Non-Parametric Parametricity

Georg Neis
MPI-SWS
neis@mpi-sws.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Andreas Rossberg
MPI-SWS
rossberg@mpi-sws.org

Abstract

Type abstraction and intensional type analysis are features seemingly at odds—type abstraction is intended to guarantee parametricity and representation independence, while type analysis is inherently non-parametric. Recently, however, several researchers have proposed and implemented "dynamic type generation" as a way to reconcile these features. The idea is that, when one defines an abstract type, one should also be able to generate at run time a fresh type name, which may be used as a dynamic representative of the abstract type for purposes of type analysis. The question remains: in a language with non-parametric polymorphism, does dynamic type generation provide us with the same kinds of abstraction guarantees that we get from parametric polymorphism?

Our goal is to provide a rigorous answer to this question. We define a step-indexed Kripke logical relation for a language with both non-parametric polymorphism (in the form of type-safe cast) and dynamic type generation. Our logical relation enables us to establish parametricity and representation independence results, even in a non-parametric setting, by attaching arbitrary relational interpretations to dynamically-generated type names. In addition, we explore how programs that are provably equivalent in a more traditional parametric logical relation may be "wrapped" systematically to produce terms that are related by our non-parametric relation, and vice versa. This leads us to a novel "polarized" form of our logical relation, which enables us to distinguish formally between positive and negative notions of parametricity.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Parametricity, intensional type analysis, representation independence, step-indexed logical relations, type-safe cast

1. Introduction

When we say that a language supports *parametric polymorphism*, we mean that "abstract" types in that language are really abstract—that is, no client of an abstract type can guess or depend on its underlying implementation [20]. Traditionally, the parametric nature of polymorphism is guaranteed statically by the language's type system, thus enabling the so-called *type-erasure* interpretation of polymorphism by which type abstractions and instantiations are erased during compilation.

However, some modern programming languages include a useful feature that appears to be in direct conflict with parametric polymorphism, namely the ability to perform *intensional type analysis* [12]. Probably the simplest and most common instance of intensional type analysis is found in the implementation of languages supporting a type `Dynamic` [1]. In such languages, any value v may be cast *to* type `Dynamic`, but the cast *from* type `Dynamic` to any type τ requires a runtime check to ensure that v's actual type equals τ. Other languages such as Acute [25] and Alice ML [23], which are designed to support dynamic loading of modules, require the ability to check dynamically whether a module implements an expected interface, which in turn involves runtime inspection of the module's type components. There have also been a number of more experimental proposals for languages that employ a `typecase` construct to facilitate *polytypic* programming (*e.g.*, [32, 29]).

There is a fundamental tension between type analysis and type abstraction. If one can inspect the identity of an unknown type at run time, then the type is not really abstract, so any invariants concerning values of that type may be broken [32]. Consequently, languages with a type `Dynamic` often distinguish between *castable* and *non-castable* types—with types that mention user-defined abstract types belonging to the latter category—and prohibit values with non-castable types from being cast to type `Dynamic`.

This is, however, an unnecessarily severe restriction, which effectively penalizes programmers for using type abstraction. Given a user-defined abstract type `t`—implemented internally, say, as `int`—it is perfectly reasonable to cast a value of type `t` $\to$ `t` to `Dynamic`, so long as we can ensure that it will subsequently be cast back only to `t` $\to$ `t` (not to, say, `int` $\to$ `int` or `int` $\to$ `t`), *i.e.*, so long as the cast is *abstraction-safe*. Moreover, such casts are useful when marshalling (or "pickling") a modular component whose interface refers to abstract types defined in other components [23]. That said, in order to ensure that casts are abstraction-safe, it is necessary to have some way of distinguishing (dynamically, when a cast occurs) between an abstract type and its implementation.

Thus, several researchers have proposed that languages with type analysis facilities should also support *dynamic type generation* [24, 21, 29, 22]. The idea is simple: when one defines an abstract type, one should also be able to generate at run time a "fresh" type name, which may be used as a dynamic representative of the abstract type for purposes of type analysis.[1] (We will see a concrete example of this in Section 2.) Intuitively, the freshness of type name generation ensures that user-defined abstract types are viewed dynamically in the same way that they are viewed statically—*i.e.*, as distinct from all other types.

[1] In languages with simple module mechanisms, such as Haskell, it is possible to generate unique type names statically. However, this is not sufficient in the presence of functors and local or first-class modules.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08…$5.00
Reprinted from ICFP'09,, [Unknown Proceedings], August 31–September 2, 2009, Edinburgh, Scotland, UK., pp. 1–14.

The question remains: how do we know that dynamic type generation *works*? In a language with intensional type analysis—*i.e., non-parametric* polymorphism—can the systematic use of dynamic type generation provably ensure abstraction safety and provide us with the same kinds of abstraction guarantees that we get from traditional parametric polymorphism?

Our goal is to provide a rigorous answer to this question. We study an extension of System F, supporting (1) a type-safe cast mechanism, which is essentially a variant of Girard's J operator [9], and (2) a facility for dynamic generation of fresh type names. For brevity, we will call this language **G**. As a practical language mechanism, the cast operator is somewhat crude in comparison to the more expressive `typecase`-style constructs proposed in the literature,[2] but it nonetheless renders polymorphism *non-parametric*. Our main technical result is that, in a language with non-parametric polymorphism, parametricity may be provably regained via judicious use of dynamic type generation.

The rest of the paper is structured as follows. In Section 2, we present our language under consideration, G, and also give an example to illustrate how dynamic type generation is useful.

In Section 3, we explain informally the approach that we have developed for reasoning about G. Our approach employs a *step-indexed Kripke logical relation*, with an unusual form of *possible world* that is a close relative of Sumii and Pierce's [26]. This section is intended to be broadly accessible to readers who are generally familiar with the basic idea of relational parametricity but not with the details of (advanced) logical relations techniques.

In Section 4, we formalize our logical relation for G and show how it may be used to reason about parametricity and representation independence. A particularly appealing feature of our formalization is that the *non*-parametricity of G is encapsulated in the notion of what it means for two *types* to be logically related to each other when viewed as *data*. The definition of this type-level logical relation is a one-liner, which can easily be replaced with an alternative "parametric" version.

In Sections 5–8, we explore how terms related by the parametric version of our logical relation may be "wrapped" systematically to produce terms related by the non-parametric version (and vice versa), thus clarifying how dynamic type generation facilitates parametric reasoning. This leads us to a novel "polarized" form of our logical relation, which enables us to distinguish formally between positive and negative notions of parametricity.

In Section 9, we extend G with iso-recursive types to form G^μ and adapt the previous development accordingly. Then, in Section 10, we discuss how the abovementioned "wrapping" function can be seen as an embedding of System F (+ recursive types) into G^μ, which we conjecture to be fully abstract.

Finally, in Section 11, we discuss related work, including recent work on the relevant concepts of dynamic sealing [27] and multi-language interoperation [13], and in Section 12, we conclude and suggest directions for future work.

2. The Language G

Figure 1 defines our non-parametric language G. For the most part, G is a standard call-by-value λ-calculus, consisting of the usual types and terms from System F [9], including pairs and existential types.[3] We also assume an unspecified set of base types b, along with suitable constants c of those types.

Two additional, non-standard constructs isolate the essential aspects of the class of languages we are interested in:

[2] That said, the implementation of dynamic modules in Alice ML, for instance, employs a very similar construct [23].

[3] We could use a Church encoding of existentials through universals, but distinguishing them gives us more leeway later (cf. Section 5).

$$
\begin{array}{lll}
\text{Types} & \tau ::= \alpha \mid b \mid \tau \to \tau \mid \tau \times \tau \mid \forall \alpha.\tau \mid \exists \alpha.\tau \\
\text{Values} & v ::= x \mid c \mid \lambda x{:}\tau.e \mid \langle v_1, v_2\rangle \mid \lambda\alpha.e \mid \mathsf{pack}\ \langle \tau, v\rangle\ \mathsf{as}\ \tau \\
\text{Terms} & e ::= v \mid e\ e \mid \langle e_1, e_2\rangle \mid e.1 \mid e.2 \mid e\ \tau \mid \\
& \qquad \mathsf{pack}\ \langle \tau, e\rangle\ \mathsf{as}\ \tau \mid \mathsf{unpack}\ \langle \alpha, x\rangle{=}e\ \mathsf{in}\ e \mid \\
& \qquad \mathsf{cast}\ \tau\ \tau \mid \mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ e \\
\text{Stores} & \sigma ::= \epsilon \mid \sigma, \alpha{\approx}\tau \\
\text{Config's} & \zeta ::= \sigma; e
\end{array}
$$

$$
\begin{array}{ll}
\text{Type Contexts} & \Delta ::= \epsilon \mid \Delta, \alpha \mid \Delta, \alpha{\approx}\tau \\
\text{Value Contexts} & \Gamma ::= \epsilon \mid \Gamma, x{:}\tau
\end{array}
$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$ $\qquad \cdots$

$$(\textsc{Ecast})\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash \mathsf{cast}\ \tau_1\ \tau_2 : \tau_1 \to \tau_2 \to \tau_2}$$

$$(\textsc{Enew})\frac{\Delta \vdash \tau \qquad \Delta, \alpha{\approx}\tau; \Gamma \vdash e : \tau' \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash \mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ e : \tau'}$$

$$(\textsc{Econv})\frac{\Delta; \Gamma \vdash e : \tau' \qquad \Delta \vdash \tau \approx \tau'}{\Delta; \Gamma \vdash e : \tau}$$

$\boxed{\Delta \vdash \tau}$

$$(\textsc{Tname})\frac{\alpha{\approx}\tau \in \Delta}{\Delta \vdash \alpha} \qquad \cdots$$

$\boxed{\Delta \vdash \tau \approx \tau}$

$$(\textsc{Cname})\frac{\alpha{\approx}\tau \in \Delta}{\Delta \vdash \alpha \approx \tau} \qquad \cdots$$

$\boxed{\vdash \zeta : \tau}$

$$(\textsc{Conf})\frac{\vdash \sigma \qquad \sigma; \epsilon \vdash e : \tau \qquad \epsilon \vdash \tau}{\vdash \sigma; e : \tau}$$

$$
\begin{array}{lrcl}
 & \sigma; (\lambda x{:}\tau.e)\ v & \hookrightarrow & \sigma; e[v/x] \\
 & \sigma; \langle v_1, v_2\rangle.i & \hookrightarrow & \sigma; v_i \\
 & \sigma; (\lambda\alpha.e)\ \tau & \hookrightarrow & \sigma; e[\tau/\alpha] \\
\multicolumn{2}{l}{\sigma; \mathsf{unpack}\ \langle \alpha, x\rangle{=}(\mathsf{pack}\ \langle \tau, v\rangle)\ \mathsf{in}\ e} & \hookrightarrow & \sigma; e[\tau/\alpha][v/x] \\
(\alpha \notin \mathrm{dom}(\sigma)) & \sigma; \mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ e & \hookrightarrow & \sigma, \alpha{\approx}\tau; e \\
(\tau_1 = \tau_2) & \sigma; \mathsf{cast}\ \tau_1\ \tau_2 & \hookrightarrow & \sigma; \lambda x_1{:}\tau_1.\lambda x_2{:}\tau_2.x_1 \\
(\tau_1 \neq \tau_2) & \sigma; \mathsf{cast}\ \tau_1\ \tau_2 & \hookrightarrow & \sigma; \lambda x_1{:}\tau_1.\lambda x_2{:}\tau_2.x_2
\end{array}
$$

(. . . plus standard "search" rules . . .)

Figure 1. Syntax and Semantics of G (excerpt)

- $\mathsf{cast}\ \tau_1\ \tau_2\ v_1\ v_2$ converts v_1 from type τ_1 to τ_2. It checks that those two types are the same at the time of evaluation. If so, the operator *succeeds* and returns v_1. Otherwise, it *fails* and defaults to v_2, which acts as an else clause of the target type τ_2.
- $\mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ e$ generates a fresh abstract type name α. Values of type α can be formed using its *representation type* τ. Both types are deemed *compatible*, but not equivalent. That is, they are considered equal as *classifiers*, but not as *data*. In particular, $\mathsf{cast}\ \alpha\ \tau\ v\ v'$ will not succeed (*i.e.*, it will return v').

Our cast operator is essentially the same as Harper and Mitchell's *TypeCond* operator [11], which was itself a variant of the non-parametric J operator that Girard studied in his thesis [9]. Our new construct is similar to previously proposed constructs for dynamic type generation [21, 29, 22]. However, we do not require *explicit* term-level type coercions to witness the isomorphism between an abstract type name α and its representation τ. Instead, our type system is simple enough that we perform this conversion *implicitly*.

For convenience, we will occasionally use expressions of the form $\mathsf{let}\, x{=}e_1 \,\mathsf{in}\, e_2$, which abbreviate the term $(\lambda x{:}\tau_1.e_2)\, e_1$ (with τ_1 being an appropriate type for e_1). We omit the type annotation for existential packages where clear from context. Moreover, we take the liberty to generalize binary tuples to n-ary ones where necessary and to use pattern matching notation to decompose tuples in the obvious manner.

2.1 Typing Rules

The typing rules for the System F fragment of G are completely standard and thus omitted from Figure 1. We focus on the non-standard rules related to cast and new. Full formal details of the type system appear in the expanded version of this paper [16].

Typing of casts is straightforward (Rule ECAST): $\mathsf{cast}\ \tau_1\, \tau_2$ is simply treated as a function of type $\tau_1 \to \tau_2 \to \tau_2$. Its first argument is the value to be converted, and its second argument is the default value returned in the case of failure. The rule merely requires that the two types be well-formed.

For an expression $\mathsf{new}\, \alpha{\approx}\tau \,\mathsf{in}\, e$, which binds α in e, Rule ENEW checks that the body e is well-typed under the assumption that α is implemented by the representation type τ. For that purpose, we enrich type contexts Δ with entries of the form $\alpha{\approx}\tau$ that keep track of the representation types tied to abstract type names. Note that τ may not mention α.

Syntactically, type names are just type variables. When viewed as data, (*i.e.,* when inspected by the cast operator), types are considered equivalent iff they are syntactically equal. In contrast, when viewed as classifiers for terms, knowledge about the representation of type names may be taken into account. Rule ECONV says that if a term e has a type τ', it may be assigned any other type that is *compatible* with τ'. Type compatibility, in turn, is defined by the judgment $\Delta \vdash \tau_1 \approx \tau_2$. We only show the rule CNAME, which discharges a compatibility assumption $\alpha{\approx}\tau$ from the context; the other rules implement the congruence closure of this axiom. The important point here is that equivalent types are compatible, but compatible types are not necessarily equivalent.

Finally, Rule ENEW also requires that the type τ' of the body e does not contain α (*i.e.,* τ' must be well formed in Δ alone). A type of this form can always be derived by applying ECONV to convert τ' to $\tau'[\tau/\alpha]$.

2.2 Dynamic Semantics

The operational semantics has to deal with generation of fresh type names. To that end, we introduce a *type store* σ to record generated type names. Hence, reduction is defined on *configurations* $(\sigma; e)$ instead of plain terms. Figure 1 shows the main reduction rules. We omit the standard "search" rules for descending into subterms according to call-by-value, left-to-right evaluation order.

The reduction rules for the F fragment are as usual and do not actually touch the store. However, types occurring in F constructs can contain type names bound in the store.

Reducing the expression $\mathsf{new}\, \alpha{\approx}\tau \,\mathsf{in}\, e$ creates a new entry for α in the type store. We rely on the usual hygiene convention for bound variables to ensure that α is fresh with respect to the current store (which can always be achieved by α-renaming).[4]

The two remaining rules are for casts. A cast takes two types and checks that they are equivalent (*i.e.,* syntactically equal). In either case, the expression reduces to a function that will return the appropriate one of the additional value arguments, *i.e.,* the value to be converted in case of success, and the default value otherwise. In the former case, type preservation is ensured because source and target types are known to be equivalent.

[4] A well-known alternative approach would omit the type store in favor of using scope extrusion rules for new binders, as in Rossberg [21].

Type preservation can be expressed using the typing rule CONF for configurations. We formulate this rule by treating the type store as a type context, which is possible because type stores are a syntactic subclass of type contexts. (In a similar manner, we can write $\vdash \sigma$ for well-formedness of store σ, by viewing it as a type context.) It is worth noting that the representation types in the store are actually never inspected by the dynamic semantics. They are only needed for specifying well-formedness of configurations and proving type soundness.

2.3 Motivating Example

Consider the following attempt to write a simple functional "binary semaphore" ADT [17] in G. Following Mitchell and Plotkin [15], we use an existential type, as we would in System F:

$$\begin{aligned}
\tau_{\text{sem}} &:= \exists\alpha.\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool})\\
e_{\text{sem}} &:= \mathsf{pack}\,\langle \mathsf{int}, \langle 1, \lambda x{:}\,\mathsf{int}\,.(1-x), \lambda x{:}\,\mathsf{int}\,.(x \neq 0)\rangle\rangle \ \mathsf{as}\ \tau_{\text{sem}}
\end{aligned}$$

A semaphore essentially is a flag that can be in two states: either *locked* or *unlocked*. The state can be toggled using the first function of the ADT, and it can be polled using the second. Our little module uses an integer value for representing the state, taking 1 for locked and 0 for unlocked. It is an invariant of the implementation that the integer never takes any other value—otherwise, the toggle function would no longer operate correctly.

In System F, the implementation invariant would be protected by the fact that existential types are parametric: there is no way to inspect the witness of α after opening the package, and hence no client could produce values of type α other than those returned by the module (nor could she apply integer operations to them).

Not so in G. The following program uses cast to forge a value s of the abstract semaphore type α:

$$\begin{aligned}
e_{\text{client}} \ := \ & \mathsf{unpack}\,\langle \alpha, \langle s_0, \mathit{toggle}, \mathit{poll}\rangle\rangle = e_{\text{sem}}\ \mathsf{in}\\
& \mathsf{let}\, s = \mathsf{cast}\ \mathsf{int}\ \alpha\ 666\ s_0\ \mathsf{in}\\
& \langle \mathit{poll}\ s, \mathit{poll}\,(\mathit{toggle}\ s)\rangle
\end{aligned}$$

Because reduction of unpack simply substitutes the representation type int for α, the consecutive cast succeeds, and the whole expression evaluates to $\langle \mathsf{true}, \mathsf{true}\rangle$—although the second component should have toggled s and thus be different from the first.

The way to prevent this in G is to create a fresh type name as witness of the abstract type:

$$\begin{aligned}
e_{\text{sem1}} \ := \ & \mathsf{new}\,\alpha' \approx \mathsf{int}\ \mathsf{in}\\
& \mathsf{pack}\,\langle \alpha', \langle 1, \lambda x{:}\,\mathsf{int}\,.(1-x), \lambda x{:}\,\mathsf{int}\,.(x \neq 0)\rangle\rangle \ \mathsf{as}\ \tau_{\text{sem}}
\end{aligned}$$

After replacing the initial semaphore implementation with this one, e_{client} will evaluate to $\langle \mathsf{true}, \mathsf{false}\rangle$ as desired—the cast expression will no longer succeed, because α will be substituted by the dynamic type name α', and $\alpha' \neq \mathsf{int}$. (Moreover, since α' is only visible statically in the scope of the new expression, the client has no access to α', and thus cannot *convert* from int to α' either.)

Now, while it is clear that new ensures proper type abstraction in the client program e_{client}, we want to prove that it does so for *any* client program. A standard way of doing so is by showing a more general property, namely *representation independence* [20]: we show that the module e_{sem1} is *contextually equivalent* to another module of the same type, meaning that no G program can observe any difference between the two modules. By choosing that other module to be a suitable reference implementation of the ADT in question, we can conclude that the "real" one behaves properly under all circumstances.

The obvious candidate for a reference implementation of the semaphore ADT is the following:

$$\begin{aligned}
e_{\text{sem2}} \ := \ & \mathsf{new}\,\alpha' \approx \mathsf{bool}\ \mathsf{in}\\
& \mathsf{pack}\,\langle \alpha', \langle \mathsf{true}, \lambda x{:}\,\mathsf{bool}\,.\neg x, \lambda x{:}\,\mathsf{bool}\,.x\rangle\rangle \ \mathsf{as}\ \tau_{\text{sem}}
\end{aligned}$$

Here, the semaphore state is represented directly by a Boolean flag and does not rely on any additional invariant. If we can show that e_{sem1} is contextually equivalent to e_{sem2}, then we can conclude that e_{sem1}'s type representation is truly being held abstract.

2.4 Contextual Equivalence

In order to be able to reason about representation independence, we need to make precise the notion of contextual equivalence.

A context C is an expression with a single hole $[\_]$, defined in the usual manner. Typing of contexts is defined by a judgment form $\vdash C : (\Delta; \Gamma; \tau) \leadsto (\Delta'; \Gamma'; \tau')$, where the triple $(\Delta; \Gamma; \tau)$ indicates the type of the hole. The judgment implies that for any expression e with $\Delta; \Gamma \vdash e : \tau$ we have $\Delta'; \Gamma' \vdash C[e] : \tau'$. The rules are straightforward, the key rule being the one for holes:

$$\frac{\Delta \subseteq \Delta' \qquad \Gamma \subseteq \Gamma'}{\vdash [\_] : (\Delta; \Gamma; \tau) \leadsto (\Delta'; \Gamma'; \tau)}$$

We can now define contextual approximation and contextual equivalence as follows (with $\sigma; e \downarrow$ asserting that $\sigma; e$ terminates):

Definition 2.1 (Contextual Approximation and Equivalence) Let $\Delta; \Gamma \vdash e_1 : \tau$ and $\Delta; \Gamma \vdash e_2 : \tau$.

$$\begin{array}{lcl}
\Delta; \Gamma \vdash e_1 \preceq e_2 : \tau & \overset{\text{def}}{\Leftrightarrow} & \forall C, \tau', \sigma. \\
 & & \vdash \sigma \wedge \vdash C : (\Delta; \Gamma; \tau) \leadsto (\sigma; \epsilon; \tau') \ \wedge \\
 & & \sigma; C[e_1] \downarrow \implies \sigma; C[e_2] \downarrow \\
\Delta; \Gamma \vdash e_1 \simeq e_2 : \tau & \overset{\text{def}}{\Leftrightarrow} & \Delta; \Gamma \vdash e_1 \preceq e_2 : \tau \ \wedge \\
 & & \Delta; \Gamma \vdash e_2 \preceq e_1 : \tau
\end{array}$$

That is, contextual approximation $\Delta; \Gamma \vdash e_1 \preceq e_2 : \tau$ means that for any well-typed program context C with a hole of appropriate type, the termination of $C[e_1]$ implies the termination of $C[e_2]$. Contextual equivalence $\Delta; \Gamma \vdash e_1 \simeq e_2 : \tau$ is just approximation in both directions.

Considering that G does not explicitly contain any recursive or looping constructs, the reader may wonder why termination is used as the notion of "distinguishing observation" in our definition of contextual equivalence. The reason is that the cast operator, together with impredicative polymorphism, makes it possible to write well-typed non-terminating programs [11]. (This was Girard's reason for studying the J operator in the first place [9].) Moreover, using cast, one can encode arbitrary recursive function definitions. Other forms of observation may then be encoded in terms of (non-)termination. See the expanded version of this paper for details [16].

3. A Logical Relation for G: Main Ideas

Following Reynolds [20] and Mitchell [14], our general approach to reasoning about parametricity and representation independence is to define a *logical relation*. Essentially, logical relations give us a tractable way of proving that two terms are contextually equivalent, which in turn gives us a way of proving that abstract types are really abstract. Of course, since polymorphism in G is non-parametric, the definition of our logical relation in the cases of universal and existential types is somewhat unusual. To place our approach in context, we first review the traditional approach to defining logical relations for languages with parametric polymorphism, such as System F.

3.1 Logical Relations for Parametric Polymorphism

Although the technical meaning of "logical relation" is rather woolly, the basic idea is to define an equivalence (or approximation) relation on programs inductively, following the structure of their types. To take the canonical example of arrow types, we would say that two functions are logically related at the type $\tau_1 \to \tau_2$ if, when passed arguments that are logically related at τ_1, either they both diverge or they both converge to values that are logically related at τ_2. The *fundamental theorem* of logical relations states that the logical relation is a congruence with respect to the constructs of the language. Together with what Pitts [17] calls *adequacy—i.e.*, the fact that logically related terms have equivalent termination behavior—the fundamental theorem implies that logically related terms are contextually equivalent, since contextual equivalence is defined precisely to be the largest adequate congruence.

Traditionally, the parametric nature of polymorphism is made clear by the definition of the logical relation for universal and existential types. Intuitively, two type abstractions, $\lambda\alpha.e_1$ and $\lambda\alpha.e_2$, are logically related at type $\forall\alpha.\tau$ if they map related *type* arguments to related results. But what does it mean for two type arguments to be related? Moreover, once we settle on two related type arguments τ_1' and τ_2', at what type do we relate the results $e_1[\tau_1'/\alpha]$ and $e_2[\tau_2'/\alpha]$?

One approach would be to restrict "related type arguments" to be the *same* type τ'. Thus, $\lambda\alpha.e_1$ and $\lambda\alpha.e_2$ would be logically related at $\forall\alpha.\tau$ iff, for any (closed) type τ', it is the case that $e_1[\tau'/\alpha]$ and $e_2[\tau'/\alpha]$ are logically related at the type $\tau[\tau'/\alpha]$. A key problem with this definition, however, is that, due to the quantification over *any* argument type τ', the type $\tau[\tau'/\alpha]$ may in fact be larger than the type $\forall\alpha.\tau$, and thus the definition of the logical relation is no longer inductive in the structure of the type. Another problem is that this definition does not tell us anything about the parametric nature of polymorphism.

Reynolds' alternative approach is a generalization of Girard's "candidates" method for proving strong normalization for System F [9]. The idea is simple: instead of defining two type arguments to be related only if they are the same, allow *any* two different type arguments to be related by an (almost) arbitrary relational interpretation (subject to certain *admissibility* constraints). That is, we parameterize the logical relation at type τ by an interpretation function ρ, which maps each free type variable of τ to a pair of types τ_1', τ_2' together with some (admissible) relation between values of those types. Then, we say that $\lambda\alpha.e_1$ and $\lambda\alpha.e_2$ are logically related at type $\forall\alpha.\tau$ under interpretation ρ iff, for any closed types τ_1' and τ_2' and any relation R between values of those types, it is the case that $e_1[\tau_1'/\alpha]$ and $e_2[\tau_2'/\alpha]$ are logically related at type τ under interpretation $\rho, \alpha \mapsto (\tau_1', \tau_2', R)$.

The miracle of Reynolds/Girard's method is that it simultaneously (1) renders the logical relation inductively well-defined in the structure of the type, and (2) demonstrates the parametricity of polymorphism: logically related type abstractions must behave the same even when passed completely different type arguments, so their behavior may not analyze the type argument and behave in different ways for different arguments. Dually, we can show that two ADTs pack $\langle\tau_1, v_1\rangle$ as $\exists\alpha.\tau$ and pack $\langle\tau_2, v_2\rangle$ as $\exists\alpha.\tau$ are logically related (and thus contextually equivalent) by exhibiting *some* relational interpretation R for the abstract type α, even if the underlying type representations τ_1 and τ_2 are different. This is the essence of what is meant by "representation independence".

Unfortunately, in the setting of G, Reynolds/Girard's method is not directly applicable, precisely because polymorphism in G is not parametric! This essentially forces us back to the first approach suggested above, namely to only consider type arguments to be logically related if they are equal. Moreover, it makes sense: the cast operator views types as data, so types may only be logically related if they are indistinguishable as data.

The natural questions, then, are: (1) what metric do we use to define the logical relation inductively, since the structure of the type no longer suffices, and (2) how do we establish that dynamic type generation regains a form of parametricity? We address these questions in the next two sections, respectively.

3.2 Step-Indexed Logical Relations for Non-Parametricity

First, in order to provide a metric for inductively defining the logical relation, we employ *step-indexing*. Step-indexed logical relations were proposed originally by Appel and McAllester [7] as a way of giving a simple operational-semantics-based model for general recursive types in the context of foundational proof-carrying code. In subsequent work by Ahmed and others [3, 6], the method has been adapted to support relational reasoning in a variety of settings, including untyped and imperative languages.

The key idea of step-indexed logical relations is to index the definition of the logical relation not only by the type of the programs being related, but also by a natural number n representing (intuitively) "the number of steps left in the computation". That is, if two terms e_1 and e_2 are logically related at type τ for n steps, then if we place them in any program context C and run the resulting programs for n steps of computation, we should not be able to produce observably different results (*e.g.,* $C[e_1]$ evaluating to 5 and $C[e_2]$ evaluating to 7). To show that e_1 and e_2 are contextually equivalent, then, it suffices to show that they are logically related for n steps, for any n.

To see how step-indexing helps us, consider how we might define a step-indexed logical relation for G in the case of universal types: two type abstractions $\lambda\alpha.e_1$ and $\lambda\alpha.e_2$ are logically related at $\forall\alpha.\tau$ for n steps iff, for any type argument τ', it is the case that $e_1[\tau'/\alpha]$ and $e_2[\tau'/\alpha]$ are logically related at $\tau[\tau'/\alpha]$ for $n-1$ steps. This reasoning is sound because the only way a program context can distinguish between $\lambda\alpha.e_1$ and $\lambda\alpha.e_2$ in n steps is by first applying them to a type argument τ'—which incurs a step of computation for the β-reduction $(\lambda\alpha.e_i)\,\tau' \hookrightarrow e_i[\tau'/\alpha]$—and then distinguishing between $e_1[\tau'/\alpha]$ and $e_2[\tau'/\alpha]$ within the next $n-1$ steps. Moreover, although the type $\tau[\tau'/\alpha]$ may be larger than $\forall\alpha.\tau$, the step index $n-1$ is smaller, so the logical relation is inductively well-defined.

3.3 Kripke Logical Relations for Dynamic Parametricity

Second, in order to establish the parametricity properties of dynamic type generation, we employ *Kripke logical relations*, *i.e.*, logical relations that are indexed by *possible worlds*.[5] Kripke logical relations are appropriate when reasoning about properties that are true only under certain conditions, such as equivalence of modules with local mutable state. For instance, an imperative ADT might only behave according to its specification if its local data structures obey certain invariants. Possible worlds allow one to codify such *local invariants* on the machine store [18].

In our setting, the local invariant we want to establish is what a dynamically generated type name *means*. That is, we will use possible worlds to assign relational interpretations to dynamically generated type names. For example, consider the programs e_{sem1} and e_{sem2} from Section 2. We want to show they are logically related at $\exists\alpha.\ \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool})$ in an empty initial world w_0 (*i.e.,* under empty type stores). The proof proceeds roughly as follows. First, we evaluate the two programs. This will have the effect of generating a fresh type name α', with $\alpha' \approx \mathsf{int}$ extending the type store of the first program and $\alpha' \approx \mathsf{bool}$ extending the type store of the second program. At this point, we correspondingly extend the initial world w_0 with a mapping from α' to the relation $R = \{(1, \mathsf{true}), (0, \mathsf{false})\}$, thus forming a new world w that specifies the semantic meaning of α'.

[5] In fact, step-indexed logical relations may already be understood as a special case of Kripke logical relations, in which the step index serves as the notion of possible world, and where n is a future world of m iff $n \leq m$.

We now must show that the values

$$\mathsf{pack}\ \langle \alpha', \langle 1, \lambda x{:}\,\mathsf{int}\,.(1-x), \lambda x{:}\,\mathsf{int}\,.(x \neq 0)\rangle\rangle\ \mathsf{as}\ \tau_{\mathsf{sem}}$$

and

$$\mathsf{pack}\ \langle \alpha', \langle \mathsf{true}, \lambda x{:}\,\mathsf{bool}\,.\neg x, \lambda x{:}\,\mathsf{bool}\,.x\rangle\rangle\ \mathsf{as}\ \tau_{\mathsf{sem}}$$

are logically related in the world w. Since G's logical relation for existential types is non-parametric, the two packages must have the *same* type representation, but of course the whole point of using new was to ensure that they do (namely, it is α'). The remainder of the proof is showing that the value components of the packages are related at the type $\alpha' \times (\alpha' \to \alpha') \times (\alpha' \to \mathsf{bool})$ under the interpretation $\rho = \alpha' \mapsto (\mathsf{int}, \mathsf{bool}, R)$ derived from the world w. This last part is completely analogous to what one would show in a standard representation independence proof.

In short, the possible worlds in our Kripke logical relations bring back the ability to assign arbitrary relational interpretations R to abstract types, an ability that was seemingly lost when we moved to a non-parametric logical relation. The only catch is that we can only assign arbitrary interpretations to *dynamic* type names, not to *static*, universally/existentially quantified type variables.

There is one minor technical matter that we glossed over in the above proof sketch but is worth mentioning. Due to nondeterminism of type name allocation, the evaluation of e_{sem1} and e_{sem2} may result in α' being replaced by α'_1 in the former and α'_2 in the latter (for some fresh $\alpha'_1 \neq \alpha'_2$). Moreover, we are also interested in proving equivalence of programs that do not necessarily allocate exactly the same number of type names in the same order.

Consequently, we also include in our possible worlds a partial bijection η between the type names of the first program and the type names of the second program, which specifies how each dynamically generated abstract type is concretely represented in the stores of the two programs. We require them to be in 1-1 correspondence because the cast construct permits the program context to observe equality on type names, as follows:

$$\begin{aligned}&\mathsf{equal?} : \forall\alpha.\forall\beta.\,\mathsf{bool} \overset{\text{def}}{=} \\ &\Lambda\alpha.\Lambda\beta.\ \mathsf{cast}\ ((\alpha\to\alpha)\to\mathsf{bool})\ ((\beta\to\beta)\to\mathsf{bool}) \\ &\qquad\qquad (\lambda x{:}(\alpha\to\alpha).\,\mathsf{true})(\lambda x{:}(\beta\to\beta).\,\mathsf{false})(\lambda x{:}\beta.x)\end{aligned}$$

We then consider types to be logically related if they are the same *up to* this bijection. For instance, in our running example, when extending w_0 to w, we would not only extend its relational interpretation with $\alpha' \mapsto (\mathsf{int}, \mathsf{bool}, R)$ but also extend its η with $\alpha' \mapsto (\alpha'_1, \alpha'_2)$. Thus, the type representations of the two existential packages, α'_1 and α'_2, though syntactically distinct, would still be logically related under w.

4. A Logical Relation for G: Formal Details

Figure 2 displays our step-indexed Kripke logical relation for G in full gory detail. It is easiest to understand this definition by making two passes over it. First, as the step indices have a way of infecting the whole definition in a superficially complex—but really very straightforward—way, we will first walk through the whole definition *ignoring* all occurrences of n's and k's (as well as auxiliary functions like the $\lfloor\cdot\rfloor_n$ operator). Second, we will pinpoint the few places where step indices actually play an important role in ensuring that the logical relation is inductively well-founded.

4.1 Highlights of the Logical Relation

The first section of Figure 2 defines the kinds of semantic objects that are used in the construction of the logical relation. Relations R are sets of *atoms*, which are pairs of terms, e_1 and e_2, indexed by a possible world w. The definition of $\mathrm{Atom}[\tau_1, \tau_2]$ requires that e_1 and e_2 have the types τ_1 and τ_2 under the type stores $w.\sigma_1$ and $w.\sigma_2$, respectively. (We use the dot notation $w.\sigma_i$ to denote the i-th

$$
\begin{array}{lcl}
\mathrm{Atom}_n[\tau_1,\tau_2] & \overset{\text{def}}{=} & \{(k,w,e_1,e_2) \mid k<n \land w \in \mathrm{World}_k \land \vdash w.\sigma_1; e_1 : \tau_1 \land \vdash w.\sigma_2; e_2 : \tau_2\} \\
\mathrm{Rel}_n[\tau_1,\tau_2] & \overset{\text{def}}{=} & \{R \subseteq \mathrm{Atom}_n^{\mathsf{val}}[\tau_1,\tau_2] \mid \forall (k,w,v_1,v_2) \in R.\ \forall (k',w') \sqsupseteq (k,w).\ (k',w',v_1,v_2) \in R\} \\
\mathrm{SomeRel}_n & \overset{\text{def}}{=} & \{r=(\tau_1,\tau_2,R) \mid \mathrm{fv}(\tau_1,\tau_2)=\emptyset \land R \in \mathrm{Rel}_n[\tau_1,\tau_2]\} \\
\mathrm{Interp}_n & \overset{\text{def}}{=} & \{\rho \in \mathrm{TVar} \overset{\text{fin}}{\to} \mathrm{SomeRel}_n\} \\
\mathrm{Conc} & \overset{\text{def}}{=} & \{\eta \in \mathrm{TVar} \overset{\text{fin}}{\to} \mathrm{TVar}\times\mathrm{TVar} \mid \forall \alpha,\alpha' \in \mathrm{dom}(\eta).\ \alpha \neq \alpha' \Rightarrow \eta^1(\alpha) \neq \eta^1(\alpha') \land \eta^2(\alpha) \neq \eta^2(\alpha')\} \\
\mathrm{World}_n & \overset{\text{def}}{=} & \{w=(\sigma_1,\sigma_2,\eta,\rho) \mid \ \vdash \sigma_1 \land \vdash \sigma_2 \land \eta \in \mathrm{Conc} \land \rho \in \mathrm{Interp}_n \land \mathrm{dom}(\eta)=\mathrm{dom}(\rho) \land \\
 & & \quad \forall \alpha \in \mathrm{dom}(\rho).\ \sigma_1 \vdash \rho^1(\alpha) \approx \eta^1(\alpha) \land \sigma_2 \vdash \rho^2(\alpha) \approx \eta^2(\alpha)\}
\end{array}
$$

$$
\begin{array}{rcl}
\lfloor(\sigma_1,\sigma_2,\eta,\rho)\rfloor_n & \overset{\text{def}}{=} & (\sigma_1,\sigma_2,\eta,\lfloor\rho\rfloor_n) \\
\lfloor\rho\rfloor_n & \overset{\text{def}}{=} & \{\alpha \mapsto \lfloor r\rfloor_n \mid \rho(\alpha)=r\} \\
\lfloor(\tau_1,\tau_2,R)\rfloor_n & \overset{\text{def}}{=} & (\tau_1,\tau_2,\lfloor R\rfloor_n) \\
\lfloor R\rfloor_n & \overset{\text{def}}{=} & \{(k,w,e_1,e_2) \in R \mid k<n\} \\
\triangleright R & \overset{\text{def}}{=} & \{(k,w,e_1,e_2) \mid k=0 \lor \\
 & & \quad (k-1,\lfloor w\rfloor_{k-1},e_1,e_2) \in R\}
\end{array}
$$

$$
\begin{array}{rcl}
(k',w') \sqsupseteq (k,w) & \overset{\text{def}}{\Leftrightarrow} & k' \leq k \land w' \in \mathrm{World}_{k'} \land \\
 & & w'.\eta \sqsupseteq w.\eta \land w'.\rho \sqsupseteq \lfloor w.\rho\rfloor_{k'} \land \\
 & & \forall i \in \{1,2\}.\ w'.\sigma_i \supseteq w.\sigma_i \land \\
 & & \quad \mathrm{rng}(w'.\eta^i) - \mathrm{rng}(w.\eta^i) \subseteq \\
 & & \quad \mathrm{dom}(w'.\sigma_i) - \mathrm{dom}(w.\sigma_i) \\
\eta' \sqsupseteq \eta & \overset{\text{def}}{\Leftrightarrow} & \forall \alpha \in \mathrm{dom}(\eta).\ \eta'(\alpha)=\eta(\alpha) \\
\rho' \sqsupseteq \rho & \overset{\text{def}}{\Leftrightarrow} & \forall \alpha \in \mathrm{dom}(\rho).\ \rho'(\alpha)=\rho(\alpha)
\end{array}
$$

$$
\begin{array}{lcl}
V_n[\![\alpha]\!]\rho & \overset{\text{def}}{=} & \lfloor\rho(\alpha).R\rfloor_n \\
V_n[\![b]\!]\rho & \overset{\text{def}}{=} & \{(k,w,c,c) \in \mathrm{Atom}_n[b,b]\} \\
V_n[\![\tau\times\tau']\!]\rho & \overset{\text{def}}{=} & \{(k,w,\langle v_1,v_1'\rangle,\langle v_2,v_2'\rangle) \in \mathrm{Atom}_n[\rho^1(\tau\times\tau'),\rho^2(\tau\times\tau')] \mid \\
 & & \quad (k,w,v_1,v_2) \in V_n[\![\tau]\!]\rho \land (k,w,v_1',v_2') \in V_n[\![\tau']\!]\rho\} \\
V_n[\![\tau'\to\tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,\lambda x{:}\tau_1.e_1,\lambda x{:}\tau_2.e_2) \in \mathrm{Atom}_n[\rho^1(\tau'\to\tau),\rho^2(\tau'\to\tau)] \mid \\
 & & \quad \forall (k',w',v_1,v_2) \in V_n[\![\tau']\!]\rho.\ (k',w') \sqsupseteq (k,w) \Rightarrow \\
 & & \quad (k',w',e_1[v_1/x],e_2[v_2/x]) \in E_n[\![\tau]\!]\rho\} \\
V_n[\![\forall\alpha.\tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,\lambda\alpha.e_1,\lambda\alpha.e_2) \in \mathrm{Atom}_n[\rho^1(\forall\alpha.\tau),\rho^2(\forall\alpha.\tau)] \mid \\
 & & \quad \forall (k',w') \sqsupseteq (k,w).\ \forall (\tau_1,\tau_2,r) \in T_{k'}[\![\Omega]\!]w'. \\
 & & \quad (k',w',e_1[\tau_1/\alpha],e_2[\tau_2/\alpha]) \in \triangleright E_n[\![\tau]\!]\rho,\alpha\mapsto r\} \\
V_n[\![\exists\alpha.\tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,\mathsf{pack}\,\langle\tau_1,v_1\rangle,\mathsf{pack}\,\langle\tau_2,v_2\rangle) \in \mathrm{Atom}_n[\rho^1(\exists\alpha.\tau),\rho^2(\exists\alpha.\tau)] \mid \\
 & & \quad \exists r.\ (\tau_1,\tau_2,r) \in T_k[\![\Omega]\!]w \land (k,w,v_1,v_2) \in \triangleright V_n[\![\tau]\!]\rho,\alpha\mapsto r\} \\
E_n[\![\tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,e_1,e_2) \in \mathrm{Atom}_n[\rho^1(\tau),\rho^2(\tau)] \mid \\
 & & \quad \forall j<k.\ \forall \sigma_1,v_1.\ (w.\sigma_1;e_1 \hookrightarrow^j \sigma_1;v_1) \Rightarrow \\
 & & \quad \exists w',v_2.\ (k-j,w') \sqsupseteq (k,w) \land w'.\sigma_1=\sigma_1 \land (w.\sigma_2;e_2 \hookrightarrow^* w'.\sigma_2;v_2) \land (k-j,w',v_1,v_2) \in V_n[\![\tau]\!]\rho\} \\
 & & \\
T_n[\![\Omega]\!]w & \overset{\text{def}}{=} & \{(w.\eta^1(\tau),w.\eta^2(\tau),(w.\rho^1(\tau),w.\rho^2(\tau),V_n[\![\tau]\!]w.\rho)) \mid \ \mathrm{fv}(\tau) \subseteq \mathrm{dom}(w.\rho)\} \\
 & & \\
G_n[\![\epsilon]\!]\rho & \overset{\text{def}}{=} & \{(k,w,\emptyset,\emptyset) \mid k<n \land w \in \mathrm{World}_k\} \\
G_n[\![\Gamma,x{:}\tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,(\gamma_1,x\mapsto v_1),(\gamma_2,x\mapsto v_2)) \mid \\
 & & \quad (k,w,\gamma_1,\gamma_2) \in G_n[\![\Gamma]\!]\rho \land (k,w,v_1,v_2) \in V_n[\![\tau]\!]\rho\} \\
D_n[\![\epsilon]\!]w & \overset{\text{def}}{=} & \{(\emptyset,\emptyset,\emptyset)\} \\
D_n[\![\Delta,\alpha]\!]w & \overset{\text{def}}{=} & \{((\delta_1,\alpha\mapsto\tau_1),(\delta_2,\alpha\mapsto\tau_2),(\rho,\alpha\mapsto r)) \mid \\
 & & \quad (\delta_1,\delta_2,\rho) \in D_n[\![\Delta]\!]w \land (\tau_1,\tau_2,r) \in T_n[\![\Omega]\!]w\} \\
D_n[\![\Delta,\alpha{\approx}\tau]\!]w & \overset{\text{def}}{=} & \{((\delta_1,\alpha\mapsto\beta_1),(\delta_2,\alpha\mapsto\beta_2),(\rho,\alpha\mapsto r)) \mid \\
 & & \quad (\delta_1,\delta_2,\rho) \in D_n[\![\Delta]\!]w \land \\
 & & \quad \exists \alpha'.\ w.\rho(\alpha')=r \land w.\eta(\alpha')=(\beta_1,\beta_2) \land \\
 & & \quad w.\sigma_1(\beta_1)=\delta_1(\tau) \land w.\sigma_2(\beta_2)=\delta_2(\tau) \land r.R=V_n[\![\tau]\!]\rho\}
\end{array}
$$

$$
\begin{array}{rcl}
\Delta;\Gamma \vdash e_1 \precsim e_2 : \tau & \overset{\text{def}}{\Leftrightarrow} & \Delta;\Gamma \vdash e_1 : \tau \land \Delta;\Gamma \vdash e_2 : \tau \land \\
 & & \forall n \geq 0.\ \forall w_0 \in \mathrm{World}_n.\ \forall (\delta_1,\delta_2,\rho) \in D_n[\![\Delta]\!]w_0.\ \forall (k,w,\gamma_1,\gamma_2) \in G_n[\![\Gamma]\!]\rho. \\
 & & (k,w) \sqsupseteq (n,w_0) \Rightarrow (k,w,\delta_1\gamma_1(e_1),\delta_2\gamma_2(e_2)) \in E_n[\![\tau]\!]\rho
\end{array}
$$

Figure 2. Logical Relation for G

type store component of w, and analogous notation for projecting out the other components of worlds.)

$\mathrm{Rel}[\tau_1,\tau_2]$ defines the set of *admissible* relations, which are permitted to be used as the semantic interpretations of abstract types. For our purposes, admissibility is simply *monotonicity—i.e.,* closure under world extension. That is, if a relation in Rel relates two values v_1 and v_2 under a world w, then the relation must relate those values in any future world of w. (We discuss the definition of world extension below.) Monotonicity is needed in order to ensure that we can extend worlds with interpretations of new dynamic type names, without interfering somehow with the interpretations of the old ones.

Worlds w are 4-tuples $(\sigma_1, \sigma_2, \eta, \rho)$, which describe a set of assumptions under which pairs of terms are related. Here, σ_1 and σ_2 are the type stores under which the terms are typechecked and evaluated. The finite mappings η and ρ share a common domain, which can be understood as the set of abstract type names that have been generated dynamically. These "semantic" type names do not exist in either store σ_1 or σ_2.[6] Rather, they provide a way of referring to an abstract type that is represented by *some* type name α_1 in σ_1 and *some* type name α_2 in σ_2. Thus, for each name $\alpha \in \mathrm{dom}(\eta) = \mathrm{dom}(\rho)$, the *concretization* η maps the "semantic" name α to a pair of "concrete" names from the stores σ_1 and σ_2, respectively. (See the end of Section 3.3 for an example of such an η.) As the definition of Conc makes clear, distinct semantic type names must have distinct concretizations; consequently, η represents a *partial bijection* between σ_1 and σ_2.

The last component of the world w is ρ, which assigns relational interpretations to the aforementioned semantic type names. Formally, ρ maps each α to a triple $r = (\tau_1, \tau_2, R)$, where R is a monotone relation between values of types τ_1 and τ_2. (Again, see the end of Section 3.3 for an example of such a ρ.) The final condition in the definition of World stipulates that the closed syntactic types in the range of ρ and the concrete type names in the range of η are compatible. As a matter of notation, we will write η^i and ρ^i to denote the type substitutions $\{\alpha \mapsto \alpha_i \mid \eta(\alpha) = (\alpha_1, \alpha_2)\}$ and $\{\alpha \mapsto \tau_i \mid \rho(\alpha) = (\tau_1, \tau_2, R)\}$, respectively.

The second section of Figure 2 displays the definition of world extension. In order for w' to extend w (written $w' \sqsupseteq w$), it must be the case that (1) w' specifies semantic interpretations for a superset of the type names that w interprets, (2) for the names that w interprets, w' must interpret them in the same way, and (3) any new semantic type names that w' interprets may only correspond to *new* concrete type names that did not exist in the stores of w. Although the third condition is not strictly necessary, we have found it to be useful when proving certain examples (*e.g.*, the "order independence" example in Section 4.4).

The last section of Figure 2 defines the logical relation itself. $V[\![\tau]\!]\rho$ is the logical relation for values, $E[\![\tau]\!]\rho$ is the one for terms, and $T[\![\Omega]\!]w$ is the one for *types as data*, as described in Section 3 (here, Ω represents the *kind* of types).

$V[\![\tau]\!]\rho$ relates values at the type τ, where the free type variables of τ are given relational interpretations by ρ. Ignoring the step indices, $V[\![\tau]\!]\rho$ is mostly very standard. For instance, at certain points (namely, in the $\to$ and $\forall$ cases), when we quantify over logically related (value or type) arguments, we must allow them to come from an arbitrary future world w' in order to ensure monotonicity. This kind of quantification over future worlds is commonplace in Kripke logical relations.

The only really interesting bit in the definition of $V[\![\tau]\!]\rho$ is the use of $T[\![\Omega]\!]w$ to characterize when the two *type* arguments (resp. components) of a universal (resp. existential) are logically related. As explained in Section 3.3, we consider two types to be logically related in world w iff they are the same up to the partial bijection $w.\eta$. Formally, we define $T[\![\Omega]\!]w$ as a relation on triples (τ_1, τ_2, r), where τ_1 and τ_2 are the two logically related types and r is a relation telling us how to relate values of those types. To be logically related means that τ_1 and τ_2 are the concretizations (according to $w.\eta$) of some "semantic" type τ'. Correspondingly, r is the logical relation $V[\![\tau']\!]w.\rho$ at that semantic type. Thus, when we write $E[\![\tau]\!]\rho, \alpha \mapsto r$ in the definition of $V[\![\forall\alpha.\tau]\!]\rho$, this is roughly equivalent to writing $E[\![\tau[\tau'/\alpha]]\!]\rho$ (which our discussion in Section 3.2 might have led the reader to expect to see here instead). The reason for our present formulation is that $E[\![\tau[\tau'/\alpha]]\!]\rho$ is not quite right: the free variables of τ are interpreted by ρ, but the free variables of τ' are *dynamic* type names whose interpretations are given by $w.\rho$. It is possible to merge ρ and $w.\rho$ into a unified interpretation ρ', but we feel our present approach is cleaner.

Another point of note: since r is uniquely determined from τ_1 and τ_2, it is not really necessary to include it in the $T[\![\Omega]\!]w$ relation. However, as we shall see in Section 6, formulating the logical relation in this way has the benefit of isolating all of the non-parametricity of our logical relation in the definition of $T[\![\Omega]\!]w$.

The term relation $E[\![\tau]\!]\rho$ is very similar to that in previous step-indexed Kripke logical relations [6]. Briefly, it says that two terms are related in an initial world w if whenever the first evaluates to a value under $w.\sigma_1$, the second evaluates to a value under $w.\sigma_2$, and the resulting stores and values are related in some future world w'.

The remainder of the definitions in Figure 2 serve to formalize a logical relation for *open* terms. $G[\![\Gamma]\!]\rho$ is the logical relation on value substitutions γ, which asserts that related γ's must map variables in $\mathrm{dom}(\Gamma)$ to related values. $D[\![\Delta]\!]w$ is the logical relation on type substitutions. It asserts that related δ's must map variables in $\mathrm{dom}(\Delta)$ to types that are related in w. For type variables α bound as $\alpha \approx \tau$, the δ's must map α to a type name whose semantic interpretation in w is precisely the logical relation at τ. Analogously to $T[\![\Omega]\!]w$, the relation $D[\![\Delta]\!]w$ also includes a relational interpretation ρ, which may be uniquely determined from the δ's.

Finally, the open logical relation $\Delta; \Gamma \vdash e_1 \precsim e_2 : \tau$ is defined in a fairly standard way. It says that for any starting world w_0, and any type substitutions δ_1 and δ_2 related in that world, if we are given related value substitutions γ_1 and γ_2 in any future world w, then $\delta_1\gamma_1 e_1$ and $\delta_2\gamma_2 e_2$ are related in w as well.

4.2 Why and Where the Steps Matter

As we explained in Section 3.2, step indices play a critical role in making the logical relation well-founded. Essentially, whenever we run into an apparent circularity, we "go down a step" by defining an n-level property in terms of an $(n-1)$-level one. Of course, this trick only works if, at all such "stepping points", the only way that an adversarial program context could possibly tell whether the n-level property holds or not is by taking one step of computation and then checking whether the underlying $(n-1)$-level property holds. Fortunately, this is the case.

Since worlds contain relations, and relations contain sets of tuples that include worlds, a naïve construction of these objects would have an inconsistent cardinality. We thus stratify both worlds and relations by a step index: n-level worlds $w \in \mathrm{World}_n$ contain n-level interpretations $\rho \in \mathrm{Interp}_n$, which map type variables to n-level relations; n-level relations $R \in \mathrm{Rel}_n[\tau_1, \tau_2]$ only contain atoms indexed by a step level $k < n$ and a world $w \in \mathrm{World}_k$. Although our possible worlds have a different structure than in previous work, the technique of mutual world and relation stratification is similar to that used in Ahmed's thesis [2], as well as recent work by Ahmed, Dreyer and Rossberg [6].

Intuitively, the reason this works in our setting is as follows. Viewed as a judgment, our logical relation asserts that two terms e_1 and e_2 are logically related for k steps in a world w at a type τ under an interpretation ρ (whose domain contains the free type variables of τ). Clearly, in order to handle the case where τ is just a type variable α, the relations r in the range of ρ must include atoms at step index k (*i.e.*, the r's must be in $\mathrm{SomeRel}_{k+1}$).

But what about the relations in the range of $w.\rho$? Those relations only come into play in the universal and existential cases of the logical relation for values. Consider the existential case (the universal one is analogous). There, $w.\rho$ pops up in the definition of the relation r that comes from $T_k[\![\Omega]\!]w$. However, that r is only needed in defining the relatedness of the values v_1 and v_2 at step level $k-1$ (note the definition of $\triangleright R$ in the second section of Figure 2). Con-

[6] In fact, technically speaking, we consider $\mathrm{dom}(\eta) = \mathrm{dom}(\rho)$ to be bound variables of the world w.

sequently, we only need r to include atoms at step $k-1$ and lower (*i.e.,* r must be in $\mathrm{SomeRel}_k$), so the world w from which r is derived need only be in World_k.

As this discussion suggests, it is *imperative* that we "go down a step" in the universal and existential cases of the logical relation. For the other cases, it is not necessary to go down a step, although we have the option of doing so. For example, we could define k-level relatedness at pair type $\tau_1 \times \tau_2$ in terms of $(k-1)$-level relatedness at τ_1 and τ_2. But since the type gets smaller, there is no need to. For clarity, we have only gone down a step in the logical relation at the points where it is absolutely necessary, and we have used the $\triangleright$ notation to underscore those points.

4.3 Key Properties

The main results concerning our logical relation are as follows:

Theorem 4.1 (Fundamental Property for $\precsim$)
If $\Delta; \Gamma \vdash e : \tau$, then $\Delta; \Gamma \vdash e \precsim e : \tau$.

Theorem 4.2 (Soundness of $\precsim$ wrt. Contextual Approximation)
If $\Delta; \Gamma \vdash e_1 \precsim e_2 : \tau$, then $\Delta; \Gamma \vdash e_1 \preceq e_2 : \tau$.

These theorems establish that our logical relation provides a sound technique for proving contextual equivalence of G programs. The proofs of these theorems rely on many technical lemmas, most of which are standard and straightforward to prove. We highlight a few of them here, and refer the reader to the expanded version of this paper for full details of the proofs [16].

One key lemma we have mentioned already is the *monotonicity* lemma, which states that the logical relation for values is closed under world extension, and therefore belongs to the Rel class of relations. Another key lemma is *transitivity of world extension*.

There are also a group of lemmas—Pitts terms them *compatibility* lemmas [17]—which show that the logical relation is a pre-congruence with respect to the constructs of the G language. Of particular note among these are the ones for cast and new.

For cast, we must show that $\mathsf{cast}\ \tau_1\ \tau_2$ is logically related to itself under a type context Δ assuming that τ_1 and τ_2 are well-formed in Δ. This boils down to showing that, for logically related type substitutions δ_1 and δ_2, it is the case that $\delta_1\tau_1 = \delta_1\tau_2$ if and only if $\delta_2\tau_1 = \delta_2\tau_2$. This follows easily from the fact that δ_1 and δ_2, by virtue of being logically related, map the variables in $\mathrm{dom}(\Delta)$ to types that are syntactically identical up to some bijection on type names.

For new, we must show that, if $\Delta, \alpha{\approx}\tau'; \Gamma \vdash e_1 \precsim e_2 : \tau$, then $\Delta; \Gamma \vdash \mathsf{new}\,\alpha{\approx}\tau'\ \mathsf{in}\ e_1 \precsim \mathsf{new}\,\alpha{\approx}\tau'\ \mathsf{in}\ e_2 : \tau$ (assuming $\Delta \vdash \Gamma$ and $\Delta \vdash \tau$). The proof involves extending the η and ρ components of some given initial world w_0 with bindings for the fresh dynamically-generated type name α. The η is extended with $\alpha \mapsto (\alpha_1, \alpha_2)$, where α_1 and α_2 are the concrete fresh names that are chosen when evaluating the left and right new expressions. The ρ is extended so that the relational interpretation of α is simply the logical relation at type τ'. The proof of this lemma is highly reminiscent of the proof of compatibility for ref (reference allocation) in a language with mutable references [6].

Finally, another important compatibility property is *type compatibility, i.e.,* that if $\Delta \vdash \tau_1 \approx \tau_2$ and $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w$, then $V_n[\![\tau_1]\!]\rho = V_n[\![\tau_2]\!]\rho$ and $E_n[\![\tau_1]\!]\rho = E_n[\![\tau_2]\!]\rho$. The interesting case is when τ_1 is a variable α bound in Δ as $\alpha \approx \tau_2$, and the result in this case follows easily from the definition of $D[\![\Delta, \alpha \approx \tau]\!]w$.

4.4 Examples

Semaphore. We now return to our semaphore example from Section 2 and show how to prove representation independence for the two different implementations $e_{\mathsf{sem}1}$ and $e_{\mathsf{sem}2}$. Recall that the former uses int, the latter bool. To show that they are contextually equivalent, it suffices by Soundness to show that each logically approximates the other. We prove only one direction, namely $\vdash e_{\mathsf{sem}1} \precsim e_{\mathsf{sem}2} : \tau_{\mathsf{sem}}$; the other is proven analogously.

Expanding the definitions, we need to show $(k, w, e_{\mathsf{sem}1}, e_{\mathsf{sem}2}) \in E_n[\![\tau_{\mathsf{sem}}]\!]\emptyset$. Note how each term generates a fresh type name α_i in one step, resulting in a package value. Hence all we need to do is come up with a world w' satisfying

- $(k-1, w') \sqsupseteq (k, w)$,
- $w'.\sigma_1 = w.\sigma_1, \alpha_1{\approx}\mathsf{int}$ and $w'.\sigma_2 = w.\sigma_2, \alpha_2{\approx}\mathsf{bool}$,
- $(k-1, w', \mathsf{pack}\langle\alpha_1, v_1\rangle, \mathsf{pack}\langle\alpha_2, v_2\rangle) \in V_n[\![\tau_{\mathsf{sem}}]\!]\emptyset$.

where v_i is the term component of $e_{\mathsf{sem}i}$'s implementation. We construct w' by extending w with mappings that establish the relation between the new type names:

$$\begin{aligned}
R &:= \{(k'', w'', v_{\mathsf{int}}, v_{\mathsf{bool}}) \in \mathrm{Atom}^{\mathsf{val}}_{k-1}[\mathsf{int}, \mathsf{bool}] \mid \\
&\qquad (v_{\mathsf{int}}, v_{\mathsf{bool}}) = (1, \mathsf{true}) \vee (v_{\mathsf{int}}, v_{\mathsf{bool}}) = (0, \mathsf{false})\} \\
r &:= (\mathsf{int}, \mathsf{bool}, R) \\
w' &:= \lfloor w \rfloor_{k-1} \uplus (\alpha_1{\approx}\mathsf{int}, \alpha_2{\approx}\mathsf{bool}, \alpha{\mapsto}(\alpha_1, \alpha_2), \alpha{\mapsto}r)
\end{aligned}$$

The first two conditions above are satisfied by construction. To show that the packages are related we need to show the existence of an r' with $(\alpha_1, \alpha_2, r') \in T_{k-1}[\![\Omega]\!]w'$ such that $(k-2, \lfloor w' \rfloor_{k-2}, v_1, v_2) \in V_n[\![\tau'_{\mathsf{sem}}]\!]\rho, \alpha{\mapsto}r'$, where $\tau'_{\mathsf{sem}} = \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool})$. Since $\alpha_i = w'.\eta^i(\alpha)$, r' must be $(\mathsf{int}, \mathsf{bool}, V_{k-1}[\![\alpha]\!]w'.\rho)$ by definition of $T[\![\Omega]\!]$. Of course, we defined w' the way we did so that this r' is exactly r.

The proof of $(k-2, \lfloor w' \rfloor_{k-2}, v_1, v_2) \in V_n[\![\tau'_{\mathsf{sem}}]\!]\rho, \alpha{\mapsto}r$ decomposes into three parts, following the structure of τ'_{sem}:

1. $(k-2, \lfloor w' \rfloor_{k-2}, 1, \mathsf{true}) \in V_n[\![\alpha]\!]\rho, \alpha{\mapsto}r$
   This holds because $V_n[\![\alpha]\!]\rho, \alpha{\mapsto}r = R$.
2. $(k-2, \lfloor w' \rfloor_{k-2}, \lambda x{:}\mathsf{int}.(1-x), \lambda x{:}\mathsf{bool}.\neg x)$
   $\in V_n[\![\alpha \to \alpha]\!]\rho, \alpha{\mapsto}r$
   - Suppose we are given related arguments in a future world: $(k'', w'', v'_1, v'_2) \in V_n[\![\alpha]\!]\rho, \alpha{\mapsto}r = R$.
   - Hence either $(v'_1, v'_2) = (1, \mathsf{true})$ or $(v'_1, v'_2) = (0, \mathsf{false})$.
   - Consequently, $1 - v'_1$ and $\neg v'_2$ will evaluate in one step, without effects, to values again related by R.
   - In other words, $(k'', w'', 1 - v'_1, \neg v'_2) \in E_n[\![\alpha]\!]\rho, \alpha{\mapsto}r$.
3. $(k-2, \lfloor w' \rfloor_{k-2}, \lambda x.(x \neq 0), \lambda x.x) \in V_n[\![\alpha \to \mathsf{bool}]\!]\rho, \alpha{\mapsto}r$
   Like in the previous part, the arguments v'_1 and v'_2 will be related by R in some future (k'', w''). Therefore $v'_1 \neq 0$ will reduce in one step without effects to v'_2, which already is a value. Because of the definition of the logical relation at type bool, this implies $(k'', w'', v'_1 \neq 0, v'_2) \in E_n[\![\mathsf{bool}]\!]\rho, \alpha{\mapsto}r$.

Partly Benign Effects. When side effects are introduced into a pure language, they often falsify various equational laws concerning repeatability and order independence of computations. In this section, we offer some evidence that the effect of dynamic type generation is partly *benign* in that it does not invalidate some of these equational laws.

First, consider the following functions:

$$\begin{aligned}
v_1 &:= \lambda x{:}(\mathsf{unit} \to \tau).\ \mathsf{let}\, x' = x\,()\ \mathsf{in}\ x\,() \\
v_2 &:= \lambda x{:}(\mathsf{unit} \to \tau).\ x\,()
\end{aligned}$$

The only difference between v_1 and v_2 is whether the argument x is applied once or twice. Intuitively, either $x\,()$ diverges, in which case both programs diverge, or else the first application of x terminates, in which case so should the second.

Second, consider the following functions:

$$v'_1 := \lambda x{:}(\mathsf{unit} \to \tau).\lambda y{:}(\mathsf{unit} \to \tau').\,\mathsf{let}\, y' = y\,()\ \mathsf{in}\ \langle x\,(), y' \rangle$$
$$v'_2 := \lambda x{:}(\mathsf{unit} \to \tau).\lambda y{:}(\mathsf{unit} \to \tau').\langle x\,(), y\,() \rangle$$

The only difference between v'_1 and v'_2 is the order in which they call their argument callbacks x and y. Those calls may both result in the generation of fresh type names, but the order in which the names are generated should not matter.

Using our logical relation, we can prove that v_1 and v_2 are contextually equivalent, and so are v'_1 and v'_2. (Due to space considerations, we refer the interested reader to the expanded version of this paper for full proof details [16].)

However, as we shall see in the example of e'_1 and e'_2 in the next section, our G language does *not* enjoy referential transparency. This is to be expected, of course, since new is an effectful operation and (in-)equality of type names is observable in the language.

5. Wrapping

We have seen that parametricity can be re-established in G by introducing name generation in the right place. But what is the "right place" in general? That is, given an arbitrary expression e with polymorphic type τ_e, how can we *systematically* transform it into an expression e' of the same type τ_e that is parametric?

One obvious—but unfortunately bogus—idea is the following: transform e such that every existential *introduction* and every universal *elimination* creates a fresh name for the respective witness or instance type. Formally, apply the following rewrite rules to e:

$$\begin{array}{lcl} \mathsf{pack}\,\langle \tau, e\rangle\ \mathsf{as}\ \tau' & \rightsquigarrow & \mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ \mathsf{pack}\,\langle \alpha, e\rangle\ \mathsf{as}\ \tau' \\ e\ \tau & \rightsquigarrow & \mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ e\ \alpha \end{array}$$

Obviously, this would make every quantified type abstract, so that any cast that tries to inspect it would fail.

Or would it? Perhaps surprisingly, the answer is no. To see why, consider the following expressions of type $(\exists\alpha.\tau') \times (\exists\alpha.\tau')$:

$$\begin{array}{lcl} e_1 & := & \mathsf{let}\, x = \mathsf{pack}\,\langle \tau, v\rangle\ \mathsf{in}\ \langle x, x\rangle \\ e_2 & := & \langle \mathsf{pack}\,\langle \tau, v\rangle, \mathsf{pack}\,\langle \tau, v\rangle\rangle \end{array}$$

They are clearly equivalent in a parametric language (and in fact they are even equivalent in G). Yet rewriting yields:

$$\begin{array}{lcl} e'_1 & := & \mathsf{let}\, x = (\mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ \mathsf{pack}\,\langle \alpha, v\rangle)\ \mathsf{in}\ \langle x, x\rangle \\ e'_2 & := & \langle \mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ \mathsf{pack}\,\langle \alpha, v\rangle, \mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ \mathsf{pack}\,\langle \alpha, v\rangle\rangle \end{array}$$

The resulting expressions are *not* equivalent anymore, because they perform different effects. Here is one distinguishing context:

$$\begin{array}{l} \mathsf{let}\, p = [\_]\ \mathsf{in}\ \mathsf{unpack}\,\langle \alpha_1, x_1\rangle = p.1\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{unpack}\,\langle \alpha_2, x_2\rangle = p.2\ \mathsf{in}\ \mathsf{equal?}\ \alpha_1\ \alpha_2 \end{array}$$

Although the representation type τ is not disclosed as such, *sharing* between the two abstract types in e'_1 is. In a parametric language, that would not be possible.

In order to introduce effects uniformly, and to hide internal sharing, the transformation we are looking for needs to be defined on the structure of types, not terms. Roughly, for each quantifier occurring in τ_e we need to generate one fresh type name. That is, instead of transforming e itself, we simply *wrap* it with some expression that introduces the necessary names at the boundary, by induction on the type τ_e.

In fact, we can refine the problem further. When looking at a G expression e, what do we actually mean by "making it parametric"? We can mean two different things: either ensuring that e *behaves* parametrically, or dually, that any context *treats* e parametrically. In the former case, we are protecting the *context* against e, in the latter we protect e against malicious contexts. The latter is what is sometimes referred to as *abstraction safety*.

$$\begin{array}{lcll} \mathrm{Wr}^{\pm}_{\tau}(e) & \overset{\text{def}}{=} & \mathsf{let}\, x{=}e\ \mathsf{in}\ \mathrm{Wr}^{\pm}_{\tau}(x) & \text{(if } e \text{ not a value)} \\ \mathrm{Wr}^{\pm}_{\alpha}(v) & \overset{\text{def}}{=} & v & \\ \mathrm{Wr}^{\pm}_{b}(v) & \overset{\text{def}}{=} & v & \\ \mathrm{Wr}^{\pm}_{\tau_1\times\tau_2}(v) & \overset{\text{def}}{=} & \langle \mathrm{Wr}^{\pm}_{\tau_1}(v.1), \mathrm{Wr}^{\pm}_{\tau_2}(v.2)\rangle & \\ \mathrm{Wr}^{\pm}_{\tau_1\to\tau_2}(v) & \overset{\text{def}}{=} & \lambda x_1{:}\tau_1.\,\mathrm{Wr}^{\pm}_{\tau_2}(v\ \mathrm{Wr}^{\mp}_{\tau_1}(x_1)) & \\ \mathrm{Wr}^{\pm}_{\forall\alpha.\tau}(v) & \overset{\text{def}}{=} & \lambda\alpha.\,\mathsf{new}^{\mp}\,\alpha\ \mathsf{in}\ \mathrm{Wr}^{\pm}_{\tau}(v\ \alpha) & \\ \mathrm{Wr}^{\pm}_{\exists\alpha.\tau}(v) & \overset{\text{def}}{=} & \mathsf{unpack}\,\langle \alpha, x\rangle{=}v\ \mathsf{in} & \\ & & \quad \mathsf{new}^{\pm}\,\alpha\ \mathsf{in}\ \mathsf{pack}\,\langle \alpha, \mathrm{Wr}^{\pm}_{\tau}(x)\rangle\ \mathsf{as}\ \exists\alpha.\tau & \\ \mathsf{new}^{+}\,\alpha\ \mathsf{in}\ e & \overset{\text{def}}{=} & \mathsf{new}\,\alpha'{\approx}\alpha\ \mathsf{in}\ e[\alpha'/\alpha] & \\ \mathsf{new}^{-}\,\alpha\ \mathsf{in}\ e & \overset{\text{def}}{=} & e & \end{array}$$

Figure 3. Wrapping

Figure 3 defines a pair of wrapping operators that correspond to these two dual requirements: Wr^{+} protects an expression $e : \tau_e$ from being *used* in a non-parametric way, by inserting fresh names for each existential quantifier. Dually, Wr^{-} forces e to *behave* parametrically by creating a fresh name for each polymorphic instantiation. The definitions extend to other types in the usual functorial manner. Both definitions are interdependent, because roles switch for function arguments. These operators are similar to the type-directed translation that Sumii and Pierce suggest for establishing type abstraction in an untyped language [27] (they propose the descriptive terms "firewall" for Wr^{+}, and "sandbox" for Wr^{-}). However, their use of dynamic sealing instead of type generation results in the insertion of runtime coercions to seal/unseal each individual value of abstract type, while our wrapping leaves such values alone.

Given these operators, we can go back to our semaphore example: e_{sem1} can now be obtained as $\mathrm{Wr}^{+}_{\tau_{\mathsf{sem}}}(e_{\mathsf{sem}})$ (modulo some harmless η-expansions). This generalises to any ADT: wrapping its implementation positively will guarantee abstraction by making it parametric. We prove that in the next section.

Positive wrapping is reminiscent of *module sealing* (or opaque signature ascription) in ML-style module languages. If we view e as a module and its type τ_e as a signature, then $\mathrm{Wr}^{+}_{\tau_e}(e)$ corresponds to the sealing operation $e :> \tau_e$. While module sealing typically only performs static abstraction, wrapping describes the dynamic equivalent [22]. In fact, positive wrapping is precisely how sealing is implemented in Alice ML [23], where the module language is non-parametric otherwise.

The correspondence to module sealing motivates our treatment of existential types. Notice that Wr^{+} causes a fresh type name to be created only once for each existentially quantified type—that is, corresponding to each existential *introduction*. Another option would be to generate type names with each existential *elimination*. In fact, such a semantics would arise naturally were we to use a Church encoding of existentials in conjunction with our wrapping for universals. However, in such a semantics, unpacking an existential value twice would have the effect of producing two distinct abstract types. While this corresponds intuitively to the "generativity" of `unpack` in System F, it is undesirable in the context of dynamic, first-class modules. In particular, in order for an abstract type `t` defined by some dynamic module M to have some permanent identity (so that it can be referenced by other dynamic modules), it is important that each unpacking of M yields a handle to the same name for `t`. Moreover, as we show in the next section, our approach to defining wrapping is sufficient to ensure abstraction safety.

6. Parametric Reasoning

The logical relation developed in Section 4 enables us to do *non-parametric* reasoning about equivalence of G programs. It also

$$T_n^\circ[\![\Omega]\!]w \quad \stackrel{\text{def}}{=} \quad \{(\tau_1, \tau_2, (\tau_1', \tau_2', R)) \mid\ \vdash \tau_i' \ \wedge\ w.\sigma_i \vdash \tau_i \approx \tau_i' \ \wedge\ R \in \mathrm{Rel}_n[\tau_1', \tau_2']\}$$

(everything else as in Figure 2)

Figure 4. Parametric Logical Relation

enables us to do *parametric* reasoning, but only indirectly: we have to explicitly deal with the effects of new and to define worlds containing relations between type names. It would be preferable if we were able to do parametric reasoning directly. For example, given two expressions e_1, e_2 that do not use casts, and assuming that the context does not do so either, we should be able to reason about equivalence of e_1 and e_2 in a manner similar to what we do when reasoning about System F.

6.1 A Parametric Logical Relation

Thanks to the modular formulation of our logical relation in Figure 2, it is easy to modify it so that it becomes parametric. All we need to do is swap out the definition of $T[\![\Omega]\!]w$, which relates types as data. Figure 4 gives an alternative definition that allows choosing an arbitrary relation between arbitrary types. Everything else stays exactly the same. We decorate the set of *parametric logical relations* thus obtained with $^\circ$ (*i.e.,* V°, E°, etc.) to distinguish them from the original ones. Likewise, we write $\precsim^\circ$ for the notion of *parametric logical approximation* defined as in Figure 2 but in terms of the parametric relations. For clarity, we will refer to the original definition as the *non-parametric* logical relation.

This modification gives us a seemingly parametric definition of logical approximation for G terms. But what does that actually *mean*? What is the relation between parametric and non-parametric logical approximation and, ultimately, *contextual* approximation? Since the language is not parametric, clearly, parametrically equivalent terms generally are not contextually equivalent.

The answer is given by the wrapping functions we defined in the previous section. The following theorem connects the two notions of logical relation and approximation that we have introduced:

Theorem 6.1 (Wrapping for $\precsim^\circ$)

1. If $\vdash e_1 \precsim^\circ e_2 : \tau$, then $\vdash \mathrm{Wr}^+_\tau(e_1) \precsim \mathrm{Wr}^+_\tau(e_2) : \tau$.
2. If $\vdash e_1 \precsim e_2 : \tau$, then $\vdash \mathrm{Wr}^-_\tau(e_1) \precsim^\circ \mathrm{Wr}^-_\tau(e_2) : \tau$.

This theorem justifies the definition of the parametric logical relation. At the same time it can be read as a correctness result for the wrapping operators: it says that whenever we can relate two terms using parametric reasoning, then the positive wrappings of the first term contextually approximates the positive wrapping of the second. Dually, once any properly related terms are wrapped negatively, they can safely be passed to any term that depends on its context behaving parametrically.

What can we say about the content of the parametric relation? Obviously, it cannot contain arbitrary non-parametric G terms—*e.g.,* cast $\tau_1\,\tau_2$ is not even related to itself in E°. However, we still obtain the following restricted form of the fundamental property:

Theorem 6.2 (Fundamental Property for $\precsim^\circ$)
If $\Delta; \Gamma \vdash e : \tau$ and e is cast-free, then $\Delta; \Gamma \vdash e \precsim^\circ e : \tau$.

In particular, this implies that any well-typed System F term is parametrically related to itself. The relation will also contain terms with cast, but only if the use of cast does not violate parametricity. (We discuss this further in Section 7.)

Along the same lines, we can show that our parametric logical relation is sound w.r.t. contextual approximation, *if* the definition of the latter is limited to quantifying only over cast-free contexts.

6.2 Examples

Semaphore. Consider our running example of the semaphore module again. Using the parametric relation, we can prove that the two implementations are related without actually reasoning about type generation. That aspect is covered once and for all by the Wrapping Theorem.

Recall the two implementations, here given in unwrapped form:

$$e'_{\mathsf{sem1}} := \mathsf{pack}\,\langle \mathsf{int}, \langle 1, \lambda x{:}\,\mathsf{int}\,.(1-x), \lambda x{:}\,\mathsf{int}\,.(x \neq 0)\rangle\rangle\ \mathsf{as}\ \tau_{\mathsf{sem}}$$
$$e'_{\mathsf{sem2}} := \mathsf{pack}\,\langle \mathsf{bool}, \langle \mathsf{true}, \lambda x{:}\,\mathsf{bool}\,.\neg x, \lambda x{:}\,\mathsf{bool}\,.x\rangle\rangle\ \mathsf{as}\ \tau_{\mathsf{sem}}$$

We can prove $\vdash e'_{\mathsf{sem1}} \precsim^\circ e'_{\mathsf{sem2}} : \tau_{\mathsf{sem}}$ using conventional parametric reasoning about polymorphic terms. Now define $e_{\mathsf{sem1}} = \mathrm{Wr}^+_{\tau_{\mathsf{sem}}}(e'_{\mathsf{sem1}})$ and $e_{\mathsf{sem2}} = \mathrm{Wr}^+_{\tau_{\mathsf{sem}}}(e'_{\mathsf{sem2}})$, which are semantically equivalent to the original definitions in Section 2.3. The Wrapping Theorem then immediately tells us that $\vdash e_{\mathsf{sem1}} \precsim e_{\mathsf{sem2}} : \tau_{\mathsf{sem}}$.

A Free Theorem. We can use the parametric relation for proving free theorems [30] in G. For example, for any $\vdash g : \forall\alpha.\alpha \to \alpha$ in G it holds that $\mathrm{Wr}^-(g)$ either diverges for all possible arguments τ and $\vdash v : \tau$, or it returns v in all cases. We first apply the Fundamental Property for $\precsim$ to relate g to itself in E, then transfer this to E° for $\mathrm{Wr}^-(g)$ using the Wrapping Theorem. From there the proof proceeds in the usual way.

7. Syntactic vs. Semantic Parametricity

The primary motivation for our parametric relation in the previous section was to enable more direct parametric reasoning about the result of (positively) wrapping System F terms. However, it is also possible to use our parametric relation to reason about terms that are *syntactically*, or *intensionally*, non-parametric (*i.e.,* that use cast's), so long as they are *semantically*, or *extensionally*, parametric (*i.e.,* the use of cast is not externally observable).

For example, consider the following two polymorphic functions of type $\forall\alpha.\tau_\alpha$ (here, let $b2i = \lambda x{:}\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 0$):

$$\begin{aligned}
\tau_\alpha &:= \exists\beta.\ (\alpha \times \alpha \to \beta) \times (\beta \to \alpha) \times (\beta \to \alpha) \\
g_1 &:= \lambda\alpha.\,\mathsf{pack}\,\langle \alpha \times \alpha, \langle \lambda p.p,\ \lambda x.(x.1),\ \lambda x.(x.2)\rangle\rangle\ \mathsf{as}\ \tau_\alpha \\
g_2 &:= \lambda\alpha.\,\mathsf{cast}\,\tau_{\mathsf{bool}}\,\tau_\alpha \\
&\qquad (\mathsf{pack}\,\langle \mathsf{int}, \langle \lambda p{:}(\mathsf{bool} \times \mathsf{bool}).\ b2i(p.1) + 2{\times}b2i(p.2), \\
&\qquad\qquad \lambda x{:}\mathsf{int}.\ x\ \mathsf{mod}\ 2 \neq 0, \\
&\qquad\qquad \lambda x{:}\mathsf{int}.\ x\ \mathsf{div}\ 2 \neq 0\rangle\rangle\ \mathsf{as}\ \tau_{\mathsf{bool}}) \\
&\qquad (g_1\,\alpha)
\end{aligned}$$

These two functions take a type argument α and return a simple generic ADT for pairs over α. But g_2 is more clever about it and specializes the representation for $\alpha = \mathsf{bool}$. In that case, it packs both components into the two least significant bits of a single integer. For all other types, g_2 falls back to the generic implementation from g_1.

Using the parametric relation, we will be able to show that $\vdash \mathrm{Wr}^+(g_1) \preceq \mathrm{Wr}^+(g_2) : \forall\alpha.\tau_\alpha$. One might find this surprising, since g_2 is syntactically non-parametric, returning different implementations for different instantiations of its type argument. However, since the two possible implementations g_2 returns are extensionally equivalent to each other, g_2 is semantically indistinguishable from the syntactically parametric g_1.

Formally: Assume that τ_1, τ_2 are the types and $R_\alpha \in \mathrm{Rel}[\tau_1, \tau_2]$ is the relation the context picks, parametrically, for α. If $\tau_2 \neq \mathsf{bool}$, the rest of the proof is straightforward. Otherwise, we do not know

$$
\begin{array}{lll}
V_n^{\pm}[\![\alpha]\!]\rho & \overset{\text{def}}{=} & \lfloor \rho(\alpha).R \rfloor_n \\
V_n^{\pm}[\![b]\!]\rho & \overset{\text{def}}{=} & \{(k,w,c,c) \in \mathrm{Atom}_n[b,b]\} \\
V_n^{\pm}[\![\tau \times \tau']\!]\rho & \overset{\text{def}}{=} & \{(k,w,\langle v_1,v_1'\rangle,\langle v_2,v_2'\rangle) \in \mathrm{Atom}_n[\rho^1(\tau\times\tau'),\rho^2(\tau\times\tau')] \mid \\
& & \quad (k,w,v_1,v_2) \in V_n^{\pm}[\![\tau]\!]\rho \wedge (k,w,v_1',v_2') \in V_n^{\pm}[\![\tau']\!]\rho\} \\
V_n^{\pm}[\![\tau' \to \tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,\lambda x{:}\tau_1.e_1,\lambda x{:}\tau_2.e_2) \in \mathrm{Atom}_n[\rho^1(\tau'\to\tau),\rho^2(\tau'\to\tau)] \mid \\
& & \quad \forall (k',w',v_1,v_2) \in V_n^{\mp}[\![\tau']\!]\rho.\, (k',w') \sqsupseteq (k,w) \Rightarrow \\
& & \quad (k',w',e_1[v_1/x],e_2[v_2/x]) \in E_n^{\pm}[\![\tau]\!]\rho\} \\
V_n^{\pm}[\![\forall\alpha.\tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,\lambda\alpha.e_1,\lambda\alpha.e_2) \in \mathrm{Atom}_n[\rho^1(\forall\alpha.\tau),\rho^2(\forall\alpha.\tau)] \mid \\
& & \quad \forall (k',w') \sqsupseteq (k,w).\, \forall(\tau_1,\tau_2,r) \in T_{k'}^{\mp}[\![\Omega]\!]w'. \\
& & \quad (k',w',e_1[\tau_1/\alpha],e_2[\tau_2/\alpha]) \in \triangleright E_n^{\pm}[\![\tau]\!]\rho,\alpha\mapsto r\} \\
V_n^{\pm}[\![\exists\alpha.\tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,\mathsf{pack}\,\langle\tau_1,v_1\rangle,\mathsf{pack}\,\langle\tau_2,v_2\rangle) \in \mathrm{Atom}_n[\rho^1(\exists\alpha.\tau),\rho^2(\exists\alpha.\tau)] \mid \\
& & \quad \exists r.\,(\tau_1,\tau_2,r) \in T_k^{\pm}[\![\Omega]\!]w \wedge (k,w,v_1,v_2) \in \triangleright V_n^{\pm}[\![\tau]\!]\rho,\alpha\mapsto r\} \\
E_n^{\pm}[\![\tau]\!]\rho & \overset{\text{def}}{=} & \{(k,w,e_1,e_2) \in \mathrm{Atom}_n[\rho^1(\tau),\rho^2(\tau)] \mid \\
& & \quad \forall j < k.\, \forall \sigma_1,v_1.\,(w.\sigma_1;e_1 \hookrightarrow^j \sigma_1;v_1) \Rightarrow \\
& & \quad \exists w',v_2.\,(k-j,w') \sqsupseteq (k,w) \wedge w'.\sigma_1 = \sigma_1 \wedge (w.\sigma_2;e_2 \hookrightarrow^* w'.\sigma_2;v_2) \wedge (k-j,w',v_1,v_2) \in V_n^{\pm}[\![\tau]\!]\rho\}
\end{array}
$$

$$
\begin{array}{lllllll}
T_n^{+}[\![\Omega]\!]w & \overset{\text{def}}{=} & T_n^{\circ}[\![\Omega]\!]w & \qquad & D_n^{+}[\![\Delta]\!]w & \overset{\text{def}}{=} & D_n^{\circ}[\![\Delta]\!]w \\
T_n^{-}[\![\Omega]\!]w & \overset{\text{def}}{=} & T_n[\![\Omega]\!]w & \qquad & D_n^{-}[\![\Delta]\!]w & \overset{\text{def}}{=} & D_n[\![\Delta]\!]w
\end{array}
$$

$$
\begin{array}{lll}
\Delta;\Gamma \vdash e_1 \precsim^{\pm} e_2 : \tau & \overset{\text{def}}{\Leftrightarrow} & \Delta;\Gamma \vdash e_1 : \tau \wedge \Delta;\Gamma \vdash e_2 : \tau \wedge \\
& & \forall n \geq 0, \forall w_0 \in \mathrm{World}_n\,.\,\forall(\delta_1,\delta_2,\rho) \in D_n^{\mp}[\![\Delta]\!]w_0.\, \forall(k,w,\gamma_1,\gamma_2) \in G_n^{\mp}[\![\Gamma]\!]\rho. \\
& & (k,w) \sqsupseteq (n,w_0) \Rightarrow (k,w,\delta_1\gamma_1(e_1),\delta_2\gamma_2(e_2)) \in E_n^{\pm}[\![\tau]\!]\rho
\end{array}
$$

Figure 5. Polarized Logical Relations

anything about τ_1 and R_α, because τ_1 and τ_2 are related in T°. Nevertheless, we can construct a suitable relational interpretation $R_\beta \in \mathrm{Rel}[\tau_1 \times \tau_1, \mathsf{int}]$ for the type β:

$$
\begin{array}{rl}
R_\beta := & \{(k,w,\langle v,v'\rangle,0) \mid (k,w,v,\mathsf{false}),(k,w,v',\mathsf{false}) \in R_\alpha\} \\
\cup & \{(k,w,\langle v,v'\rangle,1) \mid (k,w,v,\mathsf{true}),(k,w,v',\mathsf{false}) \in R_\alpha\} \\
\cup & \{(k,w,\langle v,v'\rangle,2) \mid (k,w,v,\mathsf{false}),(k,w,v',\mathsf{true}) \in R_\alpha\} \\
\cup & \{(k,w,\langle v,v'\rangle,3) \mid (k,w,v,\mathsf{true}),(k,w,v',\mathsf{true}) \in R_\alpha\}
\end{array}
$$

As it turns out, we do not need to know much about the structure of R_α to define R_β. What we are relying on here is only the knowledge that all values in R_α are well-typed, which is built into our definition of Rel. From that we know that there can never be any other value than true or false on the right side of the relation R_α. Hence we can still enumerate all possible cases to define R_β, and do a respective case distinction when proving equivalence of the projection operations.

Interestingly, it seems that our proof relies critically on the fact that our logical relations are restricted to syntactically well-typed terms. Were we to lift this restriction, we would be forced (it seems) to extend the definition of R_β with a "junk" case, but the calls to $b2i$ in g_2 would get stuck if applied to non-boolean values. We leave further investigation of this observation to future work.

8. Polarized Logical Relations

The parametric relation is useful for proving parametricity properties about (the positive wrappings of) G terms. However, it is all-or-nothing: it can only be used to prove parametricity for terms that expect to be *treated* parametrically and also *behave* parametrically—cf. the two dual aspects of parametricity described in Section 5. We might also be interested in proving representation independence for terms that do *not* behave parametrically themselves (in either the syntactic or semantic sense considered in the previous section). One situation where this might show up is if we want to show representation independence for generic ADTs that (like the ones in Section 7) return different results for different instantiations of their type arguments, but where (unlike in Section 7) the difference is not only syntactic but also semantic.

Here is a somewhat contrived example to illustrate the point. Consider the following two polymorphic functions of type $\forall\alpha.\tau_\alpha$:

$$
\begin{array}{rl}
\tau_\alpha := & \exists\beta.\,(\alpha \to \beta) \times (\beta \to \alpha) \\
f_1 := & \lambda\alpha.\,\mathsf{cast}\,\tau_{\mathsf{int}}\,\tau_\alpha\,(\mathsf{pack}\,\langle\mathsf{int},\langle\lambda x{:}\mathsf{int}.x{+}1,\lambda x{:}\mathsf{int}.x\rangle\rangle\ \mathsf{as}\ \tau_{\mathsf{int}}) \\
& \qquad (\mathsf{pack}\,\langle\alpha,\langle\lambda x{:}\alpha.x,\lambda x{:}\alpha.x\rangle\rangle\ \mathsf{as}\ \tau_\alpha) \\
f_2 := & \lambda\alpha.\,\mathsf{cast}\,\tau_{\mathsf{int}}\,\tau_\alpha\,(\mathsf{pack}\,\langle\mathsf{int},\langle\lambda x{:}\mathsf{int}.x,\lambda x{:}\mathsf{int}.x{+}1\rangle\rangle\ \mathsf{as}\ \tau_{\mathsf{int}}) \\
& \qquad (\mathsf{pack}\,\langle\alpha,\langle\lambda x{:}\alpha.x,\lambda x{:}\alpha.x\rangle\rangle\ \mathsf{as}\ \tau_\alpha)
\end{array}
$$

These functions take a type argument α and return a simple ADT β. Values of type α can be injected into β, and projected out again. However, both functions specialize the behavior of this ADT for type int—for integers, injecting n and projecting again will give back not n, but rather $n+1$. This is true for both functions, but they implement it in a different way.

We want to prove that both implementations are equivalent under wrapping using a form of parametric reasoning. However, we cannot do that using the parametric relation from the previous section—since the functions do not *behave* parametrically (*i.e.,* they return observably different packages for different instantiations of their type argument), they will not be related in E°.

To support that kind of reasoning, we need a more refined treatment of parametricity in the logical relation. The idea is to separate the two aforementioned aspects of parametricity. Consequently, we are going to have a pair of separate relations, E^+ and E^-. The former enforces parametric usage, the latter parametric behavior.

Figure 5 gives the definition of these relations. We call them *polarized,* because they are mutually dependent and the polarity ($+$ or $-$) switches for contravariant positions, *i.e.,* for function arguments and for universal quantifiers. Intuitively, in these places, term and context switch roles.

Except for the consistent addition of polarities, the definition of the polarized relations again only represents a minor modification of the original one.[7] We merely refine the definition of the type re-

[7] In fact, all four relations can easily be formulated in a single unified definition indexed by $\iota ::= \epsilon \mid \circ \mid + \mid -$. We refrained from doing so here for the sake of clarity; see the expanded version of this paper for details [16].

$$\begin{tikzcd}
 & e_{\mathrm{G}} \arrow[d, phantom, "\rotatebox{-90}{$\in$}"] & \\
 & E^+ \arrow[dl, bend right, "\mathrm{Wr}^+"'] \arrow[dr, bend left, "\mathrm{Wr}^-"] & \\
e_{\mathrm{G}} \in E \arrow[ur, bend right] \arrow[dr, bend right, "\mathrm{Wr}^-"'] & & E^\circ \ni e_{\mathrm{F}} \arrow[ul, bend left] \arrow[dl, bend left, "\mathrm{Wr}^+"] \\
 & E^- \arrow[ul, bend left] \arrow[ur, bend right] &
\end{tikzcd}$$

Figure 6. Relating the Relations

lation $T[\![\Omega]\!]w$ to distinguish polarity: in the positive case it behaves parametrically (*i.e.,* allowing an arbitrary relation) and in the negative case non-parametrically (*i.e.,* demanding r be the *logical* relation at some type). Thus, existential types behave parametrically in E^+ but non-parametrically in E^-, and vice versa for universals.

8.1 Key Properties

The way in which polarities switch in the polarized relations mirrors what is going on in the definition of wrapping. That of course is no accident, and we can show the following theorem that relates the polarized relations with the non-parametric and parametric ones through uses of wrapping:

Theorem 8.1 (Wrapping for $\precsim^{\pm}$)

1. If $\vdash e_1 \precsim^{+} e_2 : \tau$, then $\vdash \mathrm{Wr}^{+}_{\tau}(e_1) \precsim \mathrm{Wr}^{+}_{\tau}(e_2) : \tau$.
2. If $\vdash e_1 \precsim e_2 : \tau$, then $\vdash \mathrm{Wr}^{-}_{\tau}(e_1) \precsim^{-} \mathrm{Wr}^{-}_{\tau}(e_2) : \tau$.
3. If $\vdash e_1 \precsim^{+} e_2 : \tau$, then $\vdash \mathrm{Wr}^{-}_{\tau}(e_1) \precsim^{\circ} \mathrm{Wr}^{-}_{\tau}(e_2) : \tau$.
4. If $\vdash e_1 \precsim^{\circ} e_2 : \tau$, then $\vdash \mathrm{Wr}^{+}_{\tau}(e_1) \precsim^{-} \mathrm{Wr}^{+}_{\tau}(e_2) : \tau$.

Moreover, we can show that the inverse directions of these implications require no wrapping at all:

Theorem 8.2 (Inclusion for $\precsim^{\pm}$)

1. If $\vdash e_1 \precsim e_2 : \tau$ or $\vdash e_1 \precsim^{\circ} e_2 : \tau$, then $\vdash e_1 \precsim^{+} e_2 : \tau$.
2. If $\vdash e_1 \precsim^{-} e_2 : \tau$, then $\vdash e_1 \precsim e_2 : \tau$ and $\vdash e_1 \precsim^{\circ} e_2 : \tau$.

This theorem can equivalently be stated as: $E^- \subseteq E \subseteq E^+$ and $E^- \subseteq E^\circ \subseteq E^+$.

Note that Theorem 6.1 follows directly from Theorems 8.1 and 8.2. Similarly, the following property follows from Theorem 8.2 together with Theorem 4.1:

Corollary 8.3 (Fundamental Property for $\precsim^{+}$)
If $\Delta; \Gamma \vdash e : \tau$, then $\Delta; \Gamma \vdash e \precsim^{+} e : \tau$.

Interestingly, compatibility does not hold for $\precsim^{\pm}$ (consider the polarities in the rule for application), which has the consequence that we cannot show Corollary 8.3 directly. For a similar reason, we cannot show any such property for $\precsim^{-}$ at all.

Figure 6 depicts all of the above properties in a single diagram. Unlabeled arrows denote inclusion, while labeled arrows denote the wrapping that maps one relation to the other. The $\in$-operators show the fundamental properties for the respective relations, *i.e.,* which class of terms are included (G terms or F terms).

8.2 Example

Getting back to our motivating example from the beginning of the section, it is essentially straightforward to prove that $\vdash f_1 \precsim^{+} f_2 : \forall\alpha.\tau_\alpha$. The proof proceeds as usual, except that we have to make a case distinction when we want to show that the function bodies are related in E^+. At that point, we are given a triple $(\tau_1, \tau_2, r) \in T^{-}[\![\Omega]\!]w$.

If $\tau_1 = \mathsf{int}$, then we know from the definition of T^- that $\tau_2 = \mathsf{int}$, too. We hence know that both sides will evaluate to the specialized version of the ADT. Since we are in E^+, we get to pick some $(\tau_1', \tau_2', r') \in T^{+}[\![\Omega]\!]w$ as the interpretation of β, where the choice of r' is up to us. The natural choice is to use $\tau_1' = \tau_2' = \mathsf{int}$ with the relation $r' = (\mathsf{int}, \mathsf{int}, \{(k, w, n+1, n) \mid n \in \mathbb{Z}\})$. The rest of the proof is then straightforward.

If $\tau_1 \neq \mathsf{int}$ we similarly know that $\tau_2 \neq \mathsf{int}$ from the definition of T^-. Hence, both sides use the default implementations, which are trivially related in E^+, thanks to Corollary 8.3.

Finally, applying the Wrapping Theorem 8.1, we can conclude that $\vdash \mathrm{Wr}^{+}(f_1) \precsim \mathrm{Wr}^{+}(f_2) : \forall\alpha.\tau_\alpha$, and hence by Soundness, $\vdash \mathrm{Wr}^{+}(f_1) \preceq \mathrm{Wr}^{+}(f_2) : \forall\alpha.\tau_\alpha$.

Note how we relied on the knowledge that τ_1 and τ_2 can only be int at the same time. This holds for types related in T^- but not in T^+ or T°. If we had tried to do this proof in E°, the types τ_1 and τ_2 would have been related by T° only, which would give us too little information to proceed with the necessary case distinction.

9. Recursive Types

We now add iso-recursive types to G and call the result G^{μ}:

Types	τ	$::=$	$\ldots \mid \mu\alpha.\tau$
Values	v	$::=$	$\ldots \mid \mathsf{roll}\ v\ \mathsf{as}\ \tau$
Terms	e	$::=$	$\ldots \mid \mathsf{roll}\ e\ \mathsf{as}\ \tau \mid \mathsf{unroll}\ e$

The extensions to the semantics are standard and therefore omitted—they do not affect the type store. Also, the definition of contextual equivalence does not change (except there are more contexts).

9.1 Extending the Logical Relations

The step-indexing that we used in defining our logical relations makes it very easy to adapt them to G^{μ}. There are two natural ways in which we could define the value relation at a recursive type:

1. $V^{\iota}_{n}[\![\mu\alpha.\tau]\!]\rho \overset{\text{def}}{=} \{(k, w, \mathsf{roll}\ v_1, \mathsf{roll}\ v_2) \in \mathrm{Atom}_n[\ldots] \mid (k, w, v_1, v_2) \in \triangleright V^{\iota}_{k}[\![\tau]\!]\rho, \alpha \mapsto V^{\iota}_{k}[\![\mu\alpha.\tau]\!]\rho\}$
2. $V^{\iota}_{n}[\![\mu\alpha.\tau]\!]\rho \overset{\text{def}}{=} \{(k, w, \mathsf{roll}\ v_1, \mathsf{roll}\ v_2) \in \mathrm{Atom}_n[\ldots] \mid (k, w, v_1, v_2) \in \triangleright V^{\iota}_{k}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\rho\}$

For $\iota \in \{\epsilon, \circ\}$—*i.e.,* for the non-parametric and parametric forms of the logical relation—the above two formulations are equivalent due to the validity of a standard substitution property. Unfortunately, though, we do not have such a property for the polarized relation. In fact, for $\iota \in \{+, -\}$, the first definition wrongly records a fixed polarity for α. It is thus crucial that we choose the second one; only then do all key properties continue to hold in G^{μ}.

9.2 Extending the Wrapping

How can we upgrade the wrapping to account for recursive types? Given an argument of type $\mu\alpha.\tau$, the basic idea is to first unfold it to type $\tau[\mu\alpha.\tau/\alpha]$, then wrap it at that type, and finally fold the result back to type $\mu\alpha.\tau$. Of course, since $\tau[\mu\alpha.\tau/\alpha]$ may be larger than $\mu\alpha.\tau$, a direct implementation of this idea will not result in a well-founded definition.

The solution is to use a fixed-point (definable in terms of recursive types, of course), which gives us a handle on the wrapping function we are in the middle of defining. Figure 7 shows the new definition. We first index the wrapping by an environment φ that maps recursive type variables α to wrappings for those variables. Roughly, the wrapping at type $\mu\alpha.\tau$ under environment φ is a recursive function F, defined in terms of the wrapping at type τ under environment $\varphi, \alpha \mapsto F$. Since the bound variable of a recursive type may occur in positions of different polarity, we actually need two mutually recursive functions and then select the right one depending on the polarity. The cognoscenti will recognize this as a

$$\begin{array}{lll}
\mathrm{Wr}^{\pm}_{\alpha;\varphi}(v) & \stackrel{\text{def}}{=} v & (\text{if } \alpha \notin \mathrm{dom}(\varphi)) \\
\mathrm{Wr}^{\pm}_{\alpha;\varphi}(v) & \stackrel{\text{def}}{=} \varphi^{\pm}(\alpha)\ v & (\text{if } \alpha \in \mathrm{dom}(\varphi)) \\
\mathrm{Wr}^{\pm}_{\mu\alpha.\tau;\varphi}(v) & \stackrel{\text{def}}{=} \mathsf{letrec}\ f^{+} = \lambda x.\ \mathsf{roll}\ (\mathrm{Wr}^{+}_{\tau;\varphi'}(\mathsf{unroll}\, x)[\mu\alpha.\tau/\alpha]) & \\
 & \quad \mathsf{and}\ f^{-} = \lambda x.\ \mathsf{roll}\ (\mathrm{Wr}^{-}_{\tau;\varphi'}(\mathsf{unroll}\, x)[\mu\alpha.\tau/\alpha]) & \\
 & \quad \mathsf{in}\ f^{\pm}\ v & (\text{where } \varphi' = \varphi, \alpha \mapsto (f^{+}, f^{-}))
\end{array}$$

(other cases as before except for the consistent addition of φ)

Figure 7. Wrapping for G$^{\mu}$

polarized variant of the so-called *syntactic projection* function associated with a recursive type [8].

Note that the environment only plays a role for recursive types, and that for any τ that does not involve recursive types, $\mathrm{Wr}^{\pm}_{\tau;\emptyset}$ is the same as our old wrapping $\mathrm{Wr}^{\pm}_{\tau}$ from Section 5. Taking $\mathrm{Wr}^{\pm}_{\tau}$ to be shorthand for $\mathrm{Wr}^{\pm}_{\tau;\emptyset}$, all our old wrapping theorems for G continue to hold for G$^{\mu}$. Full proofs of these theorems are given in the expanded version of this paper [16].

10. Towards Full Abstraction

The definition of the parametric relation E° (including the extension for recursive types) is largely very similar to that of a typical step-indexed logical relation $E_{\mathrm{F}^{\mu}}$ for F$^{\mu}$, *i.e.,* System F extended with pairs, existentials and iso-recursive types [3]. The main difference is the presence of worlds, but they are not actually used in a particularly interesting way in E°. Therefore, one might expect that any two F$^{\mu}$ terms related by the hypothetical $E_{\mathrm{F}^{\mu}}$ would also be related by E° and vice versa.

However, this is not obvious: G$^{\mu}$ is more expressive than F$^{\mu}$, *i.e.,* terms in the parametric relation can contain non-trivial uses of casts (*e.g.,* the generic ADT for pairs from Section 7), and there is no evident way to back-translate these terms into F$^{\mu}$, as would be needed for function arguments. That invalidates a proof approach like the one taken by Ahmed and Blume [5].

Ultimately, the property we would like to be able to show is that the embedding of F$^{\mu}$ into G$^{\mu}$ by positive wrapping is *fully abstract*:

$$\vdash e_1 \simeq_{\mathrm{F}^{\mu}} e_2 : \tau \iff \vdash \mathrm{Wr}^{+}_{\tau}(e_1) \simeq \mathrm{Wr}^{+}_{\tau}(e_2) : \tau$$

This equivalence is even stronger than the one about logical relatedness in $E_{\mathrm{F}^{\mu}}$ and E°, because $\precsim$ is only sound w.r.t. contextual approximation, not complete.

Since F$^{\mu}$ is a fragment of G$^{\mu}$, and F$^{\mu}$ contexts cannot observe any difference between an F$^{\mu}$ term and its wrapping, the direction from right to left, called *equivalence reflection*, is not hard to show.

Theorem 10.1 (Equivalence Reflection)
If $\Delta;\Gamma \vdash_{\mathrm{F}^{\mu}} e_1 : \tau$ and $\Delta;\Gamma \vdash_{\mathrm{F}^{\mu}} e_2 : \tau$
and $\Delta;\Gamma \vdash \mathrm{Wr}^{+}_{\tau}(e_1) \simeq \mathrm{Wr}^{+}_{\tau}(e_2) : \tau$, then $\Delta;\Gamma \vdash e_1 \simeq_{\mathrm{F}^{\mu}} e_2 : \tau$.

Unfortunately, it is not known to us whether the other direction, *equivalence preservation*, holds as well. We conjecture that it does, but are not aware of any suitable technique to prove it.

Note that while equivalence reflection also holds for F and G—*i.e.,* in the absence of recursive types—equivalence preservation does not, because non-termination is encodable in G but not in F.

11. Related Work

Type Generation vs. Other Forms of Data Abstraction. Traditionally, authors have distinguished between two complementary forms of data abstraction, sometimes dubbed the *static* and the *dynamic* approach [13]. The former is tied to the type system and relies on parametricity (especially for existential types) to hide an ADT's representation from clients [15]. The latter approach is typically employed in untyped languages, which do not have the ability to place static restrictions on clients. Consequently, data hiding has to be enforced on the level of individual values. For that, languages provide means for generating unique names and using them as *keys* for *dynamically sealing* values. A value sealed by a given key can only be inspected by principals that have access to the key [27].

Dynamic type generation as we employ it [21, 29, 22] can be seen as a middle ground, because it bears resemblance to both approaches. As in the dynamic approach, we cannot rely on parametricity and instead generate dynamic names to protect abstractions. However, these are type-level names, not term-level names, and they only "seal" type information. In particular, individual values of abstract type are still directly represented by the underlying representation type, so that crossing abstraction boundaries has no runtime cost. In that sense, we are closer to the static approach.

Another approach to reconciling type abstraction and type analysis has been proposed by Washburn and Weirich [31]. They introduce a type system that tracks information flow for terms and types-as-data. By distinguishing security levels, the type system can statically prevent unauthorized inspection of types by clients.

Multi-Language Interoperation. The closest work to ours is that of Matthews and Ahmed [13]. They describe a pair of mutually recursive logical relations that deal with the interoperation between a typed language ("ML") and an untyped language ("Scheme"). Unlike in G, parametric behavior is hard-wired into their ML side: polymorphic instantiation unconditionally performs a form of dynamic sealing to protect against the non-parametric Scheme side. (In contrast, we treat new as its own language construct, orthogonal to universal types.) Dynamic sealing can then be defined in terms of the primitive coercion operators that bridge between the ML and Scheme sides. These coercions are similar to our (meta-level) wrapping operators, but ours perform type-level sealing, not term-level sealing.

The logical relations in Matthews and Ahmed's formalism are somewhat reminiscent of E° and E, although theirs are distinct logical relations for two languages, while ours are for a single language and differ only in the definition of $T[\![\Omega]\!]w$. In order to prove the fundamental property for their relations, they prove a "bridge lemma" transferring relatedness in one language to the other via coercions. This is analogous to our Wrapping Theorem for $\precsim^{\circ}$, but the latter is an independent theorem, not a lemma. Also, they do not propose anything like our polarized logical relations.

A key technical difference is that their formulation of the logical relations does not use possible worlds to capture the type store (the latter is left implicit in their operational semantics). Unfortunately, this resulted in a significant flaw in their paper [4]. They have since reportedly fixed the problem—independently of our work—using a technique similar to ours, but they have yet to write up the details.

Proof Methods. Logical relations in various forms are routinely used to reason about program equivalence and type abstraction [20, 14, 17, 3]. In particular, Ahmed, Dreyer and Rossberg recently applied step-indexed logical relations with possible worlds to reason about type abstraction for a language with higher-order state [6]. State in G is comparatively benign, but still requires a circular definition of worlds that we stratify using steps.

Pitts and Stark used logical relations to reason about program equivalence in a language with (term-level) name generation [18] and subsequently generalized their technique to handle mutable references [19]. Sumii and Pierce use them for proving secrecy results for a language with dynamic sealing [26], where generated names are used as keys. Their logical relation uses a form of possible world very similar to ours, but tying relational interpretations to term-level private keys instead of to type names. Their worlds come into play in the interpretation of the type `bits` of encrypted data, whereas in our setup the worlds are important in the interpretation of universal and existential types. In another line of work, Sumii and Pierce have used *bisimulations* to establish abstraction results for both untyped and polymorphic languages [27, 28]. However, none of the languages they investigate mixes the two paradigms.

Grossman, Morrisett and Zdancewic have proposed the use of *abstraction brackets* for syntactically tracing abstraction boundaries [10] during program execution. However, this is a comparatively weak method that does not seem to help in proving parametricity or representation independence results.

12. Conclusion and Future Work

In traditional static languages, type abstraction is established by parametric polymorphism. This approach no longer works when dynamic typing features like casts, `typecase`, or reflection are added to the mix. Dynamic type generation addresses this problem.

In this paper, we have shown that dynamic type generation succeeds in recovering type abstraction. More specifically: (1) we presented a step-indexed logical relation for reasoning about program equivalence in a non-parametric language with `cast` and type generation; (2) we showed that parametricity can be re-established systematically using a simple type-directed wrapping, which then can be reasoned about using a parametric variant of the logical relation; (3) we showed that parametricity can be refined into parametric *behavior* and parametric *usage* and gave a polarized logical relation that distinguishes these dual notions, thereby handling more subtle examples. The concept of a polarized logical relation seems novel, and it remains to be seen what else it might be useful for. Interestingly, all our logical relations can be defined as a single family differing only in the interpretation T of types-as-data.

An open question is whether the wrapping, when seen as an embedding of F^{μ} into G^{μ}, is fully abstract. We conjecture that it is, but we were only able to show equivalence reflection, not equivalence preservation. Proving full abstraction remains an interesting challenge for future work.

On the practical side, we would like to scale our logical relation to handle a more realistic language like ML. Unfortunately, wrapping cannot easily be extended to a type of mutable references. However, we believe that our approach still scales to a large class of languages, so long as we instrument it with a distinction between module and core levels. Specifically, note that wrapping only does something "interesting" for universal and existential types, and is the identity (modulo η-expansion) otherwise. Thus, for a language like Standard ML, which does not support first-class polymorphism—or Alice ML, which supports modules-as-first-class-values, but not existentials—wrapping could be confined to the module level (as part of the implementation of opaque signature ascription). For core-level types it could just be the identity. This is a real advantage of type generation over dynamic sealing since, for the latter, the need to seal/unseal individual values of abstract type precludes any attempt to confine wrapping to modules.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *JFP*, 5(1):111–130, 1995.

[2] Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[3] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

[4] Amal Ahmed. Personal communication, 2009.

[5] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ICFP*, 2008.

[6] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *POPL*, 2009.

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

[8] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. In *Computation, Meaning and Logic: Articles dedicated to Gordon Plotkin*. 2007.

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

[10] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *TOPLAS*, 22(6):1037–1080, 2000.

[11] Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 1999.

[12] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL*, 1995.

[13] Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *ESOP*, 2008.

[14] John C. Mitchell. Representation independence and data abstraction. In *POPL*, 1986.

[15] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *TOPLAS*, 10(3):470–502, 1988.

[16] Georg Neis. Non-parametric parametricity. Master's thesis, Universität des Saarlandes, 2009.

[17] Andrew Pitts. Typed operational reasoning. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7. MIT Press, 2005.

[18] Andrew Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *MFCS*, volume 711 of *LNCS*, 1993.

[19] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[20] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, 1983.

[21] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *PPDP*, 2003.

[22] Andreas Rossberg. Dynamic translucency with abstraction kinds and higher-order coercions. In *MFPS*, 2008.

[23] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice ML through the looking glass. In *TFP*, volume 5, 2004.

[24] Peter Sewell. Modules, abstract types, and distributed versioning. In *POPL*, 2001.

[25] Peter Sewell, James Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. *JFP*, 17(4&5):547–612, 2007.

[26] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. *JCS*, 11(4):521–554, 2003.

[27] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. *TCS*, 375(1–3):161–192, 2007.

[28] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *JACM*, 54(5):1–43, 2007.

[29] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *TLDI*, 2005.

[30] Philip Wadler. Theorems for free! In *FPCA*, 1989.

[31] Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information flow. In *LICS*, 2005.

[32] Stephanie Weirich. Type-safe cast. *JFP*, 14(6):681–695, 2004.