Free theorems and runtime type representations

Dimitrios Vytiniotis[1] Stephanie Weirich[2]

Department of Computer and Information Science
University of Pennsylvania
Philadelphia PA, USA

Abstract

Reynolds's abstraction theorem [21], often referred to as the parametricity theorem, can be used to derive properties about functional programs solely from their types. Unfortunately, in the presence of runtime type analysis, the abstraction properties of polymorphic programs are no longer valid. However, runtime type analysis can be implemented with term-level representations of types, as in the λ_R language of Crary *et al.* [10], where case analysis on these runtime representations introduces type refinement. In this paper, we show that representation-based analysis is consistent with type abstraction by extending the abstraction theorem to such a language. We also discuss the "free theorems" that result. This work provides a foundation for the more general problem of extending the abstraction theorem to languages with generalized algebraic datatypes (GADTs).

Keywords: Logical relations, polymorphism, type representations, parametricity, GADTs.

1 Introduction

Reynolds's abstraction theorem [21] serves as a characterization of *parametric polymorphism*. It asserts that parametrically polymorphic functions behave in a uniform way, independently of the types at which they are used. Importantly, the abstraction theorem can be used to derive equivalences involving functional programs, just by observing the types of these programs. Wadler [27] refers to these equivalences as the "free theorems" associated with particular types.

For example, one free theorem (not the most general one) about the polymorphic λ-calculus (also known as System F [13]) is that for any function f of type $\forall a \,.\, a \to a$ we know that for any type τ and term $h : \tau \to \tau$ we have the following equivalence

$$f[\tau] \circ h \cong h \circ f[\tau]$$

[1] Email: dimitriv@cis.upenn.edu

[2] Email: sweirich@cis.upenn.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

where $\circ$ indicates function composition and $\cong$ indicates *program equivalence*, a relation that we will make more precise later. From this free theorem we can conclude that f must behave like the identity function, as h could be any arbitrary function.

However, Reynolds's abstraction theorem does not hold for all extensions of the polymorphic λ-calculus. In particular, it does not hold in the presence of an operator for runtime type analysis, such as the *typecase* operator from Harper and Morrisett's language λ_i^{ML} [14]. In this language, polymorphic functions do not always behave parametrically. For example, the following function increments integers, but is the identity function for any other type.

$$f = \Lambda a\,.\, \mathit{typecase}\ a\ \mathit{of}\ \{\mathit{Int} \Rightarrow \lambda x{:}\mathit{Int}\,.\,\mathit{succ}\ x \ \mid\ \_ \Rightarrow \lambda x{:}a\,.\,x\}$$

Consequently, even though f is of type $\forall a\,.\,a \to a$, we can contradict its free theorem by picking $h = (\lambda x{:}\mathit{Int}\,.\,8)$.

$$f[\mathit{Int}] \circ (\lambda x{:}\mathit{Int}\,.\,8) \ncong (\lambda x{:}\mathit{Int}\,.\,8) \circ f[\mathit{Int}]$$

Runtime type analysis is a useful language feature for *datatype-generic* (also called polytypic) programming. It can be used to define type-indexed operations such as generic parsers, pretty printers, iterators and other operations that automate the boilerplate of working with algebraic datatypes. However, partly because of the negative interactions with type abstraction, few functional languages support it.

An alternative approach to generic programming is to use *term representations of types* to simulate runtime type analysis, such as in the language λ_R [10]. In this language, *typecase* analyzes terms that represent types, instead of types. The type of a term representation reveals what type it represents—if the term e represents type τ, then term e has type $R\ \tau$. Determining the identity of e simultaneously determines the identity of τ. Because polymorphic functions in λ_R treat their type arguments parametrically, Crary *et al.* conjectured that the abstraction theorem could be extended to that language, but before this paper, no one has done so.

The question about representation types and parametricity has recently become more important with the introduction of *generalized algebraic datatypes* (GADTs) [3,8,23,25,16,26], a variant of *inductive families of types* [12] originally developed in dependent type theory. With GADTs, we may implement representation types, enabling the advantages of runtime type analysis. For example, in recent work we showed how GADTs in the Glasgow Haskell Compiler (GHC) [30] may be used to implement a sophisticated library of datatype generic routines.

Using the GADT syntax of GHC, we can define the R representation type using the following datatype declaration.

$$\begin{array}{l} \mathit{data}\ R\ a\ \mathit{where} \\ \quad R_{\mathit{Int}} \ ::\ R\ \mathit{Int} \\ \quad R_{\to} \ ::\ \forall ab\,.\,R\ a \to R\ b \to R\ (a \to b) \\ \quad R_{\mathit{any}} \ ::\ \forall a\,.\,R\ a \end{array}$$

The R datatype has several constructors, each corresponding to the representation of a particular type. For example, R_{Int} serves as a runtime representation of the Int datatype. The $R_{\rightarrow}$ constructor takes the representations for the argument and the result type of a function type, and returns a representation of that function type. The R_{any} constructor is an "un-representation"—it represents *any* type, so does not actually provide any run-time information.

Functions that take arguments of type $R\ \tau$ may perform case analysis on these arguments. However, unlike ordinary pattern matching, in the types of the branches the type τ is *refined* according to the constructor matched each time. Consider the function g below of type $\forall a\,.\,R\ a \rightarrow a \rightarrow a$.

$$g = \Lambda a\,.\,\lambda y{:}R\ a\,.\,\mathit{case}\ y\ \mathit{of}\ \{R_{Int} \Rightarrow \lambda x{:}Int\,.\,\mathit{succ}\ x \mid \_ \Rightarrow \lambda x{:}a\,.\,x\}$$

In the case for R_{Int}, the type variable a is refined to be equal to Int, so the particular branch is acceptable even though the type of the case expression is $a \rightarrow a$.

Note that the free theorems about functions that take type representation arguments should be weaker than those for functions that do not. For example, for any function g of type $\forall a\,.\,R\ a \rightarrow a \rightarrow a$ we should *not* expect that

$$(g[\tau]\ r) \circ h \cong h \circ (g[\tau]\ r) \tag{1}$$

holds for any appropriately typed r and h. To see why consider the function g above. Equivalence (1) does not hold when $h = (\lambda x{:}Int\,.\,8)$ and $r = R_{Int}$. Nevertheless, we should still be able to derive a free theorem for g when it is applied to R_{any}, since R_{any} does not introduce any type refinement:

$$(g[\tau]\ (R_{any}[\tau])) \circ h \cong h \circ (g[\tau]\ (R_{any}[\tau]))$$

In this paper, we explore parametricity in the context of an explicitly typed polymorphic λ-calculus which includes various GADTs, such as R above. The language that we study is strongly normalizing, so the free theorems that we derive are not exactly the same as those for a realistic language, such as the language that GHC implements. However, working in this restricted world allows us to cleanly explain our points without the complications that non-termination would require [19]. Our work can be adapted to include non-termination and other computational effects, but we prefer not to do so here.

The specific contributions of this paper include:

- We present a parametricity result for λ_R by giving a *relational interpretation* of types as sets of pairs of closed values and showing that every well-typed expression is related to itself in the interpretation of its type. We derive free theorems from this fundamental property. This interpretation of types is sound with respect to *ground equivalence* and therefore suitable for reasoning about program equivalence.
- We show that for pure System F types, the *same free theorems* as in System F are derivable, but for types that involve the R datatype, free theorems *can still be derived*, but may be, in general, less informative than theorems for R-free types.

- However, not all GADTs produce uninformative free theorems: we give an example of a GADT *Eq*, that derives valuable theorems.
- We use the *Eq* datatype to show how our ideas extend to a language that supports arbitrary GADTs. We are confident that the ideas presented in this paper will extend to such a language by combining known techniques for relational interpretations of recursive types in an operational setting [5,9,2,1]. We sketch how this might be done, but leave the full details of the experiment as future work.
- We provide a complete formalization of the parametricity result in this paper in the Isabelle/HOL [18] proof assistant, available from:

  `www.cis.upenn.edu/~dimitriv/parametricity/`

  This formalization should not only be viewed as supporting material, but also as an extensive real-world study of representation techniques and proof methods for polymorphic languages in proof assistants [4]. Our proofs are thoroughly commented and available to other researchers.

2 Parametricity for System F

As a starting point, we review a standard parametricity result for System F extended with an integer base type. In Section 3 we will extend this foundation to a language with representation types. We begin with the syntax of System F:

Types	σ, τ	$::= a \mid \mathit{Int} \mid \tau \to \tau \mid \forall a\,.\,\tau$
Terms	e	$::= i \mid \mathit{succ}\ e \mid x \mid \lambda x{:}\tau\,.\,e \mid e\ e \mid \Lambda a\,.\,e \mid e[\tau]$
Environments	Γ, Δ	$::= \epsilon \mid \Gamma, a \mid \Gamma, (x{:}\tau)$
Values	u, v, w	$::= i \mid \lambda x{:}\tau\,.\,e \mid \Lambda a\,.\,e$

Types include type variables, the base type *Int*, arrow types, and universal types; terms include integer literals i and a successor operator *succ*, variables, abstractions, applications, type abstractions, and type applications. Environments record type variables introduced by type abstractions, and term variables introduced by term abstractions. We write ϵ for the empty environment. We write $\sigma\{\tau/a\}$ for the capture-avoiding substitution of τ for a in σ. The judgement $\Gamma \vdash \tau$ ensures that the free variables of the type τ are bound in Γ. The typing relation is given with judgements of the form $\Gamma \vdash e : \tau$. The operational semantics that we use is a standard, small-step, call-by-name. We write $e_1 \to e_2$ for the transition relation and $e_1 \to^* e_2$ for the reflexive and transitive closure of the transition relation. We write $e_1 \Downarrow e_2$ to mean that $e_1 \to^* e_2$ and $\nexists e_3\,.\,e_2 \to e_3$.

2.1 Relational interpretation of System F types

Every type of System F can be interpreted as a relation between closed values, built up from the following definitions. For a closed type τ, we define the set of all closed values of that type as $\mathit{Value}(\tau) = \{v \mid \ \epsilon \vdash v : \tau\}$ and the set of binary relations between values of two different (closed) types as $\mathit{VRel}(\tau_1, \tau_2) = \{r \mid r \subseteq \mathit{Value}(\tau_1) \times \mathit{Value}(\tau_2)\}$. We write id_τ for the set $\{(v, v) \mid v \in \mathit{Value}(\tau)\}$.

Definition 2.1 [Computation relation lifting] Let $r \in \mathit{VRel}(\tau_1, \tau_2)$. The lifting of r to a computation relation r^* between closed expressions is given by:

$$\begin{aligned} r^* = \{(e_1, e_2) \mid\ & \epsilon \vdash e_1 : \tau_1 \wedge e_1 \Downarrow v_1 \wedge \\ & \epsilon \vdash e_2 : \tau_2 \wedge e_2 \Downarrow v_2 \wedge (v_1, v_2) \in r\} \end{aligned}$$

Definition 2.2 [Semantic substitutions δ] Assume that r ranges over arbitrary value relations. Semantic substitutions are given by:

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

We view semantic substitutions as partial maps. Whenever $\delta(a) = (\tau_1, \tau_2, r)$ we write $\delta^1 a$ for τ_1, $\delta^2 a$ for τ_2 and $\delta[a]$ for r. Similarly when $\delta(x) = (e_1, e_2)$ we write $\delta^1 x$ for e_1 and $\delta^2 x$ for e_2. We extend the definition to $\delta^{1,2}\tau$ and $\delta^{1,2}e$ homomorphically (modulo capture-avoidance).

Definition 2.3 [Function space relation constructor] Let $r_a \in \mathit{VRel}(\tau_a^1, \tau_a^2)$ and $r_b \in \mathit{VRel}(\tau_b^1, \tau_b^2)$. Then let $r_a \Rightarrow r_b \in \mathit{VRel}(\tau_a^1 \to \tau_b^1, \tau_a^2 \to \tau_b^2)$ be

$$\begin{aligned} r_a \Rightarrow r_b = \{(v_1, v_2) \mid\ & \epsilon \vdash v_1 : \tau_a^1 \to \tau_b^1 \wedge \epsilon \vdash v_2 : \tau_a^2 \to \tau_b^2 \wedge \\ & \forall e_1 e_2 \,.\, (e_1, e_2) \in {r_a}^* \Longrightarrow (v_1\ e_1, v_2\ e_2) \in {r_b}^*\} \end{aligned}$$

We now give the relational interpretation of System F types.

Definition 2.4 [Relational interpretation of System F types] The interpretation of types is given as a function defined recursively on the size of types:

$$\begin{aligned} [\![\mathit{Int}]\!]_\delta &= id_{\mathit{Int}} \\ [\![a]\!]_\delta &= \delta[a] \\ [\![\sigma_1 \to \sigma_2]\!]_\delta &= [\![\sigma_1]\!]_\delta \Rightarrow [\![\sigma_2]\!]_\delta \\ [\![\forall a \,.\, \sigma]\!]_\delta &= \{(v_1, v_2) \ \mid\ \epsilon \vdash v_1 : \delta^1(\forall a \,.\, \sigma) \wedge \epsilon \vdash v_2 : \delta^2(\forall a \,.\, \sigma) \wedge \\ &\qquad \forall \tau_1\ \tau_2\ r \,.\, r \in \mathit{VRel}(\tau_1, \tau_2) \Longrightarrow \\ &\qquad\quad (v_1\ [\tau_1], v_2\ [\tau_2]) \in ([\![\sigma]\!]_{\delta, a \mapsto (\tau_1, \tau_2, r)})^*\} \end{aligned}$$

Note that the interpretation of quantified types extends the given semantic substitution δ, which is used for the interpretation of type variables. Strictly speaking, for the interpretation of open types we only need the bindings of type variables in semantic substitutions (and not bindings of term variables). However, the fact that semantic substitutions can be put in one-to-one correspondence with typing environments simplifies the formal treatment in Isabelle/HOL. In particular, we define when a substitution δ is well-formed in an environment Γ, written as $\Gamma \vdash \delta$.

Definition 2.5 [Well-formed semantic substitutions]

$$\frac{}{\epsilon \vdash \epsilon} \qquad \frac{\Delta \vdash \delta \quad r \in \mathit{VRel}(\tau_1, \tau_2)}{\Delta, a \vdash \delta, a \mapsto (\tau_1, \tau_2, r)} \qquad \frac{\Delta \vdash \delta \quad (e_1, e_2) \in [\![\tau]\!]_\delta^{\;*}}{\Delta, (x{:}\tau) \vdash \delta, x \mapsto (e_1, e_2)}$$

The parametricity theorem can be proved with the following sequence of lemmas.

Lemma 2.6 (Interpretation of types is a value relation) *If* $\Gamma \vdash \tau$ *and* $\Gamma \vdash \delta$ *then* $[\![\tau]\!]_\delta \in \mathit{VRel}(\delta^1\tau, \delta^2\tau)$.

Compositionality asserts that the interpretation of types depends only on the interpretation of structurally smaller types.

Lemma 2.7 (Compositionality) *If* $\Gamma, a \vdash \tau$ *and* $\Gamma \vdash \sigma$ *and* $\Gamma \vdash \delta$ *then* $[\![\tau]\!]_{\delta, a \mapsto (\delta^1\sigma, \delta^2\sigma, [\![\sigma]\!]_\delta)} = [\![\tau\{\sigma/a\}]\!]_\delta$

The following theorem is referred to as the "soundness lemma", or the "fundamental property of the logical relation" or the "abstraction theorem", or the "parametricity theorem". This theorem shows that well-typed terms are related in the relation which is given by the interpretation of their types, and can be proven by induction on the typing derivations appealing to the previous lemmas.

Theorem 2.8 (Fundamental property of logical relation) *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \delta$ *then* $(\delta^1 e, \delta^2 e) \in [\![\tau]\!]_\delta^{\;*}$.

Free theorems can be derived by expanding definitions in the following corollary.

Corollary 2.9 *If* $\epsilon \vdash e : \tau$ *then* $(e, e) \in [\![\tau]\!]_\epsilon^{\;*}$.

Example 2.10 [Free theorem for $\forall a \,.\, a \to a$] Let $\epsilon \vdash f : \forall a \,.\, a \to a$. The fundamental property then gives:

$$\forall \tau_1\ \tau_2\ r \in \mathit{VRel}(\tau_1, \tau_2)\ e_1\ e_2 \,.\, (e_1, e_2) \in r^* \Longrightarrow (f[\tau_1]\ e_1, f[\tau_2]\ e_2) \in r^*$$

Let us consider any τ_1, τ_2, and $\epsilon \vdash h : \tau_1 \to \tau_2$. Let $\cong$ be an equivalence relation for typed terms that is evaluation-respecting and congruent.[3] Let r be the graph of h, defined as $r = \{(v_1, v_2) \mid \epsilon \vdash v_1 : \tau_1 \wedge \epsilon \vdash v_2 : \tau_2 \wedge h\ v_1 \Downarrow v_2\}$. Then pick a value x such that $\epsilon \vdash x : \tau_1$ and $(x, h\ x) \in r^*$. By the free theorem, it has to be that $f[\tau_1]\ x \Downarrow w_1$ and $f[\tau_2]\ (h\ x) \Downarrow w_2$, such that $(w_1, w_2) \in r$, that is, $h\ w_1 \Downarrow w_2$. Because equivalence is evaluation-respecting we have $f[\tau_1]\ x \cong w_1$ and by congruence $h\ (f[\tau_1]\ x) \cong h\ w_1$. For the same reason $h\ w_1 \cong w_2$, therefore $h\ (f[\tau_1]\ x) \cong w_2$, and consequently

$$h\ (f[\tau_1]\ x) \cong f[\tau_2]\ (h\ x)$$

Moreover, because h is arbitrary, f must behave as the identity function.

We describe program equivalence here abstractly because we need only certain properties; we return to this definition in Section 3.2. The question of whether the logical

[3] Precisely, we need the properties that, for appropriately typed expressions e and e_1 and e_2, if $e_1 \to e_2$ then $e_1 \cong e_2$, and $e_1 \cong e_2$ implies $e\ e_1 \cong e\ e_2$.

relation that we have presented coincides with a specific standard notion of program equivalence (such as *ciu*-equivalence) is, to our knowledge, open. However, if we restrict the interpretation of polymorphic types to only quantify over relations that respect a particular definition of program equivalence, and close every constructor on relations over this program equivalence, it can be shown that equivalence is a sound and complete characterization of the logical relation (both for System F and the extensions presented in the next section).

Additionally, observe that it is an easy corollary of the fundamental property that all closed expressions of the language terminate. In essence, the relational interpretation of types assigns relations to type variables, just as in Girard's reducibility candidates method [13] a type variable is assigned a candidate set. Adding a recursion primitive in the language has no further complication for the soundness of the relational semantics, provided that we quantify over $\top\top$-closed relations [19], or (alternatively) admissible relations, that is, strict and limit-preserving ones. How to express admissibility syntactically has been studied elsewhere [5].

3 Type representations

We now extend System F to include type representations and show how to extend the relational interpretation of types.

$$
\begin{array}{lll}
\tau & ::= & \ldots \mid R\ \tau \\
e & ::= & \ldots \mid R_{Int} \mid R_{\rightarrow}[\tau_a][\tau_b]\ e_a\ e_b \mid R_{any}[\tau] \mid \mathit{rcase}\ e\ \mathit{of}\{e_{Int}\ ;\ e_{\rightarrow}\ ;\ e_{any}\} \\
u, v & ::= & \ldots \mid R_{Int} \mid R_{\rightarrow}[\tau_a][\tau_b]\ e_a\ e_b \mid R_{any}[\tau]
\end{array}
$$

We add the R type form, constructors for representing integer types (R_{Int}), function types ($R_{\rightarrow}$), and the "any" constructor (R_{any}). The *rcase* expression is an elimination form that performs pattern matching on values of type $R\ \tau$, selecting one of the branches (e_{Int}, $e_{\rightarrow}$, or e_{any}). The operational semantics of the extensions are given with a congruence rule (ECASE) and computation rules that select a particular branch.

$$\frac{e \rightarrow e'}{\mathit{rcase}\ e\ \mathit{of}\ \{e_{Int}\ ;\ e_{\rightarrow}\ e_{any}\} \rightarrow \mathit{rcase}\ e'\ \mathit{of}\ \{e_{Int}\ ;\ e_{\rightarrow}\ ;\ e_{any}\}}\ \text{ECASE}$$

$$\frac{}{\mathit{rcase}\ R_{\rightarrow}[\tau_a][\tau_b]\ e_a\ e_b\ \mathit{of}\{e_{Int}\ ;\ e_{\rightarrow}\ ;\ e_{any}\} \rightarrow e_{\rightarrow}[\tau_a][\tau_b]\ e_a\ e_b}\ \text{ERFUN}$$

$$\frac{}{\mathit{rcase}\ R_{Int}\ \mathit{of}\ \{e_{Int}\ ;\ e_{\rightarrow}\ ;\ e_{any}\} \rightarrow e_{Int}}\ \text{ERINT}$$

$$\frac{}{\mathit{rcase}\ R_{any}[\tau]\ \mathit{of}\{e_{Int}\ ;\ e_{\rightarrow}\ ;\ e_{any}\} \rightarrow e_{any}[\tau]}\ \text{ERANY}$$

The typing relation is extended with the following new rules.

$$\frac{}{\Gamma \vdash R_{Int} : R\ Int}\ \textsc{rint} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash R_{any}[\tau] : R\ \tau}\ \textsc{rany}$$

$$\frac{\Gamma \vdash e_a : R\ \tau_a \quad \Gamma \vdash e_b : R\ \tau_b}{\Gamma \vdash R_{\rightarrow}[\tau_a][\tau_b]\ e_a\ e_b : R\ (\tau_a \rightarrow \tau_b)}\ \textsc{rfun}$$

$$\frac{\begin{array}{c}\Gamma \vdash \forall c\,.\,\tau \qquad \Gamma \vdash e : R\ \sigma \qquad \Gamma \vdash e_{Int} : \tau\{Int/c\} \\ \Gamma \vdash e_{\rightarrow} : \forall ab\,.\,R\ a \rightarrow R\ b \rightarrow \tau\{(a \rightarrow b)/c\} \qquad \Gamma \vdash e_{any} : \forall c\,.\,\tau\end{array}}{\Gamma \vdash rcase\ e\ of\ \{e_{Int}\,;\, e_{\rightarrow}\,;\, e_{any}\} : \tau\{\sigma/c\}}\ \textsc{rcase}$$

The rules for the new constructors, RINT, RFUN, and RANY are standard. The rule RCASE is like a standard case expression, except that the type of each branch is specialized to the represented type. Rule RCASE first asserts that the type τ has one distinguished free variable c with $\Gamma \vdash \forall c.\tau$. The result type of the case expression is formed by replacing c with σ, when the scrutinee e has a $R\ \sigma$ type. In the case of e_{Int}, we know that σ is equal to Int, so c is replaced by Int in that type. Likewise, the $e_{\rightarrow}$ branch requires first two types a, and b, and two representations, $R\ a$ and $R\ b$, acting as a pattern to match against e. It then returns an expression of type τ, where c has been refined to $a \rightarrow b$. Finally, the type of e_{any} does not do any refinement. These typing rules allow us, for example, to typecheck the example from the introduction:

$$\begin{aligned} g = \Lambda a\,.\,\lambda x{:}R\ a\,.\,rcase\ x\ of\ \{\ & \lambda w{:}Int\,.\,succ\ w; \\ & \Lambda bc\,.\,\lambda z{:}R\ b\,.\,\lambda y{:}R\ c\,.\,(\lambda w{:}b \rightarrow c\,.\,w); \\ & \Lambda c\,.\,\lambda w{:}c\,.\,w\} \end{aligned}$$

3.1 Extension of relational interpretation

We next extend Definition 2.4 with the interpretation of $R\ \tau$ types. A naïve attempt at this definition merely checks the mapping of the type τ in δ to determine the related pairs. For example, part of this definition might read

$$If \quad \delta^1\tau = \delta^2\tau = Int \quad then \quad [\![R\ \tau]\!]_\delta = \{(R_{Int}, R_{Int})\} \cup \{(R_{any}[Int], R_{any}[Int])\}$$

since if $\delta^i\tau = Int$ then the only closed values of type $R\ Int$ are R_{Int}, and $R_{any}[Int]$. This naïve interpretation would have a similar case for the other constructors as well: If $\delta^1\tau = \tau_a^1 \rightarrow \tau_b^1$ and $\delta^2\tau = \tau_a^2 \rightarrow \tau_b^2$ then we would relate pairs $(R_{\rightarrow}[\tau_a^1][\tau_b^1]\ e_a^1\ e_b^1, R_{\rightarrow}[\tau_a^2][\tau_b^2]\ e_a^2\ e_b^2)$ where $(e_a^1, e_a^2) \in ([\![R\ a]\!]_{a\mapsto(\tau_a^1,\tau_a^2,r_a)})^*$, for some arbitrary relation r_a, $(e_b^1, e_b^2) \in ([\![R\ b]\!]_{b\mapsto(\tau_b^1,\tau_b^2,r_b)})^*$, for some arbitrary relation r_b. Finally, no matter what the definition of $\delta^1\tau$ and $\delta^2\tau$ is, we would always include the pair $(R_{any}[\delta^1\tau], R_{any}[\delta^2\tau])$.

However this definition is problematic. Consider the free theorem that results when $\epsilon \vdash g : \forall a \,.\, R\ a \to a \to a$.

$$\begin{array}{l}\forall \tau_1\ \tau_2\ r \,.\, r \in \mathit{VRel}(\tau_1, \tau_2)\ e_r^1\ e_r^2\ e_1\ e_2 \,.\\ \quad (e_r^1, e_r^2) \in ([\![R\ a]\!]_{a \mapsto (\tau_1, \tau_2, r)})^* \wedge (e_1, e_2) \in r^* \Longrightarrow (g[\tau_1]\ e_r^1\ e_1, g[\tau_2]\ e_r^2\ e_2) \in r^*\end{array}$$

Specifically, the above implies that:

$$\begin{array}{l}\forall r \,.\, r \in \mathit{VRel}(\mathit{Int}, \mathit{Int}) \Longrightarrow\\ \quad \forall e_1\ e_2 \,.\, (e_1, e_2) \in r^* \Longrightarrow (g[\mathit{Int}]\ R_{\mathit{Int}}\ e_1, g[\mathit{Int}]\ R_{\mathit{Int}}\ e_2) \in r^*\end{array}$$

Let r be the graph of the constant function $\mathit{eight} = \lambda x{:}\mathit{Int} \,.\, 8$. Consequently:

$$g[\mathit{Int}]\ R_{\mathit{Int}}\ (\mathit{eight}\ x) \cong \mathit{eight}\ (g[\mathit{Int}]\ R_{\mathit{Int}}\ x) \tag{2}$$

But, as we saw in Section 1, this equation is not true for every g of type $\forall a \,.\, R\ a \to a \to a$! Technically, the problem lies in the proof of the fundamental property, which states that, if $\Gamma \vdash e : \tau$ and $\Gamma \vdash \delta$ then $(\delta^1 e, \delta^2 e) \in [\![\tau]\!]_\delta{}^*$. Its proof proceeds by induction on the typing derivation. The interesting case follows:

- Case RCASE. We have that $\Gamma \vdash \mathit{rcase}\ e\ \mathit{of}\ \{e_{\mathit{Int}}\ ;\ e_{\to}\ ;\ e_{\mathit{any}}\} : \tau\{\sigma/c\}$ given that $\Gamma \vdash \forall c \,.\, \tau$, $\Gamma \vdash e : R\ \sigma$, and $\Gamma \vdash e_{\mathit{Int}} : \tau\{\mathit{Int}/c\}$ (and others). By induction $(\delta^1 e, \delta^2 e) \in [\![R\ \sigma]\!]_\delta{}^*$. We now proceed by case analysis on $\delta^1\sigma$ and $\delta^2\sigma$. Assume that $\delta^1\sigma = \delta^2\sigma = \mathit{Int}$ (the other cases would create similar problems). Then $\delta^1 e$ and $\delta^2 e$ either both evaluate to R_{Int} or the both evaluate to $R_{\mathit{any}}[\mathit{Int}]$. Suppose the former. By induction, $(\delta^1 e_{\mathit{Int}}, \delta^2 e_{\mathit{Int}}) \in [\![\tau\{\mathit{Int}/c\}]\!]_\delta{}^*$, hence $\delta^1 e_{\mathit{Int}} \Downarrow w_1$ and $\delta^2 e_{\mathit{Int}} \Downarrow w_2$ so that $(w_1, w_2) \in [\![\tau\{\mathit{Int}/c\}]\!]_\delta$. Hence $\delta^1(\mathit{rcase}\ e\ \mathit{of}\ \{e_{\mathit{Int}}\ ;\ e_{\to}\ ;\ e_{\mathit{any}}\})$ and $\delta^2(\mathit{rcase}\ e\ \mathit{of}\ \{e_{\mathit{Int}}\ ;\ e_{\to}\ ;\ e_{\mathit{any}}\})$ evaluate to w_1 and w_2 respectively. We need to establish finally that $[\![\tau\{\mathit{Int}/c\}]\!]_\delta = [\![\tau\{\sigma/c\}]\!]_\delta$. Appealing to the compositionality lemma, it suffices to show that:

  $$[\![\tau]\!]_{\delta,(c \mapsto (\mathit{Int}, \mathit{Int}, \mathit{id}_{\mathit{Int}}))} = [\![\tau]\!]_{\delta,(c \mapsto (\delta^1\sigma, \delta^2\sigma, [\![\sigma]\!]_\delta))}$$

  At this point the proof is stuck! Although we have that $\delta^1\sigma = \delta^2\sigma = \mathit{Int}$ we have *no restriction on the interpretation of* σ.

From this failed proof we see that in the interpretation of $R\ \sigma$ we can return the pair $(R_{\mathit{Int}}, R_{\mathit{Int}})$ if $\delta^1\sigma = \delta^2\sigma = \mathit{Int}$ *and* $[\![\sigma]\!]_\delta = \mathit{id}_{\mathit{Int}}$. Generalizing this idea, we see that the interpretation of $R\ \sigma$ must restrict the interpretation of σ. We define the interpretation of $R\ \sigma$ types using the operator $\mathfrak{R}\ r$ below.

Definition 3.1 [Extension for $R\ \sigma$ types]

$$\begin{array}{lcl}
[\![R\ \sigma]\!]_\delta & = & \mathfrak{R}\ [\![\sigma]\!]_\delta \\
\mathfrak{R}\ (r^{\in VRel(\tau_1,\tau_2)}) & = & \{\ (R_{Int}, R_{Int}) \mid \boxed{r = id_{Int} \wedge \tau_1 = \tau_2 = Int}\} \cup \\
 & & \{\ (R_{\rightarrow}[\tau_a^1][\tau_b^1]\ e_a^1\ e_b^1, R_{\rightarrow}[\tau_a^2][\tau_b^2]\ e_a^2\ e_b^2)\ \mid \\
 & & \quad \exists r_a \in VRel(\tau_a^1, \tau_a^2) \,.\, \exists r_b \in VRel(\tau_b^1, \tau_b^2) \,. \\
 & & \qquad \boxed{r = r_a \Rightarrow r_b \wedge \tau_1 = \tau_a^1 \rightarrow \tau_b^1 \wedge \tau_2 = \tau_a^2 \rightarrow \tau_b^2} \wedge \\
 & & \qquad (e_a^1, e_a^2) \in (\mathfrak{R}\ r_a)^* \wedge (e_b^1, e_b^2) \in (\mathfrak{R}\ r_b)^*\ \} \cup \\
 & & \{\ (R_{any}[\tau_1], R_{any}[\tau_2])\ \}
\end{array}$$

The $\mathfrak{R}\ r$ function is well-defined since the size of the types becomes smaller in sub-calls.[4] The boxed parts of the definition indicate the restrictions imposed on the interpretation of σ when interpreting $R\ \sigma$ types. Note that the constructor R_{any} does not restrict the interpretation of σ. The constructor $R_{\rightarrow}$ only restricts the interpretation of σ to be "functional".

We have validated the fundamental property for this definition, and we now show how it can be used to derive free theorems. (Note that the fundamental property also gives us type soundness and termination.)

Example 3.2 [Free theorem for $\forall a \,.\, R\ a \rightarrow a \rightarrow a$] We show here that if $\epsilon \vdash f : \forall a \,.\, R\ a \rightarrow a \rightarrow a$, then $f[\tau]\ (R_{any}[\tau])$ must behave as the identity function. Note that it is not the case that f behaves as the identity on its second argument in general. The free theorem is:

$$\begin{array}{l}
\forall \tau_1\ \tau_2\ r \in VRel(\tau_1, \tau_2) \,.\, \Longrightarrow \\
\quad (\forall e_1\ e_2 \,.\, (e_1, e_2) \in r^* \Longrightarrow (f[\tau_1]\ (R_{any}[\tau_1])\ e_1, f[\tau_2]\ (R_{any}[\tau_2])\ e_2) \in r^*) \wedge \\
\quad (\forall e_1\ e_2 \,.\, r = id_{Int} \wedge \tau_{1,2} = Int \wedge (e_1, e_2) \in r^* \Longrightarrow \\
\qquad (f[\tau_1]\ R_{Int}\ e_1, f[\tau_2]\ R_{Int}\ e_2) \in r^*) \wedge \\
\quad (\forall \tau_a^{1,2} \tau_b^{1,2} e_a^{1,2} e_b^{1,2}\ r_a \in VRel(\tau_a^1, \tau_a^2)\ r_b \in VRel(\tau_b^1, \tau_b^2)\ e_1\ e_2 \,. \\
\qquad \tau_1 = \tau_a^1 \rightarrow \tau_b^1 \wedge \tau_2 = \tau_a^2 \rightarrow \tau_b^2 \wedge r = r_a \Rightarrow r_b \wedge (e_1, e_2) \in r^* \Longrightarrow \\
\qquad\quad (f[\tau_1]\ (R_{\rightarrow}[\tau_a^1][\tau_b^1]\ e_a^1\ e_b^1)\ e_1, f[\tau_2]\ (R_{\rightarrow}[\tau_a^2][\tau_b^2]\ e_a^2\ e_b^2)\ e_2) \in r^*)
\end{array}$$

Assuming that x is of type τ_1 and r is the graph of a function h with $\epsilon \vdash h : \tau_1 \rightarrow \tau_2$ we get that

$$h\ (f[\tau_1]\ (R_{any}[\tau_1])\ x) \cong f[\tau_2]\ (R_{any}[\tau_2])\ (h\ x)$$

Taking $\tau_1 = \tau_2$, since the equation above must be true for any h we conclude that $f[\tau]\ (R_{any}[\tau])$ must behave as the identity function on any type τ.

[4] In our Isabelle/HOL formalization we deviated slightly from this definition and used an inductively defined relation, with judgements of the form $(v_1, v_2) \in \mathfrak{R}\ r$. This allowed us to derive inversion principles automatically instead of having to prove them from the definition of the function $\mathfrak{R}$.

However, the free theorems for functions that include representation types have a "you get what you pay for" feeling. Observe, for example, that the case of the theorem above for $f\ [\mathit{Int}]\ R_{\mathit{Int}}$ is not particularly informative.

$$\begin{aligned}&\forall \tau_1\ \tau_2\ r \in \mathit{VRel}(\tau_1,\tau_2)\ e_1\ e_2\,.\\&\quad r = \mathit{id}_{\mathit{Int}} \wedge \tau_{1,2} = \mathit{Int} \wedge (e_1,e_2) \in r^* \Longrightarrow (f[\tau_1]\ R_{\mathit{Int}}\ e_1, f[\tau_2]\ R_{\mathit{Int}}\ e_2) \in r^*\end{aligned}$$

This case derives the following:

$$\forall e_1\ e_2\ i\,.\, e_1 \Downarrow i \wedge e_2 \Downarrow i \Longrightarrow \exists j\,.\,(f[\mathit{Int}]\ R_{\mathit{Int}}\ e_1) \Downarrow j \wedge (f[\mathit{Int}]\ R_{\mathit{Int}}\ e_2) \Downarrow j$$

The above simply asserts that if the function takes two arguments that can be reduced to the same integer value, then the results will always be reducible to the same integer value.

3.2 Ground equivalence

An easy corollary of the fundamental property of the logical relation is that two closed related expressions are interchangeable in all program contexts that return integer values.

Definition 3.3 A *grounding context term* C_τ (or simply context with a hole τ) is an expression C_τ such that $\epsilon \vdash \tau$ and $x{:}\tau \vdash C : \mathit{Int}$ for a distinguished variable x.

Definition 3.4 [Ground equivalence] [5] We write $\vdash e_1 \cong e_2 : \tau$ iff $\epsilon \vdash e_{1,2} : \tau$ and $\forall C_\tau i\ .\ C_\tau\{e_1/x\} \Downarrow i \Longleftrightarrow C_\tau\{e_2/x\} \Downarrow i$.

We can easily establish soundness for ground equivalence by directly applying the fundamental theorem to grounding contexts.

Theorem 3.5 *If* $\epsilon \vdash \tau$ *then* $(\llbracket\tau\rrbracket_\epsilon)^* \subseteq (\vdash \cdot \cong \cdot : \tau)$.

Proof. Assume that $(e_1,e_2) \in (\llbracket\tau\rrbracket_\epsilon)^*$. Pick any context C_τ such that $x{:}\tau \vdash C_\tau : \mathit{Int}$. By the fundamental property, and since $x{:}\tau \vdash (x \mapsto (e_1,e_2))$, we get that $(C_\tau\{e_1/x\}, C_\tau\{e_2/x\}) \in (\llbracket\mathit{Int}\rrbracket_\epsilon)^*$ and therefore both evaluate to the same integer.□

It can be shown, via introduction of applicative equivalence as intermediate step between the logical relation and ground equivalence, that this notion of equivalence is evaluation-respecting and congruence, and hence can be used as the equivalence relation mentioned in the previous sections. Doing so is out of the scope of this paper, but the interested reader can verify that the methods presented by Crary and Harper [9], for example, carry over in our setting.

[5] This is not standard terminology; we use it in lack of a better one.

4 Arbitrary GADTs

The form of Definition 3.1 has a close connection to a constraint-based presentation of the constructors of the R datatype. In such presentations [31,8,24] the types of the various R-constructors would be written as:

$$\begin{aligned} R_{Int} &:: \forall a\,.\,(a = Int) \Rightarrow R\ a \\ R_{\rightarrow} &:: \forall abc\,.\,(a = b \rightarrow c) \Rightarrow R\ b \rightarrow R\ c \rightarrow R\ a \\ R_{any} &:: \forall a\,.\,true \Rightarrow R\ a \end{aligned}$$

Each of these constructors induces a certain refinement, indicated by an *equality constraint*. Our interpretation is motivated by the intuition that a constructor-induced equality constraint $\tau_1 = \tau_2$ must restrict the interpretations of τ_1 and τ_2 in the semantic substitution δ such that $[\![\tau_1]\!]_\delta = [\![\tau_2]\!]_\delta$ and $\delta^i(\tau_1) = \delta^i(\tau_2)$ for $i = 1, 2$. Note additionally that the variables b and c on the type of the $R_{\rightarrow}$ constructor can be viewed as "existentially" quantified because they only appear only negatively in the type of $R_{\rightarrow}$. This existential quantification is reflected in the logical relation with the existential quantification over relations r_a and r_b.

This same intuition is applicable to the interpretation of arbitrary GADTs. As an example, let us consider a generalized algebraic datatype $Eq\ \tau_1\ \tau_2$ with one constructor, $Refl[\tau]$, that enforces *equality at the level of types*.

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash Refl[\tau] : Eq\ \tau\ \tau}\ \textsc{refl} \qquad \frac{\Gamma \vdash e : Eq\ \sigma_1\ \sigma_2 \quad \Gamma \vdash \forall a\,.\,\tau \quad \Gamma \vdash e_{Refl} : \tau\{\sigma_2/a\}}{\Gamma \vdash rcase\ e\ of\ \{e_{Refl}\} : \tau\{\sigma_1/a\}}\ \textsc{refl-case}$$

The rule REFL-CASE ensures that in the body of the expression, the type σ_1 is replaced with σ_2. Turning to the relational interpretation, we can view *Refl* as having type $\forall ab\,.\,(a = b) \Rightarrow Eq\ a\ b$, producing the following definition

$$\begin{aligned} [\![Eq\ \tau_1\ \tau_2]\!]_\delta = \{(Refl[\sigma_1], Refl[\sigma_2]) \mid\ & \delta^1(\tau_1) = \delta^1(\tau_2) = \sigma_1 \wedge \\ & \delta^2(\tau_1) = \delta^2(\tau_2) = \sigma_2 \wedge [\![\tau_1]\!]_\delta = [\![\tau_2]\!]_\delta\} \end{aligned}$$

Using this definition, the fundamental property gives a valuable free theorem.

Example 4.1 [Free theorem for $\forall ab\,.\,Eq\ a\ b \rightarrow a \rightarrow b$] Assume that $\epsilon \vdash f : \forall ab\,.\,Eq\ a\ b \rightarrow a \rightarrow b$. Then:

$$\begin{aligned} & \forall \tau_a^1\ \tau_a^2\ r_a\ \tau_b^1\ \tau_b^2\ r_b\ \tau_1\ \tau_2\ r\,.\,r_a \in VRel(\tau_a^1, \tau_a^2) \wedge r_b \in VRel(\tau_b^1, \tau_b^2) \Longrightarrow \\ & \quad \tau_a^1 = \tau_b^1 = \tau_1 \wedge \tau_a^2 = \tau_b^2 = \tau_2 \wedge r_a = r_b = r \Longrightarrow \\ & \quad\quad (f\ (Refl[\tau_1]), f\ (Refl[\tau_2])) \in r \Rightarrow r^* \end{aligned}$$

We can now argue that for any $\epsilon \vdash x : \tau_1$ and $\epsilon \vdash h : \tau_1 \rightarrow \tau_2$, taking r to be the graph of h, we have:

$$h\ ((f[\tau_1]\ (Refl[\tau_1]))\ x) \cong (f[\tau_2]\ (Refl[\tau_2]))\ (h\ x)$$

In other words, $f[\tau]$ $(\mathit{Refl}[\tau])$ can only behave as the identity function on τ and hence f can safely be used as (a part of) a type-safe generic cast function. The theorem guarantees that its implementation is correct.

All GADTs can be written as normal datatypes augmented with equality constraints. So the technique that we describe here may be generally applied. However, in the general case, interpreting arbitrary GADTs subsumes interpreting arbitrary recursive polymorphic datatypes, which itself is not an easy problem. This problem has been extensively studied, as we discuss in the next section.

5 Related and future work

5.1 Relational parametricity and typing constraints.

The relational interpretation of representation types that we give in this paper could be used to give a general treatment of equality-constrained polymorphism, which we have never seen presented. We also are not aware of any extensions of the abstraction theorem to type systems that include arbitrary constraints, such as qualified types, although parametricity in the presence of subtyping constraints has been studied [6].

5.2 Parametricity and intensional type analysis.

Washburn and Weirich [28] address the issue of parametricity and runtime type analysis by using an information-flow type and kind system to track which types are analyzed. In that setting, the non-interference theorem generalizes Reynolds's abstraction theorem. The free theorems in that language are more informative than here because the types are more informative. However, that work did not address free theorems for languages with GADTs.

Another approach to deriving free theorems for representation types (and more general GADTs) is to use an encoding in the third-order λ-calculus. For example, in previous work [29] we showed how the R type constructor can be encoded with the following definition:

$$\begin{aligned}&\lambda a{:}*\,.\,\forall b{:}* \to *.\,(b\ \mathit{int}) \to\\ &\quad(\forall cd{:}*.\ b\ c \to b\ d \to b\ (c \to d)) \to (\forall c{:}*.\ b\ c) \to b\ a\end{aligned}$$

Using this encoding and a parametricity result for the third-order λ-calculus, one may also derive free theorems for runtime type representations. However we quickly rejected this approach because the theorems produced by this encoding are significantly more complicated—due both to the complexity of encoding as well as the use of higher-order polymorphism—and are therefore difficult to apply to reasoning about programs.

5.3 Mechanizing logical relations arguments.

There is some related work in the formalization of logical relations arguments about polymorphic λ-calculi in theorem provers. Donnelly and Xi prove termination for the simply typed λ-calculus and System F using ATS/LF [11]. Also, Sarnat and Schürmann [22] show that it is possible to use logical relation arguments in Twelf, by first encoding an appropriate assertion logic. In that way, theorems about the logical relation reduce to theorems about the consistency of the assertion logic. However, some logical relations (such as this one) require higher-order logic as their assertion logic, and it is not known how Twelf can establish the consistency of such an encoded logic.

For our particular formalization in Isabelle/HOL we used the locally nameless technique. Bound variables are represented as de Bruijn indices, and free variables as names. Our starting point was Leroy's solution to the POPLmark challenge in Coq [6], with slight modifications in the way that we distinguished between type and term variables. Additionally, we found Charguéraud's suggestions [7] for avoiding equivariance proofs helpful.

5.4 Recursive types and logical relations.

There is much related work in the area of logical relations for recursive and polymorphic types. Logical relation proofs are notoriously difficult for recursive datatypes—without even considering GADTs. The reason is that recursive datatypes prevent us from defining the logical relation inductively on the structure or size of types. To circumvent this restriction we can define such logical relations as fixpoints of certain generating functions. To solve the problem with contravariance, Pitts (based on work of Freyd) [20,19], pioneered the domain-theoretic technique of defining logical relations by a diagonalization argument as fixpoints of bi-functors, that generate relations. Showing that the "positive" fixpoint and the "negative" fixpoint of such functors coincide relies crucially on the local continuity of these functors. Harper and Birkedal [5], and later Crary and Harper [9] translate this technique into a purely operational setting. A different approach involves step-indexed models [2,1], where the logical relation relates triples consisting of an integer k and two values, to mean that the two values are actually equivalent inside any computation that runs at most for k steps, but may be distinguished later. Recently Vouillon and Mellies [17] have proposed an operationally-based relational model in which recursive types can be interpreted, the main characteristic being that the type language is augmented with interval types. Finally, Johann [15] shows how to handle positive (potentially type-parameterized) recursive datatypes.

5.5 Future work

There are several avenues that we plan to explore in future work. Most importantly, we would like to extend this work to arbitrary GADTs in a nonterminating

[6] `http://pauillac.inria.fr/~xleroy/POPLmark/locally-nameless/`

language, using a constraint-based presentation and a syntactic model for recursive polymorphic datatypes. That way our work would more directly relate to Haskell programs. We would also like to add higher-order types (as in F_ω) and a type-level type analysis operator so that we may show parametricity for full λ_R. To do so, we would start with ideas of Washburn and Weirich [28] who show how to extend a relational interpretation of a second-order language with a limited form of type-level type analysis.

Acknowledgements

Many thanks to the members of the Penn PLClub for extensive feedback and discussion. This work was supported by NSF grants CNS-545886 and CCF-541862.

References

[1] 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.

[2] 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.

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

[4] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *The 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 50–65, Oxford, UK, August 2005.

[5] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*, pages 458–490, 1997.

[6] Luca Cardelli, John C. Mitchell, Simone Martini, and Andre Scedrov. An extension of system F with subtyping. In Takayasu Ito and Albert R. Meyer, editors, *Proc. of 1st Int. Symp. on Theor. Aspects of Computer Software, TACS'91, Sendai, Japan, 24–27 Sept 1991*, volume 526, pages 750–770. Springer-Verlag, Berlin, 1991.

[7] Arthur Charguéraud, Benjamin Pierce, and Stephanie Weirich. Proof engineering: Practical techniques for mechanized metatheory, September 2006. Draft.

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

[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] Kevin Donnelly and Hongwei Xi. A formalization of strong normalization for simply typed lambda-calculus and System F. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06), Seattle, WA*, pages 98–113, Aug 2006.

[12] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In Gerard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 280–306. Prentice Hall, 1991.

[13] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[14] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

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

[16] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM Press.

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

[18] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

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

[20] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

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

[22] Jeffrey Sarnat and Carsten Schurmann. On the representation of logical relations in Twelf. Technical Report YaleU/DCS/TR-1362, Yale University, 2006.

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

[24] Vincent Simonet and François Pottier. Constraint-based type inference with guarded algebraic data types. Technical report, INRIA, July 2003.

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

[26] Martin Sulzmann, Jeremy Wazny, and Peter Stuckey. A framework for extended algebraic data types. Technical report, National University of Singapore, 2005.

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

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

[29] Stephanie Weirich. Encoding intensional type analysis. In D. Sands, editor, *10th European Symposium on Programming*, pages 92–106, Genova, Italy, April 2001.

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

[31] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Thirtieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, LA, USA, January 2003.