

Electronic Notes in Theoretical Computer Science 173 (2007) 357–373

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Free Theorems and Runtime Type Representations

# Dimitrios Vytiniotis[1](#_bookmark0) Stephanie Weirich[2](#_bookmark0)

*Department of Computer and Information Science University of Pennsylvania*

*Philadelphia PA, USA*

Abstract

Reynolds’s abstraction theorem [[21](#_bookmark30)], 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 Unfortunately, in the presence of runtime type analysis, the abstraction properties of polymorphic programs are no longer valid.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](#_bookmark20)], 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.

# Introduction

Reynolds’s abstraction theorem [[21](#_bookmark30)] serves as a characterization of *parametric poly- morphism*. It asserts that parametrically polymorphic functions behave in a uniform way, independently of the types at which they are used. Importantly, the abstrac- tion theorem can be used to derive equivalences involving functional programs, just by observing the types of these programs. Wadler [[27](#_bookmark37)] 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](#_bookmark23)]) is that for any function *f* of type ∀*a*.*a* → *a*

1 Email: [dimitriv@cis.upenn.edu](mailto:dimitriv@cis.upenn.edu)

2 Email: [sweirich@cis.upenn.edu](mailto:sweirich@cis.upenn.edu)

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2007.02.043

we know that for any type τ and term *h* : τ → τ we have the following equivalence

*f* [τ ] ◦ *h* ∼= *h* ◦ *f* [τ ]

where ◦ indicates function composition and ∼= indicates *program equivalence*,a rela-

tion 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.

polymorphic λ-calculus. In particular, it does not hold in the presence of an operator However, Reynolds’s abstraction theorem does not hold for all extensions of the language λ*ML* [[14](#_bookmark24)]. In this language, polymorphic functions do not always behave for runtime type analysis, such as the *typecase* operator from Harper and Morrisett’s parametrically. For example, the following function increments integers, but is the

*i*

identity function for any other type.

*f* = Λ*a*.*typecase a of* {*Int* ⇒ λ*x* :*Int* .*succ x* | ⇒ λ*x* :*a*.*x* }

by picking *h* = (λ*x* :*Int* .8). Consequently, even though *f* is of type ∀*a*.*a* → *a*, we can contradict its free theorem

*f* [*Int* ] ◦ (λ*x* :*Int* .8) ∼/= (λ*x* :*Int* .8) ◦ *f* [*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.

*types* to simulate runtime type analysis, such as in the language λ*R* [[10](#_bookmark20)]. In this An alternative approach to generic programming is to use *term representations of* language, *typecase* analyzes terms that represent types, instead of types. The type

type τ, then term *e* has type *R* τ . Determining the identity of *e* simultaneously of a term representation reveals what type it represents—if the term *e* represents 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 be- come more important with the introduction of *generalized algebraic datatypes* (gadts) [[3](#_bookmark13),[8](#_bookmark18),[23](#_bookmark33),[25](#_bookmark34),[16](#_bookmark26),[26](#_bookmark36)], a variant of *inductive families of types* [[12](#_bookmark22)] originally de- veloped 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](#_bookmark40)] 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.

*data R a where*

*RInt* :: *R Int*

*R*→ :: ∀*ab*.*Ra* → *Rb* → *R* (*a* → *b*) *Rany* :: ∀*a*.*Ra*

The *R* datatype has several constructors, each corresponding to the representation of a particular type. For example, *RInt* serves as a runtime representation of the *Int*

datatype. The *R*→ constructor takes the representations for the argument and the result type of a function type, and returns a representation of that function type. The *Rany* 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* τ may perform case analysis on these the type τ is *reﬁned* according to the constructor matched each time. Consider the arguments. However, unlike ordinary pattern matching, in the types of the branches function *g* below of type ∀*a*.*R a* → *a* → *a*.

*g* = Λ*a*.λ*y* :*R a*.*case y of* {*RInt* ⇒ λ*x* :*Int* .*succ x* | ⇒ λ*x* :*a*.*x* }

In the case for *RInt* , 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* → *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 ∀*a*.*Ra* → *a* → *a* we should *not* expect that

1. (*g* [τ ] *r* ) ◦ *h* ∼= *h* ◦ (*g* [τ ] *r* )

Equivalence ([1](#_bookmark2)) does not hold when *h* = (λ*x* :*Int* .8) and *r* = *RInt* . Nevertheless, we holds for any appropriately typed *r* and *h*. To see why consider the function *g* above. should still be able to derive a free theorem for *g* when it is applied to *Rany* , since

*Rany* does not introduce any type refinement:

(*g* [τ] (*Rany* [τ ])) ◦ *h* ∼= *h* ◦ (*g* [τ] (*Rany* [τ ]))

morphic λ-calculus which includes various gadts, such as *R* above. The language In this paper, we explore parametricity in the context of an explicitly typed poly- 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](#_bookmark29)]. 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 expres-

sion 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 re- spect 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 sup- ports arbitrary gadts. We are confident that the ideas presented in this paper will extend to such a language by combining known techniques for relational in- terpretations of recursive types in an operational setting [[5](#_bookmark15),[9](#_bookmark19),[2](#_bookmark11),[1](#_bookmark12)]. 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](#_bookmark27)] proof assistant, available from:

[www.cis.upenn.edu/~dimitriv/parametricity/](http://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](#_bookmark14)]. Our proofs are thoroughly commented and available to other researchers.

# 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](#_bookmark5) we will extend this foundation to a language with representation types. We begin with the syntax of System F:

|  |  |  |
| --- | --- | --- |
| Types  Terms | σ, τ  *e* | ::= *a* | *Int* | τ → τ | ∀*a*.τ  ::= *i* | *succ e* | *x* | λ*x* :τ.*e* | *ee* | Λ*a*.*e* | *e*[τ] |
| Environments | Γ, Δ | ::= ϵ | Γ, *a* | Γ, (*x* :τ) |
| Values | *u*, *v* , *w* | ::= *i* | λ*x* :τ.*e* | Λ*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 σ{τ /*a*} for the capture-avoiding substitution of τ for *a* in σ. The judgement Γ ▶ τ ensures that the free variables of the type τ are bound in Γ. The typing relation is given with judgements of the form Γ ▶ *e* : τ . The operational semantics that we use is a standard, small-step, call-by-name. We write *e*1 → *e*2 for the transition relation

and *e*1 →∗ *e*2 for the reflexive and transitive closure of the transition relation. We write *e*1 ⇓ *e*2 to mean that *e*1 →∗ *e*2 and ∄*e*3.*e*2 → *e*3.

* 1. *Relational interpretation of System F types*

built up from the following definitions. For a closed type τ , we define the set of Every type of System F can be interpreted as a relation between closed values, all closed values of that type as *Value*(τ ) = {*v* | ϵ ▶ *v* : τ} and the set of binary relations between values of two different (closed) types as *VRel* (τ1, τ2) = {*r* | *r* ⊆ *Value*(τ1) × *Value*(τ2)}. We write *id*τ for the set {(*v* , *v* ) | *v* ∈ *Value*(τ )}.

Definition 2.1 [Computation relation lifting] Let *r* ∈ *VRel* (τ1, τ2). The lifting of

*r* to a computation relation *r* ∗ between closed expressions is given by:

*r* ∗ = {(*e*1, *e*2) | ϵ ▶ *e*1 : τ1 ∧ *e*1 ⇓ *v*1 ∧

ϵ ▶ *e*2 : τ2 ∧ *e*2 ⇓ *v*2 ∧ (*v*1, *v*2) ∈ *r* }

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

γ, δ ::= ϵ | δ, *a* '→ (τ1, τ2, *r* ) | δ, *x* '→ (*e*1, *e*2)

We view semantic substitutions as partial maps. Whenever δ(*a*) = (τ1, τ2, *r* ) we write δ1*a* for τ1, δ2*a* for τ2 and δ[*a*] for *r* . Similarly when δ(*x* )= (*e*1, *e*2) we write δ1*x* for *e*1 and δ2*x* for *e*2. We extend the definition to δ1,2τ and δ1,2*e* homomorphically (modulo capture-avoidance).

Definition 2.3 [Function space relation constructor] Let *ra* ∈ *VRel* (τ 1,τ 2) and

*a a*

*rb* ∈ *VRel* (τ 1,τ 2). Then let *ra* ⇒ *rb* ∈ *VRel* (τ 1 → τ 1,τ 2 → τ 2) be

*b b a b a b*

*ra* ⇒ *rb* = {(*v*1, *v*2) | ϵ ▶ *v*1 : τ 1 → τ 1 ∧ ϵ ▶ *v*2 : τ 2 → τ 2 ∧

*a b a b*

∀*e*1*e*2.(*e*1, *e*2) ∈ *ra* ∗ =⇒ (*v*1 *e*1, *v*2 *e*2) ∈ *rb*∗}

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:

[[*Int* ]]δ = *idInt*

[[*a*]]δ = δ[*a*]

[[σ1 → σ2]]δ = [[σ1]]δ ⇒ [[σ2]]δ

[[∀*a*.σ]]δ = {(*v*1, *v*2) | ϵ ▶ *v*1 : δ1(∀*a*.σ) ∧ ϵ ▶ *v*2 : δ2(∀*a*.σ) ∧

∀τ1 τ2 *r* .*r* ∈ *VRel* (τ1, τ2) =⇒

(*v* [τ ], *v* [τ ]) ∈ ([[σ]]

∗

1 1 2 2 δ,*a*'→(τ1 ,τ2,*r* )) }

tution δ, which is used for the interpretation of type variables. Strictly speaking, Note that the interpretation of quantified types extends the given semantic substi- 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 envi-

when a substitution δ is well-formed in an environment Γ, written as Γ ▶ δ. ronments simplifies the formal treatment in Isabelle/HOL. In particular, we define

Definition 2.5 [Well-formed semantic substitutions]

ϵ ▶ ϵ

Δ ▶ δ *r* ∈ *VRel* (τ1, τ2) Δ, *a* ▶ δ, *a* '→ (τ1, τ2, *r* )

Δ ▶ δ (*e*1, *e*2) ∈ [[τ ]] ∗

Δ, (*x* :τ) ▶ δ, *x* '→ (*e*1, *e*2)

δ

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

Lemma 2.6 (Interpretation of types is a value relation) *If* Γ ▶ τ *and* Γ ▶ δ

*then* [[τ ]]δ ∈ *VRel* (δ1τ, δ2τ)*.*

Compositionality asserts that the interpretation of types depends only on the inter-

pretation of structurally smaller types.

Lemma 2.7 (Compositionality) *If* Γ, *a* ▶ τ *and* Γ ▶ σ *and* Γ ▶ δ *then*

[[τ ]]δ,*a*'→(δ1 σ,δ2 σ,[[σ]]δ ) = [[τ {σ/*a*}]]δ

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* Γ ▶ *e* : τ *and*

Γ ▶ δ *then* (δ1*e*, δ2*e*) ∈ [[τ ]] ∗

*.*

δ

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

Corollary 2.9 *If* ϵ ▶ *e* : τ *then* (*e*, *e*) ∈ [[τ ]] ∗*.*

є

Example 2.10 [Free theorem for ∀*a*.*a* → *a*] Let ϵ ▶ *f* : ∀*a*.*a* → *a*. The funda- mental property then gives:

∀τ1 τ2 *r* ∈ *VRel* (τ1, τ2) *e*1 *e*2.(*e*1, *e*2) ∈ *r* ∗ =⇒ (*f* [τ1] *e*1, *f* [τ2] *e*2) ∈ *r* ∗

Let us consider any τ1, τ2, and ϵ ▶ *h* : τ1 → τ2. Let ∼= be an equivalence relation of *h*, defined as *r* = {(*v*1, *v*2) | ϵ ▶ *v*1 : τ1 ∧ ϵ ▶ *v*2 : τ2 ∧ *h v*1 ⇓ *v*2}. Then for typed terms that is evaluation-respecting and congruent. [3](#_bookmark4) Let *r* be the graph pick a value *x* such that ϵ ▶ *x* : τ1 and (*x* , *h x* ) ∈ *r* ∗. By the free theorem, it has to be that *f* [τ1] *x* ⇓ *w*1 and *f* [τ2] (*h x* ) ⇓ *w*2, such that (*w*1, *w*2) ∈ *r* , that is, *h w*1 ⇓ *w*2. Because equivalence is evaluation-respecting we have *f* [τ1] *x* ∼= *w*1 and by congruence *h* (*f* [τ1] *x* ) ∼= *h w*1. For the same reason *h w*1 ∼= *w*2, therefore *h* (*f* [τ1] *x* ) ∼= *w*2, and consequently

*h* (*f* [τ1] *x* ) ∼= *f* [τ2] (*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 prop- erties; we return to this definition in Section [3.2](#_bookmark8). The question of whether the logical 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 interpre- tation of types assigns relations to type variables, just as in Girard’s reducibility candidates method [[13](#_bookmark23)] a type variable is assigned a candidate set. Adding a re- cursion primitive in the language has no further complication for the soundness of the relational semantics, provided that we quantify over TT-closed relations [[19](#_bookmark29)], or (alternatively) admissible relations, that is, strict and limit-preserving ones. How to express admissibility syntactically has been studied elsewhere [[5](#_bookmark15)].

3 Precisely, we need the properties that, for appropriately typed expressions *e* and *e*1 and *e*2, if *e*1 → *e*2

then *e*1 ∼= *e*2, and *e*1 ∼= *e*2 implies *e e*1 ∼= *e e*2.

# Type representations

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

τ ::= ... | *R* τ

*e* ::= ... | *RInt* | *R*→[τ*a* ][τ*b*] *ea eb* | *Rany* [τ ] | *rcase e of* {*eInt* ; *e*→ ; *eany* }

*u*, *v* ::= ... | *RInt* | *R*→[τ*a* ][τ*b*] *ea eb* | *Rany* [τ ]

We add the *R* type form, constructors for representing integer types (*RInt* ), function

form that performs pattern matching on values of type *R* τ , selecting one of the types (*R*→), and the “any” constructor (*Rany* ). The *rcase* expression is an elimination branches (*eInt* , *e*→, or *eany* ). The operational semantics of the extensions are given

with a congruence rule (ecase) and computation rules that select a particular

branch.

*rcase e of* {*e*

*Int*

; *e*→

*eany*

*e* → *e*'

}→ *rcase e*' *of* {*e*

*Int*

; *e*→

; *eany* }

ecase

*rcase R*→[τ*a* ][τ*b*] *ea eb of* {*eInt* ; *e*→ ; *eany* }→ *e*→[τ*a* ][τ*b*] *ea eb*

erfun

erint

*rcase RInt of* {*eInt* ; *e*→ ; *eany* }→ *eInt*

erany

*rcase Rany* [τ ] *of* {*eInt* ; *e*→ ; *eany* }→ *eany* [τ ]

The typing relation is extended with the following new rules.

Γ ▶ *RInt* : *R Int*

rint

Γ ▶ τ rany

Γ ▶ *Rany* [τ]: *R* τ

Γ ▶ *ea* : *R* τ*a* Γ ▶ *eb* : *R* τ*b* rfun

Γ ▶ *R*→[τ*a* ][τ*b*] *ea eb* : *R* (τ*a* → τ*b*)

Γ ▶ ∀*c* .τ Γ ▶ *e* : *R* σ Γ ▶ *eInt* : τ{*Int* /*c*}

Γ ▶ *e*→ : ∀*ab* . *Ra* → *Rb* → τ{(*a* → *b*)/*c*} Γ ▶ *eany* : ∀*c* .τ rcase

Γ ▶ *rcase e of* {*eInt* ; *e*→ ; *eany* } : τ{σ/*c*}

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 Γ ▶ ∀*c*.τ. The result type of the case expression is formed by replacing *c* with σ, when the scrutinee *e* has a *R* σ type. In the case of *eInt* , we know that σ is equal to *Int* , so *c* is replaced by *Int* in that type. Likewise, the *e*→ branch requires first two types *a*, and *b*, and two representations, *R a* and

τ, where *c* has been refined to *a* → *b*. Finally, the type of *eany* does not do any *R b*, acting as a pattern to match against *e*. It then returns an expression of type refinement. These typing rules allow us, for example, to typecheck the example

from the introduction:

*g* = Λ*a*.λ*x* :*R a*.*rcase x of* { λ*w* :*Int* .*succ w* ;

Λ*bc*.λ*z* :*R b*.λ*y* :*R c*.(λ*w* :*b* → *c*.*w* ); Λ*c* . λ*w* :*c*.*w* }

* 1. *Extension of relational interpretation*

We next extend Definition [2.4](#_bookmark3) with the interpretation of *R* τ types. A na¨ıve 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* δ1τ = δ2τ = *Int then* [[*R* τ ]]δ = {(*RInt* , *RInt* )}∪ {(*Rany* [*Int* ], *Rany* [*Int* ])} since if δ*i* τ = *Int* then the only closed values of type *R Int* are *RInt* , and *Rany* [*Int* ]. This na¨ıve interpretation would have a similar case for the other constructors

as well: If δ1τ = τ 1

→ τ 1

and δ2τ = τ 2

→ τ 2

then we would relate pairs

1 1 1 1

*a b a b*

2 2 2 2 2 ∗

(*R*→[τ*a* ][τ

] *ea e* , *R*→[τ*a* ][τ ] *ea e* ) where (*e*1, *ea* ) ∈ ([[*R a*]]

1 2 )

, for some

*b b b b a*

*a*'→(τ*a* ,τ*a* ,*ra* )

∗

arbitrary relation *ra* , (*e*1, *e*2) ∈ ([[*R b*]] ) , for some arbitrary relation *r* .

*b b b*'→(τ 1,τ 2,*rb* ) *b*

*b b*

Finally, no matter what the definition of δ1τ and δ2τ is, we would always include the pair (*Rany* [δ1τ ], *Rany* [δ2τ ]).

However this definition is problematic. Consider the free theorem that results when

ϵ ▶ *g* : ∀*a*.*Ra* → *a* → *a*.

∀τ1 τ2 *r* .*r* ∈ *VRel* (τ1, τ2) *e*1 *e*2 *e*1 *e*2.

(*e*1, *e*2) ∈ ([[*R a*]]

*r r*

)∗ ∧ (*e*1, *e*2) ∈ *r* ∗ =⇒ (*g* [τ1] *e*1 *e*1, *g* [τ2] *e*2 *e*2) ∈ *r* ∗

*r r a*'→(τ1 ,τ2,*r* ) *r r*

Specifically, the above implies that:

∀*r* .*r* ∈ *VRel* (*Int* , *Int* ) =⇒

∀*e*1 *e*2.(*e*1, *e*2) ∈ *r* ∗ =⇒ (*g* [*Int* ] *RInt e*1, *g* [*Int* ] *RInt e*2) ∈ *r* ∗

Let *r* be the graph of the constant function *eight* = λ*x* :*Int* .8. Consequently:

1. *g* [*Int* ] *RInt* (*eight x* ) ∼= *eight* (*g* [*Int* ] *RInt x* )

But, as we saw in Section [1](#_bookmark1), this equation is not true for every *g* of type ∀*a*.*Ra* → *a* → *a*! Technically, the problem lies in the proof of the fundamental property,

which states that, if Γ ▶ *e* : τ and Γ ▶ δ then (δ1*e*, δ2*e*) ∈ [[τ ]] ∗. Its proof proceeds

δ

by induction on the typing derivation. The interesting case follows:

* + Case rcase. We have that Γ ▶ *rcase e of* {*eInt* ; *e*→ ; *eany* } : τ{σ/*c*} given that

Γ ▶ ∀*c* . τ , Γ ▶ *e* : *R* σ, and Γ ▶ *eInt* : τ{*Int* /*c*} (and others). By induction

(δ1*e*, δ2*e*) ∈ [[*R* σ]] ∗. We now proceed by case analysis on δ1σ and δ2σ. Assume

δ

that δ1σ = δ2σ = *Int* (the other cases would create similar problems). Then δ1*e*

and δ2*e* either both evaluate to *RInt* or the both evaluate to *Rany* [*Int* ]. Suppose

δ

the former. By induction, (δ1*e*

*Int*

, δ2*e*

*Int*

) ∈ [[τ {*Int* /*c*}]]

∗, hence δ1*e*

*Int*

⇓ *w*1

and

δ2*eInt* ⇓ *w*2 so that (*w*1, *w*2) ∈ [[τ {*Int* /*c*}]]δ . Hence δ1(*rcase e of* {*eInt* ; *e*→ ; *eany* }) and δ2(*rcase e of* {*eInt* ; *e*→ ; *eany* }) evaluate to *w*1 and *w*2 respectively. We need to establish finally that **[**τ{*Int* /*c*}]]δ = [[τ {σ/*c*}]]δ . Appealing to the compositionality lemma, it suffices to show that:

[[τ ]]δ,(*c*'→(*Int* ,*Int* ,*idInt* )) = [[τ ]]δ,(*c*'→(δ1 σ,δ2 σ,[[σ]]δ ))

At this point the proof is stuck! Although we have that δ1σ = δ2σ = *Int* we have

*no restriction on the interpretation of* σ.

From this failed proof we see that in the interpretation of *R* σ we can return the pair (*RInt* , *RInt* ) if δ1σ = δ2σ = *Int and* [[σ]]δ = *idInt* . Generalizing this idea, we see that the interpretation of *R* σ must restrict the interpretation of σ. We define the interpretation of *R* σ types using the operator R *r* below.

Definition 3.1 [Extension for *R* σ types] [[*R* σ]]δ = R [[σ]]δ

R (*r* ∈*VRel*(τ1 ,τ2 )) = { (*R* , *R* ) | *r* = *id* ∧ τ = τ = *Int* }∪

*Int Int Int* 1 2

{ (*R*→[τ 1][τ 1] *e*1 *e*1, *R*→[τ 2][τ 2] *e*2 *e*2) |

*a b a b a b a b*

∃*ra* ∈ *VRel* (τ 1,τ 2).∃*rb* ∈ *VRel* (τ 1,τ 2).

*a a b b*

∧

*r* = *ra* ⇒ *rb* ∧ τ1 = τ 1 → τ 1 ∧ τ2 = τ 2 → τ 2

*a*

*b*

*a*

*b*

(*e*1, *e*2) ∈ (R *ra* )∗ ∧ (*e*1, *e*2) ∈ (R *r*

∗

) }∪

*a a b b b*

{ (*Rany* [τ1], *Rany* [τ2]) }

The R *r* function is well-defined since the size of the types becomes smaller in sub-

interpretation of σ when interpreting *R* σ types. Note that the constructor *Rany* calls. [4](#_bookmark7) The boxed parts of the definition indicate the restrictions imposed on the does not restrict the interpretation of σ. The constructor *R*→ 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 ∀*a*.*R a* → *a* → *a*] We show here that if ϵ ▶ *f* :

∀*a*.*R a* → *a* → *a*, then *f* [τ] (*Rany* [τ ]) 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:

∀τ1 τ2 *r* ∈ *VRel* (τ1, τ2). =⇒

(∀*e*1 *e*2.(*e*1, *e*2) ∈ *r* ∗ =⇒ (*f* [τ1] (*R* [τ1]) *e*1, *f* [τ2] (*R* [τ2]) *e*2) ∈ *r* ∗) ∧

*any any*

(∀*e*1 *e*2.*r* = *idInt* ∧ τ1,2 = *Int* ∧ (*e*1, *e*2) ∈ *r* ∗ =⇒ (*f* [τ1] *R e* , *f* [τ ] *R e* ) ∈ *r* ∗) ∧

*Int* 1 2 *Int* 2

*a a*

*b b*

(∀τ 1,2τ 1,2*e*1,2*e*1,2 *r*

*a b a b a*

∈ *VRel* (τ 1,τ 2) *rb*

∈ *VRel* (τ 1,τ 2) *e*1 *e*2.

τ1 = τ 1 → τ 1 ∧ τ2 = τ 2 → τ 2 ∧ *r* = *ra* ⇒ *rb* ∧ (*e*1, *e*2) ∈ *r* ∗ =⇒

*a b a b*

(*f* [τ1] (*R*→[τ )

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1][ | τ 1] | *e*1 *e*1) *e*1, *f* [τ2] (*R*→ [τ 2][ | | τ 2] | *e*2 *e*2) *e*2) | ∈ *r* ∗ |
| *a* | *b* | *a b* | *a* | *b* | *a b* |  |

Assuming that *x* is of type τ1 and *r* is the graph of a function *h* with ϵ ▶ *h* : τ1 → τ2

we get that

*h* (*f* [τ1] (*Rany* [τ1]) *x* ) ∼= *f* [τ2] (*Rany* [τ2]) (*h x* )

Taking τ1 = τ2, since the equation above must be true for any *h* we conclude that

*f* [τ] (*Rany* [τ ]) must behave as the identity function on any type τ .

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* [*Int* ] *RInt* is not particularly informative.

∀τ1 τ2 *r* ∈ *VRel* (τ1, τ2) *e*1 *e*2.

*r* = *idInt* ∧ τ1,2 = *Int* ∧ (*e*1, *e*2) ∈ *r* ∗ =⇒ (*f* [τ1] *R e*1, *f* [τ2] *R e*2) ∈ *r* ∗

*Int Int*

This case derives the following:

∀*e*1 *e*2 *i* .*e*1 ⇓ *i* ∧ *e*2 ⇓ *i* =⇒ ∃*j* .(*f* [*Int* ] *RInt e*1) ⇓ *j* ∧ (*f* [*Int* ] *RInt e*2) ⇓ *j*

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) ∈ R *r* . This allowed us to derive inversion principles automatically instead of having to prove them from the definition of the function R.

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.

* 1. *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*r (or simply context with a hole τ) is an expression *C*r such that ϵ ▶ τ and *x* :τ ▶ *C* : *Int* for a distinguished variable *x* . Definition 3.4 [Ground equivalence] [5](#_bookmark9) We write ▶ *e*1 ∼= *e*2 : τ iff ϵ ▶ *e*1,2 : τ and

∀*C*r *i* . *C*r {*e*1/*x* }⇓ *i* ⇐⇒ *C*r {*e*2/*x* }⇓ *i* .

We can easily establish soundness for ground equivalence by directly applying the

fundamental theorem to grounding contexts.

Theorem 3.5 *If* ϵ ▶ τ *then* ([[τ ]] )∗ ⊆ (▶· ∼= · : τ)*.*

є

Proof. Assume that (*e*1, *e*2) ∈ ([[τ ]] )∗. Pick any context *C*r such that *x* :τ ▶ *C*r :

*Int* . By the fundamental property, and since *x* :τ ▶ (*x* '→ (*e*1, *e*2)), we get that

є

∗

(*C*r {*e*1/*x* }, *C*r {*e*2/*x* }) ∈ ([[*Int* ]] ) and therefore both evaluate to the same integer.

є

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](#_bookmark19)], for example, carry over in our setting.

# Arbitrary GADTs

The form of Definition [3.1](#_bookmark6) has a close connection to a constraint-based presentation of the constructors of the *R* datatype. In such presentations [[31](#_bookmark41),[8](#_bookmark18),[24](#_bookmark35)] the types of the various *R*-constructors would be written as:

*RInt* :: ∀*a* . (*a* = *Int* ) ⇒ *Ra*

*R*→ :: ∀*abc* . (*a* = *b* → *c*) ⇒ *Rb* → *Rc* → *Ra Rany* :: ∀*a* . *true* ⇒ *Ra*

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

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 τ1 = τ2 must restrict the interpretations of τ1 and τ2 in the semantic substitution δ such that **[**τ1]]δ = [[τ2]]δ and δ*i* (τ1)= δ*i* (τ2) for *i* = 1, 2. Note additionally that the variables *b* and *c* on the type of the *R*→ constructor can

be viewed as “existentially” quantified because they only appear only negatively in

the type of *R*→. This existential quantification is reflected in the logical relation with the existential quantification over relations *ra* and *rb*.

an example, let us consider a generalized algebraic datatype *Eq* τ1 τ2 with one This same intuition is applicable to the interpretation of arbitrary gadts. As constructor, *Refl* [τ ], that enforces *equality at the level of types*.

Γ ▶ τ

refl

Γ ▶ *Refl* [τ]: *Eq* τ τ

Γ ▶ *e* : *Eq* σ1 σ2 Γ ▶ ∀*a*.τ

Γ ▶ *eRefl* : τ{σ2/*a*}

Γ ▶ *rcase e of* {*eRefl* } : τ{σ1/*a*}

refl-case

The rule

refl-case ensures that in the body of the expression, the type σ1 is

replaced with σ2. Turning to the relational interpretation, we can view *Refl* as

having type ∀*ab*.(*a* = *b*) ⇒ *Eq a b*, producing the following definition

[[*Eq* τ1 τ2]]δ = {(*Refl* [σ1], *Refl* [σ2]) | δ1(τ1)= δ1(τ2)= σ1 ∧

δ2(τ1)= δ2(τ2)= σ2 ∧ [[τ1]]δ = [[τ2]]δ }

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

Example 4.1 [Free theorem for ∀*ab*.*Eq a b* → *a* → *b*] Assume that ϵ ▶ *f* :

∀*ab*.*Eq a b* → *a* → *b*. Then:

∀τ 1 τ 2 *ra* τ 1 τ 2 *rb* τ1 τ2 *r* .*ra* ∈ *VRel* (τ 1,τ 2) ∧ *rb* ∈ *VRel* (τ 1,τ 2) =⇒

*a a b b a a b b*

τ 1 = τ 1 = τ1 ∧ τ 2 = τ 2 = τ2 ∧ *ra* = *rb* = *r* =⇒

*a b a b*

(*f* (*Refl* [τ1]), *f* (*Refl* [τ2])) ∈ *r* ⇒ *r* ∗

We can now argue that for any ϵ ▶ *x* : τ1 and ϵ ▶ *h* : τ1 → τ2, taking *r* to be the graph of *h*, we have:

*h* ((*f* [τ1] (*Refl* [τ1])) *x* ) ∼= (*f* [τ2] (*Refl* [τ2])) (*h x* )

In other words, *f* [τ ] (*Refl* [τ ]) 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.

# Related and future work

* 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](#_bookmark16)].

* 1. *Parametricity and intensional type analysis.*

Washburn and Weirich [[28](#_bookmark38)] 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.

general GADTs) is to use an encoding in the third-order λ-calculus. For example, Another approach to deriving free theorems for representation types (and more in previous work [[29](#_bookmark39)] we showed how the *R* type constructor can be encoded with

the following definition:

λ*a* : ∗ . ∀*b* : ∗→ ∗. (*b int* ) →

(∀*cd* : ∗. *bc* → *bd* → *b* (*c* → *d* )) → (∀*c* : ∗. *b c*) → *ba*

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 signif- icantly 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.

* 1. *Mechanizing logical relations arguments.*

polymorphic λ-calculi in theorem provers. Donnelly and Xi prove termination for There is some related work in the formalization of logical relations arguments about

the simply typed λ-calculus and System F using ATS/LF [[11](#_bookmark21)]. Also, Sarnat and Schu¨rmann [[22](#_bookmark32)] 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 tech- nique. 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](#_bookmark10) , with slight modifications in the way that we distinguished between type and term variables. Additionally, we found Chargu´eraud’s suggestions [[7](#_bookmark17)] for avoiding equivariance proofs helpful.

* 1. *Recursive types and logical relations.*

There is much related work in the area of logical relations for recursive and polymor- phic 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 cer- tain generating functions. To solve the problem with contravariance, Pitts (based on work of Freyd) [[20](#_bookmark31),[19](#_bookmark29)], pioneered the domain-theoretic technique of defining logi- cal 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](#_bookmark15)], and later Crary and Harper [[9](#_bookmark19)] translate this technique into a purely operational setting. A different approach involves step-indexed models [[2](#_bookmark11),[1](#_bookmark12)], 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 Mel- lies [[17](#_bookmark28)] 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](#_bookmark25)] shows how to handle positive (potentially type-parameterized) recursive datatypes.

* 1. *Future work*

There are several avenues that we plan to explore in future work. Most impor- tantly, we would like to extend this work to arbitrary gadts in a nonterminating language, using a constraint-based presentation and a syntactic model for recursive polymorphic datatypes. That way our work would more directly relate to Haskell

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

type analysis operator so that we may show parametricity for full λ*R*. To do so, programs. We would also like to add higher-order types (as in *F*ω) and a type-level we would start with ideas of Washburn and Weirich [[28](#_bookmark38)] 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 dis- cussion. 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/](http://www.cs.pdx.edu/~sheard/papers/silly.pdf)∼ [sheard/papers/silly.pdf](http://www.cs.pdx.edu/~sheard/papers/silly.pdf) , September 1994.

1. 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.
2. Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*, pages 458–490, 1997.
3. 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.

1. Arthur Chargu´eraud, Benjamin Pierce, and Stephanie Weirich. Proof engineering: Practical techniques for mechanized metatheory, September 2006. Draft.
2. James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.
3. Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types.

*Electronic Notes in Theoretical Computer Science*, 2007. (To appear.).

1. Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.
2. 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.
3. Peter Dybjer. Inductive sets and families in Martin-L¨of’s type theory and their set-theoretic semantics. In Gerard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 280–306. Prentice Hall, 1991.
4. Jean-Yves Girard. Une extension de l’interpr´etation de G¨odel `a l’analyse, et son application `a l’´elimination de coupures dans l’analyse et la th´eorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.
5. 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.
6. Patricia Johann and Janis Voigtl¨ander. Free theorems in the presence of seq. *SIGPLAN Not.*, 39(1):99– 110, 2004.
7. 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.
8. 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.
9. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for* *Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
10. 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.
11. Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in* *Computer Science*, 10:321–359, 2000.
12. 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.
13. Jeffrey Sarnat and Carsten Schurmann. On the representation of logical relations in Twelf. Technical Report YaleU/DCS/TR-1362, Yale University, 2006.
14. 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.
15. Vincent Simonet and Fran¸cois Pottier. Constraint-based type inference with guarded algebraic data types. Technical report, INRIA, July 2003.
16. 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.

1. Martin Sulzmann, Jeremy Wazny, and Peter Stuckey. A framework for extended algebraic data types. Technical report, National University of Singapore, 2005.
2. Philip Wadler. Theorems for free! In *FPCA89: Conference on Functional Programming Languages* *and Computer Architecture*, pages 347–359, London, September 1989.
3. 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.
4. Stephanie Weirich. Encoding intensional type analysis. In D. Sands, editor, *10th European Symposium on Programming*, pages 92–106, Genova, Italy, April 2001.
5. Stephanie Weirich. RepLib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, September 2006.
6. 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.