Abstract

neric functions can specialize their behaviour depending e types of their arguments, and can even recurse over the ure of the types of their arguments. Such functions can be ammed using type representations. *Generic functions pro-med this way possess certain parametricity properties, which ne interesting in the presence of higher-order polymorphism. s paper, we give a rigorous roadmap through the proof of metricity for a calculus with higher-order polymorphism and epresentations. We then use parametricity to derive the cor-ss of* type-safe cast.

Introduction

neric programming refers to the ability to specialize the iour of functions based on the *types* of their arguments. are many tools, libraries, and language extensions that sup-eneric programming, particularly for the Haskell program-language (4; 6; 16; 8; 22; 35; 34). Although the theory nderlies these mechanisms differs considerably, the common f these mechanisms is to eliminate boilerplate code. Exam-f generic programs range from generic equality functions, allers, reductions and maps, to application-specific traver-nd queries (22), user interface generators (1), XML-inspired ormations (21), and compilers (5).

presentation types (10) are an attractive mechanism for c programming. The key idea is simple: because the be-ur of parametrically polymorphic functions cannot be influ-by the types at which they are instantiated, generic func-dispatch on term arguments that *represent* types. Repre-ion types were originally proposed in the context of type-ving compilation, but they may be encoded in Haskell in l ways (6; 35; 34). The most natural implementation uses *alized algebraic datatypes* (GADTs) (7; 29), a recent exten-o the Glasgow Haskell Compiler (GHC).

r example:

```
 R a where
nt   :: R Int
nit  :: R ()
rod  :: R a -> R b -> R (a,b)
um   :: R a -> R b -> R (Either 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 :: ra0 -> ra0'
      g <- cast ra0 ra0'
      h :: rb0 -> rb0'
      h <- cast rb0 rb0'
      Just (\(a,b) -> (g a, h b))
cast (Rsum ra0 rb0) (Rsum ra0' rb0') =
 do g <- cast ra0 ra0'
    h <- cast rb0 rb0'
    Just (\x -> case x of
                 Left a -> Left (g a)
                 Right b -> Right (h b))
cast _ _ = Nothing
```

Figure 1: `cast`

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

In this paper, we focus on generic *type-safe* cast, which compares two different type representations and, if they match, produces a coercion function from one type to the other. Previously, Weirich (33) defined two different versions of type-safe cast, `cast` and `gcast`, shown in Figures 1 and 2. Our implementations differ slightly from Weirich's—namely they use Haskell's `Maybe` type to account for potential failure, instead of an `error` primitive—but the essential structure is the same.

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.

Alternatively, `gcast` produces a coercion function that never needs to decompose (or even evaluate) its argument. The key ingredient is the use of the higher-kinded type argument `c`, that allows `gcast` to return a coercion from `c a` to `c b`. As Baars and Swierstra (4), and Cheney and Hinze (6) point out, `gcast` corresponds to *Leibniz equality*. From an implementation point of view,

```
ype CL f c a d = CL (c (f d a))
 (CL e) = e
ype CR f c a d = CR (c (f a d))
 (CR e) = e

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

Figure 2: `gcast`

pe constructor `c` allows the recursive calls to `gcast` to cre- coercion that changes the type of a *part* of its argument. In a ive call, the instantiation of `c` hides the parts of the type that n unchanged. The case for sums is operationally identical, e omit the intermediate type annotations and compose all the nediate functions directly.

n important difference between the two versions has to do correctness. When the type comparison succeeds, type-safe hould behave like an identity function. Informal inspection sts that both implementations do so. However in the case of , it is possible to mess up. In particular, it is type sound to e the clause for `Rint` with:

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

e type of `gcast` more strongly constrains its implementa- We could not replace the first clause with

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

cause the type of the returned coercion must be `c Int -> c Int`, nt -> Int`. Informally, we can argue that the only coer- unction that could be returned *must* be an identity function s abstract. The only way to produce a result of type `c Int` unting divergence) is to use exactly the one that was sup-

ibutions. In this paper, we make the above arguments pre- nd rigorous. In particular, we show using a *free theorem* (31) f `gcast` returns a coercion function then that function must identity function. In fact, because we use a free theorem, unction with the type of `gcast` must behave in this manner. so, we start with a formalization of the λ-calculus with rep- ation types and higher-order polymorphism, called R_ω (10) on 2). We then extend Reynolds's abstraction theorem (28) language (Section 2). Reynolds's abstraction theorem, also ed to as the "parametricity theorem" (31), asserts that every yped expression of the second-order polymorphic λ-calculus em F) (13) satisfies a particular property directly derivable its type. After proving a version of the abstraction theorem $_\omega$, we show how to apply it to the type of `gcast` to get the d results (Section 3).

Our broader goal is not just to prove the correctness of `gcast`—there are certainly simpler ways to do so, and there are some limitations in our approach, as we describe in Section 4. Instead, our intention is to demonstrate that it is possible to use parametricity and free theorems to reason about generic functions written with representation types. In previous work (30), which was limited to the case of second-order polymorphism, we had difficulty finding free theorems for generic functions that were not trivial. This paper demonstrates a fruitful example of such reasoning when higher-order polymorphism is present, and encourages the use of variations of this method to reason about other generic functions.

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

Finally, although the ideas that we use to define parametricity for F_ω are folklore, they appear in few sources in the literature. Therefore, an additional contribution of this work is an accessible roadmap to the proof of parametricity for higher-order polymorphism using the technique of syntactic logical relations. Our development is most closely related to the proof of strong normalization of F_ω by Jean Gallier (12), but we are more explicit about the requirements from the meta-logic and the well-formedness of our definitions. Therefore, we expect our development to be particularly well-suited for mechanical verification in proof assistants, such as Coq (`http://coq.inria.fr`).

2. Parametricity for R_ω

The $\mathbf{R}_\omega$ ***calculus.*** We begin with a formal description of the R_ω calculus, an extension of Curry-style F_ω (13). The syntax of this language appears in Figure 3, but for space reasons, the semantics appears in Appendix B. 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, they simplify programming—for example, in Figure 2 we had to introduce the constructors `CL` and `CR` only because Haskell does not include type-level λ-abstractions.

Type constructor constants, $\mathcal{K}$, include standard operators, plus reprsentation types R. In the following, we write $\to$, $\times$, and $+$ using infix notation and associate applications of $\to$ to the right. We treat impredicative polymorphism with an infinite family of universal type constructors $\forall_\kappa$ indexed by kinds. We write $\forall(a_1{:}\kappa_1)\ldots(a_n{:}\kappa_n)\,.\,\sigma$ to abbreviate $\forall_{\kappa_1}(\lambda a_1{:}\kappa_1\,.\,\ldots\forall_{\kappa_n}(\lambda a_n{:}\kappa_n\,.\,\sigma)\ldots)$.

R_ω expressions e include abstractions, products, sums, integers and unit. For simplicity, type abstractions and type applications are implicit. R_ω includes type representations $\mathtt{R}_{\mathtt{int}}$, $\mathtt{R}_{()}$, $\mathtt{R}_\times$ and $\mathtt{R}_+$, which must be fully applied to their arguments. We do not include representations for function or polymorphic types in R_ω as neither are that useful for generic programming. The former can be added in a straightforward manner, but the latter significantly changes the semantics of the language, as we discuss in Section 4. The language is terminating, but includes a term `typerec` that can perform primitive recursion on type representations, and includes branches for each possible representation. For completeness, we give the R_ω implementations of *cast* and *gcast* in Appendix A.

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

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

e operational semantics of the language is standard, so we present the rules for typerec in Figure 8 (Appendix B). tially typerec performs a fold over its type representation ent. We use a big-step, call-by-name formalization. We use v for R$_\omega$ values, the syntax of which is also given in Ap- x B.

e static semantics of R$_\omega$ contains judgments for kinding, quivalence, and typing. Each of these judgements uses a uni- nvironment, Γ, containing bindings for type variables $(a{:}\kappa)$ rm variables $(x{:}\tau)$. We use $\cdot$ for the empty environment and $a\#\Gamma$ to mean that a does not appear anywhere in Γ. The g judgement $\Gamma \vdash \tau : \kappa$ (in Figure 9, Appendix B) states that well-formed type of kind κ and ensures that all the free type les of the type τ appear in the environment Γ with correct We refer to arbitrary *closed* types of a particular kind with llowing predicate:

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

e typing judgement has the form $\Gamma \vdash e : \tau$ and Figure 11, ndix B. The interesting typing rules are the introduction limination forms for type representations. The rest of this relation is standard. Notably, our typing relation includes ndard conversion rule:

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

udgement $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$ defines type equivalence as gruence relation that includes β-conversion for types. We ts definition in Figure 10, Appendix B. The presence of le T-EQ is important for R$_\omega$ because it allows expressions typed with any member of an equivalence classes of types. ehavior fits our intuition, but complicates the formalization rametricity; a significant part of this paper is devoted to ications introduced by type equivalence.

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

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

heorems result from unfolding the definition of the interpre- of types (which appears in Figure 5, using Definition 2.6). ver, before we can present that interpretation, we must first n a number of auxiliary concepts.

st, we must define a (meta-logical) type, $\mathtt{GRel}^\kappa$, to describe terpretation of types of arbitrary kind. Only types of kind interpreted as term relations—types of higher kind are in- ted as sets of morphisms. (To distinguish between R$_\omega$ and logical functions, we use the term *morphism* for the latter.)

$$
\begin{array}{lcl}
r \in \mathtt{VRel}(\tau_1, \tau_2) & \triangleq & \forall (e_1, e_2) \in r, \\
 & & \quad e_1 \text{ and } e_2 \text{ are values} \wedge \\
 & & \quad (\cdot \vdash e_1 : \tau_1) \wedge (\cdot \vdash e_2 : \tau_2) \\
 & & \\
(\tau_1, \tau_2, r) \in \mathtt{wfGRel}^\star & \triangleq & r \in \mathtt{VRel}(\tau_1, \tau_2) \\
(\tau_1, \tau_2, r) \in \mathtt{wfGRel}^{\kappa_1 \to \kappa_2} & \triangleq & \\
\multicolumn{3}{l}{\quad \text{for all } \rho \in \mathtt{wfGRel}^{\kappa_1}, (\tau_1\ \rho^1, \tau_2\ \rho^2, r\ \rho) \in \mathtt{wfGRel}^{\kappa_2} \wedge} \\
\multicolumn{3}{l}{\quad \text{for all } \pi \in \mathtt{wfGRel}^{\kappa_1}, \rho \equiv \pi \Longrightarrow r\ \rho \equiv_{\kappa_2} r\ \pi} \\
 & & \\
r \equiv_\star s & \triangleq & \text{for all } e_1\ e_2,\ (e_1, e_2) \in r \iff (e_1, e_2) \in s \\
r \equiv_{\kappa_1 \to \kappa_2} s & \triangleq & \text{for all } \rho \in \mathtt{wfGRel}^{\kappa_1},\ (r\ \rho) \equiv_{\kappa_2} (s\ \rho) \\
 & & \\
\rho \equiv \pi & \triangleq & (\cdot \vdash \rho^1 \equiv \pi^1 : \kappa) \wedge (\cdot \vdash \rho^2 \equiv \pi^2 : \kappa) \wedge \hat{\rho} \equiv_\kappa \hat{\pi}
\end{array}
$$

Figure 4: Well-formed generalized relations and equality

For example, the interpretation of a type of kind $\star \to \star$ is the set of morphisms that take term relations to appropriate term relations.

2.2 Definition [(Typed-)Generalized Relations]:

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

The notation $\mathcal{P}(\mathtt{term} \times \mathtt{term})$ stands for the space of binary relations on terms of R$_\omega$. We use $\supset$ for the function space constructor of our meta-logic, to avoid confusion with the $\to$ constructor of R$_\omega$.

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

At first glance, Definition 2.2 seems strange because it returns the term relation space at kind $\star$, while at higher kinds it returns a particular function space of the meta-logic. These two do not necessarily "type check" with a common type. However, in an expressive enough meta-logic, such as CIC (26) or ZF set theory, such a definition is indeed well-formed, as there exists a type containing both spaces (for example Type in CIC [1], or pure ZF sets in ZF set theory). In contrast, in HOL it is not clear how to build a common type "hosting" the interpretations at all kinds.

Unfortunately, not all objects of $\mathtt{GRel}^\kappa$ are suitable for the interpretation of types. In Figure 4 we define *well-formed generalized relations*, $\mathtt{wfGRel}^\kappa$, a predicate on objects in $\mathtt{TyGRel}^\kappa$. We define this predicate mutually with extensional equality on generalized relations ($\equiv_\kappa$) and on typed-generalized relations ($\equiv$). Because our $\mathtt{wfGRel}^\kappa$ conditions depend on equality for type $\mathtt{GRel}^\kappa$, we cannot include those conditions in the definition of $\mathtt{GRel}^\kappa$ itself.

At kind $\star$, $(\tau_1, \tau_2, r) \in \mathtt{wfGRel}^\star$ 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 $\mathtt{TyGRel}^{\kappa_1}$, then the

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

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

Figure 5: Relational interpretation of R_ω

must also be well-formed. (We project the three components with the notations ρ^1, ρ^2 and $\hat{\rho}$ respectively.) Second, for any of equivalent triples, ρ and π, the results $r\ \rho$ and $r\ \pi$ must e equal. This condition asserts that morphisms that satisfy $\mathtt{el}^\kappa$ *respect* the type equivalence classes of their type argu-.

uality on generalized relations is also indexed by kinds; for vo $r, s \in \mathtt{GRel}^\kappa$, the proposition $r \equiv_\kappa s$ asserts that the two alized relations are extensionally equal. Extensional equality en generalized relations asserts that at kind $\star$ the two relation nents denote the same set, whereas at higher kinds it asserts he relation arguments return equal results, when given the argument ρ which must satisfy the $\mathtt{wfGRel}^{\kappa_1}$ predicate. ping the requirement that ρ be well-formed is not possible, discuss in the proof of Coherence, Theorem 2.11.

uality for typed-generalized relations, $\rho \equiv \psi$, is defined wise. Importantly, the $\mathtt{wfGRel}^\kappa$ predicate respects this alence.

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

e turn now to the key to the abstraction theorem, the inter-ion of R_ω types as relations between closed terms. This in-tation makes use of a *substitution* δ from type variables to generalized relations. We write $dom(\delta)$ for the domain of bstitution, that is, the subset of all type variables on which ot the identity. We use $\cdot$ for the identity-everywhere substi-, and write δ, $a \mapsto \rho$ for the extension of δ that maps a to ρ quire that $a \notin dom(\delta)$. If $\delta(a) = (\tau_1, \tau_2, r)$, we define the ons $\delta^1(a) = \tau_1$, $\delta^2(a) = \tau_2$, and $\hat{\delta}(a) = r$. We also define nd $\delta^2\tau$ to be the extension of δ^1 and δ^2 to types τ.

efinition [Substitution kind checks in environment]: We at a substitution δ *kind checks in an environment* Γ, and write $\mathtt{ubst}_\Gamma$, when $dom(\delta) = dom(\Gamma)$ and for every $(a{:}\kappa) \in \Gamma$, ve $\delta(a) \in \mathtt{TyGRel}^\kappa$.

e interpretation of R_ω types is shown in Figure 5 and is de-nductively over kinding derivations for types. The interpre-function $[\![\cdot]\!]_\cdot$ accepts a derivation $\Gamma \vdash \tau : \kappa$, and a substi-$\delta \in \mathtt{Subst}_\Gamma$ and returns a generalized relation at kind κ, the meta-logical type, $\mathtt{Subst}_\Gamma \supset \mathtt{GRel}^\kappa$. We write the δ ent as a subscript to $[\![\Gamma \vdash \tau : \kappa]\!]$.

hen τ is a type variable a we project the relation compo-out of $\delta(a)$. In the case where τ is a constructor $\mathcal{K}$, we call xiliary function $[\![\mathcal{K}]\!]$, shown in Figure 6. For an application, we apply the interpretation of τ_1 to appropriate type argu-and the interpretation of τ_2. Type-level λ-abstractions are reted 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 $\delta \in \mathtt{Subst}_\Gamma$.

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

Figure 6: Operations of type constructors on relations

Furthermore, the interpretation of types gives equivalent results when given equal substitutions. We define equivalence for substitutions, $\delta_1 \equiv \delta_2$, pointwise.

2.5 Lemma: If $\Gamma \vdash \tau : \kappa$ and $\delta_1 \vDash \Gamma$, $\delta_2 \vDash \Gamma$ and $\delta_1 \equiv \delta_2$, it is the case that $[\![\Gamma \vdash \tau : \kappa]\!]_{\delta_1} \equiv_\kappa [\![\Gamma \vdash \tau : \kappa]\!]_{\delta_2}$.

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

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

For integer and unit types, $[\![\mathtt{int}]\!]$ and $[\![\,()\,]\!]$ give the identity value relations respectively on int and (). The operation $[\![\to]\!]$ lifts ρ and π to a new relation between functions that send related arguments in $\hat{\rho}$ to related results in $\hat{\pi}$. The operation $[\![\times]\!]$ lifts ρ and π to a relation between products such that the first components of the products belong in $\hat{\rho}$, and the second in $\hat{\pi}$. The operation $[\![+]\!]$ on ρ and π consists of all the pairs of left injections between elements of $\hat{\rho}$ and right injections between elements of $\hat{\pi}$. Because and products are call-by-name, their subcomponents must from the computational lifting of the value relations. For the nstructor, since its kind is $(\kappa \to \star) \to \star$ we define $\llbracket \forall_\kappa \rrbracket$ a morphism that, given a $\mathtt{TyGRel}^{\kappa \to \star}$ argument ρ, returns tersection over all well-formed π of the applications of $\hat{\rho}$ The requirement that $\pi \in \mathtt{wfGRel}^{\kappa}$ is necessary to show ne interpretation of the $\forall_\kappa$ constructor is itself well-formed ma 2.7).

r the case of representation types R, the definition relies on xiliary morphism $\mathcal{R}$, defined by induction on the size of normal form of its type arguments. The interesting property this definition is that it imposes requirements on the rela- argument r in every case of the definition. For example, in st clause of the definition of $\mathcal{R}\ (\tau, \sigma,\ r)$, the case for integer entations, r is required to be equal to $\llbracket \mathtt{int} \rrbracket$. In the case for epresentations, r is required to be equal to $\llbracket\, \mathtt{()}\, \rrbracket$. In the case oducts, r is required to be some product of relations, and in se for sums, r is required to be some sum of relations.

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

emma [Constructor interpretation is well-formed]: For $(\mathcal{K}, \mathcal{K}, \llbracket \mathcal{K} \rrbracket) \in \mathtt{wfGRel}^{kind(\mathcal{K})}$.

roof of this lemma appears in Appendix D.

ing Lemma 2.7, we wish to show that the interpretation of pe is well-formed. This result only holds for substitutions ap type variables to *well-formed* generalized relations.

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

emma [Type interpretation is well-formed]: Assume that $: \kappa$ and $\delta \vDash \Gamma$. Then $\delta^2\tau, \llbracket \Gamma \vdash \tau : \kappa \rrbracket_\delta) \in \mathtt{wfGRel}^{\kappa}$.

: Straightforward induction over the type well-formedness tions, appealing to Lemma 2.7. The only interesting case is se for type abstractions, which follows from Lemma 2.5 and a 2.3. □

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

emma [Compositionality]: If $\delta \vDash \Gamma$, $\Gamma, a{:}\kappa_a \vdash \tau : \kappa$, $_a : \kappa_a$, and $r_a = \llbracket \Gamma \vdash \tau_a : \kappa_a \rrbracket_\delta$ then $\kappa_a \vdash \tau : \kappa \rrbracket_{\delta, a \mapsto (\delta^1\tau_a, \delta^2\tau_a, r_a)} \equiv_\kappa \llbracket \Gamma \vdash \tau\{\tau_a/a\} : \kappa \rrbracket_\delta$

nally, we show that the interpretation of types respects the alence classes of types. The proof of this theorem appears in ndix D.

heorem [Coherence]: If $\Gamma \vdash \tau_1 : \kappa$, $\delta \vDash \Gamma$, and $\Gamma \vdash \tau_1 \equiv$, then $\llbracket \Gamma \vdash \tau_1 : \kappa \rrbracket_\delta \equiv_\kappa \llbracket \Gamma \vdash \tau_2 : \kappa \rrbracket_\delta$

th the above definitions and properties, we may now state straction theorem.

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

count for open terms, the theorem must be generalized in the rd manner. The proof then proceeds by induction on the typrivation, with an inner induction for the case of `typerec` ssions. It crucially relies on Coherence (Theorem 2.11) for the case of rule T-EQ. The generalization of the theorem and a proof sketch can be found in Appendix D.

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

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

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

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

$$(e\ e_1, e\ e_2) \in \mathcal{C}\,\llbracket c{:}\star \to \star \vdash c\ \mathtt{()} : \star \rrbracket_{c \mapsto \rho_c} \qquad (1)$$

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

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

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

$$\llbracket c{:}\star \to \star \vdash c\ \mathtt{()} : \star \rrbracket_{c \mapsto (\tau_c, \tau_c, f_c)} = \{(v, u) \mid \cdot \vdash v : \tau_c\ \mathtt{()}\}$$

Because (u, u) is in this set, we can pick e_1 and e_2 both to be u and use (1) to show that that $e\ e_2 \Downarrow u$, hence $e\ u \Downarrow u$ as required. □

We observe that to derive our result we had to instantiate a generalized relation to be a morphism that is itself not representable in F$_\omega$. In particular, this morphism is not parametric: it behaves differently at type $\mathtt{()}$ than at other types. Hence, despite the fact that we are discussing about a theorem for an F$_\omega$ type, we needed morphisms at higher kinds to accept *both types and morphisms* as arguments. This same idea will be used with a free theorem for the *gcast* function in the next section.

3. Free theorem for generic cast

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

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

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

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

Then, either the cast fails and

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

cast succeeds and

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

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

emma: If $\cdot \vdash e_{ra} : \mathrm{R}\ \tau_a$, $\cdot \vdash e_{rb} : \mathrm{R}\ \tau_b$, and $gcast\ e_{ra}\ e_{rb} \Downarrow$ then it follows that $\cdot \vdash \tau_a \equiv \tau_b : \star$.

: From the assumptions we get that *for any* $\tau_c \in \mathtt{ty}(\star \to \star)$, he case that $\cdot \vdash gcast\ e_{ra}\ e_{rb} : \mathtt{()} + (\tau_c\ \tau_a \to \tau_c\ \tau_b)$. Asby contradiction now that $\cdot \not\vdash \tau_a \equiv \tau_b : \star$. Then we instanhe abstraction theorem with $e^1_{ra} = e^2_{ra} = e_{ra}$, $e^1_{rb} = e^2_{rb} =$ $_a = (\tau_a, \tau_a, [\![\cdot \vdash \tau_a : \star]\!]_{\cdot})$, $\rho_b = (\tau_b, \tau_b, [\![\cdot \vdash \tau_b : \star]\!]_{\cdot})$ and $(\lambda a{:}\star\,.\,\mathtt{()}, \lambda a{:}\star\,.\,\mathtt{()}, r_c)$ where

$$_c\ (\tau,\ \sigma,\ r)\ =\ \text{if } (\cdot \vdash \tau \equiv \tau_a : \star \wedge \cdot \vdash \sigma \equiv \tau_a : \star)$$
$$\text{then } [\![\cdot \vdash (\lambda a{:}\star\,.\,\mathtt{()})\ \tau_a : \star]\!]_{\cdot} \text{ else } \emptyset$$

an confirm that $\rho_c \in \mathtt{wfGRel}^{\star\to\star}$ Moreover $(e_{ra}, e_{ra}) \in$ $_a)$ by the abstraction theorem, and similarly $(e_{rb}, e_{rb}) \in$ $_b)$. Then by the free theorem for *gcast* above we know ince $((), ()) \in \mathcal{C}(f_c\ \rho_a)$, we have $(e\ (), e\ ()) \in \mathcal{C}(f_c\ \rho_b)$ equal to both e'_1 and e'_2 in the theorem for *gcast*). But, if $\equiv \tau_b$ then $\mathcal{C}(f_c\ \rho_b) = \emptyset$, a contradiction. □

e can now show our important result about *gcast*: if *gcast* eds and returns a conversion function, then that function *must* e as the identity. Note that if the type representations agree, nnot conclude that *gcast* will succeed—it may well return implementation of *gcast* may always fail for any pair of ents and still be well typed.

emma [Correctness of *gcast*]: If $\cdot \vdash e_{ra} : \mathrm{R}\ \tau_a$, $\cdot \vdash e_{rb}$: $gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e$, and e_a is such that $\cdot \vdash e_a : \tau_c\ \tau_a$, $_a \Downarrow w$, then $e\ e_a \Downarrow w$.

: First, by Lemma 3.1 we get that $\cdot \vdash \tau_a \equiv \tau_b : \star$. We hen instantiate the free theorem for the type of *gcast* as in a 3.1. and pick the same instantiation for types and relations t for the instantiation of c. We choose c to be instantiated $_c = (\tau_c, \tau_c, f_c)$ where f_c is:

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

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

$$gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e_1 \tag{2}$$
$$gcast\ e_{ra}\ e_{rb} \Downarrow \mathtt{inr}\ e_2 \tag{3}$$
$$\forall (e'_1, e'_2) \in \mathcal{C}(f_c\ \rho_a), (e_1\ e'_1, e_2\ e'_2) \in \mathcal{C}(f_c\ \rho_b) \tag{4}$$

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

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

4. Discussion

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

To demonstrate these complications, consider what would happen if we added the representation $\mathrm{R}_{\mathtt{id}}$ of type $\mathrm{R}\ \ Rid$ to R_ω, and extended `typerec` and *gcast* accordingly, where *Rid* abbreviates the type $(\forall(a{:}\star)\,.\,\mathrm{R}\ a \to a \to a)$. Then we could encode an infinite loop in R_ω, based on an example by Harper and Mitchell (14). This example begins by using *gcast* to enable a self-application term with a concise type.

$$delta :: \forall a{:}\star\,.\,\mathrm{R}\ a \to a \to a$$
$$delta\ ra =\ \mathtt{case}\ (gcast\ \mathrm{R}_{\mathtt{id}}\ ra)\ \mathtt{of}\ \{\mathtt{inr}\ y.y\ (\lambda x\,.\,x\ \mathrm{R}_{\mathtt{id}}\ x);$$
$$\mathtt{inl}\ z.(\lambda x\,.\,x)\ \}$$

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

$$delta\ \mathrm{R}_{\mathtt{id}}\ delta \cong (\lambda x\,.\,x\ \mathrm{R}_{\mathtt{id}}\ x)\ delta \cong delta\ \mathrm{R}_{\mathtt{id}}\ delta$$

This example demonstrates that we cannot extend the relational interpretation to $\mathrm{R}_{\mathtt{id}}$ and the proof of the abstraction theorem in a straightforward manner as our proof implies termination. Thay does not mean that we cannot give any relational interpretation to $\mathrm{R}_{\mathtt{id}}$, only that our proof would have to change significantly.

Our current proof breaks in the definition of the morphism $\mathcal{R}$ in Figure **??**. The application $\mathcal{R}\ (\tau,\ \sigma,\ r)$ depends on whether r can be constructed as an application of morphisms $[\![\mathtt{int}]\!]$, $[\![\mathtt{()}]\!]$, $[\![\times]\!]$, and $[\![+]\!]$. If we are to add a new representation constructor $\mathrm{R}_{\mathtt{id}}$, we must restrict r in a similar way. To do so, it is tempting to add:

$$\mathcal{R}\ =\ \ldots \text{as before} \ldots$$
$$\cup\ \{(\mathrm{R}_{\mathtt{id}}, \mathrm{R}_{\mathtt{id}}) \mid \cdot \vdash \tau \equiv Rid : \star \wedge \cdot \vdash \sigma \equiv Rid : \star \wedge$$
$$r \equiv_\star [\![\cdot \vdash Rid : \star]\!]_{\cdot}\}$$

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

A different question is what class of polymorphic types *can* we represent with our current methodology (i.e. without breaking strong normalization)? The answer is that we can represent polymorphic types as long as those types contain only representations of *closed* types. For example, the problematic behaviour above was caused because the type $\forall a\,.\,\mathrm{R}\ \ a \to a \to a$ includes $\mathrm{R}\ \ a$, the representation of a quantified type. Such behaviour cannot happen when we only include representations of types such as $\mathrm{R}(\mathrm{R}\ \ \mathtt{int})$, $\forall a\,.\,a \to a$, $\forall a\,.\,a \to \mathrm{R}\ \mathtt{int} \to a$, or even $\forall a\,.\,a$. We can still give a definition of $\mathcal{R}$ that calls recursively the main interpretation function, but the definition must be shown well-founded using a more elaborate metric on types.

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 (24), it can be found in few sources in the literature.

Kučan (20) interprets the higher-order polymorphic λ-calculus within a second-order logic in a way similar to ours. However, the type arguments (which are important for our examples) are missing from the higher-order interpretations, and it is not clear that the particular second-order logic that Kučan employs is expressive enough to host the large type of generalized relations. On the

hand, Kučan's motivation is different: he shows the corre-
ence between free theorems obtained directly from algebraic
pe signatures and those derived from Church encodings.

llier gives a detailed formalization (12) closer to ours, al-
h his motivation is a strong normalization proof for F_ω,
on Girard's reducibility candidates method, and not free-
m reasoning about F_ω programs. Therefore the interpreta-
at he gives is a unary instead of binary relation. Our induc-
efinition of $\texttt{GRel}^\kappa$, corresponds to his definition of (general-
candidate sets. The important requirement that the general-
morphisms respect equivalence classes of types ($\texttt{wfGRel}^\kappa$) is
resent in this formalization (Definition 16.2, Condition (4)).
theless there is no explicit account of what equality means,
hat assumptions are made about the meta-logic. In contrast,
plicitly define extensional equality for $\texttt{GRel}^\kappa$ with the ex-
mplication that this must be given simultaneously with the
tion of $\texttt{wfGRel}^\kappa$.

logic for reasoning about parametricity, that extends the
-Plotkin logic (27) to the λ-cube has been proposed in a
script by Takeuti (18). Crole presents in his book (11) a cat-
al interpretation of higher-order polymorphic types, which
presumably be instantiated to the concrete syntactic rela-
used here.

ncerning the interpretation of representation types, this pa-
xtends the ideas developed in previous work by the au-
(30) to a calculus with higher-order polymorphism.

similar (but more general) approach of performing recur-
ver the type structure of the arguments for generic program-
has been employed in Generic Haskell. Free theorems about
c functions written in Generic Haskell have been explored
nze (16). Hinze derives equations about generic functions
neralizing the usual equations for base kinds using an ap-
ate logical relation at the type level, assuming a cpo model,
ing the main property for the logical relation, and assuming
typic fixpoint induction scheme. Our approach relies on no
assumptions, and our goal is slightly different: While Hinze
o generalize behaviour of Generic Haskell functions from
kind to higher kinds, we are more interested in investigating
straction properties that higher-order types carry. Represen-
types simply make programming interesting generic func-
possible.

nally, Washburn and Weirich give a relational interpretation
anguage with non-trivial type equivalence (32), but without
fication over higher-kinded types. To deal with the compli-
s of type equivalence that we explain in this paper, Washburn
Veirich use canonical forms of types (β-normal η-long forms
es (15)) as canonical representatives of equivalence classes.
gh perhaps more complicated, our analysis (especially outlin-
e necessary wfGRel conditions) provides better insight on
le of type equivalence in the interpretation of higher-order
orphism.

e work. In order for the technique in this paper to evolve
easoning technique for Haskell, several limitations need to
dressed. If we wished to use these results to reason about
ll implementations of gcast, we must extend our model
lude more—in particular, general recursion and recursive
(25; 19; 3; 2; 9). We believe that the techniques developed
re independent of those for advanced language features.

other Haskell feature lacking from R_ω is support for gen-
e types. In Haskell, these are newtypes and datatype defini-
where each declaration creates a new type that is structurally
rphic to existing types, but not equal. Dealing with these
pes in generic programming is tricky—the desired behaviour
is that generic functions should automatically extend to new type definitions based on its isomorphic structure, optionally allowing "after-the-fact" specialization for specific types (23; 17; 34). However, techniques that allow this behavior cannot define gcast. As a result, generic programming libraries that depend on gcast (22) implement it as a language extension, not directly in Haskell.

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

Acknowledgments. Thanks to Aaron Bohannon, Jeff Vaughan, Steve Zdancewic, and the anonymous POPL 2008 referees for their comments. Janis Voigtländer brought Kučan's dissertation to our attention.

References

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

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

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

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

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

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

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

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

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

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

[11] Roy Crole. *Categories for Types*. Cambridge University Press, 1994.

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

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

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

obert Harper and Frank Pfenning. On equivalence and canonical
orms in the LF type theory. *ACM Trans. Comput. Logic*, 6(1):61–
01, 2005.

alf Hinze. Polytypic values possess polykinded types. *Science
f Computer Programming*, 43(2–3):129–159, 2002. MPC Special
sue.

tefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Ro-
riguez. Generic views on data types. In *Mathematics of Program
onstruction, 8th International Conference, MPC 2006, Kuressaare,
stonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes
n Computer Science*. Springer, 2006.

akeuti Izumi. The theory of parametricity in lambda cube.
raft available at `http://www.sato.kuis.kyoto-u.ac.
p/~takeuti/art`.

atricia Johann and Janis Voigtländer. Free theorems in the presence
f seq. *SIGPLAN Not.*, 39(1):99–110, 2004.

acov Kučan. *Metatheorems about Convertibility in Typed Lambda
alculi: Applications to CPS Transform and Free Theorems*. PhD
hesis, Massachusetts Institute of Technology, February 1997.

alf Lämmel. Scrap your boilerplate with XPath-like combinators. In
OPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT
ymposium on Principles of programming languages*, pages 137–142,
ew York, NY, USA, 2007. ACM Press.

alf Lämmel and Simon Peyton Jones. Scrap your boilerplate:
practical design pattern for generic programming. In *Proc. of
he ACM SIGPLAN Workshop on Types in Language Design and
mplementation (TLDI 2003)*, 2003.

alf Lämmel and Simon Peyton Jones. Scrap your boilerplate with
ass: extensible generic functions. In *Proceedings of the ACM SIG-
LAN International Conference on Functional Programming (ICFP
005)*, pages 204–215. ACM Press, September 2005.

rik Meijer and Graham Hutton. Bananas in space: Extending fold
nd unfold to exponential types. In *FPCA95: Conference on Func-
onal Programming Languages and Computer Architecture*, pages
24–333, La Jolla, CA, June 1995.

aul-André Melliès and Jérôme Vouillon. Recursive polymorphic
ypes and parametricity in an operational framework. In *LICS '05:
roceedings of the 20th Annual IEEE Symposium on Logic in Com-
uter Science (LICS' 05)*, pages 82–91, Washington, DC, USA, 2005.
EEE Computer Society.

hristine Paulin-Mohring. Inductive definitions in the system Coq:
ules and properties. In *International Conference on Typed Lambda
alculi and Applications, TLCA '93*, volume 664 of *Lecture Notes in
omputer Science*, pages 328–345. Springer, 1993.

ordon Plotkin and Martín Abadi. A logic for parametric polymor-
hism. In *International Conference on Typed Lambda Calculi and
pplications*, pages 361–375, 1993.

ohn C. Reynolds. Types, abstraction and parametric polymorphism.
n *Information Processing '83*, pages 513–523. North-Holland, 1983.
roceedings of the IFIP 9th World Computer Congress.

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

imitrios Vytiniotis and Stephanie Weirich. Free theorems and
untime type representations. *Electron. Notes Theor. Comput. Sci.*,
73:357–373, 2007.

hilip Wadler. Theorems for free! In *FPCA89: Conference on Func-
onal Programming Languages and Computer Architecture*, pages
47–359, London, September 1989.

eoffrey Washburn and Stephanie Weirich. Generalizing parametric-
y using information flow. In *The Twentieth Annual IEEE Sym-
osium on Logic in Computer Science (LICS 2005)*, pages 62–71,
Chicago, IL, June 2005. IEEE Computer Society, IEEE Computer Society Press.

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

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

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

Definition of $cast$ in $\mathbf{R}_\omega$

e R_ω definition of $cast$ appears in Figure 7 Thanks to im-
types, the definition of $gcast$ may be obtained from this one
lacing lines 11 and 21 with $\mathtt{inr}\ (\lambda z.h_2\ (h_1\ z))$.

Additional semantics of $\mathbf{R}_\omega$

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

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

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

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

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

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

Figure 8: Operational rules for type recursion

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

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

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

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

Figure 9: Well-formed types

Generalized relations, in Coq

Coq definition of `GRel`, `wfGRel`, and `eqGRel` ($\equiv_\kappa$), fol-
n Figure 12.

e assume datatypes that encode R_ω syntax, such as `kind`,
, `type`, and `env`. Moreovere we assume constants such as

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

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

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

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

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

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

Figure 10: Type equivalence

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

D. Proof details

Proof of Lemma 2.7:
For all $\mathcal{K}$, $(\mathcal{K}, \mathcal{K}, [\![\mathcal{K}]\!]) \in \mathtt{wfGRel}^{kind(\mathcal{K})}$.

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

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

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

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

which belongs in $\mathtt{wfGRel}^\star$ since it is a relation between values of the correct types. Additionally, we need to show that $\forall_\kappa$ can only distinguish between equivalence classes of its type arguments. For this fix σ_1, σ_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:

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

We need to show that

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

To finish the case, using rule T-EQ to take care of the typing requirements, it is enough to show that, for any $\rho \in \mathtt{TyGRel}^\kappa$,

```
cast :: ∀a : ⋆.∀b : ⋆.R a → R b → () + (a → b)
cast = λx.typerec x of {
 λy.typerec y of {inr λz.z ; inl () ; inl () ; inl ()};
 λy.typerec y of {inl () ; inr λz.z ; inl () ; inl ()};
 λra1.λf1.λra2.λf2.λy.typerec y of {
   inl ();
   inl ();
   λrb1.λg1.λrb2.λg2.
     case f1 rb1 of {h.inl () ; h1.
     case f2 rb2 of {h.inl () ; h2.
      inr λz.(h1 (fst z), h2 (snd z))
     }};
   λrb1.λg1.λrb2.λg2.inl ()}
 λra1.λf1.λra2.λf2.λy.typerec y of {
   inl ();
   inl ();
   λrb1.λg1.λrb2.λg2.inl ();
   λrb1.λg1.λrb2.λg2.
     case f1 rb1 of {h.inl () ; h1.
     case f2 rb2 of {h.inl () ; h2.
      inr (λz.case z of {z1.h1 z1 ; z2.h2 z2})
     }}}}
```

Figure 7: Definition of *cast* in R_ω

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

$$\dfrac{}{\Gamma \vdash i : \mathtt{int}}\ \textsc{Int} \qquad \dfrac{(x{:}\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \dfrac{\Gamma, (x{:}\tau_1) \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : \star}{\Gamma \vdash \lambda x\,.\,e : \tau_1 \to \tau_2}\ \textsc{Abs} \qquad \dfrac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{App}$$

$$\dfrac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 \equiv \tau_2 : \star}{\Gamma \vdash e : \tau_2}\ \textsc{T-Eq} \qquad \dfrac{\Gamma \vdash e : \forall_\kappa \sigma \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e : \sigma\ \tau}\ \textsc{Inst} \qquad \dfrac{\Gamma, (a{:}\kappa) \vdash e : \sigma\ a \quad a \# \Gamma}{\Gamma \vdash e : \forall_\kappa \sigma}\ \textsc{Gen}$$

$$\dfrac{}{\Gamma \vdash \mathrm{R}_{\mathtt{int}} : \mathrm{R}\ \mathtt{int}}\ \textsc{RInt} \qquad \dfrac{}{\Gamma \vdash \mathrm{R}_{()} : \mathrm{R}\ ()}\ \textsc{RUnit}$$

$$\dfrac{\Gamma \vdash e_1 : \mathrm{R}\,\sigma_1 \quad \Gamma \vdash e_2 : \mathrm{R}\,\sigma_2}{\Gamma \vdash \mathrm{R}_\times\ e_1\ e_2 : \mathrm{R}\,(\sigma_1, \sigma_2)}\ \textsc{RProd} \qquad \dfrac{\Gamma \vdash e_1 : \mathrm{R}\,\sigma_1 \quad \Gamma \vdash e_2 : \mathrm{R}\,\sigma_2}{\Gamma \vdash \mathrm{R}_+\ e_1\ e_2 : \mathrm{R}\,(\sigma_1 + \sigma_2)}\ \textsc{RSum}$$

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

Figure 11: Typing relation for R_ω

$\rho \in \texttt{wfGRel}^\kappa$, we have $g_\tau\ \rho \equiv_\star g_\sigma\ \rho$. But this follows from vity of $\equiv_\kappa$, and the fact that g_τ and g_σ are well-formed. □

of Theorem 2.11:
$\tau_1 : \kappa$, $\delta \models \Gamma$, and $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$, then

$$[\![\Gamma \vdash \tau_1 : \kappa]\!]_\delta \equiv_\kappa [\![\Gamma \vdash \tau_2 : \kappa]\!]_\delta$$

: The proof can proceed by induction on derivations of $_1 \equiv \tau_2 : \kappa$. The case for rule BETA follows by appealing to a 2.10, and the cases for rules APP and ABS we give below. est of the cases are straightforward.

se APP. In this case we have that $\Gamma \vdash \tau_1\ \tau_2 \equiv \tau_3\ \tau_4 : \kappa_2$ ven that $\Gamma \vdash \tau_1 \equiv \tau_3 : \kappa_1 \to \kappa_2$ and $\Gamma \vdash \tau_2 \equiv \tau_4 : \kappa_1$. is easy to show as well that $\Gamma \vdash \tau_{1,3} : \kappa_1 \to \kappa_2$ and $\Gamma \vdash \tau_{2,4} : \kappa_1$. We need to show that

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

Let

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

We know by induction hypothesis that $r_1 \equiv_{\kappa_1 \to \kappa_2} r_3$ and $r_2 \equiv_{\kappa_1} r_4$. By Lemma 2.9, we have that:

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

```
Implicit Arguments.

ctive kind : Set :=
tar : kind
un  : kind -> kind -> kind.

ypes and a constant for type applications *)
meter ty : kind -> Set.
meter TyApp : forall k1 k2, ty (KFun k1 k2) -> ty k1 -> ty k2.

meter term : Set.

nvironments and constant for empty envs   *)
meter env  : Set.
meter empty : env.

meter teq  : forall k, env ->
             ty k -> ty k -> Prop.

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

alue relations as a predicate on relations *)
meter vrel : (ty KStar * ty KStar * rel) -> Prop.

oint GRel (k : kind) : Type :=
tch k with
  KStar => rel
  KFun k1 k2 => (ty k1 * ty k1 * GRel k1) -> GRel k2
d.

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

typed grels *)
oint wfGRel (k:kind) : TyGRel k -> Prop :=
tch 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 (TyApp c^1  a^1, TyApp c^2 a^2, c^3 a)) /\
    (forall b, wfGRel b ->
      teq empty a^1 b^1 ->
      teq empty a^2 b^2 -> eqGRel k1 a^3 b^3 ->
      eqGRel k2 (c^3 a) (c^3 b)))
nd
 eqGRel (k:kind) : GRel k -> GRel k -> Prop :=
tch 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))
d.
```

Figure 12: Coq definitions

nally it is not hard to show that $\cdot \vdash \delta^1\tau_2 \equiv \delta^1\tau_4 : \kappa_1$ and $- \delta^2\tau_2 \equiv \delta^2\tau_4 : \kappa_1$. Hence, by the properties of well-formed ations, and our definition of equivalence, we can show that

$$r_1\ (\delta^1\tau_2, \delta^2\tau_2, r_2) \equiv_{\kappa_2} r_3\ (\delta^1\tau_4, \delta^2\tau_4, r_4)$$

hich finishes the case.

se ABS. Here we have that

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

ven that $\Gamma, a{:}\kappa_1 \vdash \tau_1 \equiv \tau_2 : \kappa_2$. To show the required sult let us pick $\rho \in \mathtt{TyGRel}^{\kappa_1}$ with $\rho \in \mathtt{wfGRel}^{\kappa_1}$. Then $\delta_a = \delta, a \mapsto \rho$, we have $\delta_a \vDash \Gamma, (a{:}\kappa_1)$, and hence by duction hypothesis we get:

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

d the case is finished. As a side note, the important condition at $\rho \in \mathtt{wfGRel}^{\kappa_1}$ allows us to show that $\delta_a \vDash \Gamma, (a{:}\kappa_1)$ d therefore enables the use of the induction hypothesis. If $_{\kappa_1 \to \kappa_2}$ tested against *any possible* $\rho \in \mathtt{TyGRel}^{\kappa_1}$ that would longer be true, and hence the case could not be proved.

□

e first give all the rules of the main typing relation in Fig- . It is then easy to verify the following lemma.

emma [Regularity]: If $\Gamma \vdash e : \tau$ then $\Gamma \vdash \tau : \star$.

oreover, we assume a type and term substitution lemma— can be proved by straightforward inductions. We extend efinition of substitutions to include also mappings of term les to pairs of closed expressions.

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

efinition of $\mathtt{Subst}_\Gamma$ remains the same, but we add one more to $\gamma \vDash \Gamma$: for all x such that $\gamma(x) = (e_1, e_2)$, it is the hat $(e_1, e_2) \in \mathcal{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$ where $(x{:}\tau) \in \Gamma$. We write , $\gamma^2(x)$ for the left and write projections of $\gamma(x)$, and extend otation to arbitrary terms. A well-formed environment is one isjoint domain of term and type variables, and where for all $\in \Gamma$, $\Gamma \vdash \tau : \star$, so the above definition makes sense for ormed environments.

e give a detailed scetch below of the proof of the abstraction em.

of the Abstraction Theorem:

well-formed, and $\gamma \vDash \Gamma$ and $\Gamma \vdash e : \tau$ then $(\gamma^1 e, \gamma^2 e) \in$ $\tau : \star]\!]_\gamma$.

: We proceed by induction on the typing derivation $\Gamma \vdash e :$ case analysis on the last rule used.

se INT. Straightfowrard.

se VAR. The result follows immediately from the fact that e environment is well-formed and the definition of $\gamma \vDash \Gamma$.

se ABS. In this case we have that $\Gamma \vdash \lambda x \,.\, e : \tau_1 \to \tau_2$ given at $\Gamma, (x{:}\tau_1) \vdash e : \tau_2$, and where we assume w.l.o.g that $\#\Gamma, fv(\gamma)$. It suffices to show that $(\lambda x \,.\, \gamma^1 e, \lambda x \,.\, \gamma^2 e) \in$ $\vdash \tau_1 \to \tau_2 : \star]\!]_\gamma$. To show this, let us pick $(e_1, e_2) \in$ $]\!]_{\Gamma \vdash \tau_1 : \star}$, it is then enough to show that

$$((\lambda x \,.\, \gamma^1 e)\ e_1, (\lambda x \,.\, \gamma^2 e)\ e_2) \in \mathcal{C}\,[\![\Gamma \vdash \tau_2 : \star]\!]_\gamma \qquad (5)$$

t we can take $\gamma_0 = \gamma, (x \mapsto (e_1, e_2))$, which cernly satisfies $\gamma_0 \vDash \Gamma, (x{:}\tau_1)$ and by induction hypothesis: $^1_0 e, \gamma^2_0 e) \in \mathcal{C}\,[\![\Gamma, (x{:}\tau_1) \vdash \tau_2 : \star]\!]_{\gamma_0}$. By an easy weakening mma for term variables in the type interpretation we have that $(\gamma^1_0 e, \gamma^2_0 e) \in \mathcal{C}\,[\![\Gamma \vdash \tau_2 : \star]\!]_\gamma$ and by unfolding the definitions, equation (5) 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,

$$(\gamma^1 e_1, \gamma^2 e_1) \in \mathcal{C}\,[\![\Gamma \vdash \sigma \to \tau : \star]\!]_\gamma \qquad (6)$$

$$(\gamma^1 e_2, \gamma^2 e_2) \in \mathcal{C}\,[\![\Gamma \vdash \sigma : \star]\!]_\gamma \qquad (7)$$

  From (6) we get that $\gamma^1 e_1 \Downarrow w_1$ and $\gamma^2 e_1 \Downarrow w_2$ such that $(w_1\ (\gamma^1 e_2), w_2\ (\gamma^2 e_2)) \in \mathcal{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$, where we made use of equation (7) and unfolded definitions. Hence, by the operational semantics for applications, we also have that: $((\gamma^1 e_1)\ (\gamma^1 e_2), (\gamma^2 e_1)\ (\gamma^2 e_2)) \in \mathcal{C}\,[\![\Gamma \vdash \tau : \star]\!]_\gamma$, as required.
- Case T-EQ. The case follows directly from appealing to the Coherence theorem 2.11.
- Case INST. In this case we have that $\Gamma \vdash e : \sigma\ \tau$, given that $\Gamma \vdash e : \forall_\kappa \sigma$ and $\Gamma \vdash \tau : \kappa$. By induction hypothesis we get that $(\gamma^1 e, \gamma^2 e) \in \mathcal{C}([\![\forall_\kappa]\!]\ \ (\gamma^1\sigma, \gamma^2\sigma, [\![\Gamma \vdash \sigma : \kappa \to \star]\!]_\gamma))$; hence by the definition of $[\![\forall_\kappa]\!]$ and by making use of the fact that $(\gamma^1\tau, \gamma^2\tau, [\![\Gamma \vdash \tau : \kappa]\!]_\gamma) \in \mathtt{wfGRel}^\kappa$ (by Lemma 2.9), 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 $a \# \mathit{ftv}(\gamma)$ as well. We need to show that $(\gamma^1 e, \gamma^2 e) \in \mathcal{C}([\![\forall_\kappa]\!]\ \ (\gamma^1\sigma, \gamma^2\sigma, [\![\sigma]\!]_\gamma)$. Hence we can fix $\rho \in \mathtt{TyGRel}^\kappa$ such that $\rho \in \mathtt{wfGRel}^\kappa$. We can form the substitution $\gamma_0 = \gamma, (a \mapsto \rho)$, for which it is easy to show that $\gamma_0 \vDash \Gamma, (a{:}\kappa)$. Then, by induction hypothesis $(\gamma^1_0 e, \gamma^2_0 e) \in \mathcal{C}\,[\![\Gamma, (a{:}\kappa) \vdash \sigma\ a : \star]\!]_{\gamma_0}$ which means $(\gamma^1_0 e, \gamma^2_0 e) \in \mathcal{C}\,[\![\Gamma, (a{:}\kappa) \vdash \sigma : \kappa \to \star]\!]_{\gamma_0}\ \ \rho$. By an easy weakening lemma this implies $(\gamma^1_0 e, \gamma^2_0 e) \in \mathcal{C}\,[\![\Gamma \vdash \sigma : \kappa \to \star]\!]_\gamma\ \ \rho$ and moreover since terms do not contain types $\gamma^i_0 e = \gamma^i e$ and the case is finished.
- Case RINT. We have that $\Gamma \vdash \mathtt{R}_{\mathtt{int}} : \mathtt{R}\ \mathtt{int}$, hence $(\mathtt{R}_{\mathtt{int}}, \mathtt{R}_{\mathtt{int}}) \in \mathcal{R}\ (\mathtt{int}, \mathtt{int}, [\![\mathtt{int}]\!])$ by unfolding definitions.
- Case RUNIT. Similar to the case for RINT.
- Case RPROD. We have that $\Gamma \vdash \mathtt{R}_\times\ e_1\ e_2 : \mathtt{R}\ (\sigma_1 \times \sigma_2)$, given that $\Gamma \vdash e_1 : \mathtt{R}\ \sigma_1$ and $\Gamma \vdash e_2 : \mathtt{R}\ \sigma_2$. It suffices to show that $(\mathtt{R}_\times\ \gamma^1 e_1\ \gamma^1 e_2, \mathtt{R}_\times\ \gamma^2 e_1\ \gamma^2 e_2) \in \mathcal{R}\ (\gamma^1(\sigma_1 \times \sigma_2), \gamma^2(\sigma_1 \times \sigma_2), [\![\Gamma \vdash \sigma_1 \times \sigma_2 : \star]\!]_\gamma)$. The result follows by taking as $\rho_a = (\gamma^1\sigma_1, \gamma^2\sigma_1, [\![\Gamma \vdash \sigma_1 : \star]\!]_\gamma$, $\rho_b = (\gamma^1\sigma_2, \gamma^2\sigma_2, [\![\Gamma \vdash \sigma_2 : \star]\!]_\gamma$. By Lemma 2.9, regularity and inversion on the kinding relation, one can show that ρ_a and ρ_b are well-formed and hence to finish the case we only need to show that $(\gamma^1 e_1, \gamma^2 e_1) \in \mathcal{C}(\mathcal{R}\ \ \rho_a)$ and $(\gamma^1 e_2, \gamma^2 e_2) \in \mathcal{C}(\mathcal{R}\ \rho_b)$, which follow by induction hypotheses for the typing of e_1 and e_2.
- Case RSUM. Similar to the case for RPROD.
- Case TREC. This is really the only interesting case. After we decompose the premises and get the induction hypotheses, we proceed with an inner induction on the type of the scrutinee. In this case we have that:

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

  Let us introduce some abbreviations:

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

$$\sigma_+ \;=\; \forall (a{:}\star)(b{:}\star)\texttt{.}\,\mathtt{R}\ a \to \sigma\ a \to \\ \mathtt{R}\ b \to \sigma\ b \to \sigma\ (a+b)$$

the premises of the rule we have:

$$\Gamma \vdash \sigma : \star \to \star \tag{8}$$
$$\Gamma \vdash e : \mathtt{R}\ \tau \tag{9}$$
$$\Gamma \vdash e_{\mathtt{int}} : \sigma\ \mathtt{int} \tag{10}$$
$$\Gamma \vdash e_{()} : \sigma\ \mathtt{()} \tag{11}$$
$$\Gamma \vdash e_{\times} : \sigma_{\times} \tag{12}$$
$$\Gamma \vdash e_{+} : \sigma_{+} \tag{13}$$

e also know the corresponding induction hypotheses for)),(11),(12), (13). We now show that:

$$\forall e_1\ e_2\ \rho \in \mathtt{TyGRel}^{\star}, \tau_1 \in \mathtt{ty}(\star)\ \tau_2 \in \mathtt{ty}(\star)\ r, \\ \rho \in \mathtt{wfGRel}^{\star} \wedge (e_1, e_2) \in \mathcal{C}(\mathcal{R}\ \rho) \\ \Longrightarrow (\gamma^1 u[e_1], \gamma^2 u[e_2]) \in \mathcal{C}(\llbracket \Gamma \vdash \sigma : \star \to \star \rrbracket_{\gamma}\ \ \rho)$$

introducing our assumptions, and performing inner induc-n on the size of the normal form of τ_1. Let us call this operty for fixed e_1, e_2, ρ, $\mathit{INNER}(e_1, e_2, \rho)$. We have that $_1, e_2) \in \mathcal{C}(\mathcal{R}\ \ \rho)$ and hence we know that $e_1 \Downarrow w_1$ and $\Downarrow w_2$, such that:

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

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

- $w_1 = w_2 = \mathtt{R}_{\mathtt{int}}$ and $\rho \equiv (\mathtt{int}, \mathtt{int}, \llbracket \mathtt{int} \rrbracket)$. 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: $(\gamma^1 e_{\mathtt{int}}, \gamma^2 e_{\mathtt{int}}) \in \mathcal{C}(\llbracket \Gamma \vdash \sigma : \star \to \star \rrbracket_{\gamma}\ \ \rho)$. From the outer induction hypothesis for (10) we get that: $(\gamma^1 e_{\mathtt{int}}, \gamma^2 e_{\mathtt{int}}) \in \mathcal{C}\ \llbracket \Gamma \vdash \sigma\ \mathtt{int} : \star \rrbracket_{\gamma}$ And we have that:

$$\llbracket \Gamma \vdash \sigma\ \mathtt{int} : \star \rrbracket_{\gamma} \;\;= \\ \llbracket \Gamma \vdash \sigma : \star \to \star \rrbracket_{\gamma}\ (\mathtt{int}, \mathtt{int}, \llbracket \mathit{int} \rrbracket)\ \equiv_{\star} \\ \llbracket \Gamma \vdash \sigma : \star \to \star \rrbracket_{\gamma}\ \ \rho$$

  where we have made use of the properties of well-formed generalized relations to substitute equivalent types and relations in the middle step.
- $w_1 = w_2 = ()$ and $\llbracket \Gamma \vdash \tau : \star \rrbracket_{\gamma} \equiv_{\star} \llbracket\, \mathtt{()}\, \rrbracket$. The case is similar to the previous case.
- $w_1 = \mathtt{R}_{\times}\ e_a^1\ e_a^2$ and $w_2 = \mathtt{R}_{\times}\ e_b^1\ e_b^2$, such that there exist ρ_a^1 and ρ_a^2, well-formed, such that

$$\rho \equiv_{\star} ((\rho_a^1 \times \rho_b^1), (\rho_a^2 \times \rho_b^2), \llbracket \times \rrbracket\ \ \rho_a\ \rho_b \tag{14}$$
$$(e_a^1, e_a^2) \in \mathcal{C}(\mathcal{R}\ \rho_a) \tag{15}$$
$$(e_b^1, e_b^2) \in \mathcal{C}(\mathcal{R}\ \rho_b) \tag{16}$$

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

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

  By the outer induction hypothesis for (12) we will be done, as before, if we instantiate with relations r_a and r_b for the quantified variables a and b, respectively. But we need to show that, for $\gamma_0 = \gamma, (a \mapsto \rho_a), (b \mapsto \rho_b)$, $\Gamma_0 = \Gamma, (a{:}\star), (b{:}\star)$, we have:

$$(\gamma^1 u[e_a^1], \gamma^2 u[e_a^2]) \in \mathcal{C}\ \llbracket \Gamma_0 \vdash \sigma\ a : \star \rrbracket_{\gamma_0} \tag{17}$$
$$(\gamma^1 u[e_b^1], \gamma^2 u[e_b^2]) \in \mathcal{C}\ \llbracket \Gamma_0 \vdash \sigma\ b : \star \rrbracket_{\gamma_0} \tag{18}$$

  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 (15) and (16). From these, compositionality, and an easy weakening‘ lemma, we have that (17) and (18) follow. By the outer induction hypothesis for (12) we then finally have that:

$$(w_1, w_2) \in \llbracket \Gamma, (a{:}\star), (b{:}\star) \vdash \sigma\ (a \times b) : \star \rrbracket_{\gamma_0}$$

  which gives us the desired $(w_1, w_2) \in \llbracket \Gamma \vdash \sigma : \star \to \star \rrbracket_{\gamma}\ \ \rho$ by appealing to the properties of well-formed generalized relations.

We now have by the induction hypothesis for (9), that $(\gamma^1 e, \gamma^2 e) \in \mathcal{C}(\mathcal{R}\ (\gamma^1\tau, \gamma^2\tau, \llbracket \Gamma \vdash \tau : \star \rrbracket_{\gamma}))$, and hence we can get $\mathit{INNER}(\gamma^1 e, \gamma^2 e, (\gamma^1\tau, \gamma^2\tau, \llbracket \Gamma \vdash \tau : \star \rrbracket_{\gamma}))$, which gives us that: $(\gamma^1 u[e], \gamma^2 u[e]) \in \mathcal{C}(\llbracket \Gamma \vdash \sigma : \star \to \star \rrbracket_{\gamma}\ \ (\gamma^1\tau, \gamma^2\tau, \llbracket \Gamma \vdash \tau : \star \rrbracket_{\gamma}))$, or $(\gamma^1 u[e], \gamma^2 u[e]) \in \mathcal{C}(\llbracket \Gamma \vdash \sigma\ \tau : \star \rrbracket_{\gamma})$, as required.

□