JFP **21** (4 & 5): 497–562, 2011. © Cambridge University Press 2011
doi:10.1017/S0956796811000165

Non-parametric parametricity

GEORG NEIS, DEREK DREYER and ANDREAS ROSSBERG
Max Planck Institute for Software Systems (MPI-SWS), Campus E1.4, 66123 Saarbrücken, Germany
(*e-mail:* {neis,dreyer,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 runtime 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 develop a "polarized" variant of our logical relation, which enables us to distinguish formally between positive and negative notions of parametricity.

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 (Reynolds, 1983). 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* (Harper & Morrisett, 1995). Probably the simplest and most common instance of intensional type analysis is found in the implementation of languages supporting a type Dynamic (Abadi *et al.*, 1995). 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 (Sewell *et al.*, 2007) and Alice ML (Rossberg *et al.*, 2004), 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., Weirich, 2004; Vytiniotis *et al.*, 2005).

There is a fundamental tension between type analysis and type abstraction. If one can inspect the identity of an unknown type at runtime, then the type is not really abstract, so any invariants concerning values of that type may be broken (Weirich, 2004). Consequently, languages with a type Dynamic sometimes 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 $\mathsf{t} \to \mathsf{t}$ to Dynamic, so long as we can ensure that it will subsequently be cast back only to $\mathsf{t} \to \mathsf{t}$ (not to, say, $\mathsf{int} \to \mathsf{int}$ or $\mathsf{int} \to \mathsf{t}$), i.e., so long as the cast is *abstraction-safe*. Moreover, such casts are useful when marshaling (or "pickling") a modular component whose interface refers to abstract types defined in other components (Rossberg *et al.*, 2004). 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 underlying implementation.

Thus, several researchers have proposed that languages with type analysis facilities should also support *dynamic type generation* (Sewell, 2001; Rossberg, 2003, 2008; Vytiniotis *et al.*, 2005). The idea is simple: when one defines an abstract type, one should also be able to generate at runtime a "fresh" type name, which may be used as a unique 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.

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 (Girard, 1972), and (2) a facility for dynamic generation of fresh type names. For brevity, we will call this language G. As a practical

[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, local modules, or first-class modules.

language mechanism, the cast operator is somewhat crude in comparison to the more expressive typecase-style constructs proposed in the literature, but it is nonetheless useful. For instance, the implementation of dynamic modules in Alice ML (Rossberg *et al.*, 2004) relies merely on a cast-like operator, not a typecase. Moreover, the cast operator renders polymorphism *non-parametric*, and it is one of the simplest, most canonical operators that does so, making it an ideal object for formal study. Our main technical result is that, in our language G, the parametricity of polymorphism that is lost due to the presence of cast may be provably regained via judicious use of dynamic type generation. More precisely, we show that all terms that are related by a *parametric* logical relation for G can be rendered observationally equivalent by applying a type-directed "*wrapping*" function that we can construct systematically.

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* (Appel & McAllester, 2001; Ahmed *et al.*, 2009), with an unusual form of *possible world* that is a close relative of Sumii & Pierce's (2003). 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* (rather than as *classifiers*). The definition of this type-level logical relation is a one-liner, which can easily be replaced with an alternative "parametric" version.

In Sections 5–7, 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, in Section 8, to develop a "polarized" variant of our logical relation, which enables us to distinguish formally between positive and negative notions of parametricity. Essentially, positively parametric terms expect to be *treated* parametrically (by their contexts), whereas negatively parametric terms actually *behave* parametrically themselves.

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.

In Section 11, we observe that our logical relations model is *incomplete* with respect to contextual equivalence in G, but also that there are good reasons for this. Most importantly, our model is intended to generalize to the setting of a language with typecase. Thus, while there exist programs that are equivalent in the presence of a cast operator but not in the presence of the more powerful typecase, our model does not support proofs of such equivalences. (In essence, we conjecture that our model is in fact a “better fit” for typecase than for cast; we have chosen to study cast, as explained above, because it is simpler yet still interesting.)

Finally, in Section 12, we discuss related work, including recent work on the relevant concepts of dynamic sealing (Sumii & Pierce, 2007a) and multi-language interoperation (Matthews & Ahmed, 2008), and in Section 13, 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 (Girard, 1972), including pairs and existential types. (We could instead use a Church encoding of existentials via universals, but building existentials in as primitive gives us more leeway later, cf. Section 5.) We also assume an unspecified set of base types b, along with suitable constants of—and primitive operations over—those types (indicated by ... in the syntax).

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

- $\mathsf{cast}\, \tau_1\, \tau_2\, v_1\, v_2$ 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 *isomorphic*, but not equivalent. That is, they are considered equal as *classifiers*, but not as *data*. In particular, $\mathsf{cast}\, \alpha\, \tau\, v_1\, v_2$ will not succeed in casting v_1 from α to τ—it will instead return the default value v_2.

Our cast operator is essentially the same as Harper & Mitchell’s *TypeCond* operator (Harper & Mitchell, 1999), which was itself a variant of the non-parametric J operator that Girard studied in his thesis (Girard, 1972). Our new construct is similar to previously proposed constructs for dynamic type generation (Rossberg, 2003; Vytiniotis *et al.*, 2005; Rossberg, 2008). 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 can perform this conversion *implicitly* without losing significant type information.[2]

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 functions and 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] It is not obvious whether this would still be possible if the language were enriched with features such as singleton kinds (Rossberg, 2008) or type-level computations (Weirich *et al.*, 2011).

Types	τ	$::=$	$\alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \forall\alpha.\tau \mid \exists\alpha.\tau$
Values	v	$::=$	$x \mid \ldots \mid \langle v,v\rangle \mid \lambda x{:}\tau.e \mid \lambda\alpha.e \mid \mathsf{pack}\ \langle\tau,v\rangle\ \mathsf{as}\ \tau$
Terms	e	$::=$	$v \mid \ldots \mid \langle e,e\rangle \mid e.1 \mid e.2 \mid e\ e \mid e\ \tau \mid \mathsf{pack}\ \langle\tau,e\rangle\ \mathsf{as}\ \tau \mid$ $\mathsf{unpack}\ \langle\alpha,x\rangle{=}e\ \mathsf{in}\ e \mid \mathsf{cast}\ \tau\ \tau \mid \mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ e$
Stores	σ	$::=$	$\epsilon \mid \sigma, \alpha{\approx}\tau$
Config's	ζ	$::=$	$\sigma; e$
Evaluation Ctxt's	E	$::=$	$\ldots \mid \langle E,e\rangle \mid \langle v,E\rangle \mid E.1 \mid E.2 \mid E\ e \mid v\ E \mid E\ \tau \mid$ $\mathsf{pack}\ \langle\tau,E\rangle\ \mathsf{as}\ \tau \mid \mathsf{unpack}\ \langle\alpha,x\rangle{=}E\ \mathsf{in}\ e$

Type Environments	Δ	$::=$	$\epsilon \mid \Delta,\alpha \mid \Delta,\alpha{\approx}\tau$
Value Environments	Γ	$::=$	$\epsilon \mid \Gamma, x{:}\tau$

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

$\ldots$

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

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

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

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

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

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

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

$$(\textsc{Store})\frac{\Delta = \sigma \qquad \vdash \Delta}{\vdash \sigma}$$

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

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

$\ldots$

	$\sigma; E[\langle v_1,v_2\rangle.i]$	$\hookrightarrow \sigma; E[v_i]$	(Rproj)
	$\sigma; E[(\lambda x{:}\tau.e)\,v]$	$\hookrightarrow \sigma; E[e[v/x]]$	(Rapp)
	$\sigma; E[(\lambda\alpha.e)\,\tau]$	$\hookrightarrow \sigma; E[e[\tau/\alpha]]$	(Rinst)
	$\sigma; E[\mathsf{unpack}\ \langle\alpha,x\rangle{=}(\mathsf{pack}\ \langle\tau,v\rangle)\ \mathsf{in}\ e]$	$\hookrightarrow \sigma; E[e[\tau/\alpha][v/x]]$	(Runpack)
$(\alpha \notin \mathrm{dom}(\sigma))$	$\sigma; E[\mathsf{new}\,\alpha{\approx}\tau\ \mathsf{in}\ e]$	$\hookrightarrow \sigma,\alpha{\approx}\tau; E[e]$	(Rnew)
$(\tau_1 = \tau_2)$	$\sigma; E[\mathsf{cast}\ \tau_1\ \tau_2]$	$\hookrightarrow \sigma; E[\lambda x_1{:}\tau_1.\lambda x_2{:}\tau_2.x_1]$	(Rcast1)
$(\tau_1 \neq \tau_2)$	$\sigma; E[\mathsf{cast}\ \tau_1\ \tau_2]$	$\hookrightarrow \sigma; E[\lambda x_1{:}\tau_1.\lambda x_2{:}\tau_2.x_2]$	(Rcast2)

Fig. 1. Syntax and semantics of G (excerpt).

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 are given in Appendix A.

Typing of casts is straightforward (Rule ECast): 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 environments 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.) We call such environment entries *type isomorphism assumptions*.

Syntactically, type "names" are just type variables in the calculus (and like other type variables, they are α-convertible). As a matter of terminology, however, we refer as type names only to those type variables α that are bound with the syntax "$\alpha{\approx}\tau$" (that is, either by new, in a store σ, or with a respective entry in a type environment Δ).

When viewed as data (i.e., when inspected by the cast operator), types are considered equivalent iff they are syntactically equal (modulo α-conversion). 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 *isomorphic* to τ'. Type isomorphism, in turn, is defined by the judgment $\Delta \vdash \tau_1 \approx \tau_2$. We only show the rule CName, which discharges an isomorphism assumption $\alpha{\approx}\tau$ from the environment; the other rules implement the congruence closure of this axiom. The important point here is that equivalent types are isomorphic, but isomorphic 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 $\tau'[\tau/\alpha]$.

Note that the typing rules ensure that type environments are ordered and acyclic. Consequently, any type $\Delta \vdash \tau$ can be normalized to a type τ' that does not contain any type names and is isomorphic to τ, i.e., $\Delta' \vdash \tau'$ and $\Delta \vdash \tau \approx \tau'$, where Δ' is Δ without all the isomorphism assumptions. This normalization is done using the substitution Δ^* that is obtained from Δ in the following way:

$$\begin{aligned} \epsilon^* &\stackrel{\text{def}}{=} \emptyset \\ (\Delta, \alpha)^* &\stackrel{\text{def}}{=} \Delta^* \\ (\Delta, \alpha{\approx}\tau)^* &\stackrel{\text{def}}{=} \Delta^*, \alpha \mapsto \Delta^*(\tau) \end{aligned}$$

Given this normalization, it easy to see that type checking is decidable.

2.2 Dynamic semantics

The operational semantics has to deal with the 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.

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).[3] Note that this rule is the single source of non-determinism in our operational semantics.

The two remaining rules are for casts. A cast takes two types and checks whether or not 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.

Type preservation can be expressed using the typing rule CONF for configurations. We formulate this rule by treating the type store as a type environment, which is possible because type stores are a syntactic subclass of type environments. (In a similar manner, we can write $\vdash \sigma$ for well-formedness of store σ, by viewing it as a type environment.) It is worth noting that the representation types in the store are never actually inspected by the dynamic semantics. In particular, 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 (Pitts, 2005) in G. Following Mitchell & Plotkin (1988), 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 is essentially 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

[3] A well-known alternative approach would omit the type store in favor of using scope extrusion rules for new binders, as in Rossberg (2003).

those returned by the module (nor could they apply integer operations to values of type α).

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

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

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 α' as witness of the abstract type:

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

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 use type conversion to convert terms 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 result, namely, *representation independence* (Reynolds, 1983): we show that the module e_{sem1} is *contextually equivalent* to another module of the same type that implements the abstract type in a different way. Contextual equivalence means 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:

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

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 (see Section A.4). Typing of contexts is defined by a judgment form $\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau')$, 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:

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

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

Definition 1 (*Contextual Approximation and Equivalence*)
Let $\Delta;\Gamma \vdash e_1 : \tau$ and $\Delta;\Gamma \vdash e_2 : \tau$.

$$\begin{array}{lcl} \Delta;\Gamma \vdash e_1 \leqslant e_2 : \tau & \overset{\text{def}}{\Leftrightarrow} & \forall C,\tau',\sigma.\ \vdash \sigma \wedge \vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\sigma;\epsilon;\tau') \\ & & \qquad\qquad\qquad \wedge\ \sigma;C[e_1]\downarrow \Rightarrow \sigma;C[e_2]\downarrow \\ \Delta;\Gamma \vdash e_1 \equiv e_2 : \tau & \overset{\text{def}}{\Leftrightarrow} & \Delta;\Gamma \vdash e_1 \leqslant e_2 : \tau\ \wedge\ \Delta;\Gamma \vdash e_2 \leqslant e_1 : \tau \end{array}$$

That is, contextual approximation $\Delta;\Gamma \vdash e_1 \leqslant e_2 : \tau$ 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 $\Delta;\Gamma \vdash e_1 \equiv e_2 : \tau$ 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 (Harper & Mitchell, 1999). (This was Girard's reason for studying the J operator in the first place (Girard, 1972).) Moreover, using cast, one can encode arbitrary recursive function definitions (see Appendix A.5 for details). Other forms of observation may then be encoded in terms of (non-)termination.

3 A logical relation for G: Main ideas

Following Reynolds (1983) and Mitchell (1986), 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 $\tau_1 \rightarrow \tau_2$ 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 (2005) calls *adequacy*—i.e., the fact (built into the definition of the logical relation) 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 (Girard, 1972). 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 $\mathsf{pack}\ \langle\tau_1, v_1\rangle\ \mathsf{as}\ \exists\alpha.\tau$ and $\mathsf{pack}\ \langle\tau_2, v_2\rangle\ \mathsf{as}\ \exists\alpha.\tau$ 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, if not the structure of the type, 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 (2001) 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 (Ahmed, 2006; Ahmed *et al.*, 2009), 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*. (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 $n \leqslant m$.) 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 (Pitts & Stark, 1993).

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'.

We now must show that the values

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

and

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

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 non-determinism 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 one-to-one correspondence

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

$$
\begin{array}{rl}
\lfloor(\sigma_1,\sigma_2,\eta,\rho)\rfloor_n & \stackrel{\text{def}}{=} (\sigma_1,\sigma_2,\eta,\lfloor\rho\rfloor_n) \\
\lfloor\rho\rfloor_n & \stackrel{\text{def}}{=} \{\alpha \mapsto \lfloor r\rfloor_n \mid \rho(\alpha) = r\} \\
\lfloor(\tau_1,\tau_2,R)\rfloor_n & \stackrel{\text{def}}{=} (\tau_1,\tau_2,\lfloor R\rfloor_n) \\
\lfloor R\rfloor_n & \stackrel{\text{def}}{=} \{(k,w,e_1,e_2) \in R \mid k < n\}
\end{array}
$$

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

$$
\triangleright R \stackrel{\text{def}}{=} \{(k,w,e_1,e_2) \mid \forall (k',w') \sqsupset (k,w).\ (k',w',e_1,e_2) \in R\}
$$

Fig. 2. Worlds and auxiliary definitions.

because the cast construct permits the program context to observe equality on type names, as follows:

$$
\begin{array}{l}
\mathsf{equal?} : \forall\alpha.\forall\beta.\mathsf{bool} \stackrel{\text{def}}{=} \\
\quad \Lambda\alpha.\Lambda\beta.\ \mathsf{cast}\ ((\alpha \to \alpha) \to \mathsf{bool})\ ((\beta \to \beta) \to \mathsf{bool}) \\
\qquad\qquad (\lambda x{:}(\alpha \to \alpha).\mathsf{true})(\lambda x{:}(\beta \to \beta).\mathsf{false})(\lambda x{:}\beta.x)
\end{array}
$$

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

We now formalize our logical relation for G. For technical reasons related to step-indexing, we do not define it directly in terms of equivalent termination behavior. Instead, we define it in terms of approximated termination behavior such that if e_1 and e_2 are logically related, then e_1 contextually approximates e_2 (i.e., $C[e_2]$ terminates whenever $C[e_1]$ does). Logical equivalence then is just logical approximation in both directions.

Figures 2 and 3 display 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 $\text{Atom}[\tau_1, \tau_2]$ 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 ith type store component of w, and analogous notation for projecting out the other components of worlds.)

$\text{Rel}[\tau_1, \tau_2]$ 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. (In fact, technically speaking, we consider $\text{dom}(\eta) = \text{dom}(\rho)$ to be bound variables of the world w.) 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 $\alpha \in \text{dom}(\eta) = \text{dom}(\rho)$, 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.

$$
\begin{array}{lll}
V_n\llbracket\alpha\rrbracket\rho & \overset{\text{def}}{=} & \lfloor\rho(\alpha).R\rfloor_n \\
V_n\llbracket b\rrbracket\rho & \overset{\text{def}}{=} & \{(k,w,v,v)\in \mathrm{Atom}_n[b,b]\} \\
V_n\llbracket\tau\times\tau'\rrbracket\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\llbracket\tau\rrbracket\rho\wedge(k,w,v_1',v_2')\in V_n\llbracket\tau'\rrbracket\rho\} \\
V_n\llbracket\tau'\to\tau\rrbracket\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\llbracket\tau'\rrbracket\rho.\ (k',w')\sqsupseteq(k,w)\Rightarrow \\
 & & \quad (k',w',e_1[v_1/x],e_2[v_2/x])\in E_n\llbracket\tau\rrbracket\rho\} \\
V_n\llbracket\forall\alpha.\tau\rrbracket\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'}\llbracket\Omega\rrbracket w'. \\
 & & \quad (k',w',e_1[\tau_1/\alpha],e_2[\tau_2/\alpha])\in\triangleright E_n\llbracket\tau\rrbracket\rho,\alpha\mapsto r\} \\
V_n\llbracket\exists\alpha.\tau\rrbracket\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\llbracket\Omega\rrbracket w\wedge(k,w,v_1,v_2)\in\triangleright V_n\llbracket\tau\rrbracket\rho,\alpha\mapsto r\} \\
E_n\llbracket\tau\rrbracket\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\exists w',v_2.\,(k-j,w')\sqsupseteq(k,w)\wedge \\
 & & \quad 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\llbracket\tau\rrbracket\rho\} \\
 & & \\
T_n\llbracket\Omega\rrbracket w & \overset{\text{def}}{=} & \{(w.\eta^1(\tau),w.\eta^2(\tau),(w.\rho^1(\tau),w.\rho^2(\tau),V_n\llbracket\tau\rrbracket w.\rho))\mid\ \mathrm{fv}(\tau)\subseteq\mathrm{dom}(w.\rho)\} \\
 & & \\
G_n\llbracket\epsilon\rrbracket\rho & \overset{\text{def}}{=} & \{(k,w,\emptyset,\emptyset)\mid k<n\wedge w\in\mathrm{World}_k\} \\
G_n\llbracket\Gamma,x{:}\tau\rrbracket\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\llbracket\Gamma\rrbracket\rho\wedge(k,w,v_1,v_2)\in V_n\llbracket\tau\rrbracket\rho\} \\
D_n\llbracket\epsilon\rrbracket w & \overset{\text{def}}{=} & \{(\emptyset,\emptyset,\emptyset)\} \\
D_n\llbracket\Delta,\alpha\rrbracket 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\llbracket\Delta\rrbracket w\wedge(\tau_1,\tau_2,r)\in T_n\llbracket\Omega\rrbracket w\} \\
D_n\llbracket\Delta,\alpha{\approx}\tau\rrbracket 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\llbracket\Delta\rrbracket w\wedge \\
 & & \quad \exists\alpha'.\,w.\rho(\alpha')=r\wedge w.\eta(\alpha')=(\beta_1,\beta_2)\wedge \\
 & & \quad w.\sigma_1(\beta_1)=\delta_1(\tau)\wedge w.\sigma_2(\beta_2)=\delta_2(\tau)\wedge r.R=V_n\llbracket\tau\rrbracket\rho\}
\end{array}
$$

$$
\begin{array}{lll}
\Delta;\Gamma\vdash e_1\lesssim e_2:\tau & \overset{\text{def}}{\Leftrightarrow} & \Delta;\Gamma\vdash e_1:\tau\wedge\Delta;\Gamma\vdash e_2:\tau\wedge \\
 & & \forall n\geqslant 0.\ \forall w_0\in\mathrm{World}_n.\ \forall(\delta_1,\delta_2,\rho)\in D_n\llbracket\Delta\rrbracket w_0.\ \forall(k,w,\gamma_1,\gamma_2)\in G_n\llbracket\Gamma\rrbracket\rho. \\
 & & (k,w)\sqsupseteq(n,w_0)\Rightarrow(k,w,\delta_1\gamma_1(e_1),\delta_2\gamma_2(e_2))\in E_n\llbracket\tau\rrbracket\rho
\end{array}
$$

Fig. 3. Logical relation for G.

(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 isomorphic. 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. Condition (3) here corresponds to the common practice in Kripke logical relations proofs, whereby one extends a given "input" world to a future "output" world only when one wants to establish some invariants about freshly allocated entities (in the case of G, fresh type names). Although this condition is not strictly necessary for establishing soundness of the logical relation, it has not in our experience made it more difficult to prove anything. Moreover, we have found it to be useful when proving certain examples (e.g., the "order independence" example in Section 4.4) because it cuts down on the set of worlds one must consider when one universally quantifies over a future world.

Figure 3 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 $\rightarrow$ 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 (respectively, components) of a universal (respectively, 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 one-line definition of $T[\![\Omega]\!]w$, which may then easily be replaced with a more traditional parametric one.

The term relation $E[\![\tau]\!]\rho$ is very similar to that in previous step-indexed Kripke logical relations (Ahmed *et al.*, 2009). 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 3 serves 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 dom(Γ) to related values. $D[\![\Delta]\!]w$ is the logical relation on type substitutions. It asserts that related δ's must map variables in dom(Δ) 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 (2004), as well as recent work by Ahmed *et al.* (2009).

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 that the definition of $\triangleright R$ in the second section of Figure 2). Consequently, 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 *crucial* 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 Interesting properties of the logical relation

The main result concerning our logical relation is, of course, that it provides a sound technique for proving contextual equivalence of G programs. We now present the technical development necessary to establish this result. For convenience, we often omit the step annotation on the restriction operator when it is obvious from context, e.g., we will write $(k-j-1, \lfloor w \rfloor)$ instead of $(k-j-1, \lfloor w \rfloor_{k-j-1})$. Furthermore, at many places, we are required to establish the well-typedness conditions imposed by the definition of $\mathrm{Atom}[\tau_1, \tau_2]$, but since this is completely straightforward and usually tedious, we will omit this part of the proofs. If the reader is interested in seeing how the syntactic typing conditions are maintained, we would refer them to the first author's master's thesis, which shows the full gory details in two representative cases (namely, the proofs of Lemma 10.21 and Theorem 10.41).

4.3.1 Basic lemmas

We start with a few very basic lemmas that are needed ubiquitously in subsequent proofs (to the extent that we will usually not even apply them explicitly).

Lemma 1 (*Transitivity of World Extension*)
1. If $(k'', w'') \sqsupseteq (k', w')$ and $(k', w') \sqsupseteq (k, w)$, then $(k'', w'') \sqsupseteq (k, w)$.
2. If $(k'', w'') \sqsupset (k', w')$ and $(k', w') \sqsupset (k, w)$, then $(k'', w'') \sqsupset (k, w)$.

Lemma 2 (*Restriction*)
1. If $k' \leqslant k$, then $V_{k'}[\![\tau]\!]\rho = \lfloor V_k[\![\tau]\!]\rho \rfloor_{k'}$.
2. If $k' \leqslant k$, then $E_{k'}[\![\tau]\!]\rho = \lfloor E_k[\![\tau]\!]\rho \rfloor_{k'}$.

Irrelevance (Lemma 3) states that the logical relation only depends on ρ's interpretation of those variables that actually occur in τ.

Lemma 3 (*Irrelevance*)
If $\lfloor \rho' \rfloor_n \sqsupseteq \lfloor \rho \rfloor_n$ and $\mathrm{ftv}(\tau) \subseteq \mathrm{dom}(\rho)$, then

1. $V_n[\![\tau]\!]\rho' = V_n[\![\tau]\!]\rho$,
2. $E_n[\![\tau]\!]\rho' = E_n[\![\tau]\!]\rho$, and
3. $G_n[\![\tau]\!]\rho' = G_n[\![\tau]\!]\rho$.

The next lemma is a combination of the previous two, but for the type and type substitution relations.

Lemma 4
1. If $(\tau_1, \tau_2, r) \in T_n[\![\Omega]\!]w_0$ and $(k, w) \sqsupseteq (n, w_0)$, then $(\tau_1, \tau_2, \lfloor r \rfloor_k) \in T_k[\![\Omega]\!]w$.
2. If $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$ and $(k, w) \sqsupseteq (n, w_0)$, then $(\delta_1, \delta_2, \lfloor \rho \rfloor_k) \in D_k[\![\Delta]\!]w$.

Finally, Inclusion tells us that in order to show two values related in the term relation, it suffices to show them related in the value relation.

Lemma 5 (*Inclusion*)
$V_n[\![\tau]\!]\rho \subseteq E_n[\![\tau]\!]\rho$

Proof
Follows easily from the definition of $E_n[\![\tau]\!]\rho$, by choosing the final world w' to be the same as the initial world w. □

4.3.2 Validity

The first important property to show is that, under the assumption that ρ is a valid relational interpretation of the free variables of τ (i.e., $\rho \in \text{Interp}$ and $\text{ftv}(\tau) \subseteq \text{dom}(\rho)$), the logical relation for values $V_n[\![\tau]\!]\rho$ is itself a valid relation (i.e., an element of Rel).

For the sake of convenience, whenever we write $V_n[\![\tau]\!]\rho$, $E_n[\![\tau]\!]\rho$, $G_n[\![\Gamma]\!]\rho$, $D_n[\![\Delta]\!]w$, and $T_n[\![\Omega]\!]w$ from now on, we assume $\rho \in \text{Interp}$, $w \in \text{World}$, and $\text{ftv}(\tau) \subseteq \text{dom}(\rho)$.

As a first step, we note that every element of the value and term relations is a proper atom.

Lemma 6 (*Atomicity*)

1. $V_n[\![\tau]\!]\rho \subseteq \text{Atom}_n^{\text{val}}[\rho^1(\tau), \rho^2(\tau)]$
2. $E_n[\![\tau]\!]\rho \subseteq \text{Atom}_n[\rho^1(\tau), \rho^2(\tau)]$

The key property of Rel is that its elements must be closed under world extension. Proving this for the value relation is very easy because the property has mostly been built into its definition.

Lemma 7 (*Closure Under World Extension*)

1. If $(k, w, v_1, v_2) \in V_n[\![\tau]\!]\rho$ and $(k', w') \sqsupseteq (k, w)$, then $(k', w', v_1, v_2) \in V_n[\![\tau]\!]\rho$.
2. If $(k, w, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$ and $(k', w') \sqsupseteq (k, w)$, then $(k', w', \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$.

Lemma 8 (*LR-Validity*)
$V_n[\![\tau]\!]\rho \in \text{Rel}_n[\rho^1(\tau), \rho^2(\tau)]$

Proof
Follows from Atomicity and Closure Under World Extension. □

4.3.3 Compatibility

The basic building blocks for proving soundness of our logical relation are what Pitts calls *compatibility* lemmas (Pitts, 2005), which state that the logical relation is closed under the constructs of the language.

We first have three properties about syntactic type substitutions, which will be needed for proving well-formedness of different syntactic elements. Although (as mentioned earlier) we will be omitting proofs of syntactic-typing side conditions in the present paper, we include these lemmas here as they help to clarify the subtle relationship between the various substitutions inhabiting $D_n[\![\Delta]\!]w$ and $G_n[\![\Gamma]\!]\rho$.

Lemma 9
If $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w$, then $\rho^i = w.\sigma_i^* \circ \delta_i$ and $w.\sigma_i \vdash \delta_i : \Delta$ and $\epsilon \vdash \rho^i : \Delta$.

Lemma 10
If $(k, w, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$, then $w.\sigma_i; \epsilon \vdash \gamma_i : \rho^i(\Gamma)$.

The following is a standard type substitution lemma for logical relations. It is mainly needed in showing the compatibility lemmas for quantified types.

Lemma 11 (*LR-Substitution*)
1. $V_n[\![\tau]\!]\rho, \alpha \mapsto (\rho^1(\tau'), \rho^2(\tau'), V_n[\![\tau']\!]\rho) = V_n[\![\tau[\tau'/\alpha]]\!]\rho$.
2. $E_n[\![\tau]\!]\rho, \alpha \mapsto (\rho^1(\tau'), \rho^2(\tau'), V_n[\![\tau']\!]\rho) = E_n[\![\tau[\tau'/\alpha]]\!]\rho$.

The following two lemmas are needed for dealing with the particularities of the non-parametric logical relation. We know by the definition of T and D that for any $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$ and any α bound in Δ there is some τ_α such that $\delta_1(\alpha)$ and $\delta_2(\alpha)$ are the concretizations of τ_α with respect to w_0, i.e., $\delta_1(\alpha) = w_0.\eta^1(\tau)$ and $\delta_2(\alpha) = w_0.\eta^2(\tau)$. We define an operation, au, that yields the substitution δ mapping each α to its corresponding τ_α (see Lemma 12):

Definition 2 (*Anti-Unifier*)
Assume that $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w$. The anti-unifying substitution of δ_1 and δ_2 with respect to $w.\eta$, written $\mathrm{au}(\delta_1, \delta_2, w.\eta)$, is defined as follows.

$$\begin{aligned} \mathrm{au}(\epsilon, \epsilon, \eta) &\stackrel{\text{def}}{=} \epsilon \\ \mathrm{au}((\delta_1, \alpha \mapsto \tau_1), (\delta_2, \alpha \mapsto \tau_2), \eta) &\stackrel{\text{def}}{=} \mathrm{au}(\delta_1, \delta_2, \eta), \alpha \mapsto \tau \text{ where } \tau = \eta^{-1}(\tau_1) = \eta^{-2}(\tau_2) \end{aligned}$$

Here, η^{-i} is short for $(\eta^i)^{-1}$, the inverse of η^i. The latter exists because the definition of Conc ensures that η^i is injective. Furthermore, since η is a partial bijection on the generated type names, $\eta^{-1}(\tau_1)$ and $\eta^{-2}(\tau_2)$ are guaranteed to be equal.

Lemma 12
1. If $\delta = \mathrm{au}(\delta_1, \delta_2, \eta)$, then $\delta_1 = \eta^1 \circ \delta$ and $\delta_2 = \eta^2 \circ \delta$.
2. If $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$ and $\delta = \mathrm{au}(\delta_1, \delta_2, w_0.\eta)$ and $(k, w) \sqsupseteq (n, w_0)$, then $\delta = \mathrm{au}(\delta_1, \delta_2, w.\eta)$.

Proof
1. Follows easily from the definition.
2. First, note that $(\delta_1, \delta_2, \lfloor\rho\rfloor_k) \in D_k[\![\Delta]\!]w$ by Lemma 4. Furthermore, since $w.\eta$ is an extension of $w_0.\eta$, the former agrees with the latter on $\mathrm{dom}(w_0.\eta)$. As we know $\mathrm{ftv}(\delta_i(\alpha)) \subseteq \mathrm{rng}(w_0.\eta)$ for any α, it is clear that $\mathrm{au}(\delta_1, \delta_2, w_0.\eta) = \mathrm{au}(\delta_1, \delta_2, w.\eta)$. □

The motivation for defining au is the following property, which is crucial for proving compatibility of $\precsim$ for the rules Einst, Epack, and Ecast, in which its non-parametricity becomes manifest. The property essentially combines LR-Substitution (Lemma 11) with the observation that, when $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$, it means that ρ is actually highly constrained. Specifically, $\lfloor\rho(\alpha).r\rfloor_n$ must be the logical relation $V_n[\![\delta(\alpha)]\!]w_0.\rho$, where δ is the anti-unifier of δ_1 and δ_2.

Lemma 13
If $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$ and $\delta = \mathrm{au}(\delta_1, \delta_2, w_0.\eta)$ and $\Delta \vdash \tau$, then:

1. $V_n[\![\tau]\!]\rho = V_n[\![\delta(\tau)]\!]w_0.\rho$
2. $E_n[\![\tau]\!]\rho = E_n[\![\delta(\tau)]\!]w_0.\rho$

Proof
By primary induction on n and secondary induction on the derivation of $\Delta \vdash \tau$, we show the interesting cases in Appendix B. □

Many of the compatibility proofs are straightforward—they do not deal with worlds in any interesting way, and the non-parametricity does not show up because it is hidden in $T[\![\Omega]\!]$. Those proofs are thus essentially analogous to their counterparts in a parametric System F-like setting (Ahmed, 2006) and we only show one example (EUNPACK) here. The only proofs that actually involve interesting reasoning about worlds are for EINST and EPACK. We show the latter; the former is similar (and dual).

Lemma 14 (*Compatibility:* EPACK)
If $\Delta;\Gamma \vdash e_1 \precsim e_2 : \tau[\tau'/\alpha]$ and $\Delta \vdash \tau'$, then $\Delta;\Gamma \vdash \mathsf{pack}\ \langle\tau', e_1\rangle \precsim \mathsf{pack}\ \langle\tau', e_2\rangle : \exists\alpha.\tau$.

Proof

- Suppose $w_0 \in \mathrm{World}_n$, $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$, $(k, w, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$, and $(k, w) \sqsupseteq (n, w_0)$.
- To show: $(k, w, \delta_1\gamma_1(\mathsf{pack}\ \langle\tau', e_1\rangle), \delta_2\gamma_2(\mathsf{pack}\ \langle\tau', e_2\rangle)) \in E_n[\![\exists\alpha.\tau]\!]\rho$
- Assume that $w.\sigma_1; \delta_1\gamma_1(\mathsf{pack}\ \langle\tau', e_1\rangle) \hookrightarrow^j \sigma_1; \mathsf{pack}\ \langle\delta_1(\tau'), v_1\rangle$ where $j < k$.
- Instantiating the premise yields $(k, w, \delta_1\gamma_1(e_1), \delta_2\gamma_2(e_2)) \in E_n[\![\tau[\tau'/\alpha]]\!]\rho$.
- Consequently, there exists $(k-j, w') \sqsupseteq (k, w)$ such that

$$w.\sigma_2; \delta_2\gamma_2(\mathsf{pack}\ \langle\tau', e_2\rangle) \hookrightarrow^* w'.\sigma_2; \mathsf{pack}\ \langle\delta_2(\tau'), v_2\rangle$$

  with $w'.\sigma_1 = \sigma_1$ and $(k-j, w', v_1, v_2) \in V_n[\![\tau[\tau'/\alpha]]\!]\rho$.
- It remains to show $(k-j, w', \mathsf{pack}\ \langle\delta_1(\tau'), v_1\rangle, \mathsf{pack}\ \langle\delta_2(\tau'), v_2\rangle) \in V_n[\![\exists\alpha.\tau]\!]\rho$.
- Let $r := (w'.\sigma_1^*(\delta_1(\tau')), w'.\sigma_2^*(\delta_2(\tau')), V_{k-j}[\![\tau']\!]\rho)$.
- We now have to show that this witness relation actually has the shape required by the definition of $T[\![\Omega]\!]$, i.e., that $(\delta_1(\tau'), \delta_2(\tau'), r) \in T_{k-j}[\![\Omega]\!]w'$:
  - — Let $\delta := \mathrm{au}(\delta_1, \delta_2, w_0.\eta)$.
  - — It suffices to show $(\delta_1(\tau'), \delta_2(\tau'), r) = (w'.\eta^1\delta(\tau'), w'.\eta^2\delta(\tau'),$ $(w'.\rho^1\delta(\tau'), w'.\rho^2\delta(\tau'), V_{k-j}[\![\delta(\tau')]\!]w'.\rho))$.
  - — By Lemma 4, $(\delta_1, \delta_2, \lfloor\rho\rfloor) \in D_{k-j}[\![\Delta]\!]w'$.
  - — First, $\delta_i(\tau') = w'.\eta^i\delta(\tau')$ by Lemma 12.
  - — Second, $w'.\sigma_i^*(\delta_i(\tau')) = w'.\sigma_i^*(w'.\eta^i\delta(\tau')) = w'.\rho^i\delta(\tau')$ because $w' \in \mathrm{World}$.
  - — Finally, $V_{k-j}[\![\tau']\!]\rho = V_{k-j}[\![\delta(\tau')]\!]w'.\rho$ by Lemma 13.
- It thus suffices to show that $(k'', w'', v_1, v_2) \in V_n[\![\tau]\!]\rho, \alpha \mapsto r$ for any $(k'', w'') \sqsupseteq (k-j, w')$, which follows by Closure Under World Extension and LR-Substitution from $(k-j, w', v_1, v_2) \in V_n[\![\tau[\tau'/\alpha]]\!]\rho$. □

Lemma 15 (*Compatibility:* EUNPACK)
If $\Delta;\Gamma \vdash e_1 \precsim e_2 : \exists\alpha.\tau'$ and $\Delta, \alpha; \Gamma, x{:}\tau' \vdash e_3 \precsim e_4 : \tau$ with $\Delta \vdash \tau$,
then $\Delta;\Gamma \vdash \mathsf{unpack}\langle\alpha, x\rangle{=}e_1\ \mathsf{in}\ e_3 \precsim \mathsf{unpack}\langle\alpha, x\rangle{=}e_2\ \mathsf{in}\ e_4 : \tau$.

Proof

- Suppose $w_0 \in \mathrm{World}_n$, $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$, $(k, w, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$, and $(k, w) \sqsupset (n, w_0)$.
- Show: $(k, w, \delta_1\gamma_1(\mathsf{unpack}\ \langle\alpha, x\rangle{=}e_1\ \mathsf{in}\ e_3), \delta_2\gamma_2(\mathsf{unpack}\ \langle\alpha, x\rangle{=}e_2\ \mathsf{in}\ e_4))$ in $E_n[\![\tau]\!]\rho$
- Assume that $w.\sigma_1; \delta_1\gamma_1(\mathsf{unpack}\ \langle\alpha, x\rangle{=}e_1\ \mathsf{in}\ e_3)$ terminates:

$$\begin{array}{ll} & w.\sigma_1; \delta_1\gamma_1(\mathsf{unpack}\ \langle\alpha, x\rangle{=}e_1\ \mathsf{in}\ e_3) \\ \hookrightarrow^{j_1} & \sigma_1'; \mathsf{unpack}\ \langle\alpha, x\rangle{=}(\mathsf{pack}\ \langle\tau_1, v_1\rangle)\ \mathsf{in}\ \delta_1\gamma_1(e_3) \\ \hookrightarrow^{1} & \sigma_1'; \delta_1\gamma_1(e_3)[\tau_1/\alpha][v_1/x] \\ \hookrightarrow^{j_2} & \sigma_1; v_3 \end{array}$$

  and that $j_1 + 1 + j_2 =: j < k$.
- Instantiating the first premise yields the existence of $(k - j_1, w') \sqsupseteq (k, w)$ such that

$$\begin{array}{ll} & w.\sigma_2; \delta_2\gamma_2(\mathsf{unpack}\ \langle\alpha, x\rangle{=}e_2\ \mathsf{in}\ e_4) \\ \hookrightarrow^{*} & w'.\sigma_2; \mathsf{unpack}\ \langle\alpha, x\rangle{=}(\mathsf{pack}\ \langle\tau_2, v_2\rangle)\ \mathsf{in}\ \delta_2\gamma_2(e_4) \end{array}$$

  with $w'.\sigma_1 = \sigma_1'$ and $(k - j_1, w', \mathsf{pack}\ \langle\tau_1, v_1\rangle, \mathsf{pack}\ \langle\tau_2, v_2\rangle) \in V_n[\![\exists\alpha.\tau']\!]\rho$.
- Hence, there is r such that $(\tau_1, \tau_2, r) \in T_{k-j_1}[\![\Omega]\!]w'$ and $(k - j_1 - 1, \lfloor w' \rfloor, v_1, v_2) \in V_n[\![\tau']\!]\rho, \alpha \mapsto r$.
- By Lemma 4, $(\delta_1, \delta_2, \lfloor\rho\rfloor_{k-j_1}) \in D_{k-j_1}[\![\Delta]\!]w'$.
- Let $(\delta_1', \delta_2', \rho') := ((\delta_1, \alpha \mapsto \tau_1), ((\delta_2, \alpha \mapsto \tau_2), (\lfloor\rho\rfloor_{k-j_1}, \alpha \mapsto r)))$, hence $(\delta_1', \delta_2', \rho') \in D_{k-j_1}[\![\Delta, \alpha]\!]w'$.
- By Closure Under World Extension, we know $(k - j_1 - 1, \lfloor w' \rfloor, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$ and thus $(k - j_1 - 1, \lfloor w' \rfloor, \gamma_1, \gamma_2) \in G_{k-j_1}[\![\Gamma]\!]\rho'$.
- Let $\gamma_i' := \gamma_i, x \mapsto v_i$, so we get $(k - j_1 - 1, \lfloor w' \rfloor, \gamma_1', \gamma_2') \in G_{k-j_1}[\![\Gamma, x{:}\tau'']\!]\rho'$.
- Instantiating the second premise with $w' \in \mathrm{World}_{k-j_1}$, $(\delta_1', \delta_2', \rho') \in D_{k-j_1}[\![\Delta, \alpha]\!]w'$ and $(k - j_1 - 1, \lfloor w' \rfloor, \gamma_1', \gamma_2') \in G_{k-j_1}[\![\Gamma, x{:}\tau'']\!]\rho'$ now yields $(k - j_1 - 1, \lfloor w' \rfloor, \delta_1'\gamma_1'(e_3), \delta_2'\gamma_2'(e_4)) \in E_{k-j_1}[\![\tau]\!]\rho'$.
- Note that

$$\begin{array}{lr} \delta_i'\gamma_i'(e_{i+2}) & \\ = \delta_i(\gamma_i(e_{i+2})[v_i/x])[\tau_i/\alpha]) & \\ = \delta_i\gamma_i(e_{i+2})[v_i/x][\tau_i/\alpha]) \quad \text{since } \vdash w'.\sigma_i; v_i : (\rho, \alpha \mapsto V_{k-j_1}[\![\tau'']\!]w'.\rho)^i(\tau') & \\ = \delta_i\gamma_i(e_{i+2})[\tau_i/\alpha][v_i/x] & \text{ditto} \end{array}$$

- Therefore, $\sigma_1'; \delta_1\gamma_1(e_3)[\tau_1/\alpha][v_1/x] \hookrightarrow^{j_2} \sigma_1; v_3$ implies the existence of $(k - j, w'') \sqsupseteq (k - j_1 - 1, \lfloor w' \rfloor)$ such that

$$w'.\sigma_2; \delta_2\gamma_2(e_4)[\tau_2/\alpha][v_2/x] \hookrightarrow^* w''\sigma_2; v_4$$

  with $w''.\sigma_1 = \sigma_1$ and $(k - j, w'', v_3, v_4) \in V_{k-j_1}[\![\tau]\!]\rho'$.
- Since $\Delta \vdash \tau$, the latter implies $(k - j, w'', v_3, v_4) \in V_n[\![\tau]\!]\rho$. □

In the proof of compatibility for cast, we first have to argue that the argument types on the left-hand side, $\delta_1(\tau_1)$ and $\delta_1(\tau_2)$, are equal if and only if the argument types on the right-hand side, $\delta_2(\tau_1)$ and $\delta_2(\tau_2)$, are so that we have the same reduction on both sides. This is easy to see with the help of Lemma 12, which tells us that $\delta_i = w_0.\eta^i \circ \delta$ (where δ is the anti-unifying substitution of δ_1 and δ_2)—meaning that

δ_1 and δ_2 map to types that are syntactically identical up to some bijection on type names. Recall that we consider $\mathrm{dom}(w_0.\eta)$ to contain bound variables and thus can assume it to be disjoint from $\mathrm{rng}(w_0.\eta^i)$ without loss of generality. We then have to distinguish two cases. If the type arguments are not equal (the cast fails), there is not much to do, as expected. If the cast succeeds, however, we basically need to show that the argument types are also *semantically* equal, i.e., $V_n[\![\tau_1]\!]\rho = V_n[\![\tau_2]\!]\rho$. Since $\delta(\tau_1) = \delta(\tau_2)$, this follows from Lemma 13.

Lemma 16 (*Compatibility:* ECAST)
If $\Delta \vdash \Gamma$ and $\Delta \vdash \tau_1$ and $\Delta \vdash \tau_2$, then $\Delta; \Gamma \vdash \mathsf{cast}\ \tau_1\ \tau_2 \precsim \mathsf{cast}\ \tau_1\ \tau_2 : \tau_1 \to \tau_2 \to \tau_2$.

Proof

- Suppose $w_0 \in \mathrm{World}_n$, $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$, $(k, w, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$, and $(k, w) \sqsupseteq (n, w_0)$.
- To show: $(k, w, \mathsf{cast}\ \delta_1(\tau_1)\ \delta_1(\tau_2), \mathsf{cast}\ \delta_2(\tau_1)\ \delta_2(\tau_2)) \in E_n[\![\tau_1 \to \tau_2 \to \tau_2]\!]\rho$.
- Let $\delta := \mathrm{au}(\delta_1, \delta_2, w_0.\eta)$.
- Then $\delta(\tau_1) = w_0.\eta^{-i}\delta_i(\tau_1)$ and $w_0.\eta^{-i}\delta_i(\tau_2) = \delta(\tau_2)$ by Lemma 12.
- Consequently,

$$\begin{aligned} & \delta_1(\tau_1) = \delta_1(\tau_2) \\ \Longleftrightarrow\ & w_0.\eta^{-1}\delta_1(\tau_1) = w_0.\eta^{-1}\delta_1(\tau_2) \\ \Longleftrightarrow\ & \delta(\tau_1) = \delta(\tau_2) \\ \Longleftrightarrow\ & w_0.\eta^{-1}\delta_2(\tau_1) = w_0.\eta^{-1}\delta_2(\tau_2) \\ \Longleftrightarrow\ & \delta_2(\tau_1) = \delta_2(\tau_2) \end{aligned}$$

- Case $\delta_i(\tau_1) = \delta_i(\tau_2)$:
  - Then, we have the following reductions:

$$w.\sigma_i; \mathsf{cast}\ \delta_i(\tau_1)\ \delta_i(\tau_2) \hookrightarrow^1 w.\sigma_i; \lambda x_1.\lambda x_2.x_1$$

  - Hence, it suffices to show $(k-1, \lfloor w \rfloor, \lambda x_1.\lambda x_2.x_1, \lambda x_1.\lambda x_2.x_1) \in V_n[\![\tau_1 \to \tau_2 \to \tau_2]\!]\rho$.
  - So suppose $(k', w') \sqsupseteq (k-1, \lfloor w \rfloor)$ and $(k', w', v_1, v_2) \in V_n[\![\tau_1]\!]\rho$.
  - To show: $(k', w', \lambda x_2.v_1, \lambda x_2.v_2) \in V_n[\![\tau_2 \to \tau_2]\!]\rho$.
  - So suppose $(k'', w'') \sqsupseteq (k', w')$ and $(k'', w'', v_1', v_2') \in V_n[\![\tau_2]\!]\rho$.
  - To show: $(k'', w'', v_1, v_2) \in V_n[\![\tau_2]\!]\rho$
  - By Closure Under World Extension, $(k'', w'', v_1, v_2) \in V_n[\![\tau_1]\!]\rho$.
  - The claim then follows by $\delta(\tau_1) = \delta(\tau_2)$ and Lemma 13.
- Case $\delta_i(\tau_1) \neq \delta_i(\tau_2)$:
  - Then, we have the following reductions:

$$w.\sigma_i; \mathsf{cast}\ \delta_i(\tau_1)\ \delta_i(\tau_2) \hookrightarrow^1 w.\sigma_i; \lambda x_1.\lambda x_2.x_2$$

  - Hence, it suffices to show $(k-1, \lfloor w \rfloor, \lambda x_1.\lambda x_2.x_2, \lambda x_1.\lambda x_2.x_2) \in V_n[\![\tau_1 \to \tau_2 \to \tau_2]\!]\rho$.
  - So suppose $(k', w') \sqsupseteq (k-1, \lfloor w \rfloor)$ and $(k', w', v_1, v_2) \in V_n[\![\tau_1]\!]\rho$.
  - To show: $(k', w', \lambda x_2.x_2, \lambda x_2.x_2) \in V_n[\![\tau_2 \to \tau_2]\!]\rho$.
  - So suppose $(k'', w'') \sqsupseteq (k', w')$ and $(k'', w'', v_1', v_2') \in V_n[\![\tau_2]\!]\rho$.
  - To show: $(k'', w'', v_1', v_2') \in V_n[\![\tau_2]\!]\rho$, which is immediate. □

Since new is the only construct that modifies the type store, its compatibility proof is also the only one where we actually have to extend the η and ρ components of the initial world w with bindings for some fresh dynamically generated type name (here, 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 reference allocation in a language with mutable references (Ahmed *et al.*, 2009).

Lemma 17 (*Compatibility:* ENEW)
If $\Delta, \alpha{\approx}\tau' ; \Gamma \vdash e_1 \precsim e_2 : \tau$ and $\Delta \vdash \tau$ and $\Delta \vdash \Gamma$,
then $\Delta ; \Gamma \vdash \mathsf{new}\ \alpha{\approx}\tau'\ \mathsf{in}\ e_1 \precsim \mathsf{new}\ \alpha{\approx}\tau'\ \mathsf{in}\ e_2 : \tau$.

Proof

- Suppose $w_0 \in \mathrm{World}_n$, $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$, $(k, w, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$, and $(k, w) \sqsupset (n, w_0)$.
- To show: $(k, w, \delta_1\gamma_1(\mathsf{new}\ \alpha{\approx}\tau'\ \mathsf{in}\ e_1), \delta_2\gamma_2(\mathsf{new}\ \alpha{\approx}\tau'\ \mathsf{in}\ e_2)) \in E_n[\![\tau]\!]\rho$.
- Assume that $w.\sigma_1 ; \delta_1\gamma_1(\mathsf{new}\ \alpha{\approx}\tau'\ \mathsf{in}\ e_1)$ terminates:

$$\begin{aligned} & w.\sigma_1 ; \delta_1\gamma_1(\mathsf{new}\ \alpha{\approx}\tau'\ \mathsf{in}\ e_1) \\ \hookrightarrow^1\ & w.\sigma_1, \alpha_1{\approx}\delta_1(\tau') ; \delta_1\gamma_1(e_1)[\alpha_1/\alpha] \\ \hookrightarrow^{j'}\ & \sigma_1 ; v_1 \end{aligned}$$

  and $1 + j' =: j < k$.
- Note that

$$w.\sigma_2 ; \delta_2\gamma_2(\mathsf{new}\ \alpha{\approx}\tau'\ \mathsf{in}\ e_2) \hookrightarrow^1 w.\sigma_2, \alpha_2{\approx}\delta_2(\tau') ; \delta_2\gamma_2(e_2)[\alpha_2/\alpha].$$

- Let $w_\alpha := ((w.\sigma_1, \alpha_1{\approx}\delta_1(\tau')), (w.\sigma_2, \alpha_2{\approx}\delta_2(\tau')), (w.\eta, \alpha \mapsto (\alpha_1, \alpha_2)), (w.\rho, \alpha \mapsto r))$ for $r := (\rho^1(\tau'), \rho^2(\tau'), V_k[\![\tau']\!]\lfloor\rho\rfloor)$, so $(k, w_\alpha) \sqsupseteq (k, w)$ and $(\delta_1, \delta_2, \lfloor\rho\rfloor) \in D_k[\![\Delta]\!]w_\alpha$.
- Let $(\delta_1', \delta_2', \rho') := ((\delta_1, \alpha \mapsto \alpha_1), (\delta_2, \alpha \mapsto \alpha_2), (\lfloor\rho\rfloor, \alpha \mapsto r))$.
- Note that $w_\alpha.\sigma_i(\alpha_i) = \delta_i(\tau')$, $\alpha_i = w_\alpha.\eta^i(\alpha)$, and $w_\alpha.\rho(\alpha).R = V_k[\![\tau']\!]\lfloor\rho\rfloor$.
- Therefore, $(\delta_1', \delta_2', \rho') \in D_k[\![\Delta, \alpha{\approx}\tau']\!]w_\alpha$.
- By Closure Under World Extension, we know $(k-1, \lfloor w_\alpha\rfloor, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$ and therefore $(k-1, \lfloor w_\alpha\rfloor, \gamma_1, \gamma_2) \in G_k[\![\Gamma]\!]\rho'$.
- Now instantiate the premise with $w_\alpha \in \mathrm{World}_k$, $(\delta_1', \delta_2', \rho') \in D_k[\![\Delta, \alpha{\approx}\tau']\!]w_\alpha$, $(k-1, \lfloor w_\alpha\rfloor, \gamma_1, \gamma_2) \in G_k[\![\Gamma]\!]\rho'$, and $(k-1, \lfloor w_\alpha\rfloor) \sqsupset (k, w_\alpha)$ to get $(k-1, \lfloor w_\alpha\rfloor, \delta_1'\gamma_1(e_1), \delta_2'\gamma_2(e_2)) \in E_k[\![\tau]\!]\rho'$.
- Note that $\delta_i'\gamma_i(e_i) = \delta_i\gamma_i(e_i)[\alpha_i/\alpha]$.
- Consequently, there exists $(k-j, w') \sqsupseteq (k-1, w_\alpha)$ such that

$$w.\sigma_2, \alpha_2{\approx}\delta_2(\tau') ; \delta_2\gamma_2(e_2)[\alpha_2/\alpha] \hookrightarrow^* w'.\sigma_2 ; v_2$$

  with $w'.\sigma_1 = \sigma_1$ and $(k-j, w', v_1, v_2) \in V_k[\![\tau]\!]\rho'$.
- Because of $\Delta \vdash \tau$, the latter implies $(k-j, w', v_1, v_2) \in V_n[\![\tau]\!]\rho$. □

Compatibility for ECONV follows from the fact that isomorphic types are semantically equal, which we prove separately below. 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 $D[\![\Delta, \alpha{\approx}\tau]\!]w$.

Lemma 18 (*Type Isomorphism*)
If $\Delta \vdash \tau_1 \approx \tau_2$ and $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w$, then

1. $V_n[\![\tau_1]\!]\rho = V_n[\![\tau_2]\!]\rho$ and
2. $E_n[\![\tau_1]\!]\rho = E_n[\![\tau_2]\!]\rho$.

Lemma 19 (*Compatibility:* Econv)
If $\Delta;\Gamma \vdash e_1 \precsim e_2 : \tau'$ and $\Delta \vdash \tau \approx \tau'$, then $\Delta;\Gamma \vdash e_1 \precsim e_2 : \tau$.

Proof
Follows from Type Isomorphism. □

4.3.4 Soundness

Theorem 20 (*Fundamental Property of* $\precsim$)
If $\Delta;\Gamma \vdash e : \tau$, then $\Delta;\Gamma \vdash e \precsim e : \tau$.

Proof
By induction on the typing derivation, in each case using the appropriate compatibility lemma. □

The full compatibility and the Fundamental Property of $\precsim$ are at the heart of the soundness proof. Based on that and the following small lemma we can finally establish that $\precsim$ is a precongruence with respect to the constructs of the language and then prove the actual soundness theorem.

Lemma 21 (*LR-Weakening*)
If $\Delta;\Gamma \vdash e_1 \precsim e_2 : \tau$, $\Delta' \supseteq \Delta$, $\Gamma' \supseteq \Gamma$, and $\Delta' \vdash \Gamma$, then $\Delta';\Gamma' \vdash e_1 \precsim e_2 : \tau$.

Lemma 22 (*Precongruence of* $\precsim$)
If $\Delta;\Gamma \vdash e_1 \precsim e_2 : \tau$ and $\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau')$, then $\Delta';\Gamma' \vdash C[e_1] \precsim C[e_2] : \tau'$.

Proof
By induction on the derivation of the context typing, in each case using the appropriate compatibility lemma. For a context containing another term, we also need the Fundamental Property; for $C = [\_]$, we need LR-Weakening. □

Theorem 23 (*Soundness of* $\precsim$ *with respect to* $\leqslant$)
If $\Delta;\Gamma \vdash e_1 \precsim e_2 : \tau$, then $\Delta;\Gamma \vdash e_1 \leqslant e_2 : \tau$.

Proof

- Suppose $\vdash \sigma$ and $\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\sigma;\emptyset;\tau')$ and $\sigma;C[e_1]\downarrow$, i.e., $\sigma;C[e_1] \hookrightarrow^j \sigma_1;v_1$.
- To show: $\sigma;C[e_2]\downarrow$.
- By Precongruence, we have $\sigma;\epsilon \vdash C[e_1] \precsim C[e_2] : \tau'$.
- To instantiate this, we first need to create an initial world representing σ. Say, $\sigma = \alpha_1{\approx}\tau_1,\ldots,\alpha_n{\approx}\tau_n$.

- Let

$$\begin{aligned}
\sigma_0 &:= \epsilon \\
\sigma_{i+1} &:= \sigma_i, \alpha_{i+1} {\approx} \tau_{i+1} \\
\delta_0 &:= \emptyset \\
\delta_{i+1} &:= \delta_i, \alpha_{i+1} \mapsto \alpha_{i+1} \\
\rho_0 &:= \emptyset \\
\rho_{i+1} &:= \rho_i, \alpha_{i+1} \mapsto V_{j+2}[\![\tau_{i+1}]\!]\rho_i \\
w &:= (\sigma, \sigma, \{\alpha_i \mapsto (\alpha_i, \alpha_i) \mid 1 \leqslant i \leqslant n\}, \rho_n)
\end{aligned}$$

- Note that $\rho_i \in \mathrm{Interp}_{j+2}$ and $w \in \mathrm{World}_{j+2}$.
- Furthermore, given $0 \leqslant i < n$, it is easy to see that $(\delta_i, \delta_i, \rho_i) \in D_{j+2}[\![\sigma_i]\!]w$ implies $(\delta_{i+1}, \delta_{i+1}, \rho_{i+1}) \in D_{j+2}[\![\sigma_{i+1}]\!]w$.
- Together with $(\delta_0, \delta_0, \rho_0) \in D_{j+2}[\![\epsilon]\!]w$ this means $(\delta_n, \delta_n, \rho_n) \in D_{j+2}[\![\sigma]\!]w$.
- Instantiate $\sigma; \epsilon \vdash C[e_1] \lesssim C[e_2] : \tau'$ with $w \in \mathrm{World}_{j+2}$, $(\delta_n, \delta_n, \rho_n) \in D_{j+2}[\![\sigma]\!]w$ and $(j+1, \lfloor w \rfloor, \emptyset, \emptyset) \in G_{j+2}[\![\epsilon]\!]\rho_n$ to get $(j+1, \lfloor w \rfloor, \delta_n(C[e_1]), \delta_n(C[e_2])) \in E_{j+2}[\![\tau']\!]\rho_n$.
- Note that $\delta_n(C[e_i]) = C[e_i]$.
- Because of the assumption $\sigma; C[e_1] \hookrightarrow^j \sigma_1; v_1$, we therefore get $\sigma; C[e_2]\downarrow$. □

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_{sem1} and e_{sem2}. 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, $\vdash e_{\mathrm{sem1}} \lesssim e_{\mathrm{sem2}} : \tau_{\mathrm{sem}}$; the other is proven analogously.

Expanding the definitions, we need to show $(k, w, e_{\mathrm{sem1}}, e_{\mathrm{sem2}}) \in E_n[\![\tau_{\mathrm{sem}}]\!]$. 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 $w'.\sigma_2 = w.\sigma_2, \alpha_2 {\approx} \mathsf{bool}$,
- $(k-1, w', \mathsf{pack}\langle \alpha_1, v_1 \rangle, \mathsf{pack}\langle \alpha_2, v_2 \rangle) \in V_n[\![\tau_{\mathrm{sem}}]\!]$.

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

$$\begin{aligned}
R &:= \{(k'', w'', v_{\mathsf{int}}, v_{\mathsf{bool}}) \in \mathrm{Atom}^{\mathsf{val}}_{k-1}[\mathsf{int}, \mathsf{bool}] \mid \\
&\qquad (v_{\mathsf{int}}, v_{\mathsf{bool}}) = (1, \mathsf{true}) \vee (v_{\mathsf{int}}, v_{\mathsf{bool}}) = (0, \mathsf{false})\} \\
r &:= (\mathsf{int}, \mathsf{bool}, R) \\
w' &:= ((w.\sigma_1, \alpha_1 {\approx} \mathsf{int}), (w.\sigma_2, \alpha_2 {\approx} \mathsf{bool}), (w.\eta, \alpha \mapsto (\alpha_1, \alpha_2)), (\lfloor w.\rho \rfloor_{k-1}, \alpha \mapsto r))
\end{aligned}$$

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 $(k-2, \lfloor w' \rfloor, v_1, v_2) \in V_n[\![\tau'_{\mathrm{sem}}]\!](\alpha \mapsto r')$, where $\tau'_{\mathrm{sem}} = \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool})$. 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 $(k-2, \lfloor w' \rfloor, v_1, v_2) \in V_n[\![\tau'_{\text{sem}}]\!](\alpha \mapsto r)$ decomposes into three parts, following the structure of τ'_{sem}:

1. $(k-2, \lfloor w' \rfloor, 1, \mathsf{true}) \in V_n[\![\alpha]\!](\alpha \mapsto r)$
   This holds because $V_n[\![\alpha]\!](\alpha \mapsto r) = R$.
2. $(k-2, \lfloor w' \rfloor, \lambda x.(1-x), \lambda x.\neg x) \in V_n[\![\alpha \to \alpha]\!](\alpha \mapsto r)$
   - Suppose we are given related arguments in a future world: $(k'', w'', v'_1, v'_2) \in V_n[\![\alpha]\!](\alpha \mapsto r) = R$.
   - 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, $(k'', w'', 1 - v'_1, \neg v'_2) \in E_n[\![\alpha]\!](\alpha \mapsto r)$.
3. $(k-2, \lfloor w' \rfloor, \lambda x.(x \neq 0), \lambda x.x) \in V_n[\![\alpha \to \mathsf{bool}]\!](\alpha \mapsto r)$
   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 $(k'', w'', v'_1 \neq 0, v'_2) \in E_n[\![\mathsf{bool}]\!](\alpha \mapsto r)$.

Partly benign effects (repeatability). 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.

Consider the following functions (where τ is arbitrary but closed):

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

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. A detailed formal proof of v_1 and v_2's equivalence is given in Appendix B.

Partly benign effects (order independence). Now 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. Again, a formal proof of equivalence can be found in Appendix B.

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*.

Figure 4 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

$$
\begin{aligned}
\mathrm{Wr}^{\pm}_{\alpha} &\stackrel{\text{def}}{=} \lambda x{:}\alpha.x \\
\mathrm{Wr}^{\pm}_{b} &\stackrel{\text{def}}{=} \lambda x{:}b.x \\
\mathrm{Wr}^{\pm}_{\tau_1\times\tau_2} &\stackrel{\text{def}}{=} \lambda x{:}(\tau_1 \times \tau_2).\langle \mathrm{Wr}^{\pm}_{\tau_1}\,(x.1), \mathrm{Wr}^{\pm}_{\tau_2}\,(x.2)\rangle \\
\mathrm{Wr}^{\pm}_{\tau_1\to\tau_2} &\stackrel{\text{def}}{=} \lambda x{:}(\tau_1 \to \tau_2).\lambda x_1{:}\tau_1.\,\mathrm{Wr}^{\pm}_{\tau_2}\,(x\,(\mathrm{Wr}^{\mp}_{\tau_1}\;x_1)) \\
\mathrm{Wr}^{\pm}_{\forall\alpha.\tau} &\stackrel{\text{def}}{=} \lambda x{:}(\forall\alpha.\tau).\Lambda\alpha.\,\mathsf{new}^{\mp}\,\alpha\;\mathsf{in}\;\mathrm{Wr}^{\pm}_{\tau}\,(x\,\alpha) \\
\mathrm{Wr}^{\pm}_{\exists\alpha.\tau} &\stackrel{\text{def}}{=} \lambda x{:}(\exists\alpha.\tau).\,\mathsf{unpack}\,\langle\alpha, x'\rangle{=}x\;\mathsf{in} \\
&\qquad\qquad \mathsf{new}^{\pm}\,\alpha\;\mathsf{in}\;\mathsf{pack}\,\langle\alpha, \mathrm{Wr}^{\pm}_{\tau}\;x'\rangle\;\mathsf{as}\;\exists\alpha.\tau \\
\mathsf{new}^{+}\,\alpha\;\mathsf{in}\;e &\stackrel{\text{def}}{=} \mathsf{new}\;\alpha'{\approx}\alpha\;\mathsf{in}\;e[\alpha'/\alpha] \\
\mathsf{new}^{-}\,\alpha\;\mathsf{in}\;e &\stackrel{\text{def}}{=} e
\end{aligned}
$$

Fig. 4. Wrapping for G.

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 & Pierce (2007a) suggest for establishing type abstraction in an untyped language (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.

Lemma 24
If $\Delta \vdash \tau$, then $\Delta;\epsilon \vdash \mathrm{Wr}^{\pm}_{\tau} : \tau \to \tau$.

Given these operators, we can go back to our semaphore example: e_{sem1} can now be obtained as $\mathrm{Wr}^{+}_{\tau_{\text{sem}}}\ e_{\text{sem}}$ (modulo some harmless η-expansions). This generalizes to other ADTs: wrapping their implementations positively will guarantee abstraction by "making them parametric." We prove that in the next section.

Positive wrapping at existential type 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 provides the dynamic equivalent (Rossberg, 2008). In fact, positive wrapping is precisely how sealing is implemented in Alice ML (Rossberg *et al.*, 2004), 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

$$T^\pi_n[\![\Omega]\!]w \stackrel{\text{def}}{=} \{(\tau_1, \tau_2, (w.\sigma_1^*(\tau_1), w.\sigma_2^*(\tau_2), R)) \mid \\ \text{ftv}(\tau_i) \subseteq \text{dom}(w.\sigma_i) \land R \in \text{Rel}_n[w.\sigma_1^*(\tau_1), w.\sigma_2^*(\tau_2)]\}$$

(everything else as in Figure 3)

Fig. 5. Parametric logical relation.

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 (see Rossberg's thesis (2007) for illustrative examples). Moreover, as we show in the next section, our definition of 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 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 terms e_1 and 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 3, 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 5 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 $^\pi$ (i.e., V^π, E^π, etc.) to distinguish them from the original ones. Likewise, we write $\lesssim^\pi$ for the notion of *parametric logical approximation* defined as in Figure 3 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 25 (*Wrapping for $\lesssim^\pi$*)

1. If $\vdash e_1 \lesssim^\pi e_2 : \tau$, then $\vdash \text{Wr}^+_\tau\, e_1 \lesssim \text{Wr}^+_\tau\, e_2 : \tau$.

2. If $\vdash e_1 \precsim e_2 : \tau$, then $\vdash \mathrm{Wr}^-_\tau\ e_1 \precsim^\pi \mathrm{Wr}^-_\tau\ e_2 : \tau$.

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 if we can relate two terms using parametric reasoning, then the positive wrapping 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.

Rather than giving the proof of Theorem 25 now, we will wait until Section 8.1 to derive it as a corollary of a more general result (see Corollary 32).

The alert reader may wonder why this Wrapping Theorem only talks about closed terms. First of all, simply allowing open terms would not be correct. For instance, it is easy to see that we have

$$\epsilon; x{:}(\forall\alpha.\mathsf{bool}) \vdash x\ \mathsf{bool} \precsim^\pi x\ \ \mathsf{unit} : \mathsf{bool}$$

because the instantiations of x will be parametric by definition. For $\precsim$, they may, of course, be non-parametric (consider $\mathsf{equal?}$ unit being plugged in for x), hence

$$\epsilon; x{:}(\forall\alpha.\mathsf{bool}) \vdash x\ \mathsf{bool} \precsim x\ \ \mathsf{unit} : \mathsf{bool}$$

does *not* hold. However, since $\mathrm{Wr}^+_{\mathsf{bool}}$ is just the identity function, this is essentially what the naive extension of the Wrapping theorem to open terms would tell us.

The solution to this (we conjecture) is to wrap all free value variables at the inverse polarity so that the theorem would look as follows:

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

Here, the substitution $\gamma^\pm_\Gamma$ replaces each free variable $x{:}\tau$ by its wrapping $\mathrm{Wr}^\pm_\tau\ x$ and could be defined as follows:

$$\gamma^\pm_\epsilon \stackrel{\text{def}}{=} \emptyset \qquad\qquad \gamma^\pm_{\Gamma,x:\tau} \stackrel{\text{def}}{=} \gamma^\pm_\Gamma, x \mapsto (\mathrm{Wr}^\pm_\tau\ x)$$

Proving this theorem correct, however, is another matter. One problem is that if we attempt to prove the above statement, after unfolding the definition of logical approximation in part (1), we are given some $(\delta_1, \delta_2, \rho) \in D[\![\Delta]\!]$. To instantiate the assumption appropriately, $(\delta_1, \delta_2, \rho)$ needs to be in $D^\pi[\![\Delta]\!]$. In part (2), the situation is the other way around. However, $D[\![\Delta]\!]$ and $D^\pi[\![\Delta]\!]$ are only equal if Δ does not contain components of the form $\alpha{\approx}\tau'$. Another problem is that wrapped value substitutions—which arise in the proof—are no longer *value* substitutions. All in all, we believe these problems can be solved, but we leave the solution to future work.

Finally, what can we say about the content of the parametric relation? Obviously, it cannot contain arbitrary non-parametric G terms—e.g., $\Lambda\alpha_1.\Lambda\alpha_2.\mathsf{cast}\ \alpha_1\ \alpha_2$ is not even related to itself in E^π. Apart from cast, however, the parametric relation is compatible with all constructs. The corresponding compatibility proofs for the non-parametric relation carry over. The only difference is that compatibility for Epack and Einst become easier to show. In the proof of the former, for instance, it is immediate that the witness relation has the required form because $T^\pi[\![\Omega]\!]$ does not actually impose any restrictions.

Consequently, we obtain the following restricted form of the Fundamental Property:

Theorem 26 (*Fundamental Property for* $\precsim^\pi$)
If $\Delta;\Gamma \vdash e : \tau$ and e is cast-free, then $\Delta;\Gamma \vdash e \precsim^\pi e : \tau$.

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 with respect to contextual approximation, *if* the definition of the latter is limited to quantifying only over cast-free contexts:

Theorem 27 (*Soundness of* $\precsim^\pi$)
If $\Delta;\Gamma \vdash e_1 \precsim^\pi e_2 : \tau$, then for any cast-free $C : (\Delta;\Gamma;\tau) \rightsquigarrow (\sigma;\epsilon;\tau')$ with $\vdash \sigma$:

$$\sigma; C[e_1]\downarrow \Rightarrow \sigma; C[e_2]\downarrow$$

Proof
Analogous to the soundness proof for $\precsim$. The difference is that $\precsim^\pi$ is a precongruence only with respect to 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 latter aspect of the proof is covered once and for all by the Wrapping Theorem.

Recall the two implementations, here given in unwrapped form:

$$\begin{aligned}
\tau_{\mathrm{sem}} &:= \exists\alpha.\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool}) \\
e'_{\mathrm{sem1}} &:= \mathsf{pack}\,\langle \mathsf{int}, \langle 1, \lambda x{:}\mathsf{int}.(1-x), \lambda x{:}\mathsf{int}.(x \neq 0)\rangle\rangle\ \mathsf{as}\ \tau_{\mathrm{sem}} \\
e'_{\mathrm{sem2}} &:= \mathsf{pack}\,\langle \mathsf{bool}, \langle \mathsf{true}, \lambda x{:}\mathsf{bool}.\neg x, \lambda x{:}\mathsf{bool}.x\rangle\rangle\ \mathsf{as}\ \tau_{\mathrm{sem}}
\end{aligned}$$

We can prove $\vdash e'_{\mathrm{sem1}} \precsim^\pi e'_{\mathrm{sem2}} : \tau_{\mathrm{sem}}$ using conventional parametric reasoning about polymorphic terms, i.e., we immediately get to pick the relational interpretation of the abstract type and do not have to operate on worlds at all:

Proof

- Suppose $w_0 \in \mathrm{World}_n$ and $(k,w) \sqsupseteq (n,w_0)$.
- To show: $(k,w,e'_{\mathrm{sem1}},e'_{\mathrm{sem2}}) \in V^\pi_n[\![\exists\alpha.\tau]\!]$.
- Let $R := \{(k',w',v_a,v_b) \in \mathrm{Atom}_{k-1} \mid (v_a,v_b) = (\mathsf{true},1) \vee (v_a,v_b) = (\mathsf{false},0)\}$ and $r := (\mathsf{int},\mathsf{bool},R)$, such that $(\mathsf{int},\mathsf{bool},r) \in T^\pi[\![\Omega]\!]w$.
- It thus suffices to show $(k',w',v_1,v_2) \in V^\pi_n[\![\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool})]\!](\alpha \mapsto r)$ for any $(k',w') \sqsupseteq (k,w)$, where v_1 and v_2 are the term components of e'_{sem1} and e'_{sem2}, respectively.
- This decomposes into the same three parts as in Section 4.4. □

Now define $e_{\text{sem1}} = \text{Wr}^+_{\tau\text{sem}}\ e'_{\text{sem1}}$ and $e_{\text{sem2}} = \text{Wr}^+_{\tau\text{sem}}\ e'_{\text{sem2}}$, which are semantically equivalent (by some simple applications of β- and η-equivalence) to the original definitions in Section 2.3. The Wrapping Theorem then tells us that $\vdash e_{\text{sem1}} \precsim e_{\text{sem2}} : \tau_{\text{sem}}$.

A free theorem. We can use the parametric relation for proving free theorems (Wadler, 1989) in G. For example, for any $\vdash g : \forall\alpha.\alpha \to \alpha$ in G, it holds that $\text{Wr}^-\ g$ either diverges for all possible arguments τ and $\vdash v : \tau$, or it returns v in all cases.

Informally, we first apply the Fundamental Property for $\precsim$ to relate g to itself in E, then transfer this to E^π for $\text{Wr}^-\ g$ using the Wrapping Theorem. From there, the proof proceeds in the usual way.

Formally, we have to strengthen the claim slightly: Suppose $\sigma_0 \vdash v : \forall\alpha.\alpha \to \alpha$. We want to show that either

1. for all $\sigma \sqsupseteq \sigma_0$, τ, v' with $\sigma \vdash v' : \tau$, $\sigma;\text{Wr}^-_{\forall\alpha.\alpha\to\alpha}\ v\ \tau\ v' \uparrow$, or
2. for all $\sigma \sqsupseteq \sigma_0$, τ, v' with $\sigma \vdash v' : \tau$, there is σ' such that $\sigma;\text{Wr}^-_{\forall\alpha.\alpha\to\alpha}\ v\ \tau\ v' \hookrightarrow^* \sigma';v'$.

Assume that part (1) does not hold (otherwise, we are done). In this case, we know that there is at least one appropriate σ_1, τ_1, v_1 such that $\sigma_1;\text{Wr}^-\ v\ \tau_1\ v_1$ evaluates in $j := j_1 + 1 + j_2 + 1 + j_3$ steps to some $\sigma'''_1;v'_1$:

$$\begin{array}{rl} & \sigma_1;\text{Wr}^-\ v\ \tau_1\ v_1 \\ \hookrightarrow^{j_1} & \sigma'_1;(\Lambda\alpha.e_1)\ \tau_1\ v_1 \\ \hookrightarrow^{1} & \sigma'_1;e_1[\tau_1/\alpha]\ v_1 \\ \hookrightarrow^{j_2} & \sigma''_1;(\lambda x{:}\tau'_1.e'_1)\ v_1 \\ \hookrightarrow^{1} & \sigma''_1;e'_1[v_1/x] \\ \hookrightarrow^{j_3} & \sigma'''_1;v'_1 \end{array}$$

We now show that this implies that any $\sigma_2;\text{Wr}^-\ v\ \tau_2\ v_2$ will indeed evaluate to $\sigma'_2;v_2$ (for some σ'_2):

- By the Fundamental Property, $\sigma_0;\epsilon \vdash v \precsim v : \forall\alpha.\alpha \to \alpha$.
- Construct $w_0 \in \text{World}_{j+2}$ and $(\delta_1, \delta_2, \rho) \in D_{j+2}[\![\sigma_0]\!]w_0$ in the same manner as in the proof of Soundness (Theorem 23) except that $w_0.\sigma_1 = \sigma_1$ and $w_0.\sigma_2 = \sigma_2$.
- Instantiating $\sigma_0;\epsilon \vdash v \precsim v : \forall\alpha.\alpha \to \alpha$ then yields $(j+1, \lfloor w_0 \rfloor, v, v) \in V_n[\![\forall\alpha.\alpha \to \alpha]\!]\rho$
- By Wrapping, $(j+1, \lfloor w_0 \rfloor, \text{Wr}^-\ v, \text{Wr}^-\ v) \in E^\pi_n[\![\forall\alpha.\alpha \to \alpha]\!]\rho$.
- Consequently, there exists $(j+1-j_1, w') \sqsupseteq (j+1, \lfloor w_0 \rfloor)$ such that
$$\sigma_2;\text{Wr}^-\ v\ \tau_2\ v_2 \hookrightarrow^* w'.\sigma_2;(\Lambda\alpha.e_2)\ \tau_2\ v_2$$
with $w'.\sigma_1 = \sigma'_1$ and $(j+1-j_1, w', \Lambda\alpha.e_1, \Lambda\alpha.e_2) \in V^\pi_n[\![\forall\alpha.\alpha \to \alpha]\!]\rho$.
- Let $R := \{(\widehat{k}, \widehat{w}, \widehat{v_1}, \widehat{v_2}) \in \text{Atom}_{j+1-j_1} \mid \widehat{v_1} = v_1 \wedge \widehat{v_2} = v_2\}$ and $r := (\sigma^*_1(\tau_1), \sigma^*_2(\tau_2), R)$, so $(\tau_1, \tau_2, r) \in T^\pi_{j+1-j_1}[\![\Omega]\!]w'$.
- Instantiate $(j+1-j_1, w', \Lambda\alpha.e_1, \Lambda\alpha.e_2) \in V^\pi_n[\![\forall\alpha.\alpha \to \alpha]\!]\rho$ to get $(j+1-j_1-1, \lfloor w' \rfloor, e_1[\tau_1/\alpha], e_2[\tau_2/\alpha]) \in E^\pi_n[\![\alpha \to \alpha]\!]\rho, \alpha \mapsto r$.

- Consequently, there exists $(j+1-j_1-1-j_2, w'') \sqsupseteq (j+1-j_1-1, \lfloor w' \rfloor)$ such that
$$w'.\sigma_2; e_2[\tau_2/\alpha]\ v_2 \hookrightarrow^* w''.\sigma_2; (\lambda x.e_2')\ v_2$$
with $w''.\sigma_1 = \sigma_1''$ and $(j+1-j_1-1-j_2, w'', \lambda x.e_1', \lambda x.e_2') \in V_n^\pi[\![\alpha \to \alpha]\!]\rho, \alpha \mapsto r$.
- Since $(j+1-j_1-1-j_2-1, \lfloor w'' \rfloor, v_1, v_2) \in R = V_n^\pi[\![\alpha]\!]\rho, \alpha \mapsto r$, we get $(j+1-j_1-1-j_2-1, \lfloor w'' \rfloor, e_1'[v_1/x], e_2'[v_2/x]) \in E_n^\pi[\![\alpha]\!]\rho, \alpha \mapsto r$.
- Consequently, there exists $(1, w''') \sqsupseteq (j+1-j_1-1-j_2-1, \lfloor w'' \rfloor)$ such that
$$w''.\sigma_2; e_2'[v_2/x] \hookrightarrow^* w'''.\sigma_2; v_2'$$
with $w'''.\sigma_1 = \sigma_1'''$ and $(1, w''', v_1', v_2') \in V_n^\pi[\![\alpha]\!]\rho, \alpha \mapsto r = R$.
- Hence, $v_1' = v_1$ and $v_2' = v_2$ by construction of R.

7 Syntactic versus 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 $b2i = \lambda x{:}\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 0$):

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

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 $\vdash \mathrm{Wr}^+\ g_1 \leqslant \mathrm{Wr}^+\ g_2 : \forall\alpha.\tau_\alpha$. 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 anything about τ_1 and R_α, because

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

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

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 arise is if we want to show representation independence for generic ADTs that (like the one in Section 7) return different results for different instantiations of their type arguments, but where (unlike the one 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 $\forall\alpha.\tau_\alpha$:

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

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

$$
\begin{array}{ll}
V_n^{\pm}[\![\alpha]\!]\rho & \stackrel{\text{def}}{=} \lfloor \rho(\alpha).R \rfloor_n \\
V_n^{\pm}[\![b]\!]\rho & \stackrel{\text{def}}{=} \{(k, w, v, v) \in \text{Atom}_n[b, b]\} \\
V_n^{\pm}[\![\tau \times \tau']\!]\rho & \stackrel{\text{def}}{=} \{(k, w, \langle v_1, v_1' \rangle, \langle v_2, v_2' \rangle) \in \text{Atom}_n[\rho^1(\tau \times \tau'), \rho^2(\tau \times \tau')] \mid \\
& \qquad (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 & \stackrel{\text{def}}{=} \{(k, w, \lambda x{:}\tau_1.e_1, \lambda x{:}\tau_2.e_2) \in \text{Atom}_n[\rho^1(\tau' \to \tau), \rho^2(\tau' \to \tau)] \mid \\
& \qquad \forall (k', w', v_1, v_2) \in V_n^{\mp}[\![\tau']\!]\rho.\, (k', w') \sqsupseteq (k, w) \Rightarrow \\
& \qquad (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 & \stackrel{\text{def}}{=} \{(k, w, \lambda\alpha.e_1, \lambda\alpha.e_2) \in \text{Atom}_n[\rho^1(\forall\alpha.\tau), \rho^2(\forall\alpha.\tau)] \mid \\
& \qquad \forall (k', w') \sqsupseteq (k, w).\ \forall (\tau_1, \tau_2, r) \in T_{k'}^{\mp}[\![\Omega]\!]w'. \\
& \qquad (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 & \stackrel{\text{def}}{=} \{(k, w, \mathsf{pack}\,\langle \tau_1, v_1 \rangle, \mathsf{pack}\,\langle \tau_2, v_2 \rangle) \in \text{Atom}_n[\rho^1(\exists\alpha.\tau), \rho^2(\exists\alpha.\tau)] \mid \\
& \qquad \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 & \stackrel{\text{def}}{=} \{(k, w, e_1, e_2) \in \text{Atom}_n[\rho^1(\tau), \rho^2(\tau)] \mid \\
& \qquad \forall j < k.\ \forall \sigma_1, v_1.\, (w.\sigma_1; e_1 \hookrightarrow^j \sigma_1; v_1) \Rightarrow \exists w', v_2.\, (k - j, w') \sqsupseteq (k, w) \wedge \\
& \qquad 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\} \\
\\
T_n^{+}[\![\Omega]\!]w & \stackrel{\text{def}}{=} T_n^{\pi}[\![\Omega]\!]w \\
T_n^{-}[\![\Omega]\!]w & \stackrel{\text{def}}{=} T_n[\![\Omega]\!]w
\end{array}
$$

Fig. 6. Polarized logical relations.

relation from Section 6—since the functions do not *behave* parametrically (i.e., the package each function returns when instantiated with int is semantically different from the one that it returns for any other type instantiation), they will not be related in E^{π}.

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 6 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. We merely refine the definition of the type relation $T[\![\Omega]\!]w$ 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 are parametric in E^{+} but non-parametric in E^{-}, and vice versa for universals.

In fact, all four relations can easily be formulated in a single unified definition indexed by $\iota ::= \epsilon \mid \pi \mid + \mid -$ (with ϵ representing the original non-parametric relation). We refer the interested reader to the first author's master's thesis for details (Neis, 2009).

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

Fig. 7. Relating the relations.

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 28 (*Wrapping for* $\lesssim^{\pm}$)

1. If $\vdash e_1 \lesssim^{+} e_2 : \tau$, then $\vdash \mathrm{Wr}^{+}_{\tau}\ e_1 \lesssim \mathrm{Wr}^{+}_{\tau}\ e_2 : \tau$.
2. If $\vdash e_1 \lesssim e_2 : \tau$, then $\vdash \mathrm{Wr}^{-}_{\tau}\ e_1 \lesssim^{-} \mathrm{Wr}^{-}_{\tau}\ e_2 : \tau$.
3. If $\vdash e_1 \lesssim^{+} e_2 : \tau$, then $\vdash \mathrm{Wr}^{-}_{\tau}\ e_1 \lesssim^{\pi} \mathrm{Wr}^{-}_{\tau}\ e_2 : \tau$.
4. If $\vdash e_1 \lesssim^{\pi} e_2 : \tau$, then $\vdash \mathrm{Wr}^{+}_{\tau}\ e_1 \lesssim^{-} \mathrm{Wr}^{+}_{\tau}\ e_2 : \tau$.

Intuitively, the first property says that whenever two terms are related for parametric *uses*, their positive wrappings will actually be related unconditionally, even in a "hostile" non-parametric context—i.e., positive wrapping enforces parametric use. By the second property, when two terms are related unconditionally, their negative wrappings are related even in contexts that expect them to *behave* parametrically—i.e., negative wrapping enforces parametric behavior. Dually, the latter two properties characterize the effect of applying positive and negative wrappings to positively related terms in the reverse order. This is probably best understood graphically: the labeled, outer arrows in Figure 7 summarize the situation by showing how the two polarities of wrapping can take terms from one relation to another (we explain the rest of the diagram in the remainder of this section).

To show this theorem, we prove the following more general lemma. Each subitem here actually states two properties, which are obtained by first consistently ignoring the left superscript of the X^{ι_1,ι_2} notation in the whole statement and then the right one. For instance, Lemma 29 part (1a) states that the positive wrapping transports values from V^π to E^- and, independently, from V^+ to E^ϵ (that is, to E). Similarly, each proof actually represents two proofs simultaneously.

Lemma 29
Suppose $w_0 \in \text{World}_n$, $(\delta_1, \delta_2, \rho) \in D_n^\pi[\![\Delta]\!]w_0$, $(k, w) \sqsupset (n, w_0)$, and $\Delta \vdash \tau$.

1. (a) If $(k, w, v_1, v_2) \in V_n^{\pi,+}[\![\tau]\!]\rho$, then $(k, w, \delta_1(\text{Wr}_\tau^+)\ v_1, \delta_2(\text{Wr}_\tau^+)\ v_2) \in E_n^{-,\epsilon}[\![\tau]\!]\rho$.
   (b) If $(k, w, e_1, e_2) \in E_n^{\pi,+}[\![\tau]\!]\rho$, then $(k, w, \delta_1(\text{Wr}_\tau^+)\ e_1, \delta_2(\text{Wr}_\tau^+)\ e_2) \in E_n^{-,\epsilon}[\![\tau]\!]\rho$.
2. (a) If $(k, w, v_1, v_2) \in V_n^{+,\epsilon}[\![\tau]\!]\rho$, then $(k, w, \delta_1(\text{Wr}_\tau^-)\ v_1, \delta_2(\text{Wr}_\tau^-)\ v_2) \in E_n^{\pi,-}[\![\tau]\!]\rho$.
   (b) If $(k, w, e_1, e_2) \in E_n^{+,\epsilon}[\![\tau]\!]\rho$, then $(k, w, \delta_1(\text{Wr}_\tau^-)\ e_1, \delta_2(\text{Wr}_\tau^-)\ e_2) \in E_n^{\pi,-}[\![\tau]\!]\rho$.

The most interesting cases of the proof (given below) are existential types in the first part and universal types in the second part because that is where the wrapping actually has to generate a fresh type. Technically, what happens in both cases is that we have some triple $(\tau_1, \tau_2, r) \in T^{\pi,+}[\![\Omega]\!]w'$, but would like it—or something equivalent—to be in $T^{-,\epsilon}[\![\Omega]\!]w''$, i.e., $T[\![\Omega]\!]w''$, where w'' must be some extension of w' that incorporates the new names α_1 and α_2. What we do is choose w'' such that it extends w' by a new semantic name α that is connected to the concrete names α_1 and α_2 as well as their representation types and is interpreted by the relation r. Then, we can use $(\alpha_1, \alpha_2, (w''.\rho^1(\alpha), w''.\rho^2(\alpha), V[\![\alpha]\!]w''.\rho))$, which has the form required by $T[\![\Omega]\!]w''$ and, since $w''.\rho$ maps α to r, carries the same relation as (τ_1, τ_2, r).

Proof
By primary induction on n and secondary induction on the derivation of $\Delta \vdash \tau$, note that δ_i only affects the type annotations (of function arguments and package types) inside the wrapping function. We show a few representative cases:

1. (a) • Case $\tau = \tau' \to \tau''$: $v_i = \lambda x.e_i$
   — To show: $(k, w, \delta_1(\lambda x.\lambda x'.\, \text{Wr}_{\tau''}^+\ (x\ (\text{Wr}_{\tau'}^-\ x')))\ v_1,$
   $\delta_2(\lambda x.\lambda x'.\, \text{Wr}_{\tau''}^+\ (x\ (\text{Wr}_{\tau'}^-\ x')))\ v_2) \in E_n^{-,\epsilon}[\![\tau' \to \tau'']\!]\rho$
   — Since
   $$\begin{aligned} & w.\sigma_i\,;\delta_i(\lambda x.\lambda x'.\, \text{Wr}_{\tau''}^+\ (x\ (\text{Wr}_{\tau'}^-\ x')))\ v_i \\ \hookrightarrow^1\ & w.\sigma_i\,;\lambda x'.\delta_i(\text{Wr}_{\tau''}^+)\,(v_i\ (\delta_i(\text{Wr}_{\tau'}^-)\,x')) \end{aligned}$$
   it suffices to show $(k-1, \lfloor w \rfloor, \lambda x'.\delta_1(\text{Wr}_{\tau''}^+)\,(v_1\ (\delta_1(\text{Wr}_{\tau'}^-)\,x')),$
   $\lambda x'.\delta_2(\text{Wr}_{\tau''}^+)\,(v_2\ (\delta_2(\text{Wr}_{\tau'}^-)\,x'))) \in V_n^{-,\epsilon}[\![\tau' \to \tau'']\!]\rho$.
   — Suppose $(k', w', v_3, v_4) \in V_n^{+,\epsilon}[\![\tau']\!]\rho$ where $(k', w') \sqsupseteq (k-1, \lfloor w \rfloor)$.
   — To show: $(k', w', \delta_1(\text{Wr}_{\tau''}^+)\,(v_1\ (\delta_1(\text{Wr}_{\tau'}^-)\,v_3)),$
   $\delta_2(\text{Wr}_{\tau''}^+)\,(v_2\ (\delta_2(\text{Wr}_{\tau'}^-)\,v_4)))) \in E_n^{-,\epsilon}[\![\tau'']\!]\rho$
   — So suppose $w'.\sigma_1\,;\delta_1(\text{Wr}_{\tau''}^+)\,(v_1\ (\delta_1(\text{Wr}_{\tau'}^-)\,v_3))$ terminates:
   $$\begin{aligned} & w'.\sigma_1\,;\delta_1(\text{Wr}_{\tau''}^+)\,(v_1\ (\delta_1(\text{Wr}_{\tau'}^-)\,v_3)) \\ \hookrightarrow^{j_1}\ & \sigma_1''\,;\delta_1(\text{Wr}_{\tau''}^+)\,(v_1\ v_3') \\ \hookrightarrow^{1}\ & \sigma_1''\,;\delta_1(\text{Wr}_{\tau''}^+)\,e_1[v_3'/x] \\ \hookrightarrow^{j_2}\ & \sigma_1\,;v_1' \end{aligned}$$
   and $j_1 + 1 + j_2 =: j < k'$.
   — By induction, $(k', w', \delta_1(\text{Wr}_{\tau'}^-)\,v_3, \delta_2(\text{Wr}_{\tau'}^-)\,v_4) \in E_n^{\pi,-}[\![\tau']\!]\rho$.

— This implies the existence of $(k'-j_1, w'') \sqsupseteq (k', w')$ such that

$$w'.\sigma_2 ; \delta_2(\mathrm{Wr}^+_{\tau''})\,(v_2\ (\delta_2(\mathrm{Wr}^-_{\tau'})\,v_4)) \hookrightarrow^* w''.\sigma_2 ; \delta_2(\mathrm{Wr}^+_{\tau''})\,(v_2\ v'_4)$$

with $w''.\sigma_1 = \sigma''_1$ and $(k'-j_1, w'', v'_3, v'_4) \in V^{\pi,-}_n[\![\tau']\!]\rho$.

— So by assumption and Closure Under World Extension, $(k'-j_1-1, \lfloor w''\rfloor, e_1[v'_3/x], e_2[v'_4/x]) \in E^{\pi,+}_n[\![\tau'']\!]\rho$.

— By induction, $(k'-j_1-1, \lfloor w''\rfloor, \delta_1(\mathrm{Wr}^+_{\tau''})\,e_1[v'_3/x], \delta_2(\mathrm{Wr}^+_{\tau''})\,e_2[v'_4/x]) \in E^{-,\epsilon}_n[\![\tau'']\!]\rho$.

— Hence, there exists $(k'-j, w''') \sqsupseteq (k'-j_1-1, \lfloor w''\rfloor)$ such that

$$w''.\sigma_2 ; \delta_1(\mathrm{Wr}^+_{\tau''})\,e_2[v'_4/x] \hookrightarrow^* w'''.\sigma_2 ; v'_2$$

with $w'''.\sigma_1 = \sigma_1$ and $(k'-j, w''', v'_1, v'_2) \in V^{-,\epsilon}_n[\![\tau'']\!]\rho$.

- Case $\tau = \exists\alpha.\tau'$: $v_i = \mathsf{pack}\ \langle\tau_i, v'_i\rangle$

— To show:

$$\begin{aligned}&(k, w, \delta_1(\lambda x.\,\mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{new}\ \alpha{\approx}\alpha\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^+_{\tau'}\ x'\rangle)\ v_1,\\ &\qquad \delta_2(\lambda x.\,\mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{new}\ \alpha{\approx}\alpha\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^+_{\tau'}\ x'\rangle)\ v_2)\\ &\in E^{-,\epsilon}_n[\![\exists\alpha.\tau']\!]\rho\end{aligned}$$

— So suppose the first configuration terminates:

$$\begin{aligned}& w.\sigma_1 ; \delta_1(\lambda x.\,\mathsf{unpack}\langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{new}\ \alpha{\approx}\alpha\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^+_{\tau'}\ x'\rangle)v_1\\ \hookrightarrow^1\ & w.\sigma_1 ; \mathsf{unpack}\ \langle\alpha, x'\rangle{=}v_1\ \mathsf{in}\ \mathsf{new}\ \alpha{\approx}\alpha\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \delta_1(\mathrm{Wr}^+_{\tau'})\,x'\rangle\\ \hookrightarrow^1\ & w.\sigma_1 ; \mathsf{new}\ \alpha{\approx}\tau_1\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \delta_1(\mathrm{Wr}^+_{\tau'})\,v'_1\rangle\\ \hookrightarrow^1\ & w.\sigma_1, \alpha_1{\approx}\tau_1 ; \mathsf{pack}\ \langle\alpha_1, \delta'_1(\mathrm{Wr}^+_{\tau'})\,v'_1\rangle\\ \hookrightarrow^{j'}\ & \sigma_1 ; \mathsf{pack}\ \langle\alpha_1, v''_1\rangle\end{aligned}$$

where $3 + j' =: j < k$ and $\delta'_1 := \delta_1, \alpha{\mapsto}\alpha_1$

— Note that

$$\begin{aligned}& w.\sigma_2 ; \delta_2(\lambda x.\,\mathsf{unpack}\langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{new}\ \alpha{\approx}\alpha\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^+_{\tau'}\ x'\rangle)v_2\\ \hookrightarrow^1\ & w.\sigma_2 ; \mathsf{unpack}\ \langle\alpha, x'\rangle{=}v_2\ \mathsf{in}\ \mathsf{new}\ \alpha{\approx}\alpha\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \delta_2(\mathrm{Wr}^+_{\tau'})\,x'\rangle\\ \hookrightarrow^1\ & w.\sigma_2 ; \mathsf{new}\ \alpha{\approx}\tau_2\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \delta_2(\mathrm{Wr}^+_{\tau'})\,v'_2\rangle\\ \hookrightarrow^1\ & w.\sigma_2, \alpha_2{\approx}\tau_2 ; \mathsf{pack}\ \langle\alpha_2, \delta'_2(\mathrm{Wr}^+_{\tau'})\,v'_2\rangle\end{aligned}$$

where $\delta'_2 := \delta_2, \alpha{\mapsto}\alpha_2$

— By assumption, we know $(k', w', v'_1, v'_2) \in V^{\pi,+}_n[\![\tau']\!]\rho, \alpha{\mapsto}r$ for some r with $(\tau_1, \tau_2, r) \in T^{\pi,+}_k[\![\Omega]\!]w$ and any $(k', w') \sqsupset (k, w)$.

— Let $w_\alpha := ((w.\sigma_1, \alpha_1{\approx}\tau_1), (w.\sigma_2, \alpha_2{\approx}\tau_2),$
$(w.\eta, \alpha{\mapsto}(\alpha_1, \alpha_2)), \lfloor w.\rho, \alpha{\mapsto}r\rfloor_{k-2})$, so $(k-2, w_\alpha) \sqsupset (k, w)$.

— Hence, $(k-2, w_\alpha, v'_1, v'_2) \in V^{\pi,+}_n[\![\tau']\!]\rho, \alpha{\mapsto}r$.

— By Closure Under World Extension, $(k-3, \lfloor w_\alpha\rfloor, v'_1, v'_2) \in V^{\pi,+}_n[\![\tau']\!]\rho, \alpha{\mapsto}r$ and thus $(k-3, \lfloor w_\alpha\rfloor, v'_1, v'_2) \in V^{\pi,+}_n[\![\tau']\!]\rho'$ for $\rho' := \lfloor\rho\rfloor_{k-2}, \alpha{\mapsto}r'$.

— Let $r' := (w_\alpha.\rho^1(\alpha), w_\alpha.\rho^2(\alpha), V_{k-2}[\![\alpha]\!]w_\alpha) = \lfloor r\rfloor_{k-2}$, so $(\alpha_1, \alpha_2, r') \in T^{-,\epsilon}_{k-2}[\![\Omega]\!]w_\alpha \subseteq T^{\pi}_{k-2}[\![\Omega]\!]w_\alpha$.

— Furthermore $(\delta_1, \delta_2, \lfloor\rho\rfloor_{k-2}) \in D^{\pi}_{k-2}[\![\Delta]\!]w_\alpha$ by Lemma 4, so $(\delta_1', \delta_2', \rho') \in D^{\pi}_{k-2}[\![\Delta, \alpha]\!]w_\alpha$.

— Hence, induction yields $(k-3, \lfloor w_\alpha\rfloor, \delta_1'(\mathrm{Wr}^{+}_{\tau'})\, v_1', \delta_2'(\mathrm{Wr}^{+}_{\tau'})\, v_2') \in E^{-,\epsilon}_n[\![\tau']\!]\rho'$.

— Because $w_\alpha.\sigma_1 = w.\sigma_1, \alpha_1{\approx}\tau_1$, this implies the existence of $(k-j, w') \sqsupseteq (k-3, \lfloor w_\alpha\rfloor)$ such that

$$w.\sigma_2, \alpha_2{\approx}\tau_2\,;\mathsf{pack}\ \langle\alpha_2, \delta_2'(\mathrm{Wr}^{+}_{\tau'})\, v_2'\rangle \hookrightarrow^{*} w'.\sigma_2\,;\mathsf{pack}\ \langle\alpha_2, v_2''\rangle$$

with $w'.\sigma_1 = \sigma_1$ and $(k-j, w', v_1'', v_2'') \in V^{-,\epsilon}_n[\![\tau']\!]\rho'$.

— By Closure Under World Extension, $(k'', w'', v_1'', v_2'') \in V^{-,\epsilon}_n[\![\tau']\!]\rho, \alpha{\mapsto}\lfloor r'\rfloor_{k-j}$ for any $(k'', w'') \sqsupseteq (k-j, w')$.

— Since $(\alpha_1, \alpha_2, \lfloor r'\rfloor_{k-j}) \in T^{-,\epsilon}_{k-j}[\![\Omega]\!]w'$ by Lemma 4, $(k{-}j, w', \mathsf{pack}\ \langle\alpha_1, v_1''\rangle\ \mathsf{as}\ \delta_1(\tau), \mathsf{pack}\ \langle\alpha_2, v_2''\rangle\ \mathsf{as}\ \delta_2(\tau)) \in V^{-,\epsilon}_n[\![\exists\alpha.\tau']\!]\rho$.

(b) • Suppose $w.\sigma_1\,;\delta_1(\mathrm{Wr}^{+}_{\tau})\, e_1$ terminates:

$$\begin{aligned} & w.\sigma_1\,;\delta_1(\mathrm{Wr}^{+}_{\tau})\, e_1 \\ \hookrightarrow^{j_1}\ & \sigma_1'\,;\delta_1(\mathrm{Wr}^{+}_{\tau})\, v_1 \\ \hookrightarrow^{j_2}\ & \sigma_1\,;v_1' \end{aligned}$$

and $j_1 + j_2 =: j < k$ steps

• So by assumption there exists $(k-j_1, w') \sqsupseteq (k, w)$ such that

$$w.\sigma_2\,;\delta_2(\mathrm{Wr}^{+}_{\tau})\, e_2 \hookrightarrow^{*} w'.\sigma_2\,;\delta_2(\mathrm{Wr}^{+}_{\tau})\, v_2$$

with $w'.\sigma_1 = \sigma_1'$ and $(k-j_1, w', v_1, v_2) \in V^{\pi,+}_n[\![\tau]\!]\rho$.

• By part (a), $(k-j_1, w', \delta_1(\mathrm{Wr}^{+}_{\tau})\, v_1, \delta_2(\mathrm{Wr}^{+}_{\tau})\, v_2) \in E^{-,\epsilon}_n[\![\tau]\!]\rho$.

• Consequently, there exists $(k-j, w'') \sqsupseteq (k-j_1, w')$ such that

$$w'.\sigma_2\,;\delta_2(\mathrm{Wr}^{+}_{\tau})\, v_2 \hookrightarrow^{*} w''.\sigma_2\,;v_2'$$

with $w''.\sigma_1 = \sigma_1$ and $(k-j, w'', v_1', v_2') \in V^{-,\epsilon}_n[\![\tau]\!]\rho$.

2. (a) • Case $\tau = \exists\alpha.\tau'$: $v_i = \mathsf{pack}\ \langle\tau_i, v_i'\rangle$

— To show: $(k, w, \delta_1(\lambda x.\,\mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^{-}_{\tau'}\ x'\rangle)\, v_1, \delta_2(\lambda x.\,\mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^{-}_{\tau'}\ x'\rangle)\, v_2) \in E^{\pi,-}_n[\![\exists\alpha.\tau']\!]\rho$

— So suppose $w.\sigma_1\,;\delta_1(\lambda x.\,\mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^{-}_{\tau'}\ x'\rangle)\ v_1$ terminates:

$$\begin{aligned} & w.\sigma_1\,;\delta_1(\lambda x.\,\mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^{-}_{\tau'}\ x'\rangle)\, v_1 \\ \hookrightarrow^{1}\ & w.\sigma_1\,;\mathsf{unpack}\ \langle\alpha, x'\rangle{=}v_1\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \delta_1(\mathrm{Wr}^{-}_{\tau'})\, x'\rangle \\ \hookrightarrow^{1}\ & w.\sigma_1\,;\mathsf{pack}\ \langle\tau_1, \delta_1'(\mathrm{Wr}^{-}_{\tau'})\, v_1'\rangle \\ \hookrightarrow^{j'}\ & \sigma_1\,;\mathsf{pack}\ \langle\tau_1, v_1''\rangle \end{aligned}$$

where $2 + j' =: j < k$ and $\delta_1' := \delta_1, \alpha{\mapsto}\tau_1$

— Note that

$$\begin{aligned} & w.\sigma_2\,;\delta_2(\lambda x.\,\mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^{-}_{\tau'}\ x'\rangle)\, v_2 \\ \hookrightarrow^{1}\ & w.\sigma_2\,;\mathsf{unpack}\ \langle\alpha, x'\rangle{=}v_2\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \delta_2(\mathrm{Wr}^{-}_{\tau'})\, x'\rangle \\ \hookrightarrow^{1}\ & w.\sigma_2\,;\mathsf{pack}\ \langle\tau_2, \delta_2'(\mathrm{Wr}^{-}_{\tau'})\, v_2'\rangle \end{aligned}$$

where $\delta_2' := \delta_2, \alpha{\mapsto}\tau_2$

— By assumption, we know $(k-2, \lfloor w \rfloor, v_1', v_2') \in V_n^{+,\epsilon}[\![\tau']\!]\rho, \alpha \mapsto r$ for some r with $(\tau_1, \tau_2, r) \in T_k^{+,\epsilon}[\![\Omega]\!]w \subseteq T_k^{\pi}[\![\Omega]\!]w$.

— Furthermore $(\delta_1, \delta_2, \lfloor \rho \rfloor) \in D_k^{\pi}[\![\Delta]\!]w$ by Lemma 4, and therefore, we get $(\delta_1', \delta_2', (\lfloor \rho \rfloor, \alpha \mapsto r)) \in D_k^{\pi}[\![\Delta, \alpha]\!]w$.

— Hence, induction yields
$(k-2, \lfloor w \rfloor, \delta_1'(\mathrm{Wr}_{\tau'}^{-})\, v_1', \delta_2'(\mathrm{Wr}_{\tau'}^{-})\, v_2') \in E_n^{\pi,-}[\![\tau']\!]\lfloor \rho \rfloor_k, \alpha \mapsto r$.

— Consequently, there exists $(k-j, w') \sqsupseteq (k-2, \lfloor w \rfloor)$ such that

$$w.\sigma_2; \mathsf{pack}\ \langle \tau_2, \delta_2'(\mathrm{Wr}_{\tau'}^{-})\, v_2' \rangle \hookrightarrow^* w'.\sigma_2; \mathsf{pack}\ \langle \tau_2, v_2'' \rangle$$

with $w'.\sigma_1 = \sigma_1$ and $(k-1-j, w', v_1'', v_2'') \in V_n^{\pi,-}[\![\tau']\!]\lfloor \rho \rfloor_k, \alpha \mapsto r$.

— For any $(k'', w'') \sqsupset (k-j, w')$, we get $(k'', w'', v_1'', v_2'') \in V_n^{\pi,-}[\![\tau']\!]\rho, \alpha \mapsto \lfloor r \rfloor$ by Closure Under World Extension.

— Since $(\tau_1, \tau_2, \lfloor r \rfloor) \in T_{k-j}^{\pi,-}[\![\Omega]\!]w'$ Lemma 4, this implies
$(k-j, w', \mathsf{pack}\ \langle \tau_1, v_1'' \rangle, \mathsf{pack}\ \langle \tau_2, v_2'' \rangle) \in V_n^{\pi,-}[\![\exists \alpha.\tau']\!]\rho$.

(b) Symmetric to (1b). □

Corollary 30 (*aka Theorem 28*)

1. If $\vdash e_1 \precsim^{\pi,+} e_2 : \tau$, then $\vdash \mathrm{Wr}^+\ e_1 \precsim^{-,\epsilon} \mathrm{Wr}^+\ e_2 : \tau$.
2. If $\vdash e_1 \precsim^{+,\epsilon} e_2 : \tau$, then $\vdash \mathrm{Wr}^-\ e_1 \precsim^{\pi,-} \mathrm{Wr}^-\ e_2 : \tau$.

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

Theorem 31 (*Inclusion for* $\precsim^{\pm}$)

1. If $\vdash e_1 \precsim e_2 : \tau$ or $\vdash e_1 \precsim^{\pi} e_2 : \tau$, 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 $\vdash e_1 \precsim^{\pi} e_2 : \tau$.

This theorem can equivalently be stated as $E^- \subseteq E \subseteq E^+$ and $E^- \subseteq E^{\pi} \subseteq E^+$. In Figure 7, it is depicted by the unlabeled arrows between different relations, which represent inclusion.

Corollary 32 (*aka Theorem 25*)

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

Proof

Follows immediately from Theorems 28 and 31. □

Similarly, the following follows from Theorem 31 together with the Fundamental Property of $\precsim$:

Corollary 33 (*Fundamental Property of* $\precsim^{+}$)

If $\vdash e : \tau$ and $w \in \mathrm{World}_k$, then $(k, w, e, e) \in E_{k+1}^{+}[\![\tau]\!]$.

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 33 directly. For a similar reason, we cannot show any such property for E^- at all.

The $\in$-operators in Figure 7 sum up the fundamental properties for the respective relations, i.e., which class of terms (G terms or F terms) are included in which relation.

LR-Substitution does not hold for the polarized relations. Consider the case where $\tau = \alpha \to \alpha$. Then, for instance, $V_n^+[\![\tau]\!]\rho, \alpha \mapsto (\rho^1(\tau'), \rho^2(\tau'), V_n^+[\![\tau']\!]\rho)$ tells us something about how its elements behave when applied to arguments out of $V_n^+[\![\tau']\!]\rho$. $V_n^+[\![\tau[\tau'/\alpha]]\!]\rho$, on the other hand, only tells us something about how its elements behave when applied to arguments out of $V_n^-[\![\tau']\!]\rho$.

8.2 Example

Getting back to our motivating example from the beginning of the section, it is essentially straightforward to prove that $\vdash f_1 \lesssim^+ f_2 : \forall\alpha.\tau_\alpha$. 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 33.

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

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^π. If we had tried to do this proof in E^π, the types τ_1 and τ_2 would have been related by T^π only, which would give us too little information to proceed with the necessary case distinction.

9 Recursive types

In this section, we consider an interesting and non-trivial extension of G with a ubiquitous feature—namely, *(iso-)recursive types*. We call the extended language G^μ (see Figure 8). The definition of contextual equivalence does not change (except there are more contexts), but of course we must extend our logical relation, our definition of wrapping, and our meta-theory, to handle recursive types.

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

Types	$\tau ::= \ldots \mid \mu\alpha.\tau$
Values	$v ::= \ldots \mid \mathsf{roll}\ v\ \mathsf{as}\ \tau$
Terms	$e ::= \ldots \mid \mathsf{roll}\ e\ \mathsf{as}\ \tau \mid \mathsf{unroll}\ e$
Evaluation Ctxt's	$E ::= \ldots \mid \mathsf{roll}\ E\ \mathsf{as}\ \tau \mid \mathsf{unroll}\ E$

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

$$\ldots$$

$$(\textsc{ERoll})\frac{\Delta;\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Delta;\Gamma \vdash \mathsf{roll}\ e\ \mathsf{as}\ \mu\alpha.\tau : \mu\alpha.\tau} \qquad (\textsc{EUnroll})\frac{\Delta;\Gamma \vdash e : \mu\alpha.\tau}{\Delta;\Gamma \vdash \mathsf{unroll}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

$$\ldots$$

$$\sigma; E[\mathsf{unroll}\,(\mathsf{roll}\ v\ \mathsf{as}\ \tau)] \hookrightarrow \sigma; E[v]$$

Fig. 8. Syntax and semantics of G^{μ} (excerpt).

relation at a recursive type:

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

For $\iota \in \{\epsilon, \pi\}$—i.e., for the non-parametric and parametric forms of the logical relation—the above two formulations are equivalent due to LR-Substitution. 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}. Adapting the proofs of soundness, the fundamental property, and related lemmas from Section 4, to G^{μ} is straightforward.

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 9 shows the new definition. We first index the wrapping by an environment φ that maps each recursive type variable α to the appropriate wrapping and the corresponding syntactic type (we write $\varphi^{\text{val}}(\alpha)$ for the former and $\varphi^{\text{typ}}(\alpha)$ for the latter). Roughly, the wrapping at type $\mu\alpha.\tau$ under environment φ is a recursive function F, defined in terms of the wrapping at τ under environment $\varphi, \alpha \mapsto (\mu\alpha.\tau, F)$.

$$\begin{array}{lll}
F^{\varphi}_{\mu\alpha.\tau} & \stackrel{\text{def}}{=} \mathsf{fix}\, f(x').\langle \lambda x{:}(\mu\alpha.\tau).\,\mathsf{roll}\ \mathrm{Wr}^{+}_{\tau}(\varphi, \alpha \mapsto (\mu\alpha.\tau, f))\,(\mathsf{unroll}\ x)\ \mathsf{as}\ \mu\alpha.\tau, & \\
& \qquad \lambda x{:}(\mu\alpha.\tau).\,\mathsf{roll}\ \mathrm{Wr}^{-}_{\tau}(\varphi, \alpha \mapsto (\mu\alpha.\tau, f))\,(\mathsf{unroll}\ x)\ \mathsf{as}\ \mu\alpha.\tau\rangle & \\
& \qquad : \mathsf{unit} \to ((\mu\alpha.\tau) \to (\mu\alpha.\tau)) \times ((\mu\alpha.\tau) \to (\mu\alpha.\tau)) & \\
\\
\mathrm{Wr}^{+}_{\alpha}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}\varphi^{\mathrm{typ}}(\alpha).(\varphi^{\mathrm{val}}(\alpha)\ ()).1\ x & (\text{if } \alpha \in \mathrm{dom}(\varphi)) \\
\mathrm{Wr}^{-}_{\alpha}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}\varphi^{\mathrm{typ}}(\alpha).(\varphi^{\mathrm{val}}(\alpha)\ ()).2\ x & (\text{if } \alpha \in \mathrm{dom}(\varphi)) \\
\mathrm{Wr}^{\pm}_{\alpha}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}\alpha.x & (\text{if } \alpha \notin \mathrm{dom}(\varphi)) \\
\mathrm{Wr}^{\pm}_{b}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}b.x & \\
\mathrm{Wr}^{\pm}_{\tau_1 \times \tau_2}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\tau_1 \times \tau_2).\langle \mathrm{Wr}^{\pm}_{\tau_1}\ \varphi\,(x.1)), \mathrm{Wr}^{\pm}_{\tau_2}\ \varphi\,(x.2)\rangle & \\
\mathrm{Wr}^{\pm}_{\tau_1 \to \tau_2}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\tau_1 \to \tau_2).\lambda x'{:}\tau_1.\,\mathrm{Wr}^{\pm}_{\tau_2}\ \varphi\,(x\ (\mathrm{Wr}^{\mp}_{\tau_1}\ \varphi\, x')) & \\
\mathrm{Wr}^{\pm}_{\forall\alpha.\tau}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\forall\alpha.\tau).\Lambda\alpha.\,\mathsf{new}^{\mp}\ \alpha\ \mathsf{in}\ \mathrm{Wr}^{\pm}_{\tau}\ \varphi\,(x\ \alpha) & \\
\mathrm{Wr}^{\pm}_{\exists\alpha.\tau}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\exists\alpha.\tau).\,\mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{new}^{\pm}\ \alpha\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Wr}^{\pm}_{\tau}\ \varphi\, x'\rangle\ \mathsf{as}\ \exists\alpha.\tau & \\
\mathrm{Wr}^{+}_{\mu\alpha.\tau}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\mu\alpha.\tau).(F^{\varphi}_{\mu\alpha.\tau}\ ()).1\ x & \\
\mathrm{Wr}^{-}_{\mu\alpha.\tau}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\mu\alpha.\tau).(F^{\varphi}_{\mu\alpha.\tau}\ ()).2\ x & \\
\\
\mathrm{Wr}^{\pm}_{\tau} & \stackrel{\text{def}}{=} \mathrm{Wr}^{\pm}_{\tau}\ \emptyset &
\end{array}$$

Fig. 9. Wrapping for G^{μ}.

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 polarized variant of the so-called *syntactic projection* function associated with a recursive type (Birkedal & Harper, 1999).

Note that the definition of $F^{\varphi}_{\mu\alpha.\tau}$ takes a unit argument merely for simplicity so that we may encode two mutually recursive functions in terms of a single fix (whose encoding appears in Section A.5). Note also 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 $\mathrm{Wr}^{\pm}_{\tau}\ \emptyset$, we can show that our old Wrapping Theorems for G (Theorems 25 and 28) continue to hold for G^{μ}.

First of all, Lemma 24 still holds, but we can generalize it as follows:

Lemma 34
If $\Delta, \mathrm{dom}(\varphi) \vdash \tau$ and for all $\alpha \in \mathrm{dom}(\varphi)$ both $\Delta \vdash \varphi^{\mathrm{typ}}(\alpha)$ and

$$\Delta; \epsilon \vdash \varphi^{\mathrm{val}}(\alpha) : \mathsf{unit} \to (\varphi^{\mathrm{typ}}(\alpha) \to \varphi^{\mathrm{typ}}(\alpha)) \times (\varphi^{\mathrm{typ}}(\alpha) \to \varphi^{\mathrm{typ}}(\alpha)),$$

then $\Delta; \epsilon \vdash \mathrm{Wr}^{\pm}_{\tau}\ \varphi : \varphi^{\mathrm{typ}}(\tau) \to \varphi^{\mathrm{typ}}(\tau)$.

The next is a substitution lemma for the wrapping. Taking τ' to be τ (which is how it will be used), it says that wrapping at the unfolding of a recursive type $\mu\alpha.\tau$ (i.e., at $\tau[\mu\alpha.\tau/\alpha]$), relative to some environment φ, is *syntactically* the same as "moving the unfolding into the environment" and then wrapping at τ. This lemma is important for the recursive type case in the Wrapping Theorem.

Lemma 35 (*WR-Substitution*)
If $\varphi' = \varphi, \alpha \mapsto (\mu\alpha.\tau, F^{\varphi}_{\mu\alpha.\tau})$, then $\mathrm{Wr}^{\pm}_{\tau'}\,\varphi' = \mathrm{Wr}^{\pm}_{\tau'[\mu\alpha.\tau/\alpha]}\,\varphi$.

Proof
By induction on τ'. □

The proof of the Wrapping Theorem for G^{μ} is obtained from the one for G by simply extending the case analysis. Note that the wrapping theorem is stated for an empty environment φ (recall that $\mathrm{Wr}^{\pm}_{\tau}$ is just short for $\mathrm{Wr}^{\pm}_{\tau}\,\emptyset$). This may seem not general enough at first, because in the case where $\tau = \mu\alpha.\tau'$, we need an induction hypothesis that talks about wrapping relative to the non-empty environment $\varphi := (\alpha \mapsto (\tau, F^{\emptyset}_{\tau}))$. This is exactly where Lemma 35 comes in: it tells us that the terms involving $\mathrm{Wr}^{\pm}_{\tau'}\,\varphi$ that we are interested in are the same as the terms involving $\mathrm{Wr}^{\pm}_{\tau'[\tau/\alpha]}\,\emptyset$ that we know are related by the induction hypothesis.

Proof

1. (a) Case $\tau = \mu\alpha.\tau'$: $v_i = \mathsf{roll}\ v_i'$
    - To show: $(k, w, \delta_1(\lambda x.(F^{\emptyset}_{\tau}\ ()).1\ x)\ v_1, \delta_2(\lambda x.(F^{\emptyset}_{\tau}\ ()).1\ x)\ v_2) \in E^{-,\epsilon}_{n}[\![\mu\alpha.\tau']\!]\rho$
    - So suppose $w.\sigma_1 ; \delta_1(\lambda x.(F^{\emptyset}_{\tau}\ ()).1\ x)\ v_1$ terminates

    $$\begin{array}{ll} & w.\sigma_1 ; \delta_1(\lambda x.(F^{\emptyset}_{\tau}\ ()).1\ x)\ v_1 \\ \hookrightarrow^{1} & w.\sigma_1 ; (\delta_1(F^{\emptyset}_{\tau})\ ()).1\ v_1 \\ \hookrightarrow^{j_c} & w.\sigma_1 ; \mathsf{roll}\ \delta_1(\mathrm{Wr}^{+}_{\tau'}(\alpha \mapsto (\tau, F^{\emptyset}_{\tau})))\ (\mathsf{unroll}\ v_1) \\ \hookrightarrow^{1} & w.\sigma_1 ; \mathsf{roll}\ \delta_1(\mathrm{Wr}^{+}_{\tau'}(\alpha \mapsto (\tau, F^{\emptyset}_{\tau})))\ v_1' \\ \hookrightarrow^{j'} & \sigma_1 ; \mathsf{roll}\ v_1'' \end{array}$$

    and $1 + j_c + 1 + j' =: j < k$.
    - Note that

    $$\begin{array}{ll} & w.\sigma_2 ; \delta_2(\lambda x.(F^{\emptyset}_{\tau}\ ()).1\ x)\ v_2 \\ \hookrightarrow^{1} & w.\sigma_2 ; (\delta_2(F^{\emptyset}_{\tau})\ ()).1\ v_2 \\ \hookrightarrow^{j_c} & w.\sigma_2 ; \mathsf{roll}\ \delta_2(\mathrm{Wr}^{+}_{\tau'}(\alpha \mapsto (\tau, F^{\emptyset}_{\tau})))\ (\mathsf{unroll}\ v_2) \\ \hookrightarrow^{1} & w.\sigma_2 ; \mathsf{roll}\ \delta_2(\mathrm{Wr}^{+}_{\tau'}(\alpha \mapsto (\tau, F^{\emptyset}_{\tau})))\ v_2' \end{array}$$

    - By assumption, we know $(k - j, \lfloor w \rfloor, v_1', v_2') \in V^{\pi,+}_{k}[\![\tau'[\tau/\alpha]]\!]\rho$.
    - By induction,
    $(k - j, \lfloor w \rfloor, \delta_1(\mathrm{Wr}^{+}_{\tau'[\tau/\alpha]})\ v_1', \delta_2(\mathrm{Wr}^{+}_{\tau'[\tau/\alpha]})\ v_2') \in E^{-,\epsilon}_{k}[\![\tau'[\tau/\alpha]]\!]\rho$.
    - By Lemma 35, $\mathrm{Wr}^{+}_{\tau'[\tau/\alpha]} = \mathrm{Wr}^{+}_{\tau'}(\alpha \mapsto (\tau, F^{\emptyset}_{\tau}))$.
    - Consequently, there exists $(k - j, w') \sqsupseteq (k - j_c - 1, \lfloor w \rfloor)$ such that

    $$w.\sigma_2 ; \mathsf{roll}\ \delta_2(\mathrm{Wr}^{+}_{\tau'}(\alpha \mapsto (\tau, F^{\emptyset}_{\tau})))\ v_2' \hookrightarrow^{*} w'.\sigma_2 ; \mathsf{roll}\ v_2''$$

    with $w'.\sigma_1 = \sigma_1$ and $(k - j, w', v_1'', v_2'') \in V^{-,\epsilon}_{k}[\![\tau'[\tau/\alpha]]\!]\rho$.
    - By Closure Under World Extension, the latter implies
    $(k - j, w', \mathsf{roll}\ v_1'', \mathsf{roll}\ v_2'') \in V^{-,\epsilon}_{n}[\![\tau]\!]\rho$.

    (b) As before.
2. (a) Case $\tau = \mu\alpha.\tau'$: symmetric to respective case of part (1)
    (b) As before. □

10 Towards full abstraction

The definition of the parametric relation E^{π} (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^{μ}, i.e., System F extended with pairs, existentials and iso-recursive types (Ahmed, 2006). The main difference is the presence of worlds, but they are not actually used in a particularly interesting way in E^{π}. Therefore, one might expect that any two F^{μ} terms related by the hypothetical $E_{\mathrm{F}^{\mu}}$ would also be related by E^{π} and vice versa.

However, this is not obvious: G^{μ} is more expressive than F^{μ}, in the sense that 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^{μ} (as would be needed for function arguments) that invalidates a proof approach like the one taken by Ahmed & Blume (2008).

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

$$\vdash e_1 \equiv_{\mathrm{F}^{\mu}} e_2 : \tau \Leftrightarrow \vdash \mathrm{Wr}_{\tau}^{+}\, e_1 \equiv \mathrm{Wr}_{\tau}^{+}\, e_2 : \tau$$

(The semantics of F^{μ} can be obtained from G^{μ} by restricting Δ to simple variable components, ignoring all the rules related to cast and new as well as the conversion rule EConv, and dropping the type store from the reduction relation. Contextual approximation then is defined as for G^{μ} except that it does not mention a type store and the universally quantified contexts must have type $(\Delta;\Gamma;\tau) \rightsquigarrow (\epsilon;\epsilon;\tau')$.) This equivalence is even stronger than the one about logical relatedness in $E_{\mathrm{F}^{\mu}}$ and E^{π} because $\precsim$ is only sound with respect to contextual approximation, not complete.

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

Theorem 36 (*Equivalence Reflection*)
If $\Delta;\Gamma \vdash_{\mathrm{F}^{\mu}} e_1 : \tau$ and $\Delta;\Gamma \vdash_{\mathrm{F}^{\mu}} e_2 : \tau$ and $\Delta;\Gamma \vdash \mathrm{Wr}_{\tau}^{+}\, e_1 \equiv \mathrm{Wr}_{\tau}^{+}\, e_2 : \tau$, then $\Delta;\Gamma \vdash e_1 \equiv_{\mathrm{F}^{\mu}} e_2 : \tau$.

We present its proof in the remainder of this section.

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. Here is a trivial example exploiting this:

$$\begin{aligned} e_1 &:= \lambda f{:}(\mathsf{unit} \to \mathsf{unit}).f\ () \\ e_2 &:= \lambda f{:}(\mathsf{unit} \to \mathsf{unit}).() \end{aligned}$$

Clearly, e_1 and e_2 are contextually equivalent in F. Wrapping basically leaves them unmodified, because their type is simple. However, e_1 and e_2 are not contextually equivalent in G, since a G context can apply them to a diverging function.

$$\begin{array}{lll}
G^{\varphi}_{\mu\alpha.\tau} & \stackrel{\text{def}}{=} \mathsf{fix}\, f(x').\langle \lambda x{:}(\mu\alpha.\tau).\, \mathsf{roll}\ \mathrm{Sp}^{+}_{\tau}(\varphi, \alpha \mapsto (\mu\alpha.\tau, f))\,(\mathsf{unroll}\ x)\ \mathsf{as}\ \mu\alpha.\tau, & \\
 & \qquad \lambda x{:}(\mu\alpha.\tau).\, \mathsf{roll}\ \mathrm{Sp}^{-}_{\tau}(\varphi, \alpha \mapsto (\mu\alpha.\tau, f))\,(\mathsf{unroll}\ x)\ \mathsf{as}\ \mu\alpha.\tau\rangle & \\
 & \qquad : \mathsf{unit} \to ((\mu\alpha.\tau) \to (\mu\alpha.\tau)) \times ((\mu\alpha.\tau) \to (\mu\alpha.\tau)) & \\
 & & \\
\mathrm{Sp}^{+}_{\alpha}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}\varphi^{\mathrm{typ}}(\alpha).(\varphi^{\mathrm{val}}(\alpha)\ ()).1\ x & (\text{if } \alpha \in \mathrm{dom}(\varphi)) \\
\mathrm{Sp}^{-}_{\alpha}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}\varphi^{\mathrm{typ}}(\alpha).(\varphi^{\mathrm{val}}(\alpha)\ ()).2\ x & (\text{if } \alpha \in \mathrm{dom}(\varphi)) \\
\mathrm{Sp}^{\pm}_{\alpha}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}\alpha.x & (\text{if } \alpha \notin \mathrm{dom}(\varphi)) \\
\mathrm{Sp}^{\pm}_{b}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}b.x & \\
\mathrm{Sp}^{\pm}_{\tau_1\times\tau_2}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\tau_1 \times \tau_2).\langle \mathrm{Sp}^{\pm}_{\tau_1}\ \varphi\,(x.1)), \mathrm{Sp}^{\pm}_{\tau_2}\ \varphi\,(x.2)\rangle & \\
\mathrm{Sp}^{\pm}_{\tau_1\to\tau_2}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\tau_1 \to \tau_2).\lambda x'{:}\tau_1.\, \mathrm{Sp}^{\pm}_{\tau_2}\ \varphi\,(x\ (\mathrm{Sp}^{\mp}_{\tau_1}\ \varphi\, x')) & \\
\mathrm{Sp}^{\pm}_{\forall\alpha.\tau}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\forall\alpha.\tau).\Lambda\alpha.\, \mathrm{Sp}^{\pm}_{\tau}\ \varphi\,(x\ \alpha) & \\
\mathrm{Sp}^{\pm}_{\exists\alpha.\tau}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\exists\alpha.\tau).\, \mathsf{unpack}\ \langle\alpha, x'\rangle{=}x\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, \mathrm{Sp}^{\pm}_{\tau}\ \varphi\, x'\rangle\ \mathsf{as}\ \exists\alpha.\tau & \\
\mathrm{Sp}^{+}_{\mu\alpha.\tau}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\mu\alpha.\tau).(G^{\varphi}_{\mu\alpha.\tau}\ ()).1\ x & \\
\mathrm{Sp}^{-}_{\mu\alpha.\tau}\ \varphi & \stackrel{\text{def}}{=} \lambda x{:}(\mu\alpha.\tau).(G^{\varphi}_{\mu\alpha.\tau}\ ()).2\ x & \\
 & & \\
\mathrm{Sp}^{\pm}_{\tau} & \stackrel{\text{def}}{=} \mathrm{Sp}^{\pm}_{\tau}\ \emptyset &
\end{array}$$

Fig. 10. Simple wrapping for G^{μ} (new-erasure of the proper wrapping).

10.1 Equivalence reflection

Assuming $\Delta;\Gamma \vdash_{\mathrm{F}^{\mu}} e_1 : \tau$ and $\Delta;\Gamma \vdash_{\mathrm{F}^{\mu}} e_2 : \tau$, we want to show:

$$\Delta;\Gamma \vdash \mathrm{Wr}^{+}_{\tau}\ e_1 \equiv_{\mathrm{G}^{\mu}} \mathrm{Wr}^{+}_{\tau}\ e_2 : \tau \Rightarrow \Delta;\Gamma \vdash e_1 \equiv_{\mathrm{F}^{\mu}} e_2 : \tau$$

We will show the contrapositive. Since F^{μ} is a fragment of G^{μ}, it suffices to show that any context C that can distinguish e_1 and e_2 in F^{μ} will also distinguish their positive wrappings in G^{μ}. We do this in two steps. First, we prove that C will distinguish their *simple wrappings* (Lemma 40). The simple wrapping, $\mathrm{Sp}^{\pm}_{\tau}$, whose definition is given in Figure 10, is the new-erasure of the proper wrapping, i.e., obtained by replacing any $\mathsf{new}\,\alpha{\approx}\tau'$ in e' in $\mathrm{Wr}^{\pm}_{\tau}$ by $e'[\tau'/\alpha]$. In the terms of Birkedal & Harper (1999), it is precisely the syntactic projection function associated with the type τ (hence Sp for "Syntactic projection"). Subsequently, we prove that distinguishing the simple wrappings implies distinguishing the proper wrappings (Lemma 46).

For the first part, we actually show something stronger, namely, the so-called *syntactic minimal invariance* property (Birkedal & Harper, 1999), which says that the Sp function at any type is contextually equivalent to the identity and thus that any term e is contextually equivalent in G^{μ} to its simple wrapping. We do this with the help of our non-parametric logical relation, which is sound with respect to contextual approximation.

Lemma 37 (*SP-Substitution*)
If $\varphi' = \varphi, \alpha \mapsto (\mu\alpha.\tau, G^{\varphi}_{\mu\alpha.\tau})$, then $\mathrm{Sp}^{\pm}_{\tau'}\ \varphi' = \mathrm{Sp}^{\pm}_{\tau'[\mu\alpha.\tau/\alpha]}\ \varphi$.

Lemma 38
Suppose $w_0 \in \mathrm{World}_n$, $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$ and $(k, w) \sqsupseteq (n, w_0)$, where $\Delta \vdash \tau$.

1. If $(k, w, v_1, v_2) \in V_n[\![\tau]\!]\rho$,
   then $(k, w, v_1, \delta_2(\mathrm{Sp}_\tau^\pm)\, v_2) \in E_n[\![\tau]\!]\rho$ and $(k, w, \delta_1(\mathrm{Sp}_\tau^\pm)\, v_1, v_2) \in E_n[\![\tau]\!]\rho$.
2. If $(k, w, e_1, e_2) \in E_n[\![\tau]\!]\rho$,
   then $(k, w, e_1, \delta_2(\mathrm{Sp}_\tau^\pm)\, e_2) \in E_n[\![\tau]\!]\rho$ and $(k, w, \delta_1(\mathrm{Sp}_\tau^\pm)\, e_1, e_2) \in E_n[\![\tau]\!]\rho$.

Proof
By primary induction on n and secondary induction on the derivation of $\Delta \vdash \tau$. □

Lemma 39
If $\Delta; \Gamma \vdash e : \tau$, then $\Delta; \Gamma \vdash e \equiv \mathrm{Sp}_\tau^\pm\ e : \tau$.

Proof
We show $\Delta; \Gamma \vdash e \lesssim \mathrm{Sp}_\tau^\pm\ e : \tau$. The proof of $\Delta; \Gamma \vdash \mathrm{Sp}_\tau^\pm\ e \lesssim e : \tau$ is symmetric. The claim then follows by Soundness.

- Suppose $w_0 \in \mathrm{World}_n$, $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$, $(k, \gamma_1, \gamma_2) \in G_n[\![\Gamma]\!]\rho$, and $(k, w) \sqsupseteq (n, w_0)$.
- By the Fundamental Property, we know $\Delta; \Gamma \vdash e \lesssim e : \tau$.
- Instantiating this yields $(k, w, \delta_1\gamma_1(e), \delta_2\gamma_2(e)) \in E_n[\![\tau]\!]\rho$.
- By Lemma 38, $(k, w, \delta_1\gamma_1(e), \delta_2(\mathrm{Sp}_\tau^\pm)\, \delta_2\gamma_2(e)) \in E_n[\![\tau]\!]\rho$.
- Note that $\delta_2(\mathrm{Sp}_\tau^\pm)\, \delta_2\gamma_2(e) = \delta_2\gamma_2(\mathrm{Sp}_\tau^\pm\ e)$. □

Lemma 40
1. If $\Delta; \Gamma \vdash e : \tau$, $\vdash C : (\Delta; \Gamma; \tau) \rightsquigarrow (\epsilon; \epsilon; \tau')$, and $\epsilon; C[e]\downarrow$, then $\epsilon; C[\mathrm{Sp}_\tau^\pm\ e]\downarrow$.
2. If $\Delta; \Gamma \vdash e : \tau$, $\vdash C : (\Delta; \Gamma; \tau) \rightsquigarrow (\epsilon; \epsilon; \tau')$, and $\epsilon; C[e]\uparrow$, then $\epsilon; C[\mathrm{Sp}_\tau^\pm\ e]\uparrow$.

Proof
Follows from Lemma 39. □

The second part (Lemma 46) can be proven in a more direct way. Intuitively, the property holds because the only difference between the reduction of $C[\mathrm{Sp}_\tau^\pm\ e]$ and the reduction of $C[\mathrm{Wr}_\tau^\pm\ e]$ is that during the latter fresh type names are being generated and substituted. Since we assume C to be cast-free, there is no way for these type names to affect the reduction and thus the termination behavior. We will only sketch the proof and not give formal details, as this would be a very tedious job here and not reveal any insights.

The idea is to use a simulation that relates a term e_1 to a term e_2 iff e_1 is the new-erasure of e_2, i.e., e_1 is obtained from e_2 by dropping all occurrences of new. Thus, in particular, the simulation relates the simple wrapping of a term to its proper wrapping.

The definition of Erase, the new-erasure, is trivial. Its only interesting case is

$$\mathrm{Erase}(\mathsf{new}\ \alpha{\approx}\tau\ \mathsf{in}\ e) \stackrel{\mathrm{def}}{=} \mathrm{Erase}(e[\tau/\alpha]).$$

For all the other language constructs, the definition just recurses on the subterms. It is easy to see that Erase satisfies standard congruence and substitution properties:

Lemma 41
If $e_1 = \mathrm{Erase}(e_2)$ and C is new-free, then $C[e_1] = \mathrm{Erase}(C[e_2])$.

Lemma 42

1. If $e_1 = \mathrm{Erase}(e_2)$ and $e'_1 = \mathrm{Erase}(e'_2)$, then $e_1[e'_1/x] = \mathrm{Erase}(e_2[e'_2/x])$.
2. If $e_1 = \mathrm{Erase}(e_2)$, then $e_1[\tau/\alpha] = \mathrm{Erase}(e_2[\tau/\alpha])$.

The simulation argument is the following (where $\hookrightarrow^+$ denotes a reduction sequence with at least one reduction):

Lemma 43
If e_1 is cast-free and $e_1 = \sigma_2^*(\mathrm{Erase}(e_2))$ and $\sigma_1; e_1 \hookrightarrow \sigma_1; e'_1$, then there are σ'_2 and e'_2 with $e'_1 = \sigma_2'^*(\mathrm{Erase}(e'_2))$ cast-free and $\sigma_2; e_2 \hookrightarrow^+ \sigma'_2; e'_2$.

This already yields the second part of Lemma 46. For the first part, we need one more lemma and an easy induction.

Lemma 44
If $v = \sigma_2^*(\mathrm{Erase}(e))$, then $\sigma_2; e\downarrow$.

Lemma 45
If e_1 is cast-free and $e_1 = \sigma_2^*(\mathrm{Erase}(e_2))$ and $\sigma_1; e_1\downarrow$, then $\sigma_2; e_2\downarrow$.

Proof
By induction on the length of the reduction sequence, using Lemmas 44 and 43. □

Lemma 46
Suppose e and C are both cast- and new-free.

1. If $\Delta; \Gamma \vdash e : \tau$, $\vdash C : (\Delta; \Gamma; \tau) \rightsquigarrow (\epsilon; \epsilon; \tau')$ and $\epsilon; C[\mathrm{Sp}_\tau^\pm\ e]\downarrow$, then $\epsilon; C[\mathrm{Wr}_\tau^\pm\ e]\downarrow$.
2. If $\Delta; \Gamma \vdash e : \tau$, $\vdash C : (\Delta; \Gamma; \tau) \rightsquigarrow (\epsilon; \epsilon; \tau')$ and $\epsilon; C[\mathrm{Sp}_\tau^\pm\ e]\uparrow$, then $\epsilon; C[\mathrm{Wr}_\tau^\pm\ e]\uparrow$.

Proof
Since $C[\mathrm{Sp}_\tau^\pm\ e] = \mathrm{Erase}(C[\mathrm{Wr}_\tau^\pm\ e])$, the first part follows from Lemma 45 and the second from Lemma 43. □

Finally, we can prove the actual theorem:

Theorem 37 (*Equivalence Reflection*)
If $\Delta; \Gamma \vdash_{\mathrm{F}^\mu} e_1 : \tau$, $\Delta; \Gamma \vdash_{\mathrm{F}^\mu} e_2 : \tau$ and $\Delta; \Gamma \vdash \mathrm{Wr}_\tau^\pm\ e_1 \equiv \mathrm{Wr}_\tau^\pm\ e_2 : \tau$, then $\Delta; \Gamma \vdash e_1 \equiv_{\mathrm{F}^\mu} e_2 : \tau$.

Proof
Assume that $\Delta; \Gamma \vdash e_1 \equiv_{\mathrm{F}^\mu} e_2 : \tau$ does not hold, i.e., e_1 and e_2 are not contextually equivalent in F^μ. Then, there is an F^μ-context C that can tell them apart: say, $C[e_1]\downarrow$ and $C[e_2]\uparrow$. Note that C also is a valid G context. It is easy to see that C will distinguish e_1 and e_2 in G, too: $\epsilon; C[e_1]\downarrow$ and $\epsilon; C[e_2]\uparrow$. Using Lemma 40 and then Lemma 46, this implies that C also distinguishes their wrappings: $\epsilon; C[\mathrm{Wr}_\tau^\pm\ e_1]\downarrow$ and $\epsilon; C[\mathrm{Wr}_\tau^\pm\ e_2]\uparrow$. Consequently, $\Delta; \Gamma \vdash \mathrm{Wr}_\tau^\pm\ e_1 \equiv \mathrm{Wr}_\tau^\pm\ e_2 : \tau$ does not hold either. □

11 Incompleteness of the logical relation

While our logical relation for G^μ is sound with respect to contextual approximation, it is not complete. There are at least two reasons why.

First of all, we have defined our logical relation in such a way as to model a fairly general notion of non-parametricity, not tied specifically to the cast operator *per se*. Consequently, we conjecture that our logical relation (modulo potential minor tweaks) would generalize to soundly model a language with a typecase mechanism instead of a cast operator. (As explained in the introduction, we have chosen to study cast because it is simpler yet still interesting.) However, typecase is strictly more powerful than cast, in the sense that typecase is capable of distinguishing between more programs. In particular, with typecase one can *pattern-match* on an abstract type α, which one can not always do with cast (see the example below). Thus, there are programs that we cannot prove equivalent in our model—because they are not contextually equivalent in the presence of typecase—but that (we conjecture) *are* contextually equivalent in the presence of cast, and this clearly leads our model to be incomplete with respect to G^μ.

Consider the following example:

$$\begin{aligned}
\tau &:= \exists\beta.\,(\mathsf{int}\times\mathsf{int}\to\beta)\times(\beta\to\mathsf{int})\times(\beta\to\mathsf{int})\\
e_1 &:= \mathsf{new}\,\alpha{\approx}\mathsf{int}\ \mathsf{in}\ \mathsf{pack}\,\langle\alpha\times\alpha,\langle\lambda p.p,\ \lambda x.(x.1),\ \lambda x.(x.2)\rangle\rangle\ \mathsf{as}\ \tau\\
e_2 &:= \mathsf{new}\,\alpha{\approx}(\mathsf{int}\times\mathsf{int})\ \mathsf{in}\ \mathsf{pack}\,\langle\alpha,\langle\lambda p.p,\ \lambda x.(x.1),\ \lambda x.(x.2)\rangle\rangle\ \mathsf{as}\ \tau
\end{aligned}$$

We strongly conjecture that e_1 and e_2 are contextually equivalent in G^μ: Although the type components of the existential packages returned by e_1 and e_2—namely, $\alpha\times\alpha$ and α, respectively—are structurally different, there seems to be no way to observe this using cast. Specifically, after unpacking the existential and binding a name (say, β) for the existential type variable, there is no way for a client of e_1 to cast β to a pair type because, although $\beta=\alpha\times\alpha$ dynamically, the type name α is not in the client's static scope.

It is easy to see, however, that e_1 and e_2 are *not* equivalent according to our logical relation: Suppose they are, i.e., $\vdash e_1 \precsim e_2 : \tau$ (and the other way around). Instantiating this with a sufficiently large number $k+1$ and the empty world w yields $(k,\lfloor w\rfloor,e_1,e_2)\in E_{k+1}[\![\tau]\!]$. Now, since obviously $\epsilon;e_1 \hookrightarrow^1 \alpha_1{\approx}\mathsf{int};v_1[\alpha_1/\alpha]$ (where v_1 is the body of e_1), we know that there is w' such that $\epsilon;e_2 \hookrightarrow^* w'.\sigma_2;v_2$ and $(k-1,w',v_1,v_2')\in V_{k+1}[\![\tau]\!]$. Clearly, v_2' must be $v_2[\alpha_2/\alpha]$, where v_2 is the body of e_2 and α_2 is some type name. Recall that the (non-parametric) logical relation at existential type requires the type components of the two package values to be structurally equal. Clearly, this is not the case here, and so we have a contradiction.

Of course, if the language had a typecase operator, the situation would be different because a client could easily distinguish e_1 and e_2 by pattern matching the abstract type β against a pair type constructor—the pattern match would succeed for e_1 but fail for e_2. Thus, by demanding that the type components of logically related existential packages be structurally equal, our model appears to be a closer fit for a language with typecase (in which an adversarial context can perform complete structural decomposition of abstract type variables) than for one with cast (in which an adversarial context can only test for equality against "known" types).

This is fine from our perspective since our goal was never to tailor our model to the peculiarities of the cast construct. Moreover, even if we were interested in doing so, it is far from obvious to us how to go about it.

Our logical relation is also incomplete with respect to contextual approximation for reasons that have nothing to do with the non-parametric features of the language. In particular, while we have shown in this paper how our logical relation enables one to use traditional parametric reasoning when reasoning about wrapped programs, there are weird yet well-known examples—see, for instance, Pitts (2005)—of equivalences between existential packages that are not provable by direct use of logical relations. (Specifically, in these examples, there is no way to show the existential packages logically related, because there is no way of choosing a relational interpretation of the abstract type such that the ADT operations are logically related, yet the existential packages are nevertheless contextually equivalent.) Our logical relation cannot be used to directly prove those equivalences either.

A well-known technique for achieving completeness is to use *biorthogonality*, otherwise known as $\top\top$-*closure* (Pitts & Stark, 1998; Pitts, 2005). We believe it would not be difficult to incorporate biorthogonality into our present logical relations in order to render them complete. However, the completeness guaranteed by biorthogonality does not translate into a practical technique for establishing weird equivalences like the ones mentioned above. Moreover, as Benton & Tabareau (2009) have observed, biorthogonality also makes the logical relation (as a practical proof technique) sensitive to order of evaluation so that it would no longer be obvious how to use it to prove equivalences like our "order independence" result from Section 4.4.

12 Related work

Type generation versus other forms of data abstraction. Traditionally, authors have distinguished between two complementary forms of data abstraction, sometimes dubbed the *static* and the *dynamic* approach (Matthews & Ahmed, 2008). The former is tied to the type system and relies on parametricity (especially for existential types) to hide an ADT's representation from clients (Mitchell & Plotkin, 1988). 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. Toward that end, 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 (Sumii & Pierce, 2007a).

Dynamic type generation as we employ it (Rossberg, 2003, 2008; Vytiniotis *et al.*, 2005) 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 & Weirich (2005). 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 related work to ours is that of Matthews & Ahmed (2008). 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 & Ahmed's formalism are somewhat reminiscent of E^π 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—that is analogous to our Wrapping Theorem for $\lesssim^\pi$. However, 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 (A. Ahmed, 2009 personal communication). 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 (Reynolds, 1983; Mitchell, 1986; Pitts, 2005; Ahmed, 2006). In particular, Ahmed, Dreyer & Rossberg recently applied step-indexed logical relations with possible worlds to reason about type abstraction for a language with higher order state (Ahmed *et al.*, 2009). State in G is comparatively benign, but still requires a circular definition of worlds that we stratify using steps.

Pitts & Stark (1993) used logical relations to reason about program equivalence in the ν-calculus, a language with dynamic generation of term-level names in a manner similar to G. Since these names are abstract values with only an equality operator, it is sufficient in their case to index the logical relation by just the partial bijection between names, which essentially is a simple form of possible world. (In subsequent work, Pitts & Stark 1998 generalized their technique to handle mutable references.) Type names can encode term-level names via the type $\exists\alpha.1$ (Rossberg, 2003). Clearly, though this encoding is not fully abstract (in particular, $\exists\alpha.1$ is also inhabited by values not containing generated type names). Moreover, the presence of non-termination in G marks a fundamental difference from the ν-calculus that deeply affects the equational theory of the language.

Sumii & Pierce (2003) employed logical relations in proving secrecy results for a language with dynamic sealing, 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 & Pierce (2007a; 2007b) have used *bisimulations* to establish abstraction results for both untyped and polymorphic languages. However, none of the languages they investigate mixes the two paradigms.

Grossman *et al.* (2000) have proposed the use of *abstraction brackets* for syntactically tracing abstraction boundaries during program execution. However, this is a comparatively weak method that does not seem to help in proving parametricity or representation independence results.

13 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 more realistic languages, such as ML. We do not expect any problems as long as we deal only with pure language features. But unfortunately, wrapping cannot easily be extended to an impure type of mutable references, at least not without making the wrapping operator primitive in the language semantics. Nevertheless, we believe that our approach still scales to a large class of impure 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 extensions like Alice ML, which supports modules as first-class values, but not existentials—wrapping is never *needed* on the core level and could hence be confined to the module level. In such a language, wrapping can be kept implicit, as part of the implementation of opaque signature ascription—and in fact, that is exactly what Alice ML does. For core-level types, such as ref types, it can just be the identity. (Also included in "core-level" are recursive types, for which wrapping otherwise entails expensive copying.) 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

Abadi, M., Cardelli, L., Pierce, B. & Rémy, D. (1995) Dynamic typing in polymorphic languages. *J. Funct. Program.* **5**(1), 111–130.

Ahmed, A. (2004) *Semantics of Types for Mutable State*. Ph.D. thesis, Princeton University.

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

Ahmed, A. & Blume, M. (2008) Typed closure conversion preserves observational equivalence. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 157–168.

Ahmed, A., Dreyer, D. & Rossberg, A. (2009) State-dependent representation independence. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pp. 340–353.

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

Benton, N. & Tabareau, N. (2009) Compiling functional types to relational specifications for low level imperative code. In *Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pp. 3–14.

Birkedal, L. & Harper, R. W. (1999) Constructing interpretations of recursive types in an operational setting. *Inf. Comput.* **155**, 3–63.

Girard, J.-Y. (1972) *Interprétation Fonctionelle et Élimination des Coupures de L'arithmétique D'ordre Supérieur*. Ph.D. thesis, Université Paris VII.

Grossman, D., Morrisett, G. & Zdancewic, S. (2000) Syntactic type abstraction. *ACM Trans. Program. Lang. Syst.* **22**(6), 1037–1080.

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

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

Matthews, J. & Ahmed, A. (2008) Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *Proceedings of European Symposium on Programming (ESOP)*, pp. 16–31.

Mitchell, J. C. (1986) Representation independence and data abstraction. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pp. 263–276.

Mitchell, J. C. & Plotkin, G. D. (1988) Abstract types have existential type. *ACM Trans. Program. Lang. Syst.* **10**(3), 470–502.

Neis, G. (2009) *Non-Parametric Parametricity*. M.Phil. thesis, Universität des Saarlandes.

Pitts, A. (2005) Typed operational reasoning. In *Advanced Topics in Types and Programming Languages*, Benjamin C. P. (ed), chap. 2. MIT Press.

Pitts, A. & Stark, I. (1993) Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proceedings of International Symposium on Mathematical Foundations of Computer Science (MFCS)*, Lecture Notes in Computer Science, vol. 711, pp. 122–141.

Pitts, A. & Stark, I. (1998) Operational reasoning for functions with local state. In *Proceedings of Higher Order Operational Techniques in Semantics (HOOTS)*, pp. 227–274.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. *Inf. Process*, Amsterdam, pp. 513–523.

Rossberg, A. (2003) Generativity and dynamic opacity for abstract types. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, pp. 241–252.

Rossberg, A. (2007) *Typed Open Programming: A Higher-Order, Typed Approach to Dynamic Modularity and Distribution*. Ph.D. thesis, Universität des Saarlandes.

Rossberg, A. (2008) Dynamic translucency with abstraction kinds and higher-order coercions. In *Proceedings of Mathematical Foundations of Programming Semantics (MFPS)*, pp. 313–336.

Rossberg, A., Le Botlan, D., Tack, G., Brunklaus, T. & Smolka, G. (2004) Alice ML through the looking glass. In *Proceedings of Symposium on Trends in Functional Programming (TFP)*, vol. 5, pp. 79–96.

Sewell, P. (2001) Modules, abstract types, and distributed versioning. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pp. 236–247.

Sewell, P., Leifer, J., Wansbrough, K., Nardelli, F. Z., Allen-Williams, M., Habouzit, P. & Vafeiadis, V. (2007) Acute: High-level programming language design for distributed computation. *J. Funct. Program.* **17**(4–5), 547–612.

Sumii, E. & Pierce, B. C. (2003) Logical relations for encryption. *J. Comput. Secur.* **11**(4), 521–554.

Sumii, E. & Pierce, B. C. (2007a). A bisimulation for dynamic sealing. *Theor. Comput. Sci.* **375**(1–3), 161–192.

Sumii, E. & Pierce, B. C. (2007b) A bisimulation for type abstraction and recursion. *J. ACM* **54**(5), 1–43.

Vytiniotis, D., Washburn, G. & Weirich, S. (2005) An open and shut typecase. In *Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pp. 13–24.

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

Washburn, G. & Weirich, S. (2005) Generalizing parametricity using information flow. In *Proceedings of Symposium on Logic in Computer Science*, pp. 62–71.

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

Weirich, S., Vytiniotis, D., Peyton Jones, S. & Zdancewic, S. (2011) Generative type abstraction and type-level computation. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pp. 227–240.

Appendix A The languages G and G^{μ}

The differences between G and G^{μ}, i.e., everything related to recursive types, are underlined.

A.1 Syntax and semantics

Syntax

Types	$\tau ::= \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \mu\alpha.\tau$
Values	$v ::= x \mid \ldots \mid \langle v, v\rangle \mid \lambda x{:}\tau.e \mid \Lambda\alpha.e \mid \textsf{pack}\ \langle\tau, v\rangle\ \textsf{as}\ \tau \mid \underline{\textsf{roll}\ v\ \textsf{as}\ \tau}$
Expressions	$e ::= v \mid \ldots \mid \langle e, e\rangle \mid e.1 \mid e.2 \mid e\ e \mid e\ \tau \mid \textsf{pack}\ \langle\tau, v\rangle\ \textsf{as}\ \tau \mid$ $\textsf{unpack}\ \langle\alpha, x\rangle{=}e\ \textsf{in}\ e \mid \underline{\textsf{roll}\ e\ \textsf{as}\ \tau} \mid \underline{\textsf{unroll}\ e} \mid$ $\textsf{cast}\ \tau\ \tau \mid \textsf{new}\ \alpha{\approx}\tau\ \textsf{in}\ e$
Stores	$\sigma ::= \epsilon \mid \sigma, \alpha{\approx}\tau$
Evaluation Ctxt's	$E ::= \ldots \mid \langle E, e\rangle \mid \langle v, E\rangle \mid E.1 \mid E.2 \mid E\ e \mid v\ E \mid E\ \tau \mid$ $\textsf{pack}\ \langle\tau, E\rangle\ \textsf{as}\ \tau \mid \textsf{unpack}\ \langle\alpha, x\rangle{=}E\ \textsf{in}\ e \mid$ $\underline{\textsf{roll}\ E\ \textsf{as}\ \tau} \mid \underline{\textsf{unroll}\ E}$
Type Environments	$\Delta ::= \epsilon \mid \Delta, \alpha \mid \Delta, \alpha{\approx}\tau$
Value Environments	$\Gamma ::= \epsilon \mid \Gamma, x{:}\tau$

Reduction $\boxed{\sigma; e \hookrightarrow \sigma; e}$

$$\ldots$$

$$\sigma; E[\langle v_1, v_2\rangle.i] \hookrightarrow \sigma; E[v_i] \qquad (\textsc{RProj})$$

$$\sigma; E[(\lambda x{:}\tau.e)\, v] \hookrightarrow \sigma; E[e[v/x]] \qquad (\textsc{RApp})$$

$$\sigma; E[(\lambda\alpha.e)\, \tau] \hookrightarrow \sigma; E[e[\tau/\alpha]] \qquad (\textsc{RInst})$$

$$\sigma; E[\textsf{unpack}\ \langle\alpha, x\rangle{=}(\textsf{pack}\ \langle\tau, v\rangle)\ \textsf{in}\ e] \hookrightarrow \sigma; E[e[\tau/\alpha][v/x]] \qquad (\textsc{RUnpack})$$

$$\sigma; E[\textsf{unroll}\,(\textsf{roll}\ v\ \textsf{as}\ \tau)] \hookrightarrow \sigma; E[v] \qquad (\underline{\textsc{RUnroll}})$$

$$(\alpha \notin \mathrm{dom}(\sigma)) \qquad \sigma; E[\textsf{new}\,\alpha{\approx}\tau\ \textsf{in}\ e] \hookrightarrow \sigma, \alpha{\approx}\tau; E[e] \qquad (\textsc{RNew})$$

$$(\tau_1 = \tau_2) \qquad \sigma; E[\textsf{cast}\ \tau_1\ \tau_2] \hookrightarrow \sigma; E[\lambda x_1{:}\tau_1.\lambda x_2{:}\tau_2.x_1] \qquad (\textsc{RCast1})$$

$$(\tau_1 \neq \tau_2) \qquad \sigma; E[\textsf{cast}\ \tau_1\ \tau_2] \hookrightarrow \sigma; E[\lambda x_1{:}\tau_1.\lambda x_2{:}\tau_2.x_2] \qquad (\textsc{RCast2})$$

Type environments $\boxed{\vdash \Delta}$

$$\frac{}{\vdash \epsilon} \qquad \frac{\vdash \Delta \qquad \alpha \notin \mathrm{dom}(\Delta)}{\vdash \Delta, \alpha} \qquad \frac{\Delta \vdash \tau \qquad \alpha \notin \mathrm{dom}(\Delta)}{\vdash \Delta, \alpha{\approx}\tau}$$

Value environments $\boxed{\Delta \vdash \Gamma}$

$$\frac{\vdash \Delta}{\Delta \vdash \epsilon} \qquad \frac{\Delta \vdash \Gamma \qquad \Delta \vdash \tau \qquad x \notin \mathrm{dom}(\Gamma)}{\Delta \vdash \Gamma, x{:}\tau}$$

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

$$(\textsc{TVar})\,\frac{\vdash \Delta \qquad \alpha \in \Delta}{\Delta \vdash \alpha} \qquad (\textsc{TName})\,\frac{\vdash \Delta \qquad \alpha {\approx} \tau \in \Delta}{\Delta \vdash \alpha}$$

$$(\textsc{TBase})\,\frac{\vdash \Delta}{\Delta \vdash b} \qquad (\textsc{TTimes})\,\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \times \tau_2} \qquad (\textsc{TArr})\,\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}$$

$$(\textsc{TAll})\,\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall \alpha.\tau} \qquad (\textsc{TExists})\,\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \exists \alpha.\tau} \qquad (\underline{\textsc{TRec}})\,\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \mu \alpha.\tau}$$

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

$$(\textsc{CVar})\,\frac{\vdash \Delta \qquad \alpha \in \Delta}{\Delta \vdash \alpha \approx \alpha} \qquad (\textsc{CName})\,\frac{\vdash \Delta \qquad \alpha {\approx} \tau \in \Delta}{\Delta \vdash \alpha \approx \tau} \qquad (\textsc{CBase})\,\frac{\vdash \Delta}{\Delta \vdash b \approx b}$$

$$(\textsc{CTimes})\,\frac{\Delta \vdash \tau_1 \approx \tau_1' \qquad \Delta \vdash \tau_2 \approx \tau_2'}{\Delta \vdash \tau_1 \times \tau_2 \approx \tau_1' \times \tau_2'} \qquad (\textsc{CArr})\,\frac{\Delta \vdash \tau_1 \approx \tau_1' \qquad \Delta \vdash \tau_2 \approx \tau_2'}{\Delta \vdash \tau_1 \to \tau_2 \approx \tau_1' \to \tau_2'}$$

$$(\textsc{CAll})\,\frac{\Delta, \alpha \vdash \tau \approx \tau'}{\Delta \vdash \forall \alpha.\tau \approx \forall \alpha.\tau'} \qquad (\textsc{CExists})\,\frac{\Delta, \alpha \vdash \tau \approx \tau'}{\Delta \vdash \exists \alpha.\tau \approx \exists \alpha.\tau'}$$

$$(\underline{\textsc{CRec}})\,\frac{\Delta, \alpha \vdash \tau \approx \tau'}{\Delta \vdash \mu \alpha.\tau \approx \mu \alpha.\tau'}$$

$$(\textsc{CSym})\,\frac{\Delta \vdash \tau' \approx \tau}{\Delta \vdash \tau \approx \tau'} \qquad (\textsc{CTrans})\,\frac{\Delta \vdash \tau \approx \tau'' \qquad \Delta \vdash \tau'' \approx \tau'}{\Delta \vdash \tau \approx \tau'}$$

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

$$(\textsc{EVar})\,\frac{\Delta \vdash \Gamma \qquad x{:}\tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau} \qquad \ldots$$

$$(\textsc{EPair})\,\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad (\textsc{EProj})\,\frac{\Delta; \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash e.i : \tau_i}$$

$$(\textsc{EAbs})\,\frac{\Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad (\textsc{EApp})\,\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau}$$

$$(\textsc{EGen})\,\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda \alpha.e : \forall \alpha.\tau} \qquad (\textsc{EInst})\,\frac{\Delta; \Gamma \vdash e : \forall \alpha.\tau \qquad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash e\ \tau_2 : \tau[\tau_2/\alpha]}$$

$$(\textsc{EPack})\,\frac{\Delta; \Gamma \vdash e : \tau[\tau_1/\alpha] \qquad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash \mathsf{pack}\ \langle \tau_1, e \rangle\ \mathsf{as}\ \exists \alpha.\tau : \exists \alpha.\tau}$$

$$(\text{EUNPACK})\,\frac{\Delta;\Gamma \vdash e_1 : \exists\alpha.\tau_1 \qquad \Delta,\alpha;\Gamma,x{:}\tau_1 \vdash e_2 : \tau \qquad \Delta \vdash \tau}{\Delta;\Gamma \vdash \mathsf{unpack}\ \langle\alpha,x\rangle{=}e_1\ \mathsf{in}\ e_2 : \tau}$$

$$(\underline{\text{EROLL}})\,\frac{\Delta;\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Delta;\Gamma \vdash \mathsf{roll}\ e\ \mathsf{as}\ \mu\alpha.\tau : \mu\alpha.\tau} \qquad (\underline{\text{EUNROLL}})\,\frac{\Delta;\Gamma \vdash e : \mu\alpha.\tau}{\Delta;\Gamma \vdash \mathsf{unroll}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

$$(\text{ECAST})\,\frac{\Delta \vdash \Gamma \qquad \Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta;\Gamma \vdash \mathsf{cast}\ \tau_1\ \tau_2 : \tau_1 \to \tau_2 \to \tau_2}$$

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

$$(\text{ECONV})\,\frac{\Delta;\Gamma \vdash e : \tau' \qquad \Delta \vdash \tau \approx \tau'}{\Delta;\Gamma \vdash e : \tau}$$

A.2 Structural properties

$$\begin{array}{ll} \text{Type Substitutions} & \delta ::= \emptyset \mid \delta,\alpha{\mapsto}\tau \\ \text{Value Substitutions} & \gamma ::= \emptyset \mid \gamma,x{\mapsto}v \end{array}$$

Configurations $\boxed{\vdash \sigma;e : \tau}$

$$\frac{\Delta = \sigma \qquad \Delta;\epsilon \vdash e : \tau \qquad \epsilon \vdash \tau}{\vdash \sigma;e : \tau}$$

Type substitutions $\boxed{\Delta \vdash \delta : \Delta}$

$$\frac{\vdash \Delta'}{\Delta' \vdash \emptyset : \epsilon} \qquad \frac{\Delta' \vdash \delta : \Delta \qquad \Delta' \vdash \tau}{\Delta' \vdash \delta,\alpha{\mapsto}\tau : \Delta,\alpha} \qquad \frac{\Delta' \vdash \delta : \Delta \qquad \alpha'{\approx}\delta(\tau) \in \Delta'}{\Delta' \vdash \delta,\alpha{\mapsto}\alpha' : \Delta,\alpha{\approx}\tau}$$

Type substitution isomorphism $\boxed{\Delta \vdash \delta \approx \delta : \Delta}$

$$\frac{\vdash \Delta'}{\Delta' \vdash \emptyset \approx \emptyset : \epsilon} \qquad \frac{\Delta' \vdash \delta \approx \delta' : \Delta \qquad \Delta' \vdash \tau \approx \tau'}{\Delta' \vdash \delta,\alpha{\mapsto}\tau \approx \delta',\alpha{\mapsto}\tau' : \Delta,\alpha}$$

$$\frac{\Delta' \vdash \delta \approx \delta' : \Delta \qquad \alpha_1{\approx}\delta(\tau) \in \Delta' \qquad \alpha_2{\approx}\delta'(\tau) \in \Delta'}{\Delta' \vdash \delta,\alpha{\mapsto}\alpha_1 \approx \delta',\alpha{\mapsto}\alpha_2 : \Delta,\alpha{\approx}\tau}$$

Value substitutions $\boxed{\Delta;\Gamma \vdash \gamma : \Gamma}$

$$\frac{\Delta \vdash \Gamma'}{\Delta;\Gamma' \vdash \emptyset : \epsilon} \qquad \frac{\Delta;\Gamma' \vdash \gamma : \Gamma \qquad \Delta;\Gamma' \vdash v : \tau}{\Delta;\Gamma' \vdash \gamma, x \mapsto v : \Gamma, x{:}\tau}$$

Lemma 48 (*Weakening*)

1. If $\Delta \vdash \tau$ and $\Delta' \supseteq \Delta$ and $\vdash \Delta'$, then $\Delta' \vdash \tau$.
2. If $\Delta \vdash \tau \approx \tau'$ and $\Delta' \supseteq \Delta$ and $\vdash \Delta'$, then $\Delta' \vdash \tau \approx \tau'$.
3. If $\Delta \vdash \Gamma$ and $\Delta' \supseteq \Delta$ and $\vdash \Delta'$, then $\Delta' \vdash \Gamma$.
4. If $\Delta;\Gamma \vdash e : \tau$ and $\Delta' \supseteq \Delta$ and $\vdash \Delta'$, then $\Delta';\Gamma \vdash e : \tau$.
5. If $\Delta;\Gamma \vdash e : \tau$ and $\Gamma' \supseteq \Gamma$ and $\Delta \vdash \Gamma'$, then $\Delta;\Gamma' \vdash e : \tau$.
6. If $\Delta;\Gamma \vdash \gamma : \Gamma$ and $\Delta' \supseteq \Delta$ and $\vdash \Delta'$, then $\Delta';\Gamma \vdash \gamma : \Gamma$.

Lemma 49 (*Substitution*)

1. If $\Delta \vdash \tau$ and $\Delta' \vdash \delta : \Delta$, then $\Delta' \vdash \delta(\tau)$.
2. If $\Delta \vdash \tau \approx \tau'$ and $\Delta' \vdash \delta \approx \delta' : \Delta$, then $\Delta' \vdash \delta(\tau) \approx \delta'(\tau')$.
3. If $\Delta \vdash \Gamma$ and $\Delta' \vdash \delta : \Delta$, then $\Delta' \vdash \delta(\Gamma)$.
4. If $\Delta;\Gamma \vdash e : \tau$ and $\Delta' \vdash \delta : \Delta$, then $\Delta';\delta(\Gamma) \vdash \delta(e) : \delta(\tau)$.
5. If $\Delta;\Gamma \vdash e : \tau$ and $\Delta;\Gamma' \vdash \gamma : \Gamma$, then $\Delta;\Gamma' \vdash \gamma(e) : \tau$.

Lemma 50 (*Validity*)

1. If $\Delta \vdash \tau$, then $\vdash \Delta$.
2. If $\Delta \vdash \tau \approx \tau'$, then $\vdash \Delta$.
3. If $\Delta \vdash \Gamma$, then $\vdash \Delta$.
4. If $\Delta;\Gamma \vdash e : \tau$, then $\vdash \Delta$ and $\Delta \vdash \Gamma$ and $\Delta \vdash \tau$.

Lemma 51 (*Variable Containment*)

1. If $\Delta \vdash \tau$ and $\alpha \in \mathrm{ftv}(\tau)$, then $\alpha \in \mathrm{dom}(\Delta)$.
2. If $\Delta \vdash \tau \approx \tau'$ and $\alpha \in \mathrm{ftv}(\tau) \cup \mathrm{ftv}(\tau')$, then $\alpha \in \mathrm{dom}(\Delta)$.
3. If $\Delta \vdash \Gamma$ and $\alpha \in \mathrm{ftv}(\Gamma)$, then $\alpha \in \mathrm{dom}(\Delta)$.
4. If $\Delta;\Gamma \vdash e : \tau$ and $\alpha \in \mathrm{ftv}(\Gamma) \cup \mathrm{ftv}(e) \cup \mathrm{ftv}(\tau)$, then $\alpha \in \mathrm{dom}(\Delta)$.
5. If $\Delta;\Gamma \vdash e : \tau$ and $x \in \mathrm{fvv}(e)$, then $x \in \mathrm{dom}(\Gamma)$.

A.3 Type safety

Theorem 52 (*Preservation*)
If $\sigma;e \hookrightarrow \sigma';e'$ and $\vdash \sigma;e : \tau$, then $\vdash \sigma';e' : \tau$.

Lemma 53 (*Canonical Values*)
Assume $\vdash \sigma;v : \tau$. Then:

1. If $\tau = \tau_1 \times \tau_2$, then $v = \langle v_1, v_2 \rangle$.
2. If $\tau = \tau_1 \to \tau_2$, then $v = \lambda x{:}\tau_1'.e$.
3. If $\tau = \forall\alpha.\tau_1$, then $v = \Lambda\alpha.e$.
4. If $\tau = \exists\alpha.\tau_1$, then $v = \mathsf{pack}\ \langle \tau_2, v_1 \rangle\ \mathsf{as}\ \tau'$.
5. $\underline{\text{If } \tau = \mu\alpha.\tau_1\text{, then } v = \mathsf{roll}\ v'\ \mathsf{as}\ \tau'}$.

Theorem 54 (*Progress*)
If $\vdash \sigma;e : \tau$ and $e \neq v$, then $\sigma;e \hookrightarrow \sigma';e'$.

A.4 Contextual approximation and equivalence

$$
\begin{aligned}
\text{(contexts) } C ::= {} & [\_] \mid \langle C, e\rangle \mid \langle e, C\rangle \mid C.1 \mid C.2 \mid \lambda x{:}\tau.C \mid C\ e \mid e\ C \mid \\
& \Lambda\alpha.C \mid C\ \tau \mid \mathsf{pack}\langle\tau, C\rangle \mid \mathsf{unpack}\langle\alpha, x\rangle{=}C \mathsf{\ in\ } e \mid \\
& \mathsf{unpack}\ \langle\alpha, x\rangle{=}e \mathsf{\ in\ } C \mid \underline{\mathsf{roll}\ C \mathsf{\ as\ } \tau} \mid \underline{\mathsf{unroll}\ C} \mid \mathsf{new}\ \alpha{\approx}\tau \mathsf{\ in\ } C
\end{aligned}
$$

Contexts $\boxed{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta;\Gamma;\tau)}$

$$\text{(CEMPTY)}\frac{\Delta \subseteq \Delta' \quad \Gamma \subseteq \Gamma' \quad \Delta' \vdash \Gamma'}{\vdash [\_] : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau)}$$

$$\text{(CABS)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma', x{:}\tau_1;\tau_2)}{\vdash \lambda x{:}\tau_1.C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1 \to \tau_2)}$$

$$\text{(CPAIR.1)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1) \qquad \Delta';\Gamma' \vdash e : \tau_2}{\vdash \langle C, e\rangle : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1 \times \tau_2)}$$

$$\text{(CPAIR.2)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_2) \qquad \Delta';\Gamma' \vdash e : \tau_1}{\vdash \langle e, C\rangle : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1 \times \tau_2)}$$

$$\text{(CPROJ)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1 \times \tau_2)}{\vdash C.i : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_i)}$$

$$\text{(CAPP.1)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1 \to \tau_2) \qquad \Delta';\Gamma' \vdash e : \tau_1}{\vdash C\ e : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_2)}$$

$$\text{(CAPP.2)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1) \qquad \Delta';\Gamma' \vdash e : \tau_1 \to \tau_2}{\vdash e\ C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_2)}$$

$$\text{(CGEN)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta', \alpha;\Gamma';\tau')}{\vdash \Lambda\alpha.C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\forall\alpha.\tau')}$$

$$\text{(CINST)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\forall\alpha.\tau_1) \qquad \Delta' \vdash \tau_2}{\vdash C\ \tau_2 : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1[\tau_2/\alpha])}$$

$$\text{(CPACK)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_1[\tau_2/\alpha]) \qquad \Delta' \vdash \tau_2}{\vdash \mathsf{pack}\langle\tau_2, C\rangle : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\exists\alpha.\tau_1)}$$

$$\text{(CUNPACK.1)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\exists\alpha.\tau_1) \qquad \Delta', \alpha;\Gamma', x{:}\tau_1 \vdash e_2 : \tau_2 \qquad \Delta' \vdash \tau_2}{\vdash \mathsf{unpack}\ \langle\alpha, x\rangle{=}C \mathsf{\ in\ } e : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_2)}$$

$$\text{(CUNPACK.2)}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta', \alpha;\Gamma', x{:}\tau_1;\tau_2) \qquad \Delta';\Gamma' \vdash e : \exists\alpha.\tau_1 \qquad \Delta' \vdash \tau_2}{\vdash \mathsf{unpack}\ \langle\alpha, x\rangle{=}e \mathsf{\ in\ } C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_2)}$$

$$\underline{\text{(CROLL)}}\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau'[\mu\alpha.\tau'/\alpha])}{\vdash \mathsf{roll}\ C \mathsf{\ as\ } \mu\alpha.\tau' : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\mu\alpha.\tau')}$$

$$(\underline{\text{CUNROLL}})\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\mu\alpha.\tau')}{\vdash \mathsf{unroll}\ C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau'[\mu\alpha.\tau'])}$$

$$(\text{CNEW})\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta',\alpha{\approx}\tau_1;\Gamma';\tau_2) \qquad \Delta' \vdash \tau_2 \qquad \Delta' \vdash \Gamma'}{\vdash \mathsf{new}\ \alpha{\approx}\tau'\ \mathsf{in}\ C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau_2)}$$

$$(\text{CCONV})\frac{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau') \qquad \Delta' \vdash \tau' \approx \tau''}{\vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\Delta';\Gamma';\tau'')}$$

Termination and divergence $\boxed{\sigma;e\downarrow \quad \sigma;e\uparrow}$

$$\sigma;e\downarrow \overset{\text{def}}{\iff} \exists\sigma',v.\ \sigma;e \hookrightarrow^* \sigma';v$$

$$\sigma;e\uparrow \overset{\text{def}}{\iff} \nexists\sigma',v.\ \sigma;e \hookrightarrow^* \sigma';v$$

Contextual approximation $\boxed{\Delta;\Gamma \vdash e \leqslant e : \tau}$

$$\begin{aligned}\Delta;\Gamma \vdash e_1 \leqslant e_2 : \tau \overset{\text{def}}{\iff}\ & \Delta;\Gamma \vdash e_1 : \tau \wedge \Delta;\Gamma \vdash e_2 : \tau \wedge \forall\sigma,C,\tau'.\\ & \vdash \sigma \wedge \vdash C : (\Delta;\Gamma;\tau) \rightsquigarrow (\sigma;\epsilon;\tau') \wedge \sigma;C[e_1]\downarrow \Rightarrow \sigma;C[e_2]\downarrow\end{aligned}$$

Contextual equivalence $\boxed{\Delta;\Gamma \vdash e \equiv e : \tau}$

$$\Delta;\Gamma \vdash e_1 \equiv e_2 : \tau \overset{\text{def}}{\iff} \Delta;\Gamma \vdash e_1 \leqslant e_2 : \tau \wedge \Delta;\Gamma \vdash e_2 \leqslant e_1 : \tau$$

A.5 Encoding recursive functions

A.5.1 Using cast

$$\begin{aligned}\mathsf{fix}'\,f(x).e : \tau_1 \to \tau_2\ \mathsf{with}\ v_d &:= \lambda x_a{:}\tau_1.v\ (\forall\alpha.\alpha \to \tau_1 \to \tau_2)\ v\ x_a\\ \text{where } v &= \Lambda\alpha.\lambda x_s{:}\alpha.(\lambda f{:}(\tau_1 \to \tau_2).\lambda x{:}\tau_1.e)\ v'\\ \text{and } v' &= \lambda x_a{:}\tau_1.(\mathsf{cast}\ \alpha\ (\forall\alpha.\alpha \to \tau_1 \to \tau_2)\ x_s\ v_d)\ x_a\end{aligned}$$

Due to cast's required default argument, fix′ also needs to take a default value. Consequently, a fixed-point operator only exists for inhabited types. It is easy to verify the following two properties:

- $\sigma;(\mathsf{fix}'\,f(x).e : \tau_1 \to \tau_2\ \mathsf{with}\ v_d)\ v \hookrightarrow^* \sigma;e[\mathsf{fix}'\,f(x).e : \tau_1 \to \tau_2\ \mathsf{with}\ v_d/f][v/x]$, for any σ.
- If $\Delta;\Gamma,f{:}\tau_1 \to \tau_2,x{:}\tau_1 \vdash e : \tau_2$ and $\Delta;\Gamma \vdash v_d : \forall\alpha.\alpha \to \tau_1 \to \tau_2$, then $\Delta;\Gamma \vdash (\mathsf{fix}'\,f(x).e : \tau_1 \to \tau_2\ \mathsf{with}\ v_d) : \tau_1 \to \tau_2$.

A.5.2 Using recursive types

$$\begin{aligned}\mathsf{fix}\, f(x).e : \tau_1 \to \tau_2 := &\ \lambda x_a{:}\tau_1.v\ (\mathsf{roll}\ v\ \mathsf{as}\ \mu\alpha.\alpha \to \tau_1 \to \tau_2)\ x_a\\ \text{where } v\ =&\ \lambda x_s{:}(\mu\alpha.\alpha \to \tau_1 \to \tau_2).(\lambda f{:}(\tau_1 \to \tau_2).\lambda x{:}\tau_1.e)\\ &\qquad\qquad (\lambda x_a{:}\tau_1.(\mathsf{unroll}\ x_s)\ x_s\ x_a)\end{aligned}$$

It is easy to verify the following two properties:

- $\sigma;(\mathsf{fix}\, f(x).e : \tau_1 \to \tau_2)\ v \hookrightarrow^* \sigma; e[\mathsf{fix}\, f(x).e : \tau_1 \to \tau_2/f][v/x]$, for any σ.
- If $\Delta;\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e : \tau_2$, then $\Delta;\Gamma \vdash (\mathsf{fix}\, f(x).e : \tau_1 \to \tau_2) : \tau_1 \to \tau_2$.

Appendix B Some proofs

B.1 Lemma 13 from Section 4

If $(\delta_1, \delta_2, \rho) \in D_n[\![\Delta]\!]w_0$ and $\delta = \mathrm{au}(\delta_1, \delta_2, w_0.\eta)$ and $\Delta \vdash \tau$, then:

1. $V_n[\![\tau]\!]\rho = V_n[\![\delta(\tau)]\!]w_0.\rho$
2. $E_n[\![\tau]\!]\rho = E_n[\![\delta(\tau)]\!]w_0.\rho$

Proof
By primary induction on n and secondary induction on the derivation of $\Delta \vdash \tau$, we show the interesting cases.

1. • Case $\tau = \alpha$ were $\alpha \in \Delta$:
   - — Then, we know from the definition of $D_n[\![\Delta]\!]w_0$ that there is $(\tau_1, \tau_2, r) \in T_n[\![\Omega]\!]w_0$ such that $\delta_i = \delta_{i1}, \alpha \mapsto \tau_i, \delta_{i2}$ and $\rho = \rho_1, \alpha \mapsto r, \rho_2$.
   - — By definition of $T_n[\![\Omega]\!]w_0$, there is τ' such that $\tau_i = w_0.\eta^i(\tau')$ and $r.R = V_n[\![\tau']\!]w_0.\rho$.
   - — Hence, $V_n[\![\alpha]\!]\rho = V_n[\![\tau']\!]w_0.\rho$.
   - — Since $\tau_i = w_0.\eta^i(\delta(\alpha))$ by Lemma 12, the injectivity of $w_0.\eta^i$ implies $\tau' = \delta(\alpha)$.

   • Case $\tau = \alpha$ where $\alpha{\approx}\tau' \in \Delta$:
   - — Then, we know from the definition of $D_n[\![\Delta]\!]w_0$ that $\delta_i = \delta_{i1}, \alpha \mapsto \alpha_i, \delta_{i2}$ and $\rho = \rho_1, \alpha \mapsto (\rho_1^1(\tau'), \rho_1^2(\tau'), V_n[\![\tau']\!]\rho_1), \rho_2$ with $\alpha_i = w_0.\eta^i(\alpha')$ and $V_n[\![\tau']\!]\rho_1 = w_0.\rho(\alpha').R$ for some α'.
   - — Because of the injectivity of $w_0.\eta^i$, $w_0.\eta^i(\alpha') = \alpha_i = \delta_i(\alpha) = w_0.\eta^i\delta(\alpha)$ implies $\alpha' = \delta(\alpha)$.
   - — Hence, $V_n[\![\alpha]\!]\rho = V_n[\![\tau']\!]\rho_1 = V_n[\![\alpha']\!]w_0.\rho = V_n[\![\delta(\alpha)]\!]w_0.\rho$.

   • Case $\tau = \forall\alpha.\tau'$ with $\Delta, \alpha \vdash \tau'$:
   - — We show $V_n[\![\tau]\!]\rho \subseteq V_n[\![\delta(\tau)]\!]w_0.\rho$; the other direction is symmetric.
   - — Suppose $(k, w, \Lambda\alpha.e_1, \Lambda\alpha.e_2) \in V_n[\![\forall\alpha.\tau']\!]\rho$.
   - — Suppose further $(k'', w'') \sqsupset (k', w') \sqsupseteq (k, w)$ and $(\tau_1, \tau_2, r) \in T_{k'}[\![\Omega]\!]w'$.
   - — We know $(k'', w'', e_1[\tau_1/\alpha], e_2[\tau_2/\alpha]) \in E_n[\![\tau']\!]\rho, \alpha \mapsto r$.
   - — To show: $(k'', w'', e_1[\tau_1/\alpha], e_2[\tau_2/\alpha]) \in E_n[\![\delta(\tau')]\!]w_0.\rho, \alpha \mapsto r$

— This reduces to showing $E_{k'}[\![\tau']\!]\lfloor\rho\rfloor_{k'}, \alpha\mapsto r = E_{k'}[\![\delta(\tau')]\!]w'.\rho, \alpha\mapsto r$.

— By assumption and Lemma 4, $(\delta_1, \delta_2, \lfloor\rho\rfloor_{k'}) \in D_{k'}[\![\Delta]\!]w'$.

— Let $(\delta_1', \delta_2', \rho') := ((\delta_1, \alpha\mapsto\tau_1), (\delta_2, \alpha\mapsto\tau_2), (\lfloor\rho\rfloor_{k'}, \alpha\mapsto r))$, so $(\delta_1', \delta_2', \rho') \in D_{k'}[\![\Delta, \alpha]\!]w'$.

— By Lemma 12, $\delta = \mathrm{au}(\delta_1, \delta_2, w'.\eta)$.

— Since $(\tau_1, \tau_2, r) \in T_{k'}[\![\Omega]\!]w'$ we know $\tau_i = w'.\eta^i(\tau'')$ and $r = (w'.\rho^1(\tau''), w'.\rho^2(\tau''), V_{k'}[\![\tau'']\!]w'.\rho)$.

— It is easy to see then that $\delta, \alpha\mapsto\tau'' = \mathrm{au}(\delta_1', \delta_2', w'.\eta)$.

— Hence, by induction, $E_{k'}[\![\tau']\!]\rho' = E_{k'}[\![\delta(\tau')[\tau''/\alpha]]\!]w'.\rho$.

— And by LR-Substitution,

$$\begin{aligned} & E_{k'}[\![\delta(\tau')[\tau''/\alpha]]\!]w'.\rho \\ = & E_{k'}[\![\delta(\tau')]\!]w'.\rho, \alpha\mapsto(w'.\rho^1(\tau''), w'.\rho^2(\tau''), V_{k'}[\![\tau'']\!]w'.\rho) \\ = & E_{k'}[\![\delta(\tau')]\!]w'.\rho, \alpha\mapsto r. \end{aligned}$$

2. Follows immediately from part (1). □

B.2 Partly benign effects (Repeatability)

Consider the following functions (where τ is arbitrary but closed):

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

We first prove $\epsilon;\epsilon \vdash v_1 \precsim v_2 : (\mathsf{unit}\to\tau)\to\tau$. The key here is that we relate the *second* call of x in v_1—the one whose return value matters—to the single call of x in v_2. To do so, we have to construct a world w_1' that differs from the "initial" world w' in that its first type store is the one in which the second call of x is executed.

Proof

- Suppose $w_0 \in \mathrm{World}_n$ and $(k, w) \sqsupseteq (n, w_0)$.
- To show: $(k, w, v_1, v_2) \in V_n[\![(\mathsf{unit}\to\tau)\to\tau]\!]$
- So suppose $(k', w', \lambda x.e_1, \lambda x.e_2) \in V_n[\![\mathsf{unit}\to\tau]\!]$ where $(k', w') \sqsupseteq (k, w)$.
- To show: $(k', w', \mathsf{let}\ x' = (\lambda x.e_1)\,()\ \mathsf{in}\ (\lambda x.e_1)\,(), (\lambda x.e_2)\,()) \in E_n[\![\tau]\!]$
- Suppose that $w'.\sigma_1; \mathsf{let}\ x' = (\lambda x.e_1)\,()\ \mathsf{in}\ (\lambda x.e_1)\,()$ terminates:

$$\begin{aligned} & w'.\sigma_1; \mathsf{let}\ x' = (\lambda x.e_1)\,()\ \mathsf{in}\ (\lambda x.e_1)\,() \\ \hookrightarrow^1\ & w'.\sigma_1; \mathsf{let}\ x' = e_1[()/x]\ \mathsf{in}\ (\lambda x.e_1)\,() \\ \hookrightarrow^{j_1}\ & \sigma_1'; \mathsf{let}\ x' = v_1'\ \mathsf{in}\ (\lambda x.e_1)\,() \\ \hookrightarrow^1\ & \sigma_1'; (\lambda x.e_1)\,() \\ \hookrightarrow^1\ & \sigma_1'; e_1[()/x] \\ \hookrightarrow^{j_2}\ & \sigma_1; v_1'' \end{aligned}$$

  and that $3 + j_1 + j_2 =: j < k'$.
- Let $w_1' := (\sigma_1', w'.\sigma_2, w'.\eta, w'.\rho)$, so $(k', w_1') \sqsupseteq (k', w')$.
- Instantiating $(k', w', \lambda x.e_1, \lambda x.e_2) \in V_n[\![\mathsf{unit}\to\tau]\!]$ with $(k' - j_1 - 3, \lfloor w_1'\rfloor, (), ()) \in V_n[\![\mathsf{unit}]\!]$ gives us $(k' - j_1 - 3, \lfloor w_1'\rfloor, e_1[()/x], e_2[()/x]) \in E_n[\![\tau]\!]$.

- Instantiating this with $\sigma_1'; e_1[()/x] \hookrightarrow^{j_2} \sigma_1; v_1''$ yields $(k'-j, w'') \sqsupseteq (k'-j_1-3, \lfloor w_1' \rfloor)$ such that
$$w'.\sigma_2; e_2[()/x] \hookrightarrow^* w''.\sigma_2; v_2'$$
with $w''.\sigma_1 = \sigma_1$ and $(k'-j, w'', v_1'', v_2') \in V_n[\![\tau]\!]$.
- This implies $(k'-j, w'') \sqsupseteq (k', w')$ and
$$w'.\sigma_2; (\lambda x.e_2)\ () \hookrightarrow^* w''.\sigma_2; v_2'. \qquad \square$$

It remains to show the other direction, i.e., $\epsilon; \epsilon \vdash v_2 \lesssim v_1 : (\mathsf{unit} \to \tau) \to \tau$. We first relate the single call of x in v_2 (resulting in a value v_1') to the first call of x in v_1. From that we learn that the latter terminates. We can then construct a world w_2' from w' as in the previous part and use that to relate the call of x in v_2 also to the *second* call of x in v_1. From that we learn that also this call terminates and that it results in a value v_2'' to which v_1' is related.

Proof

- Suppose $w_0 \in \mathrm{World}_n$ and $(k, w) \sqsupseteq (n, w_0)$.
- To show: $(k, w, v_2, v_1) \in V_n[\![(\mathsf{unit} \to \tau) \to \tau]\!]$
- So suppose $(k', w', \lambda x.e_1, \lambda x.e_2) \in V_n[\![\mathsf{unit} \to \tau]\!]$ where $(k', w') \sqsupseteq (k, w)$.
- To show: $(k', w', (\lambda x.e_1)\ (), \mathsf{let}\ x' = (\lambda x.e_2)\ ()\ \mathsf{in}\ (\lambda x.e_2)\ ()) \in E_n[\![\tau]\!]$
- Suppose $w'.\sigma_1; (\lambda x.e_1)\ ()$ terminates:
$$\begin{aligned} & w'.\sigma_1; (\lambda x.e_1)\ () \\ \hookrightarrow^1\ & w'.\sigma_1; e_1[()/x] \\ \hookrightarrow^{j'}\ & \sigma_1; v_1' \end{aligned}$$
and that $1 + j' =: j < k'$.
- Instantiating $(k', w', \lambda x.e_1, \lambda x.e_2) \in V_n[\![\mathsf{unit} \to \tau]\!]$ with $(k'-1, \lfloor w' \rfloor, (), ()) \in V_n[\![\mathsf{unit}]\!]$ yields $(k'-1, \lfloor w' \rfloor, e_1[()/x], e_2[()/x]) \in E_n[\![\tau]\!]$.
- Consequently, there exists $(k'-j, w'') \sqsupseteq (k'-1, \lfloor w' \rfloor)$ such that
$$w'.\sigma_2; e_2[()/x] \hookrightarrow^* w''.\sigma_2; v_2'.$$
- Let $w_2' = (w'.\sigma_1, w''.\sigma_2, w'.\eta, w'.\rho)$, so $(k', w_2') \sqsupseteq (k', w')$.
- Instantiating $(k', w', \lambda x.e_1, \lambda x.e_2) \in V_n[\![\mathsf{unit} \to \tau]\!]$ with $(k'-1, \lfloor w_2' \rfloor, (), ()) \in V_n[\![\mathsf{unit}]\!]$ yields $(k'-1, \lfloor w_2' \rfloor, e_1[()/x], e_2[()/x]) \in E_n[\![\tau]\!]$.
- Consequently, there exists $(k'-j, w''') \sqsupseteq (k'-1, \lfloor w_2' \rfloor)$ such that
$$w''.\sigma_2; e_2[()/x] \hookrightarrow^* w'''.\sigma_2; v_2''$$
with $w'''.\sigma_1 = \sigma_1$ and $(k'-j, w''', v_1', v_2'') \in V_n[\![\tau]\!]$.
- Note that
$$\begin{aligned} & w'.\sigma_2; \mathsf{let}\ x' = (\lambda x.e_2)\ ()\ \mathsf{in}\ (\lambda x.e_2)\ () \\ \hookrightarrow^1\ & w'.\sigma_2; \mathsf{let}\ x' = e_2[()/x]\ \mathsf{in}\ (\lambda x.e_2)\ () \\ \hookrightarrow^*\ & w''.\sigma_2; \mathsf{let}\ x' = v_2'\ \mathsf{in}\ (\lambda x.e_2)\ () \\ \hookrightarrow^1\ & w''.\sigma_2; (\lambda x.e_2)\ () \\ \hookrightarrow^1\ & w''.\sigma_2; e_2[()/x] \\ \hookrightarrow^*\ & w'''.\sigma_2; v_2'' \end{aligned} \qquad \square$$

B.3 Partly benign effects (Order independence)

Consider the following functions (where τ and τ' are arbitrary but closed):

$$v_1' := \lambda x{:}(\mathsf{unit} \to \tau).\lambda y{:}(\mathsf{unit} \to \tau').\ \mathsf{let}\, y' = y\,()\ \mathsf{in}\ \langle x\,(), y'\rangle$$
$$v_2' := \lambda x{:}(\mathsf{unit} \to \tau).\lambda y{:}(\mathsf{unit} \to \tau').\ \langle x\,(), y\,()\rangle$$

We show $\epsilon; \epsilon \vdash v_1' \precsim v_2' : (\mathsf{unit} \to \tau) \to (\mathsf{unit} \to \tau') \to (\tau \times \tau')$. (The proof for the other direction is nearly identical.) We start by constructing a world w_2' from the "initial" world w'' that lets us relate the second application in v_1' (namely, $x\,()$) to the corresponding first application in v_2', which yields a future world w_2'' and values v_1'', v_2'' that are related in it. We then construct another world w_1' that lets us relate the first application in v_1' (namely, $y\,()$) to the corresponding second application in v_2', which yields a future world w_1'' and values v_3', v_4' that are related in it. Finally, we need to merge worlds w_1'' and w_2'' to obtain a single future world w_3 in which the resulting pairs $\langle v_1'', v_3'\rangle$, $\langle v_2'', v_4'\rangle$ are related. The well-formedness of that world is not obvious and needs to be verified by case analysis.

Proof

- Suppose $w_0 \in \mathrm{World}_n$ and $(k, w) \sqsupseteq (n, w_0)$.
- To show: $(k, w, v_1', v_2') \in V_n[\![(\mathsf{unit} \to \tau) \to (\mathsf{unit} \to \tau') \to (\tau \times \tau')]\!]$
- So suppose $(k', w', \lambda z.e_1, \lambda z.e_2) \in V_n[\![\mathsf{unit} \to \tau]\!]$ where $(k', w') \sqsupseteq (k, w)$.
- To show: $(k', w', \lambda y.\ \mathsf{let}\, y' = y\,()\ \mathsf{in}\ \langle(\lambda z.e_1)\,(), y'\rangle,$
  $\lambda y.\ \langle(\lambda z.e_2)\,(), y\,()\rangle) \in V_n[\![(\mathsf{unit} \to \tau') \to (\tau \times \tau')]\!]$
- So suppose $(k'', w'', \lambda z'.e_3, \lambda z'.e_4) \in V_n[\![\mathsf{unit} \to \tau']\!]$ where $(k'', w'') \sqsupseteq (k', w')$.
- To show: $(k'', w'', \mathsf{let}\, y' = (\lambda z'.e_3)\,()\ \mathsf{in}\ \langle(\lambda z.e_1)\,(), y'\rangle,$
  $\langle(\lambda z.e_2)\,(), (\lambda z'.e_4)\,()\rangle) \in E_n[\![\tau \times \tau']\!]$
- Suppose $w''.\sigma_1; \mathsf{let}\, y' = (\lambda z'.e_3)\,()\ \mathsf{in}\ \langle(\lambda z.e_1)\,(), y'\rangle$ terminates

$$\begin{aligned} & w''.\sigma_1; \mathsf{let}\, y' = (\lambda z'.e_3)\,()\ \mathsf{in}\ \langle(\lambda z.e_1)\,(), y'\rangle \\ \hookrightarrow^1\ & w''.\sigma_1; \mathsf{let}\, y' = e_3[()/z']\ \mathsf{in}\ \langle(\lambda z.e_1)\,(), y'\rangle \\ \hookrightarrow^{j_1}\ & \sigma_1'; \mathsf{let}\, y' = v_3'\ \mathsf{in}\ \langle(\lambda z.e_1)\,(), y'\rangle \\ \hookrightarrow^1\ & \sigma_1'; \langle(\lambda z.e_1)\,(), v_3'\rangle \\ \hookrightarrow^1\ & \sigma_1'; \langle e_1[()/z], v_3'\rangle \\ \hookrightarrow^{j_2}\ & \sigma_1; \langle v_1'', v_3'\rangle \end{aligned}$$

  and $j_1 + j_2 + 3 =: j < k''$.
- Let $(k_2', w_2') := (k'' - j_1 - 3, (\sigma_1', w''.\sigma_2, w''.\eta, \lfloor w''.\rho \rfloor))$, so $(k_2', w_2') \sqsupseteq (k'', w'')$.
- Instantiating $(k', w', \lambda z.e_1, \lambda z.e_2) \in V_n[\![\mathsf{unit} \to \tau]\!]$ with $(k_2', w_2', (), ()) \in V_n[\![\mathsf{unit}]\!]$ gives us $(k_2', w_2', e_1[()/z], e_2[()/z]) \in E_n[\![\tau]\!]$.
- Note that $w_2'.\sigma_1 = \sigma_1'$.
- Consequently, there exists $(k'' - j, w_2'') \sqsupseteq (k_2', w_2')$ such that

$$w''.\sigma_2; e_2[()/z] \hookrightarrow^* w_2''.\sigma_2; v_2''$$

  with $w_2''.\sigma_1 = \sigma_1$ and $(k'' - j, w_2'', v_1'', v_2'') \in V_n[\![\tau]\!]$.
- Let $(k_1', w_1') := (k'' - 1, (w''.\sigma_1, w_2''.\sigma_2, w''.\eta, \lfloor w''.\rho \rfloor))$, so $(k_1', w_1') \sqsupseteq (k'', w'')$.
- Instantiating $(k'', w'', \lambda z'.e_3, \lambda z'.e_4) \in V_n[\![\mathsf{unit} \to \tau']\!]$ with $(k_1', w_1', (), ()) \in V_n[\![\mathsf{unit}]\!]$ gives us $(k_1', w_1', e_3[()/z'], e_4[()/z']) \in E_n[\![\tau']\!]$.

- Note that $w'_1.\sigma_1 = w''.\sigma_1$.
- Consequently, there exists $(k'' - 1 - j_1, w''_1) \sqsupseteq (k'_1, w'_1)$ such that

$$w''_2.\sigma_2\,;e_4[()/z'] \hookrightarrow^* w''_1.\sigma_2\,;v'_4$$

  with $w''_1.\sigma_1 = \sigma'_1$ and $(k'' - 1 - j_1, w''_1, v'_3, v'_4) \in V_n[\![\tau']\!]$.
- W.l.o.g. $(\mathrm{dom}(w''_1.\eta) \setminus \mathrm{dom}(w''.\eta)) \cap (\mathrm{dom}(w''_2.\eta) \setminus \mathrm{dom}(w''.\eta)) = \emptyset$, so $w''_1.\eta \cup w''_2.\eta$ and $w''_1.\rho \cup w''_2.\rho$ are well defined.
- Let $w_3 := (w''_2.\sigma_1, w''_1.\sigma_2, w''_1.\eta \cup w''_2.\eta, \lfloor w''_1.\rho \rfloor_{k''-j} \cup w''_2.\rho)$.
- To see that w_3 is well formed, it remains to show the injectivity of $w_3.\eta^i$:
  — Note that $\mathrm{rng}(w''_1.\eta^i) \setminus \mathrm{rng}(w''.\eta^i) \subseteq \mathrm{dom}(w''_1.\sigma_i) \setminus \mathrm{dom}(w'_1.\sigma_i)$ by definition of world extension.
  — Similarly, $\mathrm{rng}(w''_2.\eta^i) \setminus \mathrm{rng}(w''.\eta^i) \subseteq \mathrm{dom}(w''_2.\sigma_i) \setminus \mathrm{dom}(w'_2.\sigma_i)$ by definition of world extension.
  — Suppose $\alpha, \alpha' \in \mathrm{dom}(w_3.\eta)$.
  — Case $\alpha, \alpha' \in \mathrm{dom}(w''.\eta)$: Trivial.
  — Case $\alpha \in \mathrm{dom}(w''.\eta)$ and $\alpha' \in \mathrm{dom}(w''_1.\eta) \setminus \mathrm{dom}(w''.\eta)$:
    - Then, $w_3.\eta^i(\alpha) \in \mathrm{dom}(w''.\sigma_i)$ and $w_3.\eta^i(\alpha') \in \mathrm{dom}(w''_1.\sigma_i) \setminus \mathrm{dom}(w'_1.\sigma_i)$.
    - Since $w'_1.\sigma_i = w''.\sigma_i$, we have $w_3.\eta^i(\alpha) \neq w_3.\eta^i(\alpha')$.
  — Case $\alpha \in \mathrm{dom}(w''.\eta)$ and $\alpha' \in \mathrm{dom}(w''_2.\eta) \setminus \mathrm{dom}(w''.\eta)$:
    - Then, $w_3.\eta^i(\alpha) \in \mathrm{dom}(w''.\sigma_i)$ and $w_3.\eta^i(\alpha') \in \mathrm{dom}(w''_2.\sigma_i) \setminus \mathrm{dom}(w'_2.\sigma_i)$.
    - Since $w'_1.\sigma_i = w''.\sigma_i$, we have $w_3.\eta^i(\alpha) \neq w_3.\eta^i(\alpha')$.
  — Case $\alpha \in \mathrm{dom}(w''_1.\eta) \setminus \mathrm{dom}(w''.\eta)$ and $\alpha' \in \mathrm{dom}(w''_2.\eta) \setminus \mathrm{dom}(w''.\eta)$:
    - Then, $w_3.\eta^i(\alpha) \in \mathrm{dom}(w''_1.\sigma_i) \setminus \mathrm{dom}(w'_1.\sigma_i)$ and $w_3.\eta^i(\alpha') \in \mathrm{dom}(w''_2.\sigma_i) \setminus \mathrm{dom}(w'_2.\sigma_i)$.
    - For $i = 1$, this means $w_3.\eta^1(\alpha) \in \mathrm{dom}(w''_1.\sigma_1) = \mathrm{dom}(\sigma'_1) = \mathrm{dom}(w'_2.\sigma_1)$, so it cannot equal $w_3.\eta^1(\alpha')$.
    - For $i = 2$, this means $w_3.\eta^2(\alpha) \in \mathrm{dom}(w''_1.\sigma_2) \setminus \mathrm{dom}(w''_2.\sigma_2)$, so it cannot equal $w_3.\eta^2(\alpha')$.
- Also note that $(k'' - j, w_3) \sqsupseteq (k'' - j, w''_2)$ and $(k'' - j, w_3) \sqsupseteq (k'' - 1 - j_1, w''_1)$.
- Hence, $(k'' - j, w_3, v''_1, v''_2) \in V_n[\![\tau]\!]$ and $(k'' - j, w_3, v'_3, v'_4) \in V_n[\![\tau']\!]$ and therefore $(k'' - j, w_3, \langle v''_1, v'_3\rangle, \langle v''_2, v'_4\rangle) \in V_n[\![\tau \times \tau']\!]$.
- And of course

$$w''.\sigma_2\,;\langle(\lambda z.e_2)\,(), (\lambda z'.e_4)\,()\rangle \hookrightarrow^* w_3.\sigma_2\,;\langle v''_2, v'_4\rangle. \qquad \square$$