.upenn.edu)

Abstract

Propositions that express type equality are a frequent ingredient of modern functional programming – they can encode generic functions, dynamic types, and GADTs. Via the Curry–Howard correspondence, these propositions are ordinary *types* inhabited by proof *terms*, computed using runtime type representations. In this paper we show that two examples of type equality propositions actually do reflect type equality; they are only inhabited when their arguments are equal and their proofs are unique (up to equivalence.) We show this result in the context of a strongly normalizing language with higher-order polymorphism and primitive recursion over runtime-type representations by proving Reynolds's abstraction theorem. We then use this theorem to derive "free" theorems about equality types.

1 Type equivalence, isomorphism, and equality

Type equivalence propositions assert that two types are isomorphic. For example, we may define such a proposition (in Haskell) as follows:

```
type EQUIV a b = (a -> b, b -> a)
```

Under the Curry–Howard correspondence, which identifies types and propositions, `EQUIV` asserts logical equivalence between two propositions: `a` implies `b` and `b` implies `a`. A proof of this equivalence, a pair of functions `f` and `g`, is a type isomorphism when the two functions compose to be the identity – in other words, when `f . g = id` and `g . f = id`. In that case, if `(f,g)` is a proof of the proposition `EQUIV a Int`, and `x` is an element of type `a`, then we can coerce `x` to be of type `Int` with `f`.

In the past 10 years, a number of authors have proposed the use of type equivalence propositions in typed programming languages (mostly Haskell). Type equivalence propositions have been used to implement heterogeneous data structures, type representations and generic functions, dynamic types, logical frameworks, metaprogramming, GADTs, and forms of lightweight dependent types (Yang 1998; Baars & Swierstra 2002; Cheney & Hinze 2002; Chen *et al.* 2004; Kiselyov *et al.* 2004; Sheard & Pasalic 2004; Weirich 2004).

Many of these authors point out that it is also possible to define a proposition that asserts that two types are not just equivalent, but that they are in fact *equal*. Equality is a stronger relation than equivalence as it must be *substitutive* as well as reflexive, symmetric, and transitive (see Kleene 1967, p. 157). Type equality propositions are also called *equality types*.

One definition of type equality is *Leibniz equality* – two types are equal iff one may be replaced with the other in all contexts. In Haskell, we may define the Leibniz equality proposition using higher-order polymorphism to quantify over all contexts.

```
type EQUAL a b = forall c. c a -> c b
```

Type equivalence and type equality propositions may be used for many of the same applications, but there are subtle differences between them. Equivalence holds for types that are not definitionally equal; for example, the types `(Int, Bool)` and `(Bool, Int)` are not equal in the Haskell type system, but they are isomorphic. One element of type `EQUIV (Int, Bool) (Bool, Int)` is two copies of a function that swaps the components of a pair. However, not all inhabitants of isomorphic types are type isomorphisms – for example, the term `(const 0, const 1)` inhabits the type `EQUIV Int Int`. Finally, some equivalent types are not isomorphic at all. For example, the proposition `EQUIV Int Bool` is provable, but not by any isomorphism between the types.

In contrast, equality only holds for equal types and equal types are trivially isomorphic. There are no (terminating) inhabitants of type `EQUAL Int Bool` or of `EQUAL (Int, Bool) (Bool, Int)`. We know this because of *parametricity*: for the latter type an inhabitant would need to know how to swap the components of the pair in an arbitrary context. Furthermore, the only inhabitants of type `EQUAL Int Int` are identity functions. Again, the reason is parametricity – because the context is abstract the function has no choice but to return its argument.

These observations about the difference between the properties of type equivalence and of type equality are informal, and we would like to do better. In this paper, we make the previous arguments about type equality rigorous by deriving *free theorems* (Reynolds 1983; Wadler 1989) about equality types from Reynolds's abstraction theorem. Reynolds's abstraction theorem (also referred to as the "parametricity theorem" (Wadler 1989) or the "fundamental theorem" of logical relations) asserts that every well-typed expression of the polymorphic λ-calculus (System F) (Girard 1972) satisfies a property directly derivable from its type.

We derive these free theorems from the parametricity theorem for a language called R_ω (Crary *et al.* 2002), which extends Girard's F_ω with constructs that are useful for programming with type equivalence propositions (see the next section). Using these constructs in R_ω we can define a *type-safe cast* operation which compares types and produces an equality proof when they are the same. This extension comes at little cost as the necessary modifications to the F_ω parametricity theorem are modest and localized. Like F_ω, R_ω is a (provably, using the results in this paper) terminating language, which simplifies our development and allows us to focus on the parametricity properties of higher-order polymorphism. Of course, our results will not carry over to full languages like Haskell without extension.

After proving a version of the abstraction theorem for R_ω, we show how to apply it to the type `EQUAL` to show that it is inhabited only when the source and target types are the same, in which case that inhabitant must be the identity.

Our use of free theorems for higher-order polymorphism exhibits an intriguing behavior. Whereas free theorems for second-order polymorphism quantify over arbitrary relations, they are often instantiated with (the graphs of) functions expressible in the polymorphic λ-calculus (Wadler 1989). By contrast, in our examples we instantiate free theorems with (the graphs of) *nonparametric* functions.

1.1 Contributions

The primary contribution of this paper is the correctness of the equality type, which implies correctness properties of a type-safe cast operation that can produce it. In addition, we use our framework to prove correctness for another equality proposition, which defines type equality as the smallest reflexive relation. We show that this latter proposition also holds only for equal types, is inhabited by a single member, and that the two equality types are isomorphic.

Along with these results, we consider our proof of parametricity for R_ω to be a significant contribution. This paper offers a fully explicit and accessible roadmap to the proof of parametricity for higher-order polymorphism, using the technique of syntactic logical relations,[1] and insisting on rigorous definitions. Rigorous definitions are not only challenging to get right but important in practice, since our examples demonstrate that the "power" of the meta-logical functions involved in instantiating the free theorems determines the expressiveness of these free theorems.

Because of our attention to formal details, our development is particularly well suited for mechanical verification in proof assistants based on Type Theory (the meta-logic of choice in this paper), such as Coq (`http://coq.inria.fr`). To this end, we offer a Coq formalization of the definitions in the Appendix.

2 Constructing equivalence and equality types

In this section we give an informal introduction to R_ω. Although we use Haskell syntax throughout the section (and all of the code is valid Haskell) our examples are intended to demonstrate R_ω programming.

Type equivalence and equality propositions can be constructed through dynamic type analysis. By comparing two types at runtime, we can produce a proof that they are isomorphic. Despite the fact that R_ω is a parametric language, dynamic type analysis is possible through *representation types* (Crary *et al.* 2002). The key idea is simple: Because the behavior of parametrically polymorphic functions cannot be influenced by the types at which they are instantiated, type analyzing functions dispatch on term arguments that *represent* types.

[1] The term "syntactic" refers to logically interpreting types as relations between syntactic terms, as opposed to semantic denotations of terms.

Although native to R_ω, representation types may be implemented in Haskell by a *Generalized Algebraic Datatype* (GADT) called `R a`, which represents its type index `a` (Sheard & Pasalic 2004; Jones *et al.* 2006).

```
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)
  Rarr  :: R a -> R b -> R (a -> b)
```

The datatype `R` includes five 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` types 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. Finally, `Rarr` represents function types. The important property of datatype `R a` is that the type index `a` changes with the data constructor. In contrast, in an ordinary datatype, all data constructors must return the same type.

Representation types may be used to define *type-safe* cast that compares two different type representations and, if they match, produces an equivalence or equality proof. Type-safe cast tests, 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.

Weirich (2004) defined two different versions of type-safe cast, `cast` and `gcast`, shown in Figure 1. 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.

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 case for products has been decorated with type annotations. Note that in the function case, a reverse cast is needed to handle the contra-variance of the function type constructor. If this cast succeeds, then it produces (half of) a type equivalence proof.

Alternatively, `gcast` produces a proof of Leibniz equality. The resulting coercion function never needs to decompose (or even evaluate) its argument. The key ingredient is the use of the higher-order type argument `c` that allows `gcast` to return a coercion from `c a` to `c b`.

In the implementation of `gcast`, the type constructor `c` allows the recursive calls to `gcast` to create a coercion that changes the type of part of its argument. Again, the case for products has been decorated with type annotations – the first recursive

```
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)
  Rarr   :: R a -> R b -> R (a -> b)

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 :: a0 -> a0') <- cast ra0 ra0'
      (h :: b0 -> b0') <- cast rb0 rb0'
      Just (\(a,b) -> (g a, h b))
cast (Rsum ra0 rb0) (Rsum ra0' rb0')
 = do g <- cast ra0 ra0'
      h <- cast rb0 rb0'
      Just (\x -> case x of Left a -> Left (g a)
                            Right b -> Right (h b))
cast (Rarr ra0 rb0) (Rarr ra0' rb0')
  = do g <- cast ra0' ra0
       h <- cast rb0  rb0'
       return (\x -> h . x . g)
cast _ _ = Nothing

type EQUAL a b = forall c. c a -> c b
newtype CL f c a d = CL { unCL :: c (f d a) }
newtype CR f c a d = CR { unCR :: c (f a d) }

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

Fig. 1. Haskell implementation of cast and gcast.

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

call changes the type of the first component of the product, the second recursive call changes the type of the second component. In each recursive call, the instantiation of `c` hides the parts of the type that remain unchanged. The newtypes `CL` and `CR` allow unification to select the right instantiation of `c`. Note that the cases for products, sums and arrow types are identical (except for the type annotations).

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.

In the rest of this paper, we make this argument formal by deriving a free theorem for `EQUAL` from the parametricity theorem for R_ω.

Of course, we do not actually need R_ω to show this result. Representation types are directly encodable in F_ω via a Church encoding (Weirich 2001) or by using type isomorphisms (Cheney & Hinze 2003). However, the definitions of `cast` and `gcast` are simpler using native representation types than either encoding as the type system (Haskell or R_ω) can implicitly use the type equalities introduced through type analysis. Furthermore, in a strongly normalizing language, such as F_ω, the native version is slightly more expressive. It is not clear how to encode the primitive recursive elimination form supported by native representation types; only iteration can be supported (Spławski & Urzyczyn 1999). Finally, extending an F_ω parametricity proof to R_ω only requires local changes to support the representation types, so the cost of this extension in minimal.

3 Parametricity for $\mathbf{R}_\omega$

3.1 *The* $\mathbf{R}_\omega$ *calculus*

The R_ω calculus is a Curry-style extension of F_ω (Girard 1972). The syntax of this language appears in Figure 2 and the static semantics appears in Figures 3 and 4. 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 1 we had to introduce the constructors `CL` and `CR` only because Haskell does not include type-level λ-abstractions.

Type constructor constants, $\mathscr{K}$, include standard operators, plus representation types R. In the following, we write $\to$, $\times$, and $+$ using infix notation and

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

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

Fig. 2. Syntax of System R_ω.

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

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

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

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

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

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

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

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

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

$$
\frac{\Gamma, a{:}\kappa_1 \vdash \tau_1 : \kappa_2 \quad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash (\lambda a{:}\kappa_1\,.\,\tau_1)\ \tau_2 \equiv \tau_1\{\tau_2/a\} : \kappa_2}\ \text{BETA}
\qquad
\frac{\Gamma \vdash \tau : \kappa_1 \to \kappa_2 \quad a \notin \mathit{fv}(\tau)}{\Gamma \vdash (\lambda a{:}\kappa_1\,.\,\tau\ a) \equiv \tau : \kappa_1 \to \kappa_2}\ \text{ETA}
$$

Fig. 3. Type well formedness and equivalence.

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

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

$$\frac{}{\Gamma \vdash i : \mathtt{int}}\ \text{INT} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}}\ \text{UNIT}$$

$$\frac{\Gamma, (x{:}\tau_1) \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : \star}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \text{ABS}$$

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 : \mathtt{R}\ \sigma_1 \quad \Gamma \vdash e_2 : \mathtt{R}\ \sigma_2}{\Gamma \vdash \mathtt{R}_\to\ e_1\ e_2 : \mathtt{R}\ (\sigma_1 \to \sigma_2)}\ \text{RARR}$$

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

Fig. 4. Typing relation for R_ω.

associate applications of $\to$ to the right. We treat impredicative polymorphism with 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

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

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

```
1  gcast :: ∀a : ⋆.∀b : ⋆.R a → R b → () + (∀c : ⋆ → ⋆.c a → c b)
2  gcast = λx.typerec x of {
3   λy.typerec y of {inr λz.z ; inl () ; inl () ; inl () ; inl ()};
4   λy.typerec y of {inl () ; inr λz.z ; inl () ; 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.h2 (h1 z))
12      }};
13    λrb1.λg1.λrb2.λg2.inl ();
14    λrb1.λg1.λrb2.λg2.inl ()};
15  λra1.λf1.λra2.λf2.λy.typerec y of {
16    inl ();
17    inl ();
18    λrb1.λg1.λrb2.λg2.inl ();
19    λrb1.λg1.λrb2.λg2.
20      case f1 rb1 of {h.inl () ; h1.
21      case f2 rb2 of {h.inl () ; h2.
22       inr (λz.h2 (h1 z))
23      }};
24    λrb1.λg1.λrb2.λg2.inl ();}
25  λra1.λf1.λra2.λf2.λy.typerec y of {
26    inl ();
27    inl ();
28    λrb1.λg1.λrb2.λg2.inl ();
29    λrb1.λg1.λrb2.λg2.inl ();
30    λrb1.λg1.λrb2.λg2.
31      case f1 rb1 of {h.inl () ; h1.
32      case f2 rb2 of {h.inl () ; h2.
33       inr (λz.h2 (h1 z))
34      }}}};
```

Fig. 5. Definition of *gcast* in R_ω. Note that lines 11, 22 and 33 are identical.

R_ω expressions *e* include abstractions, products, sums, integers, and unit. We leave type abstractions and type applications implicit to reduce notation overhead (but note that this choice has an impact on parametricity in the presence of impure features – see Section 5.4). R_ω includes type representations $\mathtt{R}_{\mathtt{int}}$, $\mathtt{R}_{()}$, $\mathtt{R}_\times$, $\mathtt{R}_+$, and $\mathtt{R}_\rightarrow$which must be fully applied to their arguments. We do not include representations for polymorphic types in R_ω because they significantly change the semantics of the language, as we discuss in Section 5.3. The R_ω 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 *gcast* in Figure 5.

The dynamic semantics of R_ω is a standard large-step nonstrict operational semantics, presented in Figure 6. Essentially `typerec` performs a fold over its type

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

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

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

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

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

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

$$
\frac{e \Downarrow \mathtt{R_{int}} \quad e_{\mathtt{int}} \Downarrow v}{\mathtt{typerec}\ e\ \mathtt{of}\ \overline{e} \Downarrow v}
\qquad
\frac{e \Downarrow \mathtt{R_{()}} \quad e_{()} \Downarrow v}{\mathtt{typerec}\ e\ \mathtt{of}\ \overline{e} \Downarrow v}
$$

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

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

$$
\frac{\begin{array}{c} e \Downarrow \mathtt{R_{\rightarrow}}\ e_1\ e_2 \\ e_{\rightarrow}\ e_1\ (\mathtt{typerec}\ e_1\ \mathtt{of}\ \overline{e})\ e_2\ (\mathtt{typerec}\ e_2\ \mathtt{of}\ \overline{e}) \Downarrow v \end{array}}{\mathtt{typerec}\ e\ \mathtt{of}\ \overline{e} \Downarrow v}
$$

Fig. 6. Operational semantics rules.

representation argument. We use u, v, w for R_ω values, the syntax of which is also given in Figure 6.

The static semantics of R_ω contains judgments for kinding, definitional type equality, 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. The notations $\Gamma, x{:}\tau$ and $\Gamma, a{:}\kappa$ are defined only when x and a are not already in the domain of Γ. The kinding judgment $\Gamma \vdash \tau : \kappa$ (in Figure 3) 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:

Definition 3.1 (*closed types*)

We write $\tau \in \mathtt{ty}(\kappa)$ iff $\cdot \vdash \tau : \kappa$.

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

The typing judgment has the form $\Gamma \vdash e : \tau$ and appears in Figure 4. 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, T-EQ. The judgment $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$ defines type equality 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 σ for a inside τ.) In addition, we implicitly identify α-equivalent types, and treat them as syntactically equal in the rest of the paper. We give the definition of type equality in Figure 3. 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 class 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 equality.

3.2 The abstraction theorem

Deriving free theorems requires first defining an appropriate interpretation of types as binary relations[2] (in the *meta-logic* that is used for reasoning) between terms and showing that these relations are reflexive. This result is the core of Reynolds's abstraction theorem:

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

Free theorems result from unfolding the definition of the interpretation of types (which appears in Figure 8, using Definition 3.5). 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.

Definition 3.2 (*(typed-)generalized relations*)

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

The notation $\mathscr{P}(\mathtt{term} \times \mathtt{term})$ 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.

[2] We use binary relations so that we can relate our definition to contextual equivalence. Note, however, that for the examples in this paper a unary interpretation is sufficient, but we chose to not sacrifice the extra generality.

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

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

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

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

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

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

Fig. 7. Well-formed generalized relations and equality.

Generalized relations are mutually defined with typed-generalized relations, $\texttt{TyGRel}^{\kappa}$, which are triples of generalized relations and types of the appropriate kind. Elements of $\texttt{GRel}^{\kappa_1 \to \kappa_2}$ accept one of these triples. These extra $\texttt{ty}(\kappa)$ arguments allow the morphisms to dispatch control depending on types as well as relational arguments. This flexibility will turn out to be important for the free theorems about R_{ω} programs that we show in this paper.

At first glance, Definition 3.2 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 (e.g. `Type` in CIC (see Appendix 7), 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.

Unfortunately, not all objects of $\texttt{GRel}^{\kappa}$ are suitable for the interpretation of types. In Figure 7, 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 $r, s \in \mathtt{GRel}^\kappa$, 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.[3] At higher kinds, equality asserts that the relation arguments return equal results when given the same argument ρ. Alternatively, equality at higher-kind could have been defined relationally (i.e. r and s are equal if they take *equal* arguments to equal results) instead of pointwise. Our version is slightly simpler, but no less expressive. We cannot simplify this definition further by dropping the requirement that ρ be well formed, as we discuss in the proof of Coherence, Theorem 3.11.

Equality for typed-generalized relations, $\rho \equiv \pi$, is defined in terms of its components. This definition is reflexive, symmetric, and transitive, and hence is an equivalence relation, by induction on the kind κ. Furthermore, the $\mathtt{wfGRel}^\kappa$ predicate respects this equality.

Lemma 3.3
For all $\rho \equiv \pi$, if $\rho \in \mathtt{wfGRel}^\kappa$ then $\pi \in \mathtt{wfGRel}^\kappa$.

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 set of type variables on which δ is defined. We use $\cdot$ for the undefined-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 homomorphic application of substitutions δ^1 and δ^2 to τ. In our development, we carefully apply substitutions on types whose free type variables belong in the domain of the substitutions.

Definition 3.3 (*substitution kind checks in environment*)
We say that a substitution δ *kind checks in an environment* Γ, and write $\delta \in \mathtt{Subst}_\Gamma$, when $dom(\delta) = dom(\Gamma)$ and for every $(a{:}\kappa) \in \Gamma$, we have $\delta(a) \in \mathtt{TyGRel}^\kappa$.

The interpretation of R_ω types is shown in Figure 8 and is defined inductively over kinding derivations for types. The interpretation function $[\![\cdot]\!]_\cdot$ accepts a derivation $\Gamma \vdash \tau : \kappa$, and a substitution $\delta \in \mathtt{Subst}_\Gamma$ and returns a generalized relation at kind κ, hence, the meta-logical type, $\mathtt{Subst}_\Gamma \supset \mathtt{GRel}^\kappa$. 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 $\mathscr{K}$, we call the auxiliary function $[\![\mathscr{K}]\!]$, shown in Figure 9. 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

[3] Observe that, in the case of kind $\star$, we use extensional equality for relations instead of the simpler intensional equality ($r = s$) to reduce the requirements on 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`).

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

Fig. 8. Relational interpretation of R_ω.

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

$$
\begin{array}{lcl}
\mathcal{R} & \triangleq & \lambda(\tau,\sigma,r) \in \texttt{TyGRel}^\star \mapsto \\
 & & \{(\texttt{R}_{\texttt{int}}, \texttt{R}_{\texttt{int}}) \mid (\tau,\sigma,r) \equiv (\texttt{int}, \texttt{int}, [\![\texttt{int}]\!])\} \\
 & \cup & \{(\texttt{R}_{()}, \texttt{R}_{()}) \mid (\tau,\sigma,r) \equiv ((),(),[\![()]\!])\} \\
 & \cup & \{(\texttt{R}_\times\ e_a^1\ e_b^1, \texttt{R}_\times\ e_a^2\ e_b^2) \ \mid \\
 & & \quad \exists\rho_a,\rho_b \in \texttt{wfGRel}^\star \wedge \\
 & & \quad\ \cdot \vdash \tau \equiv \rho_a^1 \times \rho_b^1 : \star \wedge \cdot \vdash \sigma \equiv \rho_a^2 \times \rho_b^2 : \star \wedge r \equiv_\star [\![\times]\!]\ \rho_a\ \rho_b \wedge \\
 & & \quad\ (e_a^1,e_a^2) \in \mathcal{C}(\mathcal{R}\ \rho_a) \ \wedge\ (e_b^1,e_b^2) \in \mathcal{C}(\mathcal{R}\ \rho_b)\ \} \\
 & \cup & \{(\texttt{R}_+\ e_a^1\ e_b^1, \texttt{R}_+\ e_a^2\ e_b^2) \ \mid \\
 & & \quad \exists\rho_a,\rho_b \in \texttt{wfGRel}^\star \wedge \\
 & & \quad\ \cdot \vdash \tau \equiv \rho_a^1 + \rho_b^1 : \star \wedge \cdot \vdash \sigma \equiv \rho_a^2 + \rho_b^2 : \star \wedge r \equiv_\star [\![+]\!]\ \rho_a\ \rho_b \wedge \\
 & & \quad\ (e_a^1,e_a^2) \in \mathcal{C}(\mathcal{R}\ \rho_a) \ \wedge\ (e_b^1,e_b^2) \in \mathcal{C}(\mathcal{R}\ \rho_b)\ \} \\
 & \cup & \{(\texttt{R}_\to\ e_a^1\ e_b^1, \texttt{R}_\to\ e_a^2\ e_b^2) \ \mid \\
 & & \quad \exists\rho_a,\rho_b \in \texttt{wfGRel}^\star \wedge \\
 & & \quad\ \cdot \vdash \tau \equiv \rho_a^1 \to \rho_b^1 : \star \wedge \cdot \vdash \sigma \equiv \rho_a^2 \to \rho_b^2 : \star \wedge r \equiv_\star [\![\to]\!]\ \rho_a\ \rho_b \wedge \\
 & & \quad\ (e_a^1,e_a^2) \in \mathcal{C}(\mathcal{R}\ \rho_a) \ \wedge\ (e_b^1,e_b^2) \in \mathcal{C}(\mathcal{R}\ \rho_b)\ \}
\end{array}
$$

Fig. 9. Operations of type constructors on relations.

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

as abstractions in the meta-logic. We use λ and $\mapsto$ for meta-logic abstractions. Confirming that $[\![\Gamma \vdash \tau : \kappa]\!]_\delta \in \mathtt{GRel}^\kappa$ is straightforward using the fact that $\delta \in \mathtt{Subst}_\Gamma$.

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

Definition 3.5 (*computational lifting*)
The *computational lifting* of a relation $r \in \mathtt{VRel}(\tau_1, \tau_2)$, written as $\mathscr{C}(r)$, 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, $[\![\mathtt{int}]\!]$ and $[\![\mathtt{()}]\!]$ give the identity value relations respectively on $\mathtt{int}$ and $\mathtt{()}$. The operation $[\![\rightarrow]\!]$ 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 liftings of the value relations. For the $\forall_\kappa$ constructor, since its kind is $(\kappa \rightarrow \star) \rightarrow \star$ we define $[\![\forall_\kappa]\!]$ to be a morphism that, given a $\mathtt{TyGRel}^{\kappa \rightarrow \star}$ 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 3.6).

For the case of representation types $\mathtt{R}$, the definition relies on an auxiliary morphism $\mathscr{R}$, 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 $\mathscr{R}$ (τ, σ, r), the case for integer representations, r is required to be equal to $[\![\mathtt{int}]\!]$. The $\mathscr{R}$ definition is carefully crafted to validate the abstraction theorem – alternative definitions, such as one that leaves the relational argument of $\mathtt{R}$ completely unconstrained, do not validate the abstraction theorem (Vytiniotis & Weirich 2007).

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

Lemma 3.6
For all $\mathscr{K}$, $(\mathscr{K}, \mathscr{K}, [\![\mathscr{K}]\!]) \in \mathtt{wfGRel}^{kind(\mathscr{K})}$.

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

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

Let us fix τ_1, $\tau_2 \in \mathtt{ty}(\kappa \rightarrow \star)$, and a generalized relation $g_\tau \in \mathtt{GRel}^{\kappa \rightarrow \star}$, with $(\tau_1, \tau_2, g_\tau) \in \mathtt{wfGRel}^{\kappa \rightarrow \star}$. Then we know that

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

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

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

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

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 $\rho \in \mathtt{TyGRel}^\kappa$, with $\rho \in \mathtt{wfGRel}^\kappa$, we have $g_\tau\ \rho \equiv_\star g_\sigma\ \rho$. This holds by 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.

Definition 3.7 (*environment-respecting substitution*)
We write $\delta \vDash \Gamma$ iff $\delta \in \mathtt{Subst}_\Gamma$ and for every $a \in dom(\delta)$, it is the case that $\delta(a) \in \mathtt{wfGRel}^\kappa$.

With this definition we can now state the lemma.

Lemma 3.8 (*type interpretation is well formed*)
If $\Gamma \vdash \tau : \kappa$ then

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

Proof
Straightforward induction over the type well-formedness derivations, appealing to Lemma 3.6. The only interesting case is the case for type abstractions, which follows from Lemma 3.3. □

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

Lemma 3.9 (*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.

Lemma 3.10 (*extensionality*)
Given an environment-respecting $\delta \vDash \Gamma$, a well-formed type $\Gamma \vdash \tau : \kappa_1 \to \kappa_2$, and a fresh variable $a \notin fv(\tau), \Gamma$, 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

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

Pick $\rho \in \mathtt{wfGRel}^{\kappa_1}$. 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 an easy weakening property, this is definitionally equal 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.

Theorem 3.11 (*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 3.9, the case for rule ETA follows from Lemma 3.10, 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

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

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

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 3.8, we have that

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

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 $\rho \in \mathtt{TyGRel}^{\kappa_1}$ with $\rho \in \mathtt{wfGRel}^{\kappa_1}$. 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 $\rho \in \mathtt{wfGRel}^{\kappa_1}$ (Figure 7) 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* $\rho \in \mathtt{TyGRel}^{\kappa_1}$ that would no longer be true, and hence the case could not be proved. $\square$

We may now state the abstraction theorem.

Theorem 3.12 (*abstraction theorem for* R_ω)
Assume $\cdot \vdash e : \tau$. Then $(e, e) \in \mathscr{C}\,[\![\cdot \vdash \tau : \star]\!]_\cdot$.

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 $(\gamma^1 e, \gamma^2 e) \in \mathscr{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$.

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

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

The definition of $\mathtt{Subst}_\Gamma$ 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 $(e_1, e_2) \in \mathscr{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$ 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 where for all $(x{:}\tau) \in \Gamma$ it is $\Gamma \vdash \tau : \star$; 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 3.11) 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 $x \notin \Gamma, fv(\gamma)$. 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
$$((\lambda x\,.\,\gamma^1 e)\; e_1, (\lambda x\,.\,\gamma^2 e)\; e_2) \in \mathscr{C}\,[\![\Gamma \vdash \tau_2 : \star]\!]_\gamma \tag{1}$$
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: $(\gamma_0^1 e, \gamma_0^2 e) \in \mathscr{C}\,[\![\Gamma, (x{:}\tau_1) \vdash \tau_2 : \star]\!]_{\gamma_0}$. By an easy weakening lemma for term variables in the type interpretation we have that $(\gamma_0^1 e, \gamma_0^2 e) \in \mathscr{C}\,[\![\Gamma \vdash \tau_2 : \star]\!]_\gamma$ 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,
$$\begin{aligned}(\gamma^1 e_1, \gamma^2 e_1) &\in \mathscr{C}\,[\![\Gamma \vdash \sigma \to \tau : \star]\!]_\gamma &(2)\\ (\gamma^1 e_2, \gamma^2 e_2) &\in \mathscr{C}\,[\![\Gamma \vdash \sigma : \star]\!]_\gamma &(3)\end{aligned}$$
From (2) we get $\gamma^1 e_1 \Downarrow w_1$ and $\gamma^2 e_1 \Downarrow w_2$ such that $(w_1\; (\gamma^1 e_2), w_2\; (\gamma^2 e_2)) \in \mathscr{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$, where we made use of Equation (3) and unfolded definitions. Hence, by the operational semantics for applications, we also have that $((\gamma^1 e_1)\; (\gamma^1 e_2), (\gamma^2 e_1)\; (\gamma^2 e_2)) \in \mathscr{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$, as required.
- Case T-EQ. The case follows directly from appealing to the Coherence theorem 3.11.
- 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 $(\gamma^1 e, \gamma^2 e) \in \mathscr{C}([\![\forall_\kappa]\!]\;\; (\gamma^1 \sigma, \gamma^2 \sigma, [\![\Gamma \vdash \sigma : \kappa \to \star]\!]_\gamma))$; 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 3.8), 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 $a \notin ftv(\gamma)$ as well. We need to show that $(\gamma^1 e, \gamma^2 e) \in \mathscr{C}([\![\forall_\kappa]\!]\;\; (\gamma^1 \sigma, \gamma^2 \sigma, [\![\sigma]\!]_\gamma)$. 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 $(\gamma_0^1 e, \gamma_0^2 e) \in \mathscr{C}\,[\![\Gamma, (a{:}\kappa) \vdash \sigma\; a : \star]\!]_{\gamma_0}$ which means $(\gamma_0^1 e, \gamma_0^2 e) \in \mathscr{C}\,[\![\Gamma, (a{:}\kappa) \vdash \sigma : \kappa \to \star]\!]_{\gamma_0}\;\; \rho$. By an easy weakening lemma this implies $(\gamma_0^1 e, \gamma_0^2 e) \in \mathscr{C}\,[\![\Gamma \vdash \sigma : \kappa \to \star]\!]_\gamma\;\; \rho$ 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 $\Gamma \vdash \mathtt{R_{int}} : \mathtt{R}\ \mathtt{int}$, hence $(\mathtt{R_{int}}, \mathtt{R_{int}}) \in \mathscr{R}\,(\mathtt{int}, \mathtt{int}, [\![\mathtt{int}]\!])$ 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 $(\mathtt{R}_\times\ \gamma^1 e_1\ \gamma^1 e_2, \mathtt{R}_\times\ \gamma^2 e_1\ \gamma^2 e_2) \in \mathscr{R}\,(\gamma^1(\sigma_1 \times \sigma_2), \gamma^2(\sigma_1 \times \sigma_2), [\![\Gamma \vdash \sigma_1 \times \sigma_2 : \star]\!]_\gamma)$. 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 3.8, 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 $(\gamma^1 e_1, \gamma^2 e_1) \in \mathscr{C}(\mathscr{R}\ \rho_a)$ and $(\gamma^1 e_2, \gamma^2 e_2) \in \mathscr{C}(\mathscr{R}\ \rho_b)$, which follow by induction hypotheses for the typing of e_1 and e_2.
- Case RSUM. Similar to the case for RPROD.
- Case RARR. 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 scrutiny. In this case we have that

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

Let us introduce some abbreviations:

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

By the premises of the rule we have

$$\Gamma \vdash \sigma : \star \rightarrow \star \tag{4}$$
$$\Gamma \vdash e : \mathtt{R}\ \tau \tag{5}$$
$$\Gamma \vdash e_{\mathtt{int}} : \sigma\ \mathtt{int} \tag{6}$$
$$\Gamma \vdash e_{()} : \sigma\ () \tag{7}$$
$$\Gamma \vdash e_\times : \sigma_\times \tag{8}$$
$$\Gamma \vdash e_+ : \sigma_+ \tag{9}$$
$$\Gamma \vdash e_\rightarrow : \sigma_\rightarrow \tag{10}$$

We also know the corresponding induction hypotheses for (6), (7), (8), (9), and (10). We now show that:

$$\begin{array}{l} \forall e_1\ e_2\ \rho \in \mathtt{TyGRel}^\star, \rho \in \mathtt{wfGRel}^\star \wedge (e_1, e_2) \in \mathscr{C}(\mathscr{R}\ \rho) \\ \Longrightarrow (\gamma^1 u[e_1], \gamma^2 u[e_2]) \in \mathscr{C}([\![\Gamma \vdash \sigma : \star \rightarrow \star]\!]_\gamma\ \ \rho) \end{array}$$

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 $(e_1, e_2) \in \mathscr{C}(\mathscr{R}\ \rho)$ and hence we know that

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

$e_1 \Downarrow w_1$ and $e_2 \Downarrow w_2$, such that

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

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

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

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

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

— $w_1 = w_2 = ()$ and $[\![\Gamma \vdash \tau : \star]\!]_\gamma \equiv_\star [\![()]\!]$. Similarly 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 and ρ_b, well formed, such that

$$\rho \equiv_\star ((\rho_a^1 \times \rho_b^1), (\rho_a^2 \times \rho_b^2), [\![\times]\!]\ \ \rho_a\ \rho_b) \tag{11}$$

$$(e_a^1, e_a^2) \in \mathscr{C}(\mathscr{R}\ \rho_a) \tag{12}$$

$$(e_b^1, e_b^2) \in \mathscr{C}(\mathscr{R}\ \rho_b) \tag{13}$$

In this case we know that $\gamma^1 u[e_1] \Downarrow w_1$ and $\gamma^2 u[e_2] \Downarrow w_2$ where

$$\begin{array}{c} (\gamma^1 e_\times)\ e_a^1\ (\gamma^1 u[e_a^1])\ e_b^1\ (\gamma^1 u[e_b^1]) \Downarrow w_1 \\ (\gamma^2 e_\times)\ e_a^2\ (\gamma^2 u[e_a^2])\ e_b^2\ (\gamma^2 u[e_b^2]) \Downarrow w_2 \end{array}$$

By the outer induction hypothesis for Equation (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

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

$$(\gamma^1 u[e_b^1], \gamma^2 u[e_b^2]) \in \mathscr{C}\,[\![\Gamma_0 \vdash \sigma\ b : \star]\!]_{\gamma_0} \tag{15}$$

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 Equations (12) and (13). From these, compositionality, and an easy weakening lemma, we have that Equations (14) and (15) follow. By the outer induction hypothesis for Equation (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.

— The case for the $+$ and $\to$ constructors are similar to the case for $\times$.

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

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

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

3.3 Behavioral equivalence

As a corollary to the abstraction theorem, we can establish that the interpretation of types at kind $\star$ is contained in a suitable *behavioral equivalence* relation for closed terms. Intuitively, two terms are behaviorally equivalent if all uses of them produce the same result.[4]

To capture the idea of uses of terms, we define *elimination contexts* with the following syntax:

$$\begin{array}{rcl} E & ::= & \bullet \mid \mathtt{typerec}\ E\ \mathtt{of}\ \{e_{\mathtt{int}}\,; e_{()}\,; e_{\times}\,; e_{+}\,; e_{\to}\} \mid E\ v \\ & \mid & \mathtt{fst}\ E \mid \mathtt{snd}\ E \mid \mathtt{case}\ E\ \mathtt{of}\ \{x.e_l\,; x.e_r\} \end{array}$$

In R_ω, we cannot use termination behavior in our observations, so we only observe uses that produce integers. Therefore, a simple definition of behavioral equivalence for R_ω is the following. (As syntactic sugar, we will write $E[\bullet] : \tau \to \mathtt{int}$ for the derivation $\cdot \vdash \lambda x.E[x] : \tau \to \mathtt{int}$.)

Definition 3.13 (*behavioral equivalence*)
We write $e_1 \approx e_2 : \tau$ iff $\cdot \vdash e_1 : \tau$ and $\cdot \vdash e_2 : \tau$ and for all derivations $\vdash E[\bullet] : \tau \to \mathtt{int}$, it is $E[e_1] \Downarrow i$ iff $E[e_2] \Downarrow i$.

Theorem 3.14
If $(e_1, e_2) \in \mathscr{C}\,[\![\cdot \vdash \tau : \star]\!]_\cdot$ then $e_1 \approx e_2 : \tau$.

Proof
By Theorem 3.12, for any suitable context $E[\bullet]$ it is $(\lambda x.E[x], \lambda x.E[x]) \in \mathscr{C}\,[\![\cdot \vdash \tau \to \mathtt{int} : \star]\!]_\cdot$, and the result follows by unfolding definitions. □

Thus, showing that two expressions belong in the interpretation of their type provides a way to establish their behavioral equivalence.

[4] We conjecture that if this definition is extended to open terms via closing substitutions, then it may be shown equivalent to a suitable definition of *contextual equivalence* for R_ω following the techniques of Pitts (Pitts 2005).

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

4 Free theorems for type equality

4.1 Leibniz equality

We are now ready to use the abstraction theorem to reason about the equality type `EQUAL`. The parametricity theorem instantiated at type $\forall c : \star \to \star . c\ \tau_a \to c\ \tau_b$ reads as follows:

Corollary 4.1 (*free theorem for Leibniz equality*)
Suppose $\cdot \vdash e : \forall c : \star \to \star . c\ \tau_a \to c\ \tau_b$. Then given any $\rho_c \in \texttt{wfGRel}^{\star\to\star}$ and any $(e_1, e_2) \in \mathscr{C}(\hat{\rho}_c\ [\![\cdot \vdash \tau_a : \star]\!]_\cdot)$ we have that $(e\ e_1, e\ e_2) \in \mathscr{C}(\hat{\rho}_c\ [\![\cdot \vdash \tau_b : \star]\!]_\cdot)$

The first result that we show using this corollary is that if we have a proof of `EQUAL` $\tau_a\ \tau_b$ for two closed types, then those two types must actually be equal.

Theorem 4.2 (*Leibniz equality implies definitional equality*)
If $\cdot \vdash e : \forall c : \star \to \star . c\ \tau_a \to c\ \tau_b$ then $\cdot \vdash \tau_a \equiv \tau_b : \star$.

Proof
Assume by contradiction that $\cdot \not\vdash \tau_a \equiv \tau_b : \star$. Then we instantiate the abstraction theorem with $\rho_c = (\lambda a{:}\star . (), \lambda a{:}\star . (), f_c)$ where

$$f_c\ (\tau, \sigma, r) \quad = \quad \text{if } (\cdot \vdash \tau \equiv \tau_a : \star \wedge \cdot \vdash \sigma \equiv \tau_a : \star) \text{ then } [\![\cdot \vdash () : \star]\!]_\cdot \text{ else } \emptyset$$

One can confirm that $\rho_c \in \texttt{wfGRel}^{\star\to\star}$. Then by the free theorem above we know that, since $((), ()) \in \mathscr{C}(f_c\ [\![\cdot \vdash \tau_a : \star]\!]_\cdot)$, we have $(e\ (), e\ ()) \in \mathscr{C}(f_c\ [\![\cdot \vdash \tau_b : \star]\!]_\cdot)$ if $\cdot \not\vdash \tau_a \equiv \tau_b$ then $\mathscr{C}(f_c\ [\![\cdot \vdash \tau_b : \star]\!]_\cdot) = \emptyset$, a contradiction. □

We next use this free theorem again to show that the only inhabitant of the Leibniz equality proposition is an identity function.

Theorem 4.3 (*Leibniz proof is identity*)
If $\cdot \vdash e : \forall c : \star \to \star . c\ \tau_a \to c\ \tau_b$ then $e \approx \lambda x . x : \forall c : \star \to \star . c\ \tau_a \to c\ \tau_b$.

Proof
First, by Lemma 4.2 we get that $\cdot \vdash \tau_a \equiv \tau_b : \star$. As our logical relation implies equivalence, we show our result by showing that

$$(e, \lambda x . x) \in [\![\cdot \vdash \forall c . c\ \tau_a \to c\ \tau_a : \star]\!]_\cdot$$

Unfolding definitions, we need to show that for any $\rho \in \texttt{wfGRel}^{*\to*}$ and any

$$(e_1, e_2) \in [\![c\ \tau_a]\!]_{c \mapsto \rho} \qquad \text{we must have} \qquad (e\ e_1, (\lambda x.x)\ e_2) \in \mathscr{C}\, [\![c\ \tau_a]\!]_{c \mapsto \rho}$$

Suppose $e_1 \Downarrow w$ and $e_2 \Downarrow v$. Because $(\lambda x . x)\ v \Downarrow v$ and these sets are closed under evaluation, the result holds if we can show that $e\ w \Downarrow w$.

We prove this last fact using the free theorem about the type of e. By the free theorem, we know that *for all* well-formed ρ_c, we have

$$(e, e) \in [\![c\ \tau_a \to c\ \tau_a]\!]_{c \mapsto \rho_c}$$

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

Therefore, we choose c to be instantiated with $\rho_c = (\lambda\_.\,\rho^1(\tau_a), \lambda\_.\,\rho^1(\tau_a), f_c)$ where $f_c = \{(w,w)\}$. It is easy to see that this generalized relation is well formed. Then, unfolding definitions, because $(w,w) \in \rho_c(\tau_a, \tau_a, [\![\tau_a]\!]_\cdot)$, we know that $(e\ w, e\ w) \in \mathscr{C}(\rho_c(\tau_a, \tau_a, [\![\tau_a]\!]_\cdot))$. However, because (w,w) is the only value in this last set, we must have $e\ w \Downarrow w$. $\square$

Remark 4.4

To derive Theorem 4.2 we had to instantiate a generalized relation to be a morphism that is not the interpretation of any F_ω type function. In particular, this morphism is nonparametric since it dispatches on its type arguments. Hence, despite the fact that we are showing a theorem about an F_ω type, we need morphisms at higher kinds to accept *both types and morphisms* as arguments and dispatch on their type argument – a novel use of type-dispatching interpretations compared to recent work on free theorems for higher-order polymorphic functions (Voigtländer 2009). On the other hand, as soon as type equality was established, the proof of Theorem 4.3 did not use a nonparametric relation.

Remark 4.5

A weaker theorem than Theorem 4.3, namely that $e \approx \lambda x.x : \forall a : \star.a \to a$ can be shown *without* any use of higher-order instantiations. We may implicitly generalize over a and instantiate c with a function that returns a to show that e has also type $\forall a{:}\star\,.a \to a$. We may then apply first-order parametricity, which still holds in our language to show the theorem. However, we are interested in the equivalence at a *different type* and it is unclear under which conditions the equivalence at a more specialized type (such as $\forall a{:}\star\,.a \to a$) implies equivalence at a more general type (such as $\forall c.c\ \tau_a \to c\ \tau_b$).

Remark 4.6

Observe that the condition that the function f_c has to operate uniformly for *equivalence classes* of type α and β, 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. If the abstraction theorem remained true when this condition was violated then we could derive a false statement. Consider an expression e of type

$$\forall (c{:}\star \to \star).\,c\ () \to c\ ((\lambda d{:}\star\,.d)\ ())$$

Let $\tau_c = \lambda c{:}\star\,.c$. We instantiate c in the free theorem for the Leibniz equality type with $\rho_c = (\tau_c, \tau_c, f)$ where

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

The important detail is that f can return different results for *equivalent but syntactically different type arguments*. In particular, the type $(\lambda d{:}\star\,.d)\ ()$ is not syntactically equal to $()$, so $f((\lambda d{:}\star\,.d)\ (), (\lambda d{:}\star\,.d)\ (), r)$ returns the empty set for any r. Then, by the free theorem for the equality type, it must be that $(e\ (), e\ ()) \in \emptyset$, a contradiction to the abstraction theorem! Hence the abstraction theorem breaks when generalized morphisms at higher kinds do not respect type equivalence classes of their type arguments.

We can use these two theorems to directly prove two correctness properties about *any* function with same type as *gcast*. The first property that we show is that if *gcast* returns a function then the two types that instantiated *gcast* must be equal. (Note that even if the type representations are equivalent, 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.) We can also show the second part of the correctness property of *gcast*, that if *gcast* succeeds and returns a conversion function, then that function *must* be equivalent to an identity function.

Corollary 4.7 (*correctness of gcast I*)
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 $\cdot \vdash \tau_a \equiv \tau_b : \star$.

Corollary 4.8 (*correctness of gcast II*)
If $\cdot \vdash e_{ra} : \mathtt{R}\ \tau_a$, $\cdot \vdash e_{rb} : \mathtt{R}\ \tau_b$, $gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e$, then $e \approx \lambda x.x : \forall c.c\ \tau_a \to c\ \tau_b$.

Remark 4.9
Similar theorems would be true for any term e such that

$$\cdot \vdash e : \forall (a{:}\star)(b{:}\star).\,() + (\forall (c{:}\star \to \star).\,c\ a \to c\ b)$$

if such a term could be constructed that would return a right injection. However, all terms of this type may only return `inl ()`. What is important in R_ω is that the extra $\mathtt{R}\ a$ and $\mathtt{R}\ b$ arguments and `typerec` make the programming of `gcast` possible!

4.2 Another definition of type equality

We have seen applications of the free theorem for the type $\forall c.c\ \tau_1 \to c\ \tau_2$, but this type is not the only way to define type equality. In this section we discuss the properties of another proposition that defines type equality as the *smallest reflexive* relation. This definition also uses higher-order polymorphism to quantify over all binary relations c that can be shown to be reflexive (through the argument $(\forall (d{:}\star).\,c\ d\ d)$. Equality is the intersection of all such relations.

$$\mathtt{REQUAL}\ a\ b = \forall (c{:}\star \to \star \to \star).\,(\forall (d{:}\star).\,c\ d\ d) \to c\ a\ b$$

This definition of equality is interesting because it is a Church encoding of a commonly used definition for propositional equality in Haskell (and other dependently typed languages such as Coq and Agda). The code shown in Figure 10 includes a definition of the `REqual` GADT (of which `REQUAL` is the encoding). This datatype has a single constructor `Refl`, which produces a proof that some type is equal to itself. Pattern matching on an object of type `REqual a b` instructs the type checker to unify the types `a` and `b`. For example, in the product branch, pattern matching on the result of `pcast ra0 ra0'` unifies the types `a0` and `a0'`. Likewise for the types `b0` and `b0'` in the second recursive call. Therefore, the branch may return `Refl` as a proof of equality for `(a0, b0)` and `(a0', b0')` as these types

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

```
data REqual a b where
     Refl :: Equal a a

pcast :: R a -> R b -> Maybe (REqual a b)
pcast Rint Rint = Just Refl
pcast Runit Runit = Just Refl
pcast (Rprod (ra0  :: R a0)  (rb0  :: R b0))
      (Rprod (ra0' :: R a0') (rb0' :: R b0')) =
  do Refl <- pcast ra0 ra0'
     Refl <- pcast rb0 rb0'
     return Refl
pcast (Rsum ra0 rb0) (Rsum ra0' rb0') =
  do Refl <- pcast ra0 ra0'
     Refl <- pcast rb0 rb0'
     return Refl
pcast (Rarr ra0 rb0) (Rarr ra0' rb0') =
  do Refl <- pcast ra0 ra0'
     Refl <- pcast rb0 rb0'
     return Refl
pcast _ _ = Nothing
```

Fig. 10. `pcast`.

are identical to the Haskell type checker. Because of the integration between this equality predicate and the Haskell type checker, a proof of type `REqual t1 t2` is often easier to use than one of type `EQUAL t1 t2`.

As before, we show that if $\texttt{REQUAL}\ \tau_1\ \tau_2$ is inhabited, then the two types are indeed definitionally equal and that the proof is an identity function.

Theorem 4.10 (*reflexive proposition implies definitional equality*)
If $\cdot \vdash e : \texttt{REQUAL}\ \tau_1\ \tau_2$ then $\cdot \vdash \tau_a \equiv \tau_b : \star$.

Proof sketch
Similar to the proof of Lemma 4.2. □

Theorem 4.11 (*reflexive proof is identity*)
If $\cdot \vdash e : \texttt{REQUAL}\ \tau_1\ \tau_2$ then $e \approx \lambda x.x : \texttt{REQUAL}\ \tau_1\ \tau_2$.

Proof sketch
Similar to the proof of Theorem 4.3. □

Furthermore, we can also show that these two definitions of equality are logically equivalent. In particular, we can define F_ω terms i and j that witness the implications in both directions as follows. (For clarity, we write these terms in a Church-style variant, where all type abstractions and applications are explicit.)

$$
\begin{aligned}
& i : \forall a{:}\star.\forall b{:}\star.\texttt{REQUAL}\ a\ b \rightarrow \texttt{EQUAL}\ a\ b \\
& i = \Lambda a{:}\star.\Lambda b{:}\star.\lambda x{:}\texttt{REQUAL}\ a\ b\,.\,\Lambda c{:}\star \rightarrow \star. \\
& \qquad x\ [\lambda b\,.\,c\ a \rightarrow c\ b]\ (\lambda y{:}c\ a\,.\,y)
\end{aligned}
$$

$$
\begin{aligned}
& j : \forall a{:}\star.\forall b{:}\star.\texttt{EQUAL}\ a\ b \rightarrow \texttt{REQUAL}\ a\ b \\
& j = \Lambda a{:}\star.\Lambda b{:}\star.\lambda x{:}\texttt{EQUAL}\ a\ b\,.\,\Lambda c{:}\star \rightarrow \star \rightarrow \star. \\
& \qquad \lambda w{:}(\forall d{:}\star.c\ d\ d)\,. \\
& \qquad\quad x[\lambda c.g\ c\ a \rightarrow g\ c\ b]\ (w[a])
\end{aligned}
$$

Furthermore, by Theorems 4.3 and 4.11, we know that i and j form an isomorphism between the two equality types.

5 Discussion

5.1 *Injectivity*

Although the higher-order types `EQUAL` and `REQUAL` encode type equality, 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:

$$inj : \forall ab\,.(\forall c\,.c\ (a \times \texttt{int}) \rightarrow c\ (b \times \texttt{int})) \rightarrow (\forall c\,.c\ a \rightarrow c\ b)$$

However, it does not seem possible to construct such a term in F_ω or R_ω. Given the ability to write an intensional type constructor (Harper & Morrisett 1995), such as the following, which maps product types to their first component but leaves other types alone,

$$
\begin{array}{lcl}
D & : & \star \rightarrow \star \\
D\ (a \times b) & = & a \\
D\ a & = & a
\end{array}
$$

one could write such a injectivity term (in an explicitly typed calculus) as:

$$inj = \Lambda ab.\lambda x{:}\texttt{EQUAL}\ a\ b.\Lambda c.\lambda y{:}c\ a.x[D]$$

But without such capability, such an injection does not seem possible. On the other hand, we do not know how to show that the type of inj is uninhabited – we cannot assume the existence of a term inj and derive that $(inj, inj) \in \emptyset$ by using the fundamental theorem as we can for other empty types.

In fact, we conjecture that such an injection is consistent with R_ω and F_ω, but we have not extended our parametricity proof to a language with type level type analysis.[5]

The lack of injectivity hinders practical use of the `EQUAL` type. Some authors propose that the `EQUIV` type, which can define injectivity, be used instead. Fortunately, because the typing rules for GADTs in Haskell are more expressive

[5] However, see Washburn and Weirich (Washburn & Weirich 2005) for a related language that does show parametricity in the presence of such a construct.

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

than that of the Church encoding, the `REqual` type in Figure 10 does support injectivity. In particular, the following code typechecks in GHC.

```
inj1 :: REqual (a, c) (b, d) -> REqual a b
inj1 Refl = Refl
```

5.2 Relational interpretation and contextual equivalence

How does the relational interpretation of types given here relate to contextual equivalence? Theorem 3.14 shows that it is sound with respect to our notion of behavioral equivalence. We conjecture that for closed values our behavioral equivalence coincides with contextual equivalence. On the other hand, it is an open problem to determine whether the interpretation of types that we give is complete with respect to contextual equivalence (i.e. contains contextual equivalence). In fact the same problem is open even for System F even without any datatypes or representations. A potential solution to this problem would involve modifying the clauses of the definition that correspond to sums (such as the $[\![+]\!]$ and $\mathscr{R}$ operations) by $\top\top$-closing them as Pitts suggests (Pitts 2000; 2005). 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 $\star$ (this should be 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 Pitts's roadmap) and we have not yet carried out the experiment.

5.3 Representations of polymorphic types and nontermination

R_ω does not include representations of all types for a good reason. While representing function types poses no problem, adding representations of *polymorphic* types has subtle consequences for the semantics of the language.

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

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

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

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

to get an infinite loop:

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

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 (Neis *et al.* 2009) gives a way to reconcile Girard's J operator and parametricity, using step-indexed logical relations to account for nontermination.

Our current proof breaks in the definition of the morphism $\mathscr{R}$ in Figure 9. The application $\mathscr{R}\ (\tau,\ \sigma,\ r)$ 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:

$$\begin{array}{lcl} \mathscr{R} & = & \ldots \text{as before} \ldots \\ & \cup & \{(\mathtt{R_{id}}, \mathtt{R_{id}}) | \cdot \vdash \tau \equiv Rid : \star \wedge \cdot \vdash \sigma \equiv Rid : \star\ \wedge\ r \equiv_{\star} [\![\cdot \vdash Rid : \star]\!].\} \end{array}$$

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

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

5.4 Implicit versus explicit generalization and instantiation

Parametricity in the presence of impure features, such as nontermination or exceptions, is known to be affected by whether type application and generalization is kept explicit or implicit. For example, a term of type $\forall a\,.\,a$ is only inhabited by a diverging term if type generalization is implicit, whereas it may be also be inhabited by a converging term $\Lambda a\,.\,e$ where $e\{\tau/a\}$ has to be diverging for every τ, in an explicit setting. Hence, it is to be expected that the derived free theorems in this paper will only be "morally" true (Danielsson *et al.* 2006) in a setting with nontermination.

5.5 Arbitrary gadts

Equality types, along with existential types and standard recursive datatypes, are the foundation of arbitrary GADTs (Johann & Ghani 2008). In fact, the earliest examples of GADTs were defined in this way (Cheney & Hinze 2003; Xi *et al.* 2003). Therefore, although the language R_ω only contains the specific example of the representation type, the parametricity results in this paper could be extended to languages that include arbitrary GADTs.

The easiest GADTs to incorporate in this way are those that, like representation types, have inductive structure. Such types do not introduce nontermination, so the necessary extensions to the definitions in this paper are localized. Alternatively, we believe that such types may also be defined in R_ω using a Church encoding.

Recursive datatypes require more change to the proofs as they introduce nontermination. Crary and Harper (Crary & Harper 2007) and Ahmed (Ahmed 2006) describe necessary extensions to to support their inclusion.

6 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), our presentation is technically more precise in dealing with equality and well formedness, and employs a dependently typed meta-logic for the interpretation of the morphisms.

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.

In recent work (Voigtländer 2009), Voigtländer shows interesting free theorems about higher-order polymorphic functions where the higher-order types satisfy extra axioms (e.g. they are monads), but he never has to interpret them as nonparametric morphisms as we do – and he elides the formal setup of parametricity altogether.

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 $\texttt{GRel}^\kappa$, corresponds to his definition of (generalized) candidate sets. The important requirement that the generalized morphisms respect equivalence classes of types ($\texttt{wfGRel}^\kappa$) 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 *is*, and hence elides the extra complication of it be defined simultaneously with $\texttt{wfGRel}^\kappa$.

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.

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

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.

Washburn and Weirich give a relational interpretation for a language with nontrivial 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.

Neis *et al.* (2009) show that it *is* possible to reconcile parametricity and ordinary case analysis on types (and not on type representations) using generative types. Going one step further, Neis *et al.* introduce *polarized* logical relations in order to produce more interesting free theorems. For example, the fact that in the presence of type analysis the type $\forall a.a \to a$ is inhabited by terms other than the identity does not preclude the context that *uses* a value of that type to be parametric. Polarized logical relations make the distinction between contexts and expressions explicit, and would be an orthogonal but interesting extension in our setting as well.

7 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 (Appel & McAllester 2001; Johann & Voigtländer 2004; Melliès & Vouillon 2005; Ahmed 2006; Crary & Harper 2007). We believe that the techniques developed here are independent of those for advanced language features.

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. Furthermore and we have shown important applications of parametricity, in particular to reason about the properties of equality types.

Acknowledgments

Thanks to Aaron Bohannon, Jeff Vaughan, Steve Zdancewic, and anonymous reviewers for their feedback and suggestions. Janis Voigtländer brought Kučan's dissertation to our attention. This work was partially supported by NSF grants 0347289, 0702545, and 0716469.

References

Ahmed, A. J. (2006) Step-indexed syntactic logical relations for recursive and quantified types. In *Proceedings of ESOP*. Springer, pp. 69–83.

Appel, A. W. & McAllester, D. (2001) An indexed model of recursive types for foundational proof-carrying code, *ACM Trans. Program. Lang. Syst.*, 23 (5): 657–683.

Baars, A. I. & Swierstra, S. D. (2002) Typing dynamic typing. In *ICFP '02: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 157–166.

Chen, C., Zhu, D. & Xi, H. (2004) Implementing cut elimination: A case study of simulating dependent types in Haskell. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science, vol. 3057. Springer-Verlag, pp. 239–254.

Cheney, J. & Hinze, R. (2002) A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. ACM, pp. 90–104.

Cheney, J. & Hinze, R. (2003) In *First-Class Phantom Types*. CUCIS TR2003-1901. Cornell University.

Crary, K. & Harper, R. (2007) Syntactic logical relations for polymorphic and recursive types, *Electr. Notes Theor. Comput. Sci.*, 172: 259–299. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

Crary, K., Weirich, S. & Morrisett, G. (2002) Intensional polymorphism in type erasure semantics. *J. Funct. Program.*, 12 (6): 567–600.

Crole, R. (1994) *Categories for Types*. Cambridge University Press.

Danielsson, N. A., Hughes, J., Jansson, P. & Gibbons, J. (2006) Fast and loose reasoning is morally correct. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, vol. 41, ACM, pp. 206–217.

Gallier, J. H. (1990) On Girard's "Candidats de Reductibilité". In *Logic and Computer Science*, P. Odifreddi (ed.), The APIC Series 31. Academic Press, pp. 123–203.

Girard, J.-Y. (1972) *Interprétation fonctionelle et é limination des coupures de l'arithmétique d'ordre Supérieur*. PhD thesis, Université Paris VII.

Harper, R. & Mitchell, J. C. (1999) Parametricity and variants of Girard's J operator, *Inf. Process. Lett.*, 70 (1): 1–5.

Harper, R. & Morrisett, G. (1995) Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, pp. 130–141.

Harper, R. & Pfenning, F. (2005) On equivalence and canonical forms in the LF type theory, *ACM Trans. Comput. Logic*, 6 (1): 61–101.

Hinze, R. (2002) Polytypic values possess polykinded types, *Sci. Comput. Program.*, 43 (2–3): 129–159. MPC Special Issue.

Johann, P. & Ghani, N. (2008) Foundations for structured programming with gadts. In *POPL*, Necula, G. C. & Wadler, P. (eds), ACM, pp. 297–308.

Johann, P. & Voigtländer, J. (2004) Free theorems in the presence of seq. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, pp. 99–110.

Jones, S. P., Vytiniotis, D., Weirich, S. & Washburn, G. (2006) Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 50–61.

Kiselyov, O., Lämmel, R. & Schupke, K. (2004) Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*. ACM, pp. 96–107.

Kleene, S. C. (1967) *Mathematical Logic*. Wiley.

Kučan, J. (1997) *Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS Transform and Free Theorems*. PhD thesis, Massachusetts Institute of Technology.

Meijer, E. & Hutton, G. (1995) Bananas in space: Extending fold and unfold to exponential types. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture*. ACM, pp. 324–333.

Melliès, P.-A. & Vouillon, J. (2005) Recursive polymorphic types and parametricity in an operational framework. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. IEEE Computer Society, pp. 82–91.

Neis, G., Dreyer, D. & Rossberg, A. (2009) Non-parametric parametricity. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 135–148.

Paulin-Mohring, C. (1993) Inductive definitions in the system Coq: Rules and properties. In *International Conference on Typed Lambda Calculi and Applications, TLCA '93*, Lecture Notes in Computer Science, vol. 664. Springer, pp. 328–345.

Pitts, A. M. (2000) Parametric polymorphism and operational equivalence, *Math. Struct. Comput. Sci.*, 10: 321–359.

Pitts, A. M. (2005) Typed operational reasoning. , In *Chap. 7 of: Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed), MIT Press, pp. 245–289.

Plotkin, G. & Abadi, M. (1993) A logic for parametric polymorphism. In *International Conference on Typed Lambda Calculi and Applications*. Springer-Verlag, pp. 361–375.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. In *Information Processing '83*. Proceedings of the IFIP 9th World Computer Congress. North-Holland, pp. 513–523.

Sheard, T. & Pasalic, E. (2004) Meta-programming with built-in type equality. In *Proc 4th International Workshop on Logical Frameworks and Meta-languages (LFM'04), Cork*. pp. 106–124.

Spławski, Z. & Urzyczyn, P. (1999) Type fixpoints: Iteration versus recursion. *Fourth ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 102–113.

Takeuti, I. (2001) *The Theory of Parametricity in Lambda Cube (Towards new interaction between category theory and proof theory)*. Technical Report. Kyoto University Research Information Repository.

Voigtländer, J. (2009) Free theorems involving type constructor classes: functional pearl. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 173–184.

Vytiniotis, D. & Weirich, S. (2007) Free theorems and runtime type representations, *Electron. Notes Theor. Comput. Sci.*, 173: 357–373.

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press

Wadler, P. (1989) Theorems for free! In *FPCA89: Conference on Functional Programming Languages and Computer Architecture*. ACM, pp. 347–359.

Washburn, G. & Weirich, S. (2005) Generalizing parametricity using information flow. In *The Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*. Chicago, IL: IEEE Computer Society Press, pp. 62–71.

Weirich, S. (2001) Encoding intensional type analysis. In *10th European Symposium on Programming*, Sands, D. (ed), pp. 92–106.

Weirich, S. (2004) Type-safe cast, *J. Funct. Program.* 14 (6): 681–695.

Xi, H., Chen, C. & Chen, G. (2003) Guarded recursive datatype constructors. In *POPL*. ACM, pp. 224–235.

Yang, Z. (1998) Encoding types in ML-like languages. In *1998 ACM SIGPLAN International Conference on Functional Programming*. ACM SIGPLAN Notices 34, pp. 289–300.

Appendix A Generalized relations, in Coq

A Coq definition of `GRel`, `wfGRel`, and `eqGRel` ($\equiv_\kappa$), follows.[6] 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 *)
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.
```

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

```
(* 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. The rel datatype 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. 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) :=
  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 3.2 *)
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 generalized relations and equality (Fig. 7) *)
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
```

https://doi.org/10.1017/S0956796810000079 Published online by Cambridge University Press