The Marriage of Bisimulations and Kripke Logical Relations

Chung-Kil Hur   Derek Dreyer   Georg Neis   Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS)
{gil,dreyer,neis,viktor}@mpi-sws.org

Abstract

There has been great progress in recent years on developing effective techniques for reasoning about program equivalence in ML-like languages—that is, languages that combine features like higher-order functions, recursive types, abstract types, and general mutable references. Two of the most prominent types of techniques to have emerged are *bisimulations* and *Kripke logical relations (KLRs)*. While both approaches are powerful, their complementary advantages have led us and other researchers to wonder whether there is an essential tradeoff between them. Furthermore, both approaches seem to suffer from fundamental limitations if one is interested in scaling them to inter-language reasoning.

In this paper, we propose *relation transition systems (RTSs)*, which marry together some of the most appealing aspects of KLRs and bisimulations. In particular, RTSs show how bisimulations' support for reasoning about recursive features via *coinduction* can be synthesized with KLRs' support for reasoning about local state via *state transition systems*. Moreover, we have designed RTSs to avoid the limitations of KLRs and bisimulations that preclude their generalization to inter-language reasoning. Notably, unlike KLRs, RTSs are transitively composable.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Kripke logical relations, bisimulations, relation transition systems, contextual equivalence, higher-order state, recursive types, abstract types, transitivity, global vs. local knowledge

1. Introduction

One of the grand challenges in programming language semantics is to find scalable techniques for reasoning about the observational equivalence of programs. Even when the intuitive principles of local reasoning suggest that a change to some program module should not be observable to any client, it can be fiendishly difficult to establish that formally. Denotational semantics offers a tractable way of proving equivalence of programs by showing that they *mean* the same thing in some adequate model of their language. However, traditional denotational methods do not scale well to general-purpose languages like ML that combine support for functional, *value-oriented* programming (*e.g.,* higher-order functions, polymorphism, abstract data types, recursive types) with support for imperative, *effect-oriented* programming (*e.g.,* mutable state and control effects, among other things).

Fortunately, in recent years, there has been a groundswell of interest in the problem of developing effective methods for reasoning about program equivalence in ML-like languages. A variety of promising techniques have emerged [29, 36, 19, 20, 34, 33, 23, 5, 35, 12, 25], and while some of these methods are denotational, most support direct reasoning about the operational semantics of programs. In particular, there has been a healthy rivalry between techniques based on **Kripke logical relations (KLRs)** [29, 5, 26, 13, 12, 17, 37] and **bisimulations** [36, 19, 34, 33, 23, 35].

This paper is motivated by two high-level concerns:

(1) KLRs and bisimulations offer complementary advantages, which we would like to synthesize in a single proof method.

(2) There is a specific sense in which both KLRs and bisimulations appear to be fundamentally limited, and some fresh idea seems necessary to circumvent this limitation.

Concerning motivation (1): The latest KLR techniques [5, 12] use *state transition systems* to provide more flexible principles for reasoning about local state than bisimulations do. However, in order to account for the presence of recursive features, such as recursive types and higher-order state, KLRs require tedious manipulation of tricky "step-indexed" constructions [6, 2].[1] In contrast, bisimulation techniques use *coinduction* to model such recursive features very elegantly, but their support for reasoning about local state is weaker than KLRs' (see Section 9). These complementary strengths have led us and other researchers to wonder whether there is some fundamental tradeoff between KLRs and bisimulations.

Concerning motivation (2): We are interested in scaling equational reasoning techniques to the setting of *inter-language reasoning*, *i.e.,* reasoning about equivalences between programs in different languages. Inter-language reasoning is essential to the development of *compositional certified compilers* [7, 17], and may also have applications to the verification of multi-language (interoperating) programs [4]. Unfortunately, both KLR and bisimulation methods rely on technical devices that prevent them (it seems) from scaling to the inter-language setting. Specifically, in order to deal with higher-order functions, bisimulation methods employ various "syntactic" devices that restrict the applicability of the methods to single-language reasoning (see Section 3 for details). KLRs, in contrast, *have* been shown to generalize to inter-language reasoning [17], but there remains a key problem: KLR proofs are in general *not transitively composable*, at least in part due to the use of step-indexed constructions as mentioned above. In order to prove compositional correctness of *multi-phase* compilers for ML-like

[1] This has led to a series of papers—some written by authors of the present paper—on how to hide the "ugliness" of step-indices [11, 13, 8].

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

languages, it is crucial to be able to prove the correctness of each phase individually and then compose them transitively, so KLRs' lack of transitivity is a showstopper for that application.

1.1 Contributions

In this paper, we present a new technique for reasoning about program equivalence, called **Relation Transition Systems (RTSs)**. RTSs marry together some of the most appealing features of KLRs and bisimulations, while circumventing their limitations.

In particular, RTSs show how the use of state transition systems (from KLRs) can be synthesized with the coinductive, step-index-free style of reasoning (from bisimulations), thereby enabling clean and elegant proofs about local state *and* recursive features simultaneously. Thus, concerning the long-standing open question of whether there is a fundamental tradeoff between KLRs and bisimulations, we provide a definitive answer: no, there is not.

We explore RTSs here in the setting of $\mathsf{F}^{\mu!}$—a CBV λ-calculus with general recursive types, products, sums, universals, existentials, and general references [5]—as this provides a clear point of comparison with recent work on both KLRs [12] and bisimulations [35]. With one notable exception (see Section 9), we believe RTSs are capable of reasoning effectively about all the challenging $\mathsf{F}^{\mu!}$ equivalences studied in the aforementioned papers, and we demonstrate RTSs' effectiveness on several such equivalences.

Although we do not study inter-language reasoning in this paper, we have designed RTSs so as to avoid the use of any technical devices that would preclude a future generalization to the inter-language setting. To achieve this goal, we had to come up with a novel way of accounting for higher-order functions in the context of a coinductive bisimulation-like proof method, without relying on the "syntactic" devices that previous bisimulation methods use. Our solution—a new technique we call **global vs. local knowledge**—is one of the major contributions of this paper. Relying heavily on this new technique, we have proven that RTS equivalence proofs *are* transitively composable, which suggests they may serve as a superior foundation to KLRs for inter-language reasoning.

The remainder of this paper is structured as follows. In Section 2, we define $\mathsf{F}^{\mu!}$, the language under consideration. In Section 3, we motivate our key novel technical idea of global vs. local knowledge. We then present the formal development of RTSs. For pedagogical reasons, we begin in Section 4 with the presentation of a relational model for λ^{μ} (a pure subset of $\mathsf{F}^{\mu!}$ with recursive types), and then proceed in Sections 5 and 6 to extend that model to handle the full language $\mathsf{F}^{\mu!}$. In Section 7, we demonstrate the expressive power of our method by proving several challenging equivalences from the literature. In Section 8, we briefly sketch our proof of transitivity for the λ^{μ} model. (The transitivity proof for full $\mathsf{F}^{\mu!}$ is complex and sophisticated, meriting a detailed discussion that is beyond the scope of the present paper. We will present it in a follow-on paper.) Finally, in Sections 9 and 10, we discuss related and future work, and conclude.

Our online appendix provides detailed proofs of the metatheory of RTSs, both on paper and machine-checked in Coq.

2. The Language $\mathsf{F}^{\mu!}$

In Figure 1, we present the syntax and the operational semantics of $\mathsf{F}^{\mu!}$, a completely standard PCF-like language extended with products, sums, universals, existentials, general recursive types, and general reference types. Formally, we distinguish between "static" *programs* p, which are explicitly typed and do not include memory locations ℓ (since the programmer cannot write them), and "dynamic" *expressions* (or *terms*) e, which include memory locations and in which all type information is erased. The static semantics of $\mathsf{F}^{\mu!}$ (see the online appendix) defines the program typing judgment $\Delta; \Gamma \vdash p : \tau$, wherein $\Delta ::= \cdot \mid \Delta, \alpha$ and $\Gamma ::= \cdot \mid \Gamma, x : \tau$, while the dynamic semantics says how to execute expressions e. There is a straightforward erasure of programs p into expressions, written $|p|$; and when discussing equivalence informally, we will often gloss over the program/expression distinction.

$$
\begin{array}{lll}
\tau_{\mathrm{base}} & ::= & \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \\
\tau \in \mathrm{Typ} & ::= & \alpha \mid \tau_{\mathrm{base}} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \mu\alpha.\,\tau \mid \\
 & & \forall\alpha.\,\tau \mid \exists\alpha.\,\tau \mid \mathsf{ref}\ \tau \\
p \in \mathrm{Prog} & ::= & x \mid \langle\rangle \mid n \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}\ p_0\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \mid \\
 & & \langle p_1, p_2\rangle \mid p.1 \mid p.2 \mid \mathsf{inj}^1_\tau\ p \mid \mathsf{inj}^2_\tau\ p \mid \\
 & & (\mathsf{case}\ p\ \mathsf{of}\ \mathsf{inj}^1\ x \Rightarrow p_1 \mid \mathsf{inj}^2\ x \Rightarrow p_2) \mid \mathsf{roll}_\tau\ p \mid \\
 & & \mathsf{unroll}\ p \mid \mathsf{fix}\ f(x{:}\tau_1){:}\tau_2.\,p \mid p_1\ p_2 \mid \Lambda\alpha.\,p \mid p[\tau] \mid \\
 & & \mathsf{pack}\ \langle\tau, p\rangle\ \mathsf{as}\ \exists\alpha.\,\tau' \mid \mathsf{unpack}\ p_1\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ p_2 \mid \\
 & & \mathsf{ref}\ p \mid !p \mid p_1 := p_2 \mid p_1 == p_2 \\
v \in \mathrm{Val} & ::= & x \mid \langle\rangle \mid n \mid \mathsf{tt} \mid \mathsf{ff} \mid \langle v_1, v_2\rangle \mid \mathsf{inj}^1\ v \mid \mathsf{inj}^2\ v \mid \mathsf{roll}\ v \mid \\
 & & \mathsf{fix}\ f(x).\,e \mid \Lambda.\,e \mid \mathsf{pack}\ v \mid \ell \\
e \in \mathrm{Exp} & ::= & v \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \\
 & & \langle e_1, e_2\rangle \mid e.1 \mid e.2 \mid \mathsf{inj}^1\ e \mid \mathsf{inj}^2\ e \mid \\
 & & (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}^1\ x \Rightarrow e_1 \mid \mathsf{inj}^2\ x \Rightarrow e_2) \mid \mathsf{roll}\ e \mid \mathsf{unroll}\ e \mid \\
 & & e_1\ e_2 \mid e[] \mid \mathsf{pack}\ e \mid \mathsf{unpack}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 \mid \\
 & & \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid e_1 == e_2 \\
K \in \mathrm{Cont} & ::= & \bullet \mid \mathsf{if}\ K\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \langle K, e\rangle \mid \langle v, K\rangle \mid K.1 \mid K.2 \mid \\
 & & \mathsf{inj}^1\ K \mid \mathsf{inj}^2\ K \mid \mathsf{case}\ K\ \mathsf{of}\,[\mathsf{inj}^i\ x \Rightarrow e_i] \mid \\
 & & \mathsf{roll}\ K \mid \mathsf{unroll}\ K \mid K\ e \mid v\ K \mid K[] \mid \\
 & & \mathsf{pack}\ K \mid \mathsf{unpack}\ K\ \mathsf{as}\ x\ \mathsf{in}\ e \mid \\
 & & \mathsf{ref}\ K \mid !K \mid K := e \mid v := K \mid K == e \mid v == K \\
h \in \mathrm{Heap} & ::= & \mathrm{Loc} \overset{\mathrm{fin}}{\rightharpoonup} \mathrm{CVal} \qquad \mathrm{Loc} = \{\ell_1, \ell_2, \ldots\}
\end{array}
$$

$$
\begin{array}{rcll}
h, \mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \hookrightarrow & h, e_1 & \\
h, \mathsf{if}\ \mathsf{ff}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \hookrightarrow & h, e_2 & \\
h, \langle v_1, v_2\rangle.i & \hookrightarrow & h, v_i & \\
h, \mathsf{case}\ \mathsf{inj}^j\ v\ \mathsf{of}\,[\mathsf{inj}^i\ x \Rightarrow e_i] & \hookrightarrow & h, e_j[v/x] & \\
h, (\mathsf{fix}\ f(x).\,e)\ v & \hookrightarrow & h, e[(\mathsf{fix}\ f(x).\,e)/f, v/x] & \\
h, (\Lambda.\,e)[] & \hookrightarrow & h, e & \\
h, \mathsf{unpack}\ (\mathsf{pack}\ v)\ \mathsf{as}\ x\ \mathsf{in}\ e & \hookrightarrow & h, e[v/x] & \\
h, \mathsf{unroll}\ (\mathsf{roll}\ v) & \hookrightarrow & h, v & \\
h, \mathsf{ref}\ v & \hookrightarrow & h \uplus [\ell \mapsto v], \ell & \text{where } \ell \notin \mathrm{dom}(h) \\
h \uplus [\ell \mapsto v], !\ell & \hookrightarrow & h \uplus [\ell \mapsto v], v & \\
h \uplus [\ell \mapsto v], \ell := v' & \hookrightarrow & h \uplus [\ell \mapsto v'], \langle\rangle & \\
h, \ell == \ell & \hookrightarrow & h, \mathsf{tt} & \\
h, \ell == \ell' & \hookrightarrow & h, \mathsf{ff} & \text{where } \ell \neq \ell' \\
h, K[e] & \hookrightarrow & h', K[e'] & \text{where } h, e \hookrightarrow h', e'
\end{array}
$$

Figure 1. The syntax and semantics of $\mathsf{F}^{\mu!}$.

Using evaluation contexts (aka continuations) K, we define the small-step reduction relation, $\hookrightarrow$, between configurations consisting of a heap and an expression. The reduction rules shown in Figure 1 are deterministic except for the rule for allocating reference cells, which is completely non-deterministic. For technical reasons, we find it convenient to assume that allocation is in fact deterministic, but we do not care which deterministic allocator is used. Thus, we will assume that $\hookrightarrow$ is some unknown determinization of the rules shown in the figure, and our model will be parametric w.r.t. that determinization. It is easy to show that if two $\mathsf{F}^{\mu!}$ programs are contextually equivalent under all deterministic allocators, then they are contextually equivalent under a non-deterministic allocator, so nothing is lost by this assumption.

In the following, we write $\hookrightarrow^*$ for the reflexive, transitive closure of $\hookrightarrow$. We also say that a configuration diverges, denoted $h, e\!\uparrow$, if it can perform an infinite sequence of $\hookrightarrow$-reductions.

Two programs are contextually equivalent if, under any well-typed closing context C (as defined in the appendix), they either both terminate or both diverge.

Definition 1 (Contextual equivalence)**.**
Let $\Delta; \Gamma \vdash p_1 : \tau$ and $\Delta; \Gamma \vdash p_2 : \tau$. Then:

$$
\begin{array}{l}
\Delta; \Gamma \vdash p_1 \sim_{\mathrm{ctx}} p_2 : \tau \overset{\mathrm{def}}{=} \forall C, h, \tau'. \\
\quad \vdash C : (\Delta; \Gamma; \tau) \rightsquigarrow (\cdot; \cdot; \tau') \Longrightarrow (h, |C[p_1]|\!\uparrow \iff h, |C[p_2]|\!\uparrow)
\end{array}
$$

3. Global vs. Local Knowledge

Our new method of relation transition systems (RTSs) is essentially *coinductive*, following the style of existing bisimulation techniques in many respects. As explained in the introduction, coinductive reasoning makes it easy to deal with recursive features (such as recursive types and higher-order state) without requiring the use of step-indexed constructions.

The two main ways in which RTSs differ from existing bisimulation techniques are in their treatment of:

Local state. From recent work on KLRs [5, 12], we borrow the idea of using *state transition systems (STSs)* to establish invariants on how a module's local state may evolve over time. STSs enable one to encode more flexible state invariants than are expressible using "environmental" bisimulations [33, 35].

Higher-order functions. In order to reason about higher-order functions in a coinductive style, but without confining ourselves to single-language reasoning, we employ a novel technical idea: *global vs. local knowledge*.

The treatment of local state using state transition systems follows prior work very closely, so we postpone further discussion of that idea until Section 5 and focus attention here instead on motivating our new idea of global vs. local knowledge.

Coinductive Reasoning One way of formulating contextual equivalence is as the *largest adequate congruence relation* [28]. Being adequate means that if two terms of base type are related, then either they both diverge (run forever) or they both evaluate to the same value (*e.g.,* if one term evaluates to 3, then the other must evaluate to 3 as well). Being a congruence means the relation is closed under all the constructs of the language (*e.g.,* if f_1 and f_2 are related at $\tau' \to \tau$, and e_1 and e_2 are related at τ', then $f_1\,e_1$ and $f_2\,e_2$ are related at τ).

To prove using coinduction that two terms e_1 and e_2 are contextually equivalent at type τ, one must exhibit a (type-indexed) term relation L that contains (τ, e_1, e_2) and then prove that L is an adequate congruence. The relation L serves as a "generalized coinduction hypothesis", by which one proves equivalence for all pairs of terms related by L simultaneously. However, while it is possible for one to employ this kind of "brute-force" coinductive proof, it is typically not very pleasant, because proving a relation to be a congruence directly can be incredibly tedious.

Bisimulation techniques help make coinductive proofs manageable by lightening the congruence proof burden. Typically, this is achieved by only requiring one to show that L is closed under type-directed *uses* (*i.e.,* evaluation or deconstruction) of the terms it relates. This results in proof obligations that look like the following:

(1) If $(\tau, e_1, e_2) \in L$, then either $e_1\!\uparrow$ and $e_2\!\uparrow$,
or $\exists v_1, v_2.\ e_1 \hookrightarrow^* v_1$ and $e_2 \hookrightarrow^* v_2$ and $(\tau, v_1, v_2) \in L$.

(2) If $(\mathsf{int}, v_1, v_2) \in L$, then $\exists n.\ v_1 = v_2 = n$.

(3) If $(\tau' \times \tau'', v_1, v_2) \in L$, then $\exists v_1', v_1'', v_2', v_2''.\ v_i = \langle v_i', v_i'' \rangle$
and $(\tau', v_1', v_2') \in L$ and $(\tau'', v_1'', v_2'') \in L$.

(4) If $(\mu\alpha.\,\tau, v_1, v_2) \in L$, then $\exists v_1', v_2'.\ v_i = \mathsf{roll}\ v_i'$
and $(\tau[\mu\alpha.\,\tau/\alpha], v_1', v_2') \in L$.

The most problematic proof obligation is the one for function values. It usually looks something like this (simplifying fix to λ):

(5) If $(\tau' \to \tau, v_1, v_2) \in L$, then $\exists x, e_1, e_2.\ v_i = \lambda x.e_i$ and
$\forall v_1', v_2'.\ (\tau', v_1', v_2') \in \boxed{G} \Rightarrow (\tau, e_1[v_1'/x], e_2[v_2'/x]) \in L$.

In other words, if L relates function values v_1 and v_2, then applying them to any "equivalent arguments" v_1' and v_2' should produce results that are also related by L. The big question is: what is this relation G from which the arguments v_1' and v_2' are drawn?

Global vs. Local Knowledge First, some (non-standard) terminology: There are many equivalent terms in the world, but when we do a bisimulation proof, we only make a claim about some of them. So let us make a distinction between "local" and "global" knowledge about term equivalence. The relation L describes our *local knowledge*: these are the terms whose equivalence we aim to validate in our proof. The relation G, on the other hand, embodies the *global knowledge* about all terms that are equivalent in the world. In proof obligation (5), we draw equivalent function arguments from G (rather than L) since they might indeed originate from "somewhere else" in the program (some unknown client code), and thus our local knowledge L may not be sufficient to justify their equivalence. This leaves us with the question of how to define G.

Whence Global Knowledge? Coming up with a sound (and practically usable) choice for G is far from obvious, and existing bisimulation methods make a variety of different choices. For example:

- *Applicative* bisimulations [1] define G to be the syntactic identity relation on closed values.

This is a nice, simple choice, which works well for pure λ-calculus. Unfortunately, for higher-order stateful languages like $\mathsf{F}^{\mu!}$, it is unsound [18], so more advanced approaches are needed:

- *Environmental* bisimulations [36, 19, 33, 35] take G to be the "context closure" of L, *i.e.,* the relation that extends the syntactic identity relation on closed values by including closures of open values v with pairs of values (w_1, w_2) that are related by L (formally: $\{(\sigma, v[\overline{w_1/y}], v[\overline{w_2/y}]) \mid \{\overline{(\sigma', w_1, w_2)}\} \subseteq L\}$).[2]
- *Normal form* (or *open*) bisimulations [21, 34, 22, 23] sidestep the whole question by choosing a fresh variable name x and representing equivalent arguments by the same x. As a result, these bisimulations are built over open terms, and proof obligation (1) above must be updated to account for the possibility that the evaluations of e_1 and e_2 get stuck trying to deconstruct the same free variable x (more about that below).

All of these methods define global knowledge in a very "syntactic" way that is well suited to proving contextual equivalences. However, as explained in the introduction, we wish to develop a method that will be capable of generalizing to the setting of inter-language reasoning, where G may relate different languages. We therefore seek an account of global knowledge that is more "semantic".

Parameterizing Over Global Knowledge The essential difficulty in choosing G has to do with higher-order functions: if the argument type τ' is (or contains) a function type, then equivalence at τ' is very hard to characterize directly.[3] Our solution is simple: we don't try to define the global knowledge at all; instead, we take G to be a *parameter* of our model!

Our key observation is that it is not necessary to pin down exactly what G is, so long as we make our coinductive proof for L as parametric as possible with respect to it. (We will clarify what "as parametric as possible" means in Section 4.) This parametricity makes our proofs quite robust by allowing G to be instantiated in a variety of different ways. In particular, we make no assumptions whatsoever about the values that G relates at function type. For all we know, G might even include "garbage" like $(\mathsf{int} \to \mathsf{int}, 4, \mathsf{tt})$.[4]

Our approach can be viewed as a more semantic account of the idea behind normal form bisimulations (see above), which is

[2] We are glossing over a lot of details here. To be precise, environmental bisimulations are actually *sets* of L's. For more details, see Section 9.

[3] Conversely, if τ' were arrow-free (*e.g.,* in a first-order language), it would be easy to characterize equivalence at τ' directly.

[4] The ability to instantiate G with a "trashy" relation is surprisingly useful. We will make critical use of it in our transitivity proof in Section 8.

to model "equivalent arguments" as black boxes about which nothing is known. Consequently, just as for normal form bisimulations, we need to adapt proof obligation (1) above to account for the possibility that e_1 and e_2 get stuck. For normal form bisimulations, e_1 and e_2 may get stuck if they try to deconstruct a free variable x, and so normal form bisimulations loosen proof obligation (1) to allow e_1 and e_2 to reduce to terms of the form $K_1[x\ v_1]$ and $K_2[x\ v_2]$, where K_1 and K_2 are equivalent continuations and v_1 and v_2 are equivalent values. In our case, e_1 and e_2 may get stuck if they try to apply some bogus functions that are equivalent according to the global knowledge but that turn out (like 4 and tt) to not even be functions. Hence, we will allow e_1 and e_2 to reduce to terms of the form $K_1[f_1\, v_1]$ and $K_2[f_2\, v_2]$, where K_1 and K_2 are equivalent continuations, and where $\{(\tau' \to \tau, f_1, f_2), (\tau', v_1, v_2)\} \subseteq G$. In this way, the parameter G serves as a semantic analogue of free variables in normal form bisimulations.

Intuitively, although the idea of parameterizing over the global knowledge may seem surprising at first, we find it to be comfortingly reminiscent of Girard's method for modeling System F [14]. In Girard's method, a potential cycle in the definition of the logical relation for impredicative universal types $\forall\alpha.\tau$ is avoided by parameterizing over an arbitrary relational interpretation of the abstract type α. In our scenario, the problem of how to define the global knowledge is avoided by parameterizing over an arbitrary relational interpretation of *function types*. In essence, we are treating a function type $\tau_1 \to \tau_2$ as an unusual kind of abstract type: the coinductive proofs about different "modules" in a program all treat the global interpretation of $\tau_1 \to \tau_2$ abstractly, while simultaneously they each contribute to defining it.

Parameterizing over the global knowledge turns out to be *very* useful. First and foremost, it makes it easy to soundly compose our coinductive proofs for different "modules" together (and hence prove soundness of our method w.r.t. contextual equivalence). Second, it enables us to reason about open terms (Section 4) and higher-order state invariants (Section 6), replacing the use of context closure or free variables for those purposes in environmental and normal form bisimulations, respectively. Finally, it is the key to establishing transitivity for our proof method (Section 8).

4. Warmup: A Relational Model for λ^μ

To ease the presentation of relation transition systems (RTSs), we begin in this section by using the idea of global vs. local knowledge, motivated in the previous section, to define a relational model for λ^μ, a sub-language of $\mathsf{F}^{\mu!}$ containing base, function, product, sum, and recursive types, but not universal, existential, or reference types. This model cannot properly be called an RTS model, since it does not include any transition systems! The transition systems will come into play when dealing with state in Sections 5 and 6. However, by ignoring the transition-systems aspect of RTSs for the time being, we can focus attention on other aspects of the model.

Figure 2 lists the various semantic domains we will be using. Here, Type denotes the types of λ^μ, and CType denotes closed types (*i.e.,* types with no free type variables α). The next four are standard: relations on closed values, closed expressions, closed continuations, and heaps, indexed by the relevant types (in case of KRel, input and output types).

Next, we define what we call the *flexible* types, CTypeF, along with the *flexible relations*, VRelF, which are just relations on closed values indexed by such flexible types. Whereas bisimulation methods typically allow terms of arbitrary type to be included in the bisimulation, we find it useful to restrict local and global knowledges to relate only values of "flexible" types. Intuitively, these are the types at which value equivalence may depend on module-specific knowledge. In $\mathsf{F}^{\mu!}$, there will be several kinds of flexible types, but in λ^μ, the only flexible types are function types.

$$
\begin{array}{lcl}
\tau \in \mathrm{Type} & ::= & \alpha \mid \tau_{\text{base}} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\,\tau \\
\mathrm{VRel} & := & \mathrm{CType} \to \mathbb{P}(\mathrm{CVal} \times \mathrm{CVal}) \\
\mathrm{ERel} & := & \mathrm{CType} \to \mathbb{P}(\mathrm{CExp} \times \mathrm{CExp}) \\
\mathrm{KRel} & := & \mathrm{CType} \times \mathrm{CType} \to \mathbb{P}(\mathrm{CCont} \times \mathrm{CCont}) \\
\mathrm{HRel} & := & \mathbb{P}(\mathrm{Heap} \times \mathrm{Heap}) \\
\mathrm{CTypeF} & := & \{\, (\tau_1 \to \tau_2) \in \mathrm{CType} \,\} \\
\mathrm{VRelF} & := & \mathrm{CTypeF} \to \mathbb{P}(\mathrm{CVal} \times \mathrm{CVal})
\end{array}
$$

Figure 2. Semantic domains for λ^μ.

$$
\begin{array}{lcl}
\overline{R}(\tau) & := & R(\tau) \qquad \text{if } \tau \in \mathrm{CTypeF} \\
\overline{R}(\tau_{\text{base}}) & := & \mathrm{ID}_{\tau_{\text{base}}} \\
\overline{R}(\tau_1 \times \tau_2) & := & \{\, ((v_1, v_1'), (v_2, v_2')) \mid \\
 & & \qquad (v_1, v_2) \in \overline{R}(\tau_1) \wedge (v_1', v_2') \in \overline{R}(\tau_2) \,\} \\
\overline{R}(\tau_1 + \tau_2) & := & \{\, (\mathsf{inj}^1\, v_1, \mathsf{inj}^1\, v_2) \mid (v_1, v_2) \in \overline{R}(\tau_1) \,\} \cup \\
 & & \{\, (\mathsf{inj}^2\, v_1, \mathsf{inj}^2\, v_2) \mid (v_1, v_2) \in \overline{R}(\tau_2) \,\} \\
\overline{R}(\mu\alpha.\,\tau) & := & \{\, (\mathsf{roll}\ v_1, \mathsf{roll}\ v_2) \mid (v_1, v_2) \in \overline{R}(\tau[\mu\alpha.\,\tau/\alpha]) \,\}
\end{array}
$$

Figure 3. Value closure for λ^μ (if $R \in \mathrm{VRelF}$, then $\overline{R} \in \mathrm{VRel}$).

$$
\begin{array}{lcl}
\mathit{beta}(e) & := & \begin{cases} e' & \text{if } e \hookrightarrow e' \\ \text{undef} & \text{otherwise} \end{cases} \\
\mathrm{FunVal} & := & \{\, f \in \mathrm{CVal} \mid \forall v.\ \mathit{beta}(f\ v) \text{ defined} \,\} \\
R' \supseteq R & := & \forall \tau.\ R'(\tau) \supseteq R(\tau) \\
\mathrm{LK} & := & \{\, L \in \mathrm{VRelF} \to \mathrm{VRelF} \mid L \text{ is monotone w.r.t.} \subseteq \wedge \\
 & & \quad \forall G.\, \forall (f_1, f_2) \in L(G)(\tau_1 \to \tau_2).\ f_1, f_2 \in \mathrm{FunVal} \,\} \\
\mathrm{GK}(L) & := & \{\, G \in \mathrm{VRelF} \mid G \supseteq L(G) \,\}
\end{array}
$$

Figure 4. Definition of local and global knowledge for λ^μ.

In contrast, value equivalence at the remaining types—base, product, sum, and recursive types, which we call *rigid*—is fixed and agreed upon by all modules once the meaning of the flexible types is defined. This is achieved by a closure operation that takes a relation $R \in \mathrm{VRelF}$ and returns its closure $\overline{R} \in \mathrm{VRel}$. It is defined as the least fixed-point of the set of equations in Figure 3. Note that R only occurs covariantly, so $\overline{R}$ is inductively well defined, even though the type gets bigger on the r.h.s. in the case of $\mu\alpha.\,\tau$.

Local and Global Knowledge In λ^μ, a local knowledge $L \in \mathrm{LK}$ is essentially a flexible relation, except that, as shown in Figure 4, this relation is actually parameterized by the global knowledge, G. In effect, $L(G)$ describes the values that we wish to prove are equivalent, assuming that G correctly represents the global knowledge. This parameterization is necessary in order to reason about open terms, and we will see its utility below in the proof of compatibility for fix. We require that L is monotone w.r.t. G: intuitively, passing in a larger global knowledge should never result in fewer terms being related by L.

We also require that the values related by the local knowledge at function type are indeed functions, in the sense that their application to an arbitrary value should not be a stuck configuration, but should reduce at least for one step. This is a technical requirement that is used in our transitivity proof in Section 8.

We want to restrict attention to global knowledges that are closed w.r.t. the local knowledge in question: For a particular L, we define $\mathrm{GK}(L)$ to be the set of flexible relations G s.t. $G \supseteq L(G)$. This requirement makes sense since the global knowledge must by definition be a superset of any local knowledge. Observe, however, that we do not restrict what other values G relates. Indeed, G may relate values at function type that are not actually functions, or that are obviously inequivalent (*e.g.,* 4 and tt, cf. Section 3).

Relating Expressions and Continuations Figure 5 shows how equivalence is defined for expressions, e, and continuations, K.

$$
\begin{array}{lcl}
\mathbf{E}(G)(\tau) & := & \{\, (e_1, e_2) \mid (e_1{\uparrow} \wedge e_2{\uparrow}) \;\vee\; (\exists v_1, v_2.\, e_1 \hookrightarrow^* v_1 \wedge e_2 \hookrightarrow^* v_2 \wedge (v_1, v_2) \in \overline{G}(\tau)) \\
 & & \vee\, (\exists \tau', K_1, K_2, e_1', e_2'.\, e_1 \hookrightarrow^* K_1[e_1'] \wedge e_2 \hookrightarrow^* K_2[e_2'] \wedge (e_1', e_2') \in \mathbf{S}(G, G)(\tau') \wedge (K_1, K_2) \in \mathbf{K}(G)(\tau', \tau)) \,\} \\
\mathbf{K}(G)(\tau_1, \tau_2) & := & \{\, (K_1, K_2) \mid \forall (v_1, v_2) \in \overline{G}(\tau_1).\, (K_1[v_1], K_2[v_2]) \in \mathbf{E}(G)(\tau_2) \,\} \\
\mathbf{S}(R_f, R_v)(\tau) & := & \{\, (f_1\ v_1, f_2\ v_2) \mid \exists \tau'.\, (f_1, f_2) \in R_f(\tau' \to \tau) \wedge (v_1, v_2) \in \overline{R_v}(\tau') \,\} \\
\mathit{consistent}(L) & := & \forall G \in \mathrm{GK}(L).\, \forall (e_1, e_2) \in \mathbf{S}(L(G), G)(\tau).\, (\mathit{beta}(e_1), \mathit{beta}(e_2)) \in \mathbf{E}(G)(\tau) \\
\Gamma \vdash e_1 \sim_L e_2 : \tau & := & \mathit{consistent}(L) \wedge \forall G \in \mathrm{GK}(L).\, \forall \gamma_1, \gamma_2 \in \mathrm{dom}(\Gamma) \to \mathrm{CVal}. \\
 & & \quad (\forall x{:}\tau' \in \Gamma.\, (\gamma_1(x), \gamma_2(x)) \in \overline{G}(\tau')) \implies (\gamma_1 e_1, \gamma_2 e_2) \in \mathbf{E}(G)(\tau) \\
\Gamma \vdash e_1 \sim e_2 : \tau & := & \exists L.\ \Gamma \vdash e_1 \sim_L e_2 : \tau
\end{array}
$$

Figure 5. Mutually coinductive definitions of (closed) expression equivalence, $\mathbf{E} \in \mathrm{VRelF} \to \mathrm{ERel}$, and continuation equivalence, $\mathbf{K} \in \mathrm{VRelF} \to \mathrm{KRel}$, and definitions of consistency and program equivalence for λ^μ.

Specifically, we introduce two new relations, $\mathbf{E} \in \mathrm{VRelF} \to \mathrm{ERel}$ and $\mathbf{K} \in \mathrm{VRelF} \to \mathrm{KRel}$, which are defined coinductively.

Given a type τ, a local knowledge $L \in \mathrm{LK}$, and a global knowledge $G \in \mathrm{GK}(L)$, we say that two expressions are "locally" equivalent, written $(e_1, e_2) \in \mathbf{E}(G)(\tau)$, if they either both diverge or both terminate producing related values. Along the way, however, they may make calls to "external" functions, that is, functions that are related by G, but not necessarily by the local knowledge $L(G)$. More precisely, we say two closed expressions are equivalent if and only if one of the following three cases holds:

1. Both expressions diverge (run forever).
2. Both expressions run successfully to completion, producing related values.
3. Both expressions reduce after some number of steps to some expressions of the form $K_i[f_i\ v_i]$, where both the f_i and v_i are related by the global knowledge G at the appropriate types, and the continuations, K_1 and K_2, are equivalent. We say that two continuations are equivalent if instantiating them with equivalent values (according to the global knowledge G) yields equivalent expressions.

As $\mathbf{E}$ and $\mathbf{K}$ are defined mutually dependent over a complete lattice and all operations involved are monotone, we can take the meaning of these definitions to be either the least or the greatest fixed-point. We choose the greatest fixed-point, corresponding to coinduction, because this can in principle relate more terms and is somewhat easier to work with.[5]

Consistency and Program Equivalence We say that a local knowledge L is *consistent* (in Figure 5) if and only if any two functions that it declares equivalent do in fact beta-reduce to equivalent expressions when applied to equivalent arguments. In the formal definition, we parameterize over an arbitrary global knowledge $G \in \mathrm{GK}(L)$; the functions being tested for equivalence are drawn from the local knowledge, $L(G)$, while the arguments to which they are applied are drawn from the global knowledge, G.

We say that two expressions are equivalent at type τ in the context Γ, written $\Gamma \vdash e_1 \sim e_2 : \tau$, if and only if there exists a consistent local knowledge, L, that shows that $\gamma_1 e_1$ and $\gamma_2 e_2$ are equivalent at type τ for arbitrary value substitutions γ_1 and γ_2 that are related at Γ by an arbitrary global knowledge G extending L.

Two programs are equivalent simply if their type-erased versions are equivalent expressions: $\Gamma \vdash |p_1| \sim |p_2| : \tau$.

4.1 Properties of Program Equivalence and Soundness

We move on to some properties of our constructions.

$$
\frac{\Gamma, f{:}\tau' \to \tau, x{:}\tau' \vdash e_1 \sim e_2 : \tau}{\Gamma \vdash \mathsf{fix}\, f(x).\, e_1 \sim \mathsf{fix}\, f(x).\, e_2 : \tau' \to \tau}\ \textsc{Fix}
$$

$$
\frac{\Gamma \vdash e_1 \sim e_2 : \tau' \to \tau \qquad \Gamma \vdash e_1' \sim e_2' : \tau'}{\Gamma \vdash e_1\ e_1' \sim e_2\ e_2' : \tau}\ \textsc{App}
$$

$$
\frac{\Gamma \vdash p : \tau}{\Gamma \vdash |p| \sim |p| : \tau}\ \textsc{Refl} \qquad \frac{\Gamma \vdash e_2 \sim e_1 : \tau}{\Gamma \vdash e_1 \sim e_2 : \tau}\ \textsc{Symm}
$$

$$
\frac{\Gamma \vdash e_1 \sim e_2 : \tau \qquad \vdash C : (\Gamma; \tau) \rightsquigarrow (\Gamma'; \tau')}{\Gamma' \vdash C[e_1] \sim C[e_2] : \tau'}\ \textsc{Cong}
$$

$$
\frac{\Gamma, x{:}\tau' \vdash e_1 \sim e_2 : \tau \qquad \Gamma \vdash v_1 \sim v_2 : \tau'}{\Gamma \vdash e_1[v_1/x] \sim e_2[v_2/x] : \tau}\ \textsc{Subst}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash e_1' \sim e_2' : \tau \\ \forall \gamma.\, \gamma e_1 \hookrightarrow^* \gamma e_1' \qquad \forall \gamma.\, \gamma e_2 \hookrightarrow^* \gamma e_2'\end{array}}{\Gamma \vdash e_1 \sim e_2 : \tau}\ \textsc{Expand}
$$

$$
\frac{\Gamma, x{:}\tau' \vdash e_1 \sim e_2 : \tau \qquad \Gamma \vdash v_1 \sim v_2 : \tau'}{\Gamma \vdash (\lambda x.\, e_1)\ v_1 \sim e_2[v_2/x] : \tau}\ \textsc{Beta}
$$

Figure 6. Some basic properties of our equational model.

The following lemma states that consistency of local knowledges is preserved under (pointwise) union. This is important for ensuring that equivalence proofs for different subterms, which rely on different local knowledges, can be soundly composed.

Lemma 1.
If $\mathit{consistent}(L)$ and $\mathit{consistent}(L')$, then $\mathit{consistent}(L \cup L')$.

Figure 6 shows some of the basic properties of our program equivalence relation. First, we have a set of rules stating that equivalence is compatible with all the language constructs. These rules state that if two terms start with the same term constructor and their immediate subterms are component-wise equivalent, then so are the composite terms. For brevity, we just present the rules for recursive function definition (Fix) and function application (App), whose proofs are the most interesting.

We briefly sketch the proof of the Fix rule. From the premise, there exists L such that $\Gamma, f{:}\tau' \to \tau, x{:}\tau' \vdash e_1 \sim_L e_2 : \tau$. Define

$$
\begin{array}{l}
L'(G) := \{\, (\tau' \to \tau, \gamma_1 \mathsf{fix}\, f(x).\, e_1, \gamma_2 \mathsf{fix}\, f(x).\, e_2) \mid \\
\qquad \gamma_i \in \mathrm{dom}(\Gamma) \to \mathrm{CVal} \wedge \\
\qquad \forall y{:}\tau'' \in \Gamma.\, (\gamma_1(y), \gamma_2(y)) \in \overline{G}(\tau'') \,\}.
\end{array}
$$

Note here how the parameterization of L' over G provides it with a source from which to draw the closing substitutions γ_1 and γ_2.

The goal now is to prove

$$
\Gamma \vdash \mathsf{fix}\, f(x).\, e_1 \sim \mathsf{fix}\, f(x).\, e_2 : \tau' \to \tau.
$$

Showing that L' (and thus $L \cup L'$) relates any appropriately closed instances of these two values is simply a matter of unfolding def-

[5] In particular, were we to extend the language with other forms of recursion (such as while loops or primitive recursion), the coinductive interpretation would be essential for proving congruence of expression equivalence.

initions. It therefore remains to establish $\mathit{consistent}(L \cup L')$. By Lemma 1, this boils down to showing

$$(\gamma_1' e_1, \gamma_2' e_2) \in \mathbf{E}(G)(\tau)$$

for any $G \in \mathrm{GK}(L \cup L')$, where

- $\gamma_i \in \mathrm{dom}(\Gamma) \to \mathrm{CVal}$,
- $\forall y{:}\tau' \in \Gamma.\ (\gamma_1(y), \gamma_2(y)) \in \overline{G}(\tau')$,
- $(v_1, v_2) \in \overline{G}(\tau')$,
- $\gamma_i' = \gamma_i, f{\mapsto}(\gamma_i \mathsf{fix}\, f(x).\, e_i), x{\mapsto}v_i$.

Finally, as $(\gamma_1 \mathsf{fix}\, f(x).\, e_1, \gamma_2 \mathsf{fix}\, f(x).\, e_2) \in L'(G)(\tau' \to \tau) \subseteq \overline{G}(\tau' \to \tau)$, we can instantiate $\Gamma, f{:}\tau' \to \tau, x{:}\tau' \vdash e_1 \sim_L e_2 : \tau$ with γ_i' and are done.

The proof of rule APP relies on Lemma 1 as well, in order to show that the consistent local knowledges for its two premises combine to form a consistent local knowledge for the conclusion. In addition, the proof relies on the following lemma about plugging equivalent expressions or continuations into equivalent continuations, proved by mutual coinduction and case analysis.

Lemma 2. If $(K_1, K_2) \in \mathbf{K}(G)(\tau', \tau)$, then:

1. $(e_1, e_2) \in \mathbf{E}(G)(\tau')$
   $\implies (K_1[e_1], K_2[e_2]) \in \mathbf{E}(G)(\tau)$.
2. $(K_1', K_2') \in \mathbf{K}(G)(\tau'', \tau')$
   $\implies (K_1[K_1'], K_2[K_2']) \in \mathbf{K}(G)(\tau'', \tau)$.

To prove APP, we apply the first case of this lemma with $e_i := \gamma_i\, e_i$ and $K_i := \bullet\ \gamma_i e_i'$, which leaves us to prove K_1 and K_2 to be equivalent according to $\mathbf{K}$. Unfolding the definition of $\mathbf{K}$, we have to show that for arbitrary equivalent values v_1 and v_2, $(v_1\ \ \gamma_1 e_1', v_2\ \ \gamma_2 e_2')$ is in $\mathbf{E}$, for which we apply Lemma 2 again with $e_i := \gamma_i\, e_i'$ and $K_i := v_i\ \bullet$. Then we are left to prove $v_1\ \bullet$ and $v_2\ \bullet$ equivalent, *i.e.*, that $(v_1\ \ v_1', v_2\ \ v_2')$ is in $\mathbf{E}$ for arbitrary equivalent values v_1' and v_2', which follows from the third disjunct of the $\mathbf{E}$ definition.

As a consequence of these "compatibility" rules, by a straightforward induction on the typing derivation, we can show that equivalence is reflexive on well-typed programs (rule REFL). This corresponds to the "fundamental property" of logical relations. Equivalence is also symmetric: this follows trivially from the symmetric nature of our definition. Likewise, by induction on the typing derivation of contexts, we can show that our equivalence is a congruence: if two equivalent terms are placed in the same contexts, the resulting compositions are equivalent.

Next, we have a substitutivity property for values, an expansion law for pure execution steps, and finally a direct corollary of these two, namely β-equivalence (on value arguments).

We move to a key lemma about $\mathbf{E}$. Given a consistent local knowledge L, if the global knowledge extends L with some additional external knowledge $\boldsymbol{\mathcal{R}}$, then the third case in the definition of $\mathbf{E}$ can be restricted so that it applies only to external function calls (*i.e.,* calls to functions related by $\boldsymbol{\mathcal{R}}$, not by L).

Lemma 3 (External call)**.** For any $\mathit{consistent}(L)$, any $G \in \mathrm{GK}(L)$ and $\boldsymbol{\mathcal{R}} \in \mathrm{VRelF}$, we have:

$$G = L(G) \cup \boldsymbol{\mathcal{R}} \implies \mathbf{E}(G) = \mathbf{E}^{\boldsymbol{\mathcal{R}}}(G)$$

where the definition of $\mathbf{E}^{\boldsymbol{\mathcal{R}}}$ is the same as $\mathbf{E}$ except that, in the third disjunct, $\mathbf{S}(G, G)$ is replaced by $\mathbf{S}(\boldsymbol{\mathcal{R}}, G)$.

The $\supseteq$ follows directly from the observation that $\boldsymbol{\mathcal{R}} \subseteq G$. To prove the other direction, we essentially have to eliminate all uses of the third disjunct of $\mathbf{E}$ where the functions being invoked are related by $G \setminus \boldsymbol{\mathcal{R}}$. Since all such functions are by definition in $L(G)$, and since we know $\mathit{consistent}(L)$, we can in fact always "inline" the equivalence proofs for all such function calls.

A corollary of Lemma 3 (for $G = \mu R.L(R)$ and $\boldsymbol{\mathcal{R}} = \emptyset$) is adequacy, which says that equivalent closed terms either both diverge or both terminate returning proper values. In particular, they never get stuck during evaluation.

Lemma 4 (Adequacy)**.** If $\vdash e_1 \sim e_2 : \tau$, then:

$$(e_1\!\uparrow \wedge\ e_2\!\uparrow) \vee (\exists v_1, v_2.\ e_1 \hookrightarrow^* v_1 \wedge e_2 \hookrightarrow^* v_2)$$

Finally, combining adequacy and congruence, we show our main soundness theorem: for well-typed programs, our equivalence relation is included in contextual equivalence.

Theorem 5 (Soundness)**.** Let $\Gamma \vdash p_1 : \tau$ and $\Gamma \vdash p_2 : \tau$. If $\Gamma \vdash |p_1| \sim |p_2| : \tau$, then $\Gamma \vdash p_1 \sim_{\mathrm{ctx}} p_2 : \tau$.

4.2 Example

Consider the following example concerning streams as functions (taken from Sumii and Pierce [36]):

$$\begin{array}{rcl}
\tau & := & \mu\alpha.\, \mathsf{unit} \to \mathsf{int} \times \alpha \\
\mathsf{ones} : \mathsf{unit} \to \mathsf{int} \times \tau & := & \mathsf{fix}\, f(x).\, \langle 1, \mathsf{roll}\ f \rangle \\
\mathsf{twos} : \mathsf{unit} \to \mathsf{int} \times \tau & := & \mathsf{fix}\, f(x).\, \langle 2, \mathsf{roll}\ f \rangle \\
\mathsf{succ} : \tau \to \tau & := & \mathsf{fix}\, f(s).\, \mathsf{let}\ \langle n, s' \rangle = \mathsf{unroll}\ s\ \langle\rangle\ \mathsf{in} \\
 & & \qquad\qquad \mathsf{roll}\ \lambda x.\, \langle n{+}1, f\ s' \rangle
\end{array}$$

The goal is to show $\vdash \mathsf{roll\ twos} \sim \mathsf{succ}\ (\mathsf{roll\ ones}) : \tau$.

Constructing a Suitable Local Knowledge Note that we have

$$\mathsf{succ}\ (\mathsf{roll\ ones}) \hookrightarrow^* \mathsf{roll\ twos}'$$

for $\mathsf{twos}' := \lambda x.\, \langle 1 + 1, \mathsf{succ}\ \ (\mathsf{roll}\ \ \mathsf{ones}) \rangle$. We define a local knowledge L that relates exactly twos and twos':

$$L(G) := \{\, (\mathsf{unit} \to \mathsf{int} \times \tau, \mathsf{twos}, \mathsf{twos}')\, \}$$

Proving Its Consistency For $G \in \mathrm{GK}(L)$ we must show:

$$(\langle 2, \mathsf{roll\ twos} \rangle, \langle 1{+}1, \mathsf{succ}\ (\mathsf{roll\ ones}) \rangle) \in \mathbf{E}(G)(\mathsf{int} \times \tau)$$

Using the second case in $\mathbf{E}$ and the definition of $\overline{G}$, this reduces to showing $(2, 2) \in \overline{G}(\mathsf{int})$ and $(\mathsf{roll\ twos}, \mathsf{roll\ twos}') \in \overline{G}(\tau)$. The former is trivial. The latter is equivalent to $(\mathsf{twos}, \mathsf{twos}') \in \overline{G}(\mathsf{unit} \to \mathsf{int} \times \tau)$, which holds by construction because G extends $L(G)$.

Showing the Programs Related By It It remains to show:

$$\forall G \in \mathrm{GK}(L).\ (\mathsf{roll\ twos}, \mathsf{succ}\ (\mathsf{roll\ ones})) \in \mathbf{E}(G)(\tau)$$

Again using the second case in $\mathbf{E}$ we end up having to show $(\mathsf{roll\ twos}, \mathsf{roll\ twos}') \in \overline{G}(\tau)$, which we have already done above.

5. Local State Transition Systems: A Review

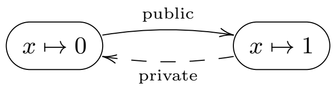
In the next section, we will extend our model to account for abstract types and state. The key extension there will be to incorporate support for *state transition systems (STSs)* in the style of Dreyer *et al.*'s recent work on KLRs [5, 12]. We use this section to review the basic idea behind that work by means of a concrete example.

Perhaps the simplest example that demonstrates the utility of STSs for reasoning about local state is the "well-bracketed state change" example, due originally to Jacob Thamsborg (which is itself a variant of Pitts and Stark's "awkward" example) [5, 29]:

$$\begin{array}{lcl}
\tau & := & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int} \\
v_1 & := & \lambda f.\, (f\ \langle\rangle; f\ \langle\rangle; 1) \\
e_2 & := & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda f.\, (x := 0; f\ \langle\rangle; x := 1; f\ \langle\rangle; !x)
\end{array}$$

(Here and in later examples we write $\lambda x.\, e$ for $\mathsf{fix}\, f(x).\, e$ when f is not free in e.) The goal is to show that v_1 and e_2 are contextually equivalent (at the type τ). To see intuitively why they are equivalent, observe that e_2 allocates a fresh (local) location x, which initially points to 0, and then returns a function value—call it v_2. When v_1 and v_2 are applied (at any point in the future) to some callback function f, they both call f twice. Before the first and second calls, v_2 will set x to 0 and 1, respectively. Thus, even if the second call to f internally applies v_2 again, the last thing x will get set to (before it is dereferenced) is always 1. Note that this reasoning assumes that control flow in the language is "well-bracketed"—in the sense that the call to f cannot escape its current continuation—and also that the language lacks exceptions (both are true for $\mathsf{F}^{\mu!}$).

In order to prove this example, we need to be able to establish an invariant concerning the local state of v_2. However, we really need more than just a simple fixed invariant because the only such invariant that holds here is the one stating that x points to *either* 0 or 1, and that is not strong enough. This is where STSs come in. Dreyer *et al.* [12] prove this example using the following STS:



The states of this STS represent the possible "abstract" states in which the functions we are proving equivalent may find themselves, and associated with each abstract state is a "physical" relation on heaps. In the left state, the heap of the second program must contain $[x \mapsto 0]$, and in the right state, it must contain $[x \mapsto 1]$. (No restrictions are placed on the heap of the first program, since v_1 does not manipulate any local state.)

The accessibility relation between these abstract states is governed by two transition relations (preorders on the state space): a *private* and a *public* one. The rough intuition is that the private (or "full") transition relation includes all legal state transitions, while the smaller, public transition relation governs the legal transitions that *function calls* can make (when their behavior is viewed extensionally). For proving equivalence of v_1 and v_2, we require transitions of some kind from each of these states to the other because repeated applications of v_2 will indeed result in x flip-flopping back and forth between 0 and 1. However, the transition from $x \mapsto 1$ to $x \mapsto 0$ may be considered private, not public, because when the behavior of v_1 is viewed end-to-end, it will never start with $x \mapsto 1$ and end with $x \mapsto 0$. Moreover, restricting the public transition relation in this way is essential to making the proof go through: since the second call to f starts with $x \mapsto 1$ and is required (by definition) to make a public transition, we know that, when it returns, x must still point to 1, and thus $!x$ will evaluate to 1.

What we have described here is the high-level idea of the equivalence proof of v_1 and e_2, which is essentially the same when using our RTSs as it is when using STS-indexed KLRs. One difference is that, with KLRs, the proof is driven by the need to show that (v_1, e_2)—and thus also (v_1, v_2)—are in the logical relation at the type τ. With RTSs, there is no logical relation defining what it means for functions to be related at type τ. Instead, in typical coinductive style, we need to enter v_1 and v_2 into the "local knowledge" of our RTS, which in turn generates a proof obligation to show that these function values do in fact behave equivalently when passed arguments that are related by the global knowledge. Fundamentally, this ends up being only a minor change to the structure of the proof. (We will see the formal RTS proof of this example in Section 7.1.)

6. Relation Transition Systems for $\mathsf{F}^{\mu!}$

In this section, we present our full-blown relation transition system (RTS) model for $\mathsf{F}^{\mu!}$. This RTS model generalizes the model from Section 4 in a superficially very simple way: whereas previously we proved two terms equivalent by exhibiting a consistent local knowledge L, we now do so by exhibiting a consistent *world* W.

$$
\begin{array}{lll}
\tau \in \text{Type} & ::= & \ldots \mid \forall\alpha.\,\tau \mid \exists\alpha.\,\tau \mid \mathsf{ref}\,\tau \mid \mathbf{n} \\
\text{CTypeF} & := & \ldots \cup \{\, (\forall\alpha.\,\tau) \in \text{CType} \,\} \cup \{\, \mathsf{ref}\,\tau \in \text{CType} \,\} \\
 & & \cup \{\, \mathbf{n} \in \text{TypeName} \,\}
\end{array}
$$

Figure 7. Semantic domains for $\mathsf{F}^{\mu!}$.

$$
\begin{array}{lll}
\overline{R}(\tau) & := & R(\tau) \qquad \text{if } \tau \in \text{CTypeF} \\
 & \vdots & \\
\overline{R}(\exists\alpha.\,\tau) & := & \{\, (\mathsf{pack}\ v_1, \mathsf{pack}\ v_2) \mid \exists\tau'.\,(v_1, v_2) \in \overline{R}(\tau[\tau'/\alpha]) \,\}
\end{array}
$$

Figure 8. Value closure for $\mathsf{F}^{\mu!}$ (if $R \in \text{VRelF}$, then $\overline{R} \in \text{VRel}$).

$$
\begin{array}{l}
\text{DepWorld}(P) \ := \ \{\, (\mathsf{S}, \sqsubseteq, \sqsubseteq_{\text{pub}}, \mathsf{N}, \mathsf{L}, \mathsf{H}) \in \\
\quad \text{Set} \times \mathbb{P}(\mathsf{S} \times \mathsf{S}) \times \mathbb{P}(\mathsf{S} \times \mathsf{S}) \times \mathbb{P}(\text{TypeName}) \times \\
\quad (\mathsf{S}_P \to \mathsf{S} \to \text{VRelF} \to \text{VRelF}) \times (\mathsf{S}_P \to \mathsf{S} \to \text{VRelF} \to \text{HRel}) \mid \\
\ \sqsubseteq, \sqsubseteq_{\text{pub}} \text{ are preorders} \wedge \sqsubseteq_{\text{pub}} \text{ is a subset of } \sqsubseteq \wedge \\
\ \mathsf{L} \text{ monotone in 1st arg w.r.t. } \sqsubseteq_P\text{, in 2nd w.r.t. } \sqsubseteq\text{, in 3rd w.r.t. } \subseteq \wedge \\
\ \mathsf{H} \text{ monotone in 3rd arg w.r.t. } \subseteq \wedge \\
\ \forall s_{\text{P}}, s, G. \\
\quad (\forall \mathbf{n} \notin \mathsf{N}.\ \mathsf{L}(s_{\text{P}})(s)(G)(\mathbf{n}) = \emptyset) \wedge \\
\quad (\forall \tau_1, \tau_2.\, \forall (f_1, f_2) \in \mathsf{L}(s_{\text{P}})(s)(G)(\tau_1 \to \tau_2).\ f_1, f_2 \in \text{FunVal}) \wedge \\
\quad (\forall \alpha, \tau.\, \forall (v_1, v_2) \in \mathsf{L}(s_{\text{P}})(s)(G)(\forall\alpha.\,\tau).\ v_1, v_2 \in \text{TyFunVal}) \,\}
\end{array}
$$

where

$$
\begin{array}{lll}
\text{FunVal} & := & \{\, f \in \text{CVal} \mid \forall v.\ \mathit{beta}(f\ v) \text{ defined} \,\} \\
\text{TyFunVal} & := & \{\, v \in \text{CVal} \mid \mathit{beta}(v[]) \text{ defined} \,\} \\
\mathit{beta}(e) & := & \begin{cases} e' & \text{if } \forall h.\ h, e \hookrightarrow h, e' \\ \text{undef} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lll}
\text{World} & := & \{\, W \in \text{DepWorld}(\{*\}, \{(*, *)\}) \,\} \\
\text{LWorld} & := & \{\, w \in \text{DepWorld}(W_{\text{ref}}.\mathsf{S}, W_{\text{ref}}.\sqsubseteq) \mid \\
 & & \qquad \forall s_{\text{ref}}, s, G, \tau.\ w.\mathsf{L}(s_{\text{ref}})(s)(G)(\mathsf{ref}\,\tau) = \emptyset \,\}
\end{array}
$$

Figure 9. Definition of worlds (relation transition systems) and auxiliary RTS definitions.

Worlds Worlds are state transition systems (equipped with "public" and "private" transitions, just as described in Section 5) that control how the local knowledge of a module and the properties of its local state may evolve over time. Formally (Figure 9), a world consists of: the transition system's (possibly infinite) state space (S); the private (or full) transition relation ($\sqsubseteq$) and a smaller public transition relation ($\sqsubseteq_{\text{pub}}$), both preorders; a set of type names that are used to represent abstract types (N); a mapping from states to local knowledges (L); and a mapping from states to heap relations (H). For now, ignore the distinction between different kinds of worlds as well as the S_P, s_P, and $\sqsubseteq_P$ objects in that figure.

As before, the local knowledge (at each state) is parameterized by—and must be monotone in—the global knowledge G. The same applies to the heap relation, which describes pairs of subheaps that are "owned" by the RTS. The parameter G here provides a way of referring to the global equivalence on values when establishing invariants on the contents of local heaps; this is especially critical in dealing with higher-order state. While the local knowledge mapping must be monotone in its state index (w.r.t. $\sqsubseteq$), the heap relation mapping need not be. Indeed, since a module's local state is hidden from the environment, there is no reason to require that heaps related in one state will continue to be related in future states (*e.g.,* in the example in Section 5, $x \mapsto 0$ or $x \mapsto 1$ but not both).

We have seen in the previous section section how a local knowledge and its closure relate values at λ^{μ} types. This carries over to the full setting. But how do we deal with the additional types of $\mathsf{F}^{\mu!}$, *i.e.,* with universal, existential, and reference types?

$$
\begin{array}{lcl}
W_{\mathrm{ref}}.\mathsf{S} & := & \{\, s_{\mathrm{ref}} \in \mathbb{P}_{\mathrm{fin}}(\mathrm{CType} \times \mathrm{Loc} \times \mathrm{Loc}) \mid \\
& & \quad \forall(\tau, \ell_1, \ell_2) \in s_{\mathrm{ref}}.\, \forall(\tau', \ell_1', \ell_2') \in s_{\mathrm{ref}}. \\
& & \qquad (\ell_1 = \ell_1' \implies \tau = \tau' \wedge \ell_2 = \ell_2') \wedge \\
& & \qquad (\ell_2 = \ell_2' \implies \tau = \tau' \wedge \ell_1 = \ell_1') \,\} \\
W_{\mathrm{ref}}.\sqsubseteq & := & \subseteq \\
W_{\mathrm{ref}}.\sqsubseteq_{\mathrm{pub}} & := & \subseteq \\
W_{\mathrm{ref}}.\mathsf{N} & := & \emptyset \\
W_{\mathrm{ref}}.\mathsf{L}(s_{\mathrm{ref}})(G)(\mathsf{ref}\,\tau) & := & \{\, (\ell_1, \ell_2) \mid (\tau, \ell_1, \ell_2) \in s_{\mathrm{ref}} \,\} \\
W_{\mathrm{ref}}.\mathsf{H}(s_{\mathrm{ref}})(G) & := & \{\, (h_1, h_2) \mid \\
& & \quad \mathrm{dom}(h_1) = \{\, \ell_1 \mid \exists \tau, \ell_2.\, (\tau, \ell_1, \ell_2) \in s_{\mathrm{ref}} \,\} \\
& & \quad \wedge\ \mathrm{dom}(h_2) = \{\, \ell_2 \mid \exists \tau, \ell_1.\, (\tau, \ell_1, \ell_2) \in s_{\mathrm{ref}} \,\} \\
& & \quad \wedge\ \forall(\tau, \ell_1, \ell_2) \in s_{\mathrm{ref}}.\, (h_1(\ell_1), h_2(\ell_2)) \in \overline{G}(\tau) \,\}
\end{array}
$$

Figure 10. W_{ref} provides the meaning of reference types.

Treatment of Universal and Existential Types Universal types, like function types, are considered flexible. That is, the local knowledge can relate any values at any closed type $\forall\alpha.\,\tau$, as long as, when instantiated, those values can run for at least one step.

Existential types, like product types, are considered rigid, and thus their interpretation is given by the value closure (Figure 8). Note that the witness of related packages must be the *same* type τ'. How, then, do we support reasoning about parametricity?

The key is that the witness type τ' may be an abstract *type name*. We extend the syntax of types in our RTS model with type names $\mathbf{n}$ (Figure 7), and the local knowledge of a world can pick a subset of these names and interpret them however it wants. To avoid conflicts with other worlds, the choice of names must be recorded in the N component of the world (no other names may be interpreted).

Treatment of Reference Types Reference types are considered flexible, but they really are a special case. Intuitively, the collection of all reference types can be seen as a separate module that is used by all other modules. Consequently, we construct a designated world W_{ref} (explained below) that interprets $\mathsf{ref}\,\tau$, and bar ordinary worlds from relating anything at such types. We therefore distinguish between two kinds of worlds: *local worlds* and *full worlds*. For conciseness, both are defined in terms of the same underlying structure of *dependent worlds*.

A dependent world is a world as described above, except that it is parameterized by a preorder $P = (\mathsf{S}_P, \sqsubseteq_P)$ that its local knowledge and heap relation may depend on (via the $s_P \in \mathsf{S}_P$ argument of L and H in Figure 9). Intuitively, P is the state transition system of some other world and s_P is that world's current state. Thus, a full world $W \in \mathrm{World}$ is simply a world depending on nothing (a singleton set). In contrast, a local world $w \in \mathrm{LWorld}$ is a world that depends on—will later be "linked" with—W_{ref} and does not itself relate any values at reference types.

As a matter of notational convenience: if $W \in \mathrm{World}$ and $s \in W.\mathsf{S}$, then we will often just write $W.\mathsf{L}(s)$ for $W.\mathsf{L}(*)(s)$, and similarly for the H component. (We use the dot notation to project components out of a world.)

The World for Reference Types Figure 10 defines $W_{\mathrm{ref}} \in \mathrm{World}$, the world that provides the meaning of reference types. Its states are finite ternary relations (between a type τ and two locations ℓ_1, ℓ_2) that are functional in the location arguments. They associate each allocated location on the left with the corresponding one on the right and the type of values stored. The relations are finite because only a finite number of locations can ever be allocated. And, as dictated by the language, they can only grow over time.

Its local knowledge $W_{\mathrm{ref}}.\mathsf{L}(s_{\mathrm{ref}})$ relates precisely the locations related by the current state s_{ref}, at the corresponding reference types. The heap relation $W_{\mathrm{ref}}.\mathsf{H}(s_{\mathrm{ref}})$ relates heaps that contain exactly the locations related by the current state s_{ref} and that store (globally) related values at those locations. Note the critical use of the global knowledge parameter G in defining $W_{\mathrm{ref}}.\mathsf{H}$.

$$
H_1 \otimes H_2 := \{(h_1 \uplus h_1', h_2 \uplus h_2') \mid (h_1, h_2) \in H_1 \wedge (h_1', h_2') \in H_2\}
$$

$$
\begin{array}{lcl}
w{\uparrow}.\mathsf{S} & := & W_{\mathrm{ref}}.\mathsf{S} \times w.\mathsf{S} \\
w{\uparrow}.\sqsubseteq & := & \{\, (p, p') \mid p.1 \sqsubseteq p'.1 \wedge p.2 \sqsubseteq p'.2 \,\} \\
w{\uparrow}.\sqsubseteq_{\mathrm{pub}} & := & \{\, (p, p') \mid p.1 \sqsubseteq_{\mathrm{pub}} p'.1 \wedge p.2 \sqsubseteq_{\mathrm{pub}} p'.2 \,\} \\
w{\uparrow}.\mathsf{N} & := & w.\mathsf{N} \\
w{\uparrow}.\mathsf{L}(s_{\mathrm{ref}}, s)(G) & := & W_{\mathrm{ref}}.\mathsf{L}(s_{\mathrm{ref}})(G) \cup w.\mathsf{L}(s_{\mathrm{ref}})(s)(G) \\
w{\uparrow}.\mathsf{H}(s_{\mathrm{ref}}, s)(G) & := & W_{\mathrm{ref}}.\mathsf{H}(s_{\mathrm{ref}})(G) \otimes w.\mathsf{H}(s_{\mathrm{ref}})(s)(G)
\end{array}
$$

$$
\begin{array}{lcl}
(w_1 \otimes w_2).\mathsf{S} & := & w_1.\mathsf{S} \times w_2.\mathsf{S} \\
(w_1 \otimes w_2).\sqsubseteq & := & \{\, (p, p') \mid p.1 \sqsubseteq p'.1 \wedge p.2 \sqsubseteq p'.2 \,\} \\
(w_1 \otimes w_2).\sqsubseteq_{\mathrm{pub}} & := & \{\, (p, p') \mid p.1 \sqsubseteq_{\mathrm{pub}} p'.1 \wedge p.2 \sqsubseteq_{\mathrm{pub}} p'.2 \,\} \\
(w_1 \otimes w_2).\mathsf{N} & := & w_1.\mathsf{N} \uplus w_2.\mathsf{N} \\
(w_1 \otimes w_2).\mathsf{L}(s_{\mathrm{ref}})(s_1, s_2)(G) & := & w_1.\mathsf{L}(s_{\mathrm{ref}})(s_1)(G) \\
& \cup & w_2.\mathsf{L}(s_{\mathrm{ref}})(s_2)(G) \\
(w_1 \otimes w_2).\mathsf{H}(s_{\mathrm{ref}})(s_1, s_2)(G) & := & w_1.\mathsf{H}(s_{\mathrm{ref}})(s_1)(G) \\
& \otimes & w_2.\mathsf{H}(s_{\mathrm{ref}})(s_2)(G)
\end{array}
$$

Figure 11. Lifting ($\uparrow \in \mathrm{LWorld} \to \mathrm{World}$) and separating conjunction ($\otimes \in \mathrm{LWorld} \times \mathrm{LWorld} \to \mathrm{LWorld}$) of worlds.

Lifting and Separating Conjunction of Local Worlds Now, if we have a local world $w \in \mathrm{LWorld}$, then we can link it with W_{ref}, thereby *lifting* it to a full world $w{\uparrow} \in \mathrm{World}$. This operation is defined in Figure 11. The full world's transition system is the synchronous product of W_{ref}'s and w's. Its local knowledge relates values iff they are related by either component's local knowledge, and its heap relation relates heaps iff they can be split into disjoint parts that are related by $W_{\mathrm{ref}}.\mathsf{H}$ and $w.\mathsf{H}$, respectively. Note how the state s_{ref} of the reference world W_{ref} is passed to $w.\mathsf{L}$ and $w.\mathsf{H}$ along with the state of w itself.

Similarly, given two local worlds $w_1, w_2 \in \mathrm{LWorld}$ that own disjoint sets of abstract types (*i.e.,* $w_1.\mathsf{N} \cap w_2.\mathsf{N} = \emptyset$), we can construct their separating conjunction $w_1 \otimes w_2 \in \mathrm{LWorld}$. The definition is given in Figure 11. Note how the same shared state s_{ref} is passed to the L and H components of both w_1 and w_2. Separating conjunction of worlds is a generalization of the union operation on local knowledges, which we have seen in Section 4 to be critical for composing proofs (cf. Lemma 1).

Program Equivalence With these constructions in hand, we can now describe the definition of program equivalence in Figure 12.

We say that two expressions are equivalent ($\sim$) iff there exists a local world w that (1) does not depend on a particular choice of names to represent its abstract types; (2) is *stable*; and (3) when lifted, relates the expressions. Stability means that the local world's heap relation in some sense tolerates "environmental" changes: whenever the shared world W_{ref} is advanced to a future state, s'_{ref}, then w should be able to respond to that change by moving to a public future state, s', such that any local heaps that were related previously by $w.\mathsf{H}$ are still related at s'. This is a very technical condition that is required for soundness but is satisfied trivially in the common case that $w.\mathsf{H}$ does not actually depend on its s_{ref} parameter. See the end of this section for further discussion.

A world W (such as $w{\uparrow}$) relates two expressions ($\sim_W$) iff (3a) it is *inhabited*; (3b) it is *consistent*; and (3c) the expressions, when closed using related substitutions, are related by the expression relation (see below). Inhabitance says there exists a state at which $W.\mathsf{H}$ relates the empty heaps. Consistency is essentially the same as for λ^{μ}, but extended straightforwardly to universal types. In all these definitions, the global knowledge G is drawn from $\mathrm{GK}(W)$. As before, this enforces that G must *contain* the local knowledge. At reference types, and at abstract type names owned by W, however, $\mathrm{GK}(W)$ also enforces that G must *not extend* the local knowledge. Intuitively, this is because W should completely control the meaning of those types. Furthermore, since G is state-indexed, it must, like the local knowledge of W, be monotone w.r.t. $\sqsubseteq$.

$$
\begin{array}{lll}
R' \geq^{\mathcal{N}}_{\mathrm{ref}} R & := & R' \supseteq R \wedge \forall \tau.\, R'(\mathsf{ref}\,\tau) = R(\mathsf{ref}\,\tau) \wedge \forall \mathbf{n} \in \mathcal{N}.\, R'(\mathbf{n}) = R(\mathbf{n}) \\
\mathrm{GK}(W) & := & \{\, G \in W.\mathsf{S} \to \mathrm{VRelF} \mid G \text{ is monotone w.r.t. } \sqsubseteq \wedge \forall s.\, G(s) \geq^{W.\mathsf{N}}_{\mathrm{ref}} W.\mathsf{L}(s)(G(s)) \,\} \\
\mathbf{E}_W(G)(s_0, s)(\tau) & := & \{\, (e_1, e_2) \mid \forall (h_1, h_2) \in W.\mathsf{H}(s)(G(s)).\, \forall h_1^{\mathrm{F}}, h_2^{\mathrm{F}}.\, h_1 \uplus h_1^{\mathrm{F}} \text{ defined} \wedge h_2 \uplus h_2^{\mathrm{F}} \text{ defined} \implies \\
& & \quad ((h_1 \uplus h_1^{\mathrm{F}}, e_1)\uparrow \wedge (h_2 \uplus h_2^{\mathrm{F}}, e_2)\uparrow) \\
& & \quad \vee (\exists h_1', h_2', v_1, v_2.\, (h_1 \uplus h_1^{\mathrm{F}}, e_1) \hookrightarrow^* (h_1' \uplus h_1^{\mathrm{F}}, v_1) \wedge (h_2 \uplus h_2^{\mathrm{F}}, e_2) \hookrightarrow^* (h_2' \uplus h_2^{\mathrm{F}}, v_2) \wedge \\
& & \qquad \exists s'.\, s' \sqsupseteq s \wedge s' \sqsupseteq_{\mathrm{pub}} s_0.\, (h_1', h_2') \in W.\mathsf{H}(s')(G(s')) \wedge (v_1, v_2) \in \overline{G(s')}(\tau)) \\
& & \quad \vee (\exists h_1', h_2', \tau', K_1, K_2, e_1', e_2'. \\
& & \qquad (h_1 \uplus h_1^{\mathrm{F}}, e_1) \hookrightarrow^* (h_1' \uplus h_1^{\mathrm{F}}, K_1[e_1']) \wedge (h_2 \uplus h_2^{\mathrm{F}}, e_2) \hookrightarrow^* (h_2' \uplus h_2^{\mathrm{F}}, K_2[e_2']) \wedge \\
& & \qquad \exists s' \sqsupseteq s.\, (h_1', h_2') \in W.\mathsf{H}(s')(G(s')) \wedge (e_1', e_2') \in \mathbf{S}(G(s'), G(s'))(\tau') \wedge \\
& & \qquad\qquad \forall s'' \sqsupseteq_{\mathrm{pub}} s'.\, \forall G' \supseteq G.\, (K_1, K_2) \in \mathbf{K}_W(G')(s_0, s'')(\tau', \tau)) \,\} \\
\mathbf{K}_W(G)(s_0, s)(\tau_1, \tau_2) & := & \{\, (K_1, K_2) \mid \forall (v_1, v_2) \in \overline{G(s)}(\tau_1).\, (K_1[v_1], K_2[v_2]) \in \mathbf{E}_W(G)(s_0, s)(\tau_2) \,\} \\
\mathbf{S}(R_f, R_v)(\tau) & := & \{\, (f_1\ v_1, f_2\ v_2) \mid \exists \tau'.\, (f_1, f_2) \in R_f(\tau' \to \tau) \wedge (v_1, v_2) \in \overline{R_v}(\tau') \,\} \\
& \cup & \{\, (f_1[], f_2[]) \mid \exists \tau_1, \tau_2.\, \tau = \tau_1[\tau_2/\alpha] \wedge (f_1, f_2) \in R_f(\forall \alpha.\, \tau_1) \,\} \\
\mathit{inhabited}(W) & := & \forall G \in \mathrm{GK}(W).\, \exists s_0.\, (\emptyset, \emptyset) \in W.\mathsf{H}(s_0)(G(s_0)) \\
\mathit{consistent}(W) & := & \forall G \in \mathrm{GK}(W).\, \forall s.\, \forall \tau.\, \forall (e_1, e_2) \in \mathbf{S}(W.\mathsf{L}(s)(G(s)), G(s))(\tau).\, (\mathit{beta}(e_1), \mathit{beta}(e_2)) \in \mathbf{E}_W(G)(s, s)(\tau) \\
\mathit{stable}(w) & := & \forall G \in \mathrm{GK}(w\!\uparrow).\, \forall s_{\mathrm{ref}}, s.\, \forall (h_1, h_2) \in w.\mathsf{H}(s_{\mathrm{ref}})(s)(G(s_{\mathrm{ref}}, s)). \\
& & \forall s'_{\mathrm{ref}} \sqsupseteq s_{\mathrm{ref}}.\, \forall (h^1_{\mathrm{ref}}, h^2_{\mathrm{ref}}) \in W_{\mathrm{ref}}.\mathsf{H}(s'_{\mathrm{ref}})(G(s'_{\mathrm{ref}}, s)).\, h^1_{\mathrm{ref}} \uplus h_1 \text{ defined} \wedge h^2_{\mathrm{ref}} \uplus h_2 \text{ defined} \implies \\
& & \exists s' \sqsupseteq_{\mathrm{pub}} s.\, (h_1, h_2) \in w.\mathsf{H}(s'_{\mathrm{ref}})(s')(G(s'_{\mathrm{ref}}, s')) \\
\Delta; \Gamma \vdash e_1 \sim_W e_2 : \tau & := & \mathit{inhabited}(W) \wedge \mathit{consistent}(W) \wedge \forall G \in \mathrm{GK}(W).\, \forall s.\, \forall \delta \in \Delta \to \mathrm{CType}.\, \forall \gamma_1, \gamma_2 \in \mathrm{dom}(\Gamma) \to \mathrm{CVal}. \\
& & \quad (\forall x{:}\tau' \in \Gamma.\, (\gamma_1(x), \gamma_2(x)) \in \overline{G(s)}(\delta\tau')) \implies (\delta\tau, \gamma_1 e_1, \gamma_2 e_2) \in \mathbf{E}_W(G)(s, s) \\
\Delta; \Gamma \vdash e_1 \sim e_2 : \tau & := & \forall \mathcal{N} \in \mathbb{P}(\mathrm{TypeName}).\, \mathcal{N} \text{ countably infinite} \implies \exists w.\, w.\mathsf{N} \subseteq \mathcal{N} \wedge \mathit{stable}(w) \wedge \Delta; \Gamma \vdash e_1 \sim_{w\uparrow} e_2 : \tau
\end{array}
$$

Figure 12. Mutually coinductive definitions of expression equivalence, $\mathbf{E}_W \in \mathrm{GK}(W) \to W.\mathsf{S} \times W.\mathsf{S} \to \mathrm{ERel}$, and continuation equivalence, $\mathbf{K}_W \in \mathrm{GK}(W) \to W.\mathsf{S} \times W.\mathsf{S} \to \mathrm{KRel}$, and definitions of world consistency and program equivalence for $\mathsf{F}^{\mu!}$.

Expression and Continuation Equivalence The new definitions of $\mathbf{E}$ and $\mathbf{K}$ are also given in Figure 12. Notice that they are now defined relative to a world W (as $\mathbf{E}_W$ and $\mathbf{K}_W$) and that their types have changed to $\mathrm{GK}(W) \to W.\mathsf{S} \times W.\mathsf{S} \to \mathrm{ERel}$ and $\mathrm{GK}(W) \to W.\mathsf{S} \times W.\mathsf{S} \to \mathrm{KRel}$, respectively: they take both an "initial" state, s_0, and a "current" state, s, as arguments.

Given a world W, a global knowledge $G \in \mathrm{GK}(W)$, states $s_0, s \in W.\mathsf{S}$, and a type τ, we say that two expressions are "locally" equivalent, written $(e_1, e_2) \in \mathbf{E}_W(G)(s_0, s)(\tau)$, iff, when executed starting in heaps that satisfy the heap relation of W at the current state s, then (as before) one of three cases holds:

1. Both expressions diverge (run forever).
2. Both expressions run successfully to completion, producing related values. In this case, the values need not be related in the current state s, but rather in some future state, $s' \sqsupseteq s$, which, however, must also be a public future state of the initial state of the expression: $s' \sqsupseteq_{\mathrm{pub}} s_0$. Moreover, this future state must be consistent with the resulting heaps: $(h_1', h_2') \in W.\mathsf{H}(s')(G(s'))$.
3. Both expressions reduce after some number of steps to some expressions of the form $K_i[e_i']$, where e_i' are either both applications or both instantiations that are related at some future state $s' \sqsupseteq s$. This state must be consistent with the corresponding heaps. Finally, the continuations, K_1 and K_2, are equivalent under any public future state, $s'' \sqsupseteq_{\mathrm{pub}} s'$, and any (pointwise) larger global knowledge, $G' \supseteq G$.

   We restrict s'' to be a public future state of s' rather than an arbitrary future state because the end-to-end effect of a function call (or universal instantiation) is assumed to always be a public transition. For this assumption to be sound, in return we will have to ensure that the end-to-end behaviors of equivalent function bodies indeed change the state only into public future states. This is why we thread the s_0 argument through the coinduction and check that the final RTS state in the previous case (2) is a public future state of s_0.

   The intuitive reason for quantifying over a larger global knowledge G' is this: At the point when the continuations are run, not only might W's state s' have changed to a future state s'', but also the states of all other "modules", which is reflected by the growth of the global knowledge.

In all three cases, the definition quantifies over frame heaps, h_1^{F} and h_2^{F}: the execution of e_1 and e_2 should not update any disjoint part of the heap that they do not own according to the heap relation of the current state. This framing aspect of our definition is a semantic version of the frame rule of separation logic and allows us to concentrate the reasoning about the heaps only on the parts of the heaps accessed by the program. (Baking the frame rule into the semantic model is quite common in more recent models of separation logic [9, 39], essentially because it allows one to avoid proving any "safety monotonicity" or "frame" properties of the operational semantics itself.)

Properties of Program Equivalence and Soundness The properties presented in Section 4 for λ^{μ}, as well as the corresponding soundness proofs, extend naturally to the full model, but we omit further details here due to space considerations. We plan to present them in an expanded version of this paper. Meanwhile, we refer the reader to our online appendix (see the link at the end of the paper).

A Word on Dependency In the examples in the next section, we will not actually rely on the ability of local knowledges and heap relations to depend on the state of the reference world W_{ref}. Consequently, the stability property in the definition of program equivalence will be trivially satisfied (by choosing $s' := s$). However, dependent worlds are of critical importance in the RTS transitivity proof for full $\mathsf{F}^{\mu!}$. We will report on this issue in a future paper.

7. Examples

In this section, we present several example RTS equivalence proofs. For convenience, we drop the s_{ref} argument from the L and H components of local worlds since we do not use it in our examples.

7.1 Well-Bracketed State Change

Recall the example from Section 5 and its high-level proof-sketch using an STS. We will now show in some formal detail how this proof is done using our method. Concretely, we prove $\vdash v_1 \sim e_2 : \tau$, where:

$$\begin{array}{lll} \tau & := & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int} \\ v_1 & := & \lambda f.\, (f\ \langle\rangle; f\ \langle\rangle; 1) \\ v_2 & := & \lambda f.\, (x := 0; f\ \langle\rangle; x := 1; f\ \langle\rangle; !x) \\ e_2 & := & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ v_2 \end{array}$$

Constructing a Suitable RTS We construct an RTS w that we will then show to be consistent and to relate v_1 and e_2.

Since the programs don't involve abstract types, we can define $w.\mathsf{N}$ to be empty. The STS that we build into w is essentially the one from Section 5. A state $s \in w.\mathsf{S}$ is to be understood as follows: for each running instance of e_2, identified by the location ℓ that that instance initially allocated, $s(\ell)$ says whether it is in the (pictorially) left state (ℓ points to 0) or in the right one (ℓ points to 1). Accordingly, the heap relation $w.\mathsf{H}$ at state s is just $\{(\emptyset, s)\}$. Finally, the local knowledge $w.\mathsf{L}$ at state s relates v_1 with $v_2[\ell/x]$ for any location ℓ belonging to an instance.

$$\begin{array}{ll} w.\mathsf{S} & := \mathrm{Loc} \xrightarrow{\text{fin}} \{0,1\} \subseteq \mathrm{Heap} \\ w.\sqsubseteq & := \{(s, s') \mid \mathrm{dom}(s) \subseteq \mathrm{dom}(s')\} \\ w.\sqsubseteq_{\mathrm{pub}} & := \{(s, s') \in w.\sqsubseteq \mid \forall(\ell, 1) \in s.\ (\ell, 1) \in s'\} \\ w.\mathsf{N} & := \emptyset \\ w.\mathsf{L}(s)(G)(\tau) & := \{(v_1, v_2[\ell/x]) \mid \ell \in \mathrm{dom}(s)\} \\ w.\mathsf{H}(s)(G) & := \{(\emptyset, s)\} \end{array}$$

It is easy to see that $w \in$ LWorld. In particular, $w.\mathsf{L}$ and $w.\mathsf{H}$ are monotone as required. Note that $\mathit{stable}(w)$ (the dependency is vacuous) and that $\mathit{inhabited}(w\!\uparrow)$ for $s_0 = (\emptyset, \emptyset)$. To show $\vdash v_1 \sim_{w\uparrow} e_2 : \tau$, two parts remain.

Proving Its Consistency Establishing $\mathit{consistent}(w\!\uparrow)$ is the real meat of the proof. Consider two functions related by $w\!\uparrow\!.\mathsf{L}$ at a state $(s^1_{\mathrm{ref}}, s_1)$. Clearly, one is v_1 and the other is $v_2[\ell/x]$ for some $\ell \in \mathrm{dom}(s_1)$. Now suppose we are given related arguments $(v'_1, v'_2) \in \overline{G_1(s^1_{\mathrm{ref}}, s_1)}(\mathsf{unit} \to \mathsf{unit})$. We need to show:

$$\begin{array}{c} ((v'_1\ \langle\rangle; v'_1\ \langle\rangle; 1), (\ell := 0; v'_2\ \langle\rangle; \ell := 1; v'_2\ \langle\rangle; !\ell)) \\ \in \mathbf{E}_{w\uparrow}(G_1)((s^1_{\mathrm{ref}}, s_1), (s^1_{\mathrm{ref}}, s_1))(\mathsf{int}) \end{array}$$

Note that for $(h_1, h_2) \in w.\mathsf{H}(s_1)(G(s^1_{\mathrm{ref}}, s_1))$ we know by construction that $h_1 = \emptyset$ and $h_2 = s_1$. Consequently, for any frame heaps $h^{\mathrm{F}}_1, h^{\mathrm{F}}_2$, we have

$$\begin{array}{l} h_2 \uplus h^{\mathrm{F}}_2, (\ell := 0; v'_2\ \langle\rangle; \ell := 1; v'_2\ \langle\rangle; !\ell) \hookrightarrow^* \\ (s_1 \backslash \ell) \uplus [\ell {\mapsto} 0] \uplus h^{\mathrm{F}}_2, (v'_2\ \langle\rangle; \ell := 1; v'_2\ \langle\rangle; !\ell) \end{array}$$

where $s_1 \backslash \ell$ denotes the restriction of s_1 to domain $\mathrm{dom}(s_1) \setminus \{\ell\}$. Since $h_1 \uplus h^{\mathrm{F}}_1, (v'_1\ \langle\rangle; v'_1\ \langle\rangle; 1) \hookrightarrow^* h_1 \uplus h^{\mathrm{F}}_1, (v'_1\ \langle\rangle; v'_1\ \langle\rangle; 1)$, it suffices, by the third disjunct in the definition of $\mathbf{E}_{w\uparrow}$, to find $s'_1 \sqsupseteq s_1$ such that:

1. $(\emptyset, (s_1 \backslash \ell) \uplus [\ell {\mapsto} 0]) \in w.\mathsf{H}(s'_1)(G_1(s^1_{\mathrm{ref}}, s'_1))$
2. $\forall(s^2_{\mathrm{ref}}, s_2) \sqsupseteq_{\mathrm{pub}} (s^1_{\mathrm{ref}}, s'_1). \forall G_2 \supseteq G_1.$
   $((\bullet; v'_1\ \langle\rangle; 1), (\bullet; \ell := 1; v'_2\ \langle\rangle; !\ell)) \in$
   $\mathbf{K}_{w\uparrow}(G_2)((s^1_{\mathrm{ref}}, s_1), (s^2_{\mathrm{ref}}, s_2))(\mathsf{unit}, \mathsf{int})$

Naturally, we pick $s'_1 = (s_1 \backslash \ell) \uplus [\ell {\mapsto} 0] \sqsupseteq s_1$. Then (1) holds by construction of w and it remains to show (2).

After repeating the whole procedure one more time, we arrive at the goal of finding $s'_2 \sqsupseteq s_2$ such that:

3. $(\emptyset, (s_2 \backslash \ell) \uplus [\ell {\mapsto} 1]) \in w.\mathsf{H}(s'_2)(G_2(s^2_{\mathrm{ref}}, s'_2))$
4. $\forall(s^3_{\mathrm{ref}}, s_3) \sqsupseteq_{\mathrm{pub}} (s^2_{\mathrm{ref}}, s'_2). \forall G_3 \supseteq G_2.$
   $((\bullet; 1), (\bullet; !\ell)) \in$
   $\mathbf{K}_{w\uparrow}(G_3)((s^1_{\mathrm{ref}}, s_1), (s^3_{\mathrm{ref}}, s_3))(\mathsf{unit}, \mathsf{int})$

Naturally, we pick $s'_2 = (s_2 \backslash \ell) \uplus [\ell {\mapsto} 1] \sqsupseteq s_2$. Then (3) holds by construction of w and it remains to show (4).

We observe that, for any $s_3 \sqsupseteq_{\mathrm{pub}} s'_2$, it must be that $s_3(\ell) = 1$ since $s'_2(\ell) = 1$. Hence for $(h'_1, h'_2) \in w.\mathsf{H}(s_3)(G(s^3_{\mathrm{ref}}, s_3))$ we know by construction $h'_2(\ell) = 1$. Consequently, for any frame heaps $h^{\mathrm{F}'}_1, h^{\mathrm{F}'}_2$, we have:

$$h'_2 \uplus h^{\mathrm{F}'}_2, (\langle\rangle; !\ell) \hookrightarrow^* h'_2 \uplus h^{\mathrm{F}'}_2, 1$$

Since of course $h'_1 \uplus h^{\mathrm{F}'}_1, (\langle\rangle; 1) \hookrightarrow^* h'_1 \uplus h^{\mathrm{F}'}_1, 1$ and $(1, 1) \in \overline{G_3(s^3_{\mathrm{ref}}, s_3)}(\mathsf{int})$ by definition, we are done if we can show $(s^3_{\mathrm{ref}}, s_3) \sqsupseteq_{\mathrm{pub}} (s^1_{\mathrm{ref}}, s_1)$. Indeed, this is easy to verify.

Showing the Programs Related By It Given how we constructed our RTS, this final goal is fairly easy to accomplish. Formally, we must show

$$(v_1, e_2) \in \mathbf{E}_{w\uparrow}(G)((s_{\mathrm{ref}}, s), (s_{\mathrm{ref}}, s))(\tau)$$

for any G, s_{ref}, s. Note that if $(h_1, h_2) \in w.\mathsf{H}(s)(G(s_{\mathrm{ref}}, s))$, then for some fresh ℓ we have $h_2, e_2 \hookrightarrow h_2 \uplus [\ell {\mapsto} 0], v_2[\ell/x]$ and, of course, $h_1, v_1 \hookrightarrow^* h_1, v_1$. It therefore suffices to find $s' \sqsupseteq_{\mathrm{pub}} s$ such that the following hold:

5. $(h_1, h_2 \uplus [\ell {\mapsto} 0]) \in w.\mathsf{H}(s')(G(s_{\mathrm{ref}}, s'))$
6. $(v_1, v_2[\ell/x]) \in \overline{G(s_{\mathrm{ref}}, s')}(\tau)$

We pick $s' = s \uplus [\ell {\mapsto} 0]$. Note that s' is well-defined because ℓ is fresh (so $\ell \notin \mathrm{dom}(s)$), and that $s' \sqsupseteq_{\mathrm{pub}} s$ as required. To show (6), it suffices by definition of GK to show $(v_1, v_2[\ell/x]) \in w.\mathsf{L}(s')(G(s_{\mathrm{ref}}, s'))(\tau)$. This holds by construction of w and s', and so does (5).

7.2 A Free Theorem

The next example demonstrates the treatment of universal types, and the fact that our method may be used to prove at least some simple so-called "free theorems" [40]. Suppose $\vdash p : \forall\alpha.\, \alpha$ and $|p| = v$. We want to prove that $h, v[]\!\uparrow$ for any h.

We start by applying REFL to obtain $\vdash v \sim v : \forall\alpha.\, \alpha$. This gives us w with $\vdash v \sim_{w\uparrow} v : \forall\alpha.\, \alpha$ and $w.\mathsf{N} \subseteq \mathrm{TypeName} \backslash \{\mathbf{n}\}$ for some arbitrary $\mathbf{n}$ of our choosing. We now instantiate $G \in \mathrm{GK}(w\!\uparrow)$ to be the least solution of $\forall s.\ G(s) = w\!\uparrow\!.\mathsf{L}(s)(G(s))$. From $\vdash v \sim_{w\uparrow} v : \forall\alpha.\, \alpha$ we then get

$$(v, v) \in \mathbf{E}_{w\uparrow}(G)(s_0, s_0)(\forall\alpha.\, \alpha)$$

where s_0 is the state witnessing $\mathit{inhabited}(w\!\uparrow)$. Consequently, we get $s \sqsupseteq_{\mathrm{pub}} s_0$ such that:

1. $(v, v) \in \overline{G(s)}(\forall\alpha.\, \alpha) = G(s)(\forall\alpha.\, \alpha)$
2. $(\emptyset, \emptyset) \in (w\!\uparrow).\mathsf{H}(s)(G(s))$

From (1), the construction of G, and $\mathit{consistent}(w\!\uparrow)$, we derive

$$(\mathit{beta}(v[]), \mathit{beta}(v[])) \in \mathbf{E}_{w\uparrow}(G)(s, s)(\tau)$$

for any τ. So, in particular, we have

$$(\mathit{beta}(v[]), \mathit{beta}(v[])) \in \mathbf{E}_{w\uparrow}(G)(s, s)(\mathbf{n}).$$

In fact, due to the construction of G, Lemma 3 tells us

$$(\mathit{beta}(v[]), \mathit{beta}(v[])) \in \mathbf{E}^{\boldsymbol{\mathcal{R}}}_{w\uparrow}(G)(s, s)(\mathbf{n})$$

for $\boldsymbol{\mathcal{R}} = \lambda s.\emptyset$. Together with (2), this allows only two cases for any heap h: either $h, \mathit{beta}(v[])$ diverges (then we are done), or it terminates and the resulting values are related by $\overline{G(s')}(\mathbf{n})$ for some $s' \sqsupseteq_{\mathrm{pub}} s$. However, because $W_{\mathrm{ref}}.\mathsf{N} = \emptyset$ and $w.\mathsf{N} \subseteq \mathrm{TypeName} \backslash \{\mathbf{n}\}$, we know $\overline{G(s')}(\mathbf{n}) = \overline{w\!\uparrow\!.\mathsf{L}(s')(G(s'))}(\mathbf{n}) = \emptyset$, which rules out that second case.

7.3 Twin Abstraction

This final example (originally due to Ahmed *et al.* [5]) demonstrates the interaction of local state with abstract types.

$$\begin{aligned}
\tau \quad &:= \quad \exists\alpha.\,\exists\beta.\,(\mathsf{unit} \to \alpha) \times (\mathsf{unit} \to \beta) \times (\alpha \times \beta \to \mathsf{bool}) \\
e_1 \quad &:= \quad \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\
&\qquad \mathsf{pack}\ \langle \mathsf{int}, \mathsf{pack}\ \langle \mathsf{int}, \lambda\_.\, x := !x + 1; !x, \\
&\qquad\qquad\qquad\qquad\quad \lambda\_.\, x := !x + 1; !x, \\
&\qquad\qquad\qquad\qquad\quad \lambda p.\, p.1 = p.2\rangle\rangle \\
e_2 \quad &:= \quad \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\
&\qquad \mathsf{pack}\ \langle \mathsf{int}, \mathsf{pack}\ \langle \mathsf{int}, \lambda\_.\, x := !x + 1; !x, \\
&\qquad\qquad\qquad\qquad\quad \lambda\_.\, x := !x + 1; !x, \\
&\qquad\qquad\qquad\qquad\quad \lambda p.\, \mathsf{ff}\rangle\rangle
\end{aligned}$$

Both e_1 and e_2 return a name generator ADT consisting of two abstract types α and β, together with a function for generating a fresh name of type α, a function for generating a fresh name of type β, and a function for comparing an α name and a β name for equality. Both implementations represent names as integers, and in e_1, the comparison operation really tests the names for equality. In e_2, however, the comparison just always returns false right away. Nevertheless, the two programs are contextually equivalent because the α names and the β names are generated by the *same* underlying integer counter, and thus no value can be both an α name and a β name at the same time.

We now show the construction of an RTS w that can be used to prove $\vdash e_1 \sim e_2 : \tau$. Let a countably infinite $\mathcal{N} \in \mathbb{P}(\mathrm{TypeName})$ be given. Since Loc is also countably infinite, we can think of $\mathcal{N}$ as being split into $\{\mathbf{n}^\alpha_\ell \mid \ell \in \mathrm{Loc}\} \uplus \{\mathbf{n}^\beta_\ell \mid \ell \in \mathrm{Loc}\}$. We define w as follows:

$$\begin{aligned}
w.\mathsf{S} \quad &:= \{s \in \mathrm{Loc} \times \mathrm{Loc} \overset{\mathrm{fin}}{\rightharpoonup} \mathbb{P}(\mathbb{N}_{>0}) \times \mathbb{P}(\mathbb{N}_{>0}) \mid \\
&\qquad \mathrm{dom}(s)\ \text{partial bijection}\ \wedge \\
&\qquad \forall(\ell_1, \ell_2, S_1, S_2) \in s.\ S_1 \cap S_2 = \emptyset\} \\
w.\sqsubseteq \quad &:= w.\sqsubseteq_{\mathrm{pub}} \\
w.\sqsubseteq_{\mathrm{pub}} \quad &:= \{(s, s') \in w.\mathsf{S} \times w.\mathsf{S} \mid \forall(\ell_1, \ell_2, S_1, S_2) \in s. \\
&\qquad \exists S_1' \supseteq S_1, S_2' \supseteq S_2.\ (\ell_1, \ell_2, S_1', S_2') \in s'\} \\
w.\mathsf{N} \quad &:= \mathcal{N} \\
w.\mathsf{L}(s)(G) \ &:= \\
&\quad \{(\mathbf{n}^\alpha_{\ell_1}, n, n) \mid \exists\ell_2, S_1, S_2.\ (\ell_1, \ell_2, S_1, S_2) \in s \wedge n \in S_1\} \\
&\ \uplus \{(\mathbf{n}^\beta_{\ell_1}, n, n) \mid \exists\ell_2, S_1, S_2.\ (\ell_1, \ell_2, S_1, S_2) \in s \wedge n \in S_2\} \\
&\ \uplus \{((\mathsf{unit} \to \mathbf{n}^\alpha_{\ell_1}), {+\!\!+}\ell_1, {+\!\!+}\ell_2) \mid (\ell_1, \ell_2) \in \mathrm{dom}(s)\} \\
&\ \uplus \{((\mathsf{unit} \to \mathbf{n}^\beta_{\ell_1}), {+\!\!+}\ell_1, {+\!\!+}\ell_2) \mid (\ell_1, \ell_2) \in \mathrm{dom}(s)\} \\
&\ \uplus \{((\mathbf{n}^\alpha_{\ell_1} \times \mathbf{n}^\beta_{\ell_1} \to \mathsf{bool}), (\lambda p.\, p.1 = p.2), (\lambda p.\, \mathsf{ff})) \mid \\
&\qquad \exists\ell_2.\ (\ell_1, \ell_2) \in \mathrm{dom}(s)\} \\
w.\mathsf{H}(s)(G) &:= \{(h_1, h_2) \mid \mathrm{dom}(h_1) = \Pi_1(\mathrm{dom}(s)) \wedge \\
&\qquad \mathrm{dom}(h_2) = \Pi_2(\mathrm{dom}(s)) \wedge \\
&\qquad \forall(\ell_1, \ell_2, S_1, S_2) \in s. \\
&\qquad\quad h_1(\ell_1) = h_2(\ell_2) = \max(\{0\} \uplus S_1 \uplus S_2)\}
\end{aligned}$$

Here, ${+\!\!+}\ell$ is short for $\lambda\_.\, \ell := !\ell{+}1; !\ell$; and $\Pi_i : \mathbb{P}(\mathrm{Loc}\times\mathrm{Loc}) \to \mathbb{P}(\mathrm{Loc})$ for the appropriate projection function.

Similar to the world construction in Section 7.1, states $s \in w.\mathsf{S}$ are functions defined for those locations (ℓ_1, ℓ_2) that, intuitively, were allocated in an instance of e_1 and e_2, respectively. They are mapped to sets S_1 and S_2 of positive integers, representing[6] the current inhabitants of the abstract types α and β, respectively, for that instance. The crucial invariant here is that S_1 and S_2 are always disjoint. The local knowledge $w.\mathsf{L}$ declares e_1's functions equivalent to those of e_2; it also defines the meaning of type $\mathbf{n}^\alpha_{\ell_1}$ as the identity relation restricted to those numbers that inhabit α in the instance pair identified by ℓ_1 (and similarly for $\mathbf{n}^\beta_{\ell_1}$ and β).

According to this interpretation, the transition relation only allows S_1 and S_2 to grow (the distinction between public and private transitions is not needed for this example). Finally, $w.H$ says that the related heaps at state s are any (h_1, h_2) where h_i contains exactly those locations allocated in instances of e_i, and each such location stores the largest value handed out so far (no matter if at α or β). This latter condition is critical to ensure that S_1 and S_2 stay disjoint in each instance.

Using this world, it is straightforward to finish the proof. Details can be found in the online appendix.

7.4 World Generators

As one may observe from the examples in Sections 7.1 and 7.3, worlds must often describe "n-ary" state spaces, where each state consists of n copies of states drawn from some simpler state space, one copy for each dynamic instance of the object or ADT. Thus, it would be convenient if one were able to reason about program equivalence under the degenerate case of a single copy (*i.e.,* $n = 1$). Fortunately, it is not hard to (i) define a *world generator* that, given a single-instance world, automatically performs the multiplexing; and (ii) show that program equivalence in a single-instance world implies equivalence in the automatically multiplexed world. For space reasons, we do not present the details here but refer the interested reader to our online appendix, where the proofs of the examples from Sections 7.1 and 7.3 are simplified greatly with the help of such a world generator.

8. Transitivity

In this section, we briefly sketch our proof of transitivity of RTS equivalence in the pure, simplified setting of λ^μ (as defined in Section 4). Transitivity also holds for the full RTS model described in Section 6, but the proof of that result is much more complex (involving a notion of weak isomorphism between worlds), so we leave its presentation to a future paper.

The main lemma we would like to prove is the following:

Lemma 6. If $\Gamma \vdash e_1 \sim_{L_1} e_2 : \tau$ and $\Gamma \vdash e_2 \sim_{L_2} e_3 : \tau$, then there exists L such that $\Gamma \vdash e_1 \sim_L e_3 : \tau$.

Naturally, we can expect L to be some sort of composition of the given local knowledges L_1 and L_2. Defining this composition is, however, quite subtle. The problem is that the local knowledge takes the global knowledge G as a parameter, but then what global knowledges G_{L_1} and G_{L_2} should be passed on to L_1 and L_2 in constructing L? Assuming we somehow pick G_{L_1} and G_{L_2} appropriately, L can be defined as follows:

$$L(G)(\tau) \ := \ L_1(G_{L_1})(\tau) \circ L_2(G_{L_2})(\tau)$$

where $\circ$ stands for ordinary relation composition.

The key part of the proof is showing transitivity of $\mathbf{E}$:

$$\begin{aligned}
\forall G \in \mathrm{GK}(L).\ &(e_1', e_2') \in \mathbf{E}(G_{L_1})(\tau) \wedge \\
&(e_2', e_3') \in \mathbf{E}(G_{L_2})(\tau) \implies (e_1', e_3') \in \mathbf{E}(G)(\tau)
\end{aligned}$$

In order to prove this, we want the disjunct of $\mathbf{E}$ by which e_1' and e_2' are related to match the disjunct of $\mathbf{E}$ by which e_2' and e_3' are related (recall the three disjuncts in the definition of $\mathbf{E}$). To illustrate, say e_1' and e_2' are related because they reduce to related values (second disjunct). Now consider the three cases regarding e_2' and e_3':

(1) They both diverge. Fortunately, this contradicts our assumption about e_2', so this case cannot arise.

(2) They are related for the same reason as e_1' and e_2' are—*i.e.,* e_2' and e_3' reduce to related values. This is the "good" case. Relying on determinacy of reduction, we are done if we can show transitivity of the value relation. Formally, we need to show that $\overline{G_{L_1}}(\tau) \circ \overline{G_{L_2}}(\tau) \subseteq \overline{G}(\tau)$.

[6] To keep the definitions as simple as possible, the state space includes some states that actual program behaviour cannot result in (but that nevertheless are consistent with the property we want to prove).

(3) They reduce to related function calls with related continuations. It is unclear how to make sense of this situation, so we want to rule it out!

In order to make case (2) straightforward to show, while simultaneously ruling out case (3) from consideration, we will need to define G_{L_1} and G_{L_2} carefully.

The key idea is as follows: for each pair of function values $(f_1, f_3) \in G$, come up with a value, v_2, that (1) uniquely identifies f_1 and f_3, and that (2) is not a normal function, but rather a "bad" value that gets stuck when applied to an argument. The first requirement allows us to ensure $G_{L_1}(\tau) \circ G_{L_2}(\tau) = G(\tau)$, as needed in proving transitivity of the value relation, by relating $(f_1, v_2) \in G_{L_1}$ and $(v_2, f_3) \in G_{L_2}$. The second requirement, together with Lemma 3, rules out the "bad" case (3) above.

Formally, since Type and CVal are countable sets, there exists an injective function $\mathbf{I} \in \text{Type} \times \text{Type} \times \text{CVal} \times \text{CVal} \rightarrow \mathbb{N}$. Using this function, one can decompose $G \in \text{VRelF}$ as follows:

$$G_{(1)}(\tau_1 \rightarrow \tau_2) := \{\, (f_1, \mathbf{I}(\tau_1, \tau_2, f_1, f_3)) \mid (f_1, f_3) \in G(\tau_1 \rightarrow \tau_2) \,\}$$
$$G_{(2)}(\tau_1 \rightarrow \tau_2) := \{\, (\mathbf{I}(\tau_1, \tau_2, f_1, f_3), f_3) \mid (f_1, f_3) \in G(\tau_1 \rightarrow \tau_2) \,\}$$

Taking G_{L_1} to be $G_{(1)}$ is, however, incorrect because if L_1 relates any values at function type, then the global knowledge will not be closed w.r.t. it (*i.e.,* $G_{(1)} \notin \text{GK}(L_1)$). To address this problem, we simply close $G_{(1)}$ accordingly, *i.e.,* we take G_{L_1} to be the least solution to the fixed-point equation $G_{L_1} = L_1(G_{L_1}) \cup G_{(1)}$ (and similarly for G_{L_2}).

With these definitions, we can show $\overline{G_{L_1}}(\tau) \circ \overline{G_{L_2}}(\tau) = \overline{G}(\tau)$ (if $G \in \text{GK}(L)$). Moreover, if we are in case (3) above, then Lemma 3 tells us that the functions being called—say, (f_1, f_2)—are *really* external, meaning that they are related by $G_{(2)}$. But by construction, this means that f_2 is an integer and thus e'_2 gets stuck, contradicting the prior assumption that e'_1 and e'_2 reduce to values.

For further details, we refer the interested reader to our online appendix.

9. Related Work and Discussion

Our method of relation transition systems builds closely on ideas from several prior techniques. We compare here only to the most immediately related work.

Kripke Logical Relations The method of *logical relations* is an old and fundamentally important technique for proving a variety of deep properties in *higher-typed* languages, such as strong normalization [14] and parametricity [31]. Although they were originally geared toward reasoning about pure λ-calculi, logical relations have been successfully generalized to reason about state. In Pitts and Stark's seminal work on *Kripke logical relations (KLRs)* [29], logical relations are indexed by *possible worlds*, which characterize the runtime environment (*e.g.,* the assumptions about heaps) under which two programs are considered to be equivalent.

In the most recent work on KLRs, Dreyer *et al.* [5, 12] showed how to generalize Pitts and Stark's technique to reason about (1) modules whose correctness proofs require fine-grained control over how local state evolves over time, and (2) ML-like languages with higher-order state. W.r.t. point (1), they model possible worlds as *state transition systems (STSs)*, as we have described in Section 5. RTSs adopt Dreyer *et al.*'s STS technique directly, and thus it is relatively straightforward to port all the $\mathsf{F}^{\mu!}$ equivalence proofs given in their papers from using KLRs to using RTSs.

W.r.t. point (2), the challenge of supporting higher-order state in Kripke logical relations is that a naive attempt to construct a model of general reference types leads to a circularity. Intuitively, ℓ_1 and ℓ_2 are related at $\mathsf{ref}\,\tau$ under a possible world W iff W encodes the invariant that the heaps of the two programs map ℓ_1 and ℓ_2 to values v_1 and v_2 that are logically related at type τ. But how can the logical relation be indexed by a possible world W, which itself is defined in terms of the logical relation? If τ is restricted to base type (*e.g.,* int), there's no issue because the logical relation at int is simply the identity relation, but at higher type we have a problem.

Dreyer *et al.* handle higher-order state by means of Appel, McAllester, and Ahmed's technique of *step-indexed logical relations (SILRs)* [6, 2]. That is, they cut the aforementioned semantic circularity by indexing the model by a natural number ("step index") k, which represents the number of steps left on "the clock" and which gets decremented every time around the cycle between logical relations and possible worlds.

While step-indexing is a powerful weapon, it can be somewhat annoying to work with, due to the tedious threading of step counting throughout proofs [11]. More importantly, it seems fundamentally difficult to compose SILR proofs transitively. Ahmed studied the transitivity problem in her first paper on binary SILRs [3]. There, she observed a serious problem in naively proving that Appel and McAllester's original binary SILRs formed a PER. She proposed a way of regaining transitivity of SILRs for a pure language with recursive types [3], but her approach relies on baking syntactic typing assumptions into the model. Such an approach is unlikely to scale to reasoning about the intermediate and low-level languages of a certified compiler (one of our ultimate goals), which in general may be untyped. Moreover, we are not aware of any successful attempts to generalize her technique to SILRs for richer languages.

RTSs employ the idea of global knowledge in order to avoid the need for step-indexing in modeling higher-order state. Specifically, by parameterizing the heap relations in our worlds over the global knowledge G, we give heap invariants a way of referring to the global value equivalence, which is essentially what the step-indexed stratification of Kripke worlds is trying to achieve as well. In contrast to SILRs, we already have a proof that RTS equivalence for $\mathsf{F}^{\mu!}$ is transitive, and our method does not bake in any syntactic typing assumptions. In fact, our transitivity proof *depends* on instantiating global knowledge parameters with "trashy" (syntactically ill-typed) relations.

Bisimulations Aside from their general coinductive flavor, RTSs are closely related to two different bisimulation techniques.

From *normal form* (or *open*) bisimulations [32, 21, 34, 22, 23], we take the idea of treating unknown equivalent functions as black boxes. In particular, our expression equivalence relation $\mathbf{E}$, which deals explicitly with the possibility (in its third disjunct) that related terms may get stuck by calling unknown functions, is *highly* reminiscent of the formulation of normal form bisimulations. The main difference is that we express the notion of "stuckness" semantically, via the global knowledge parameter G, whereas normal form bisimulations express it syntactically by requiring related stuck terms to share a common head variable.

Normal form bisimulations draw much inspiration from game-semantics models [25], and our distinction between global and local knowledge has a seemingly gamey flavor as well. We leave a deeper study of the connection to game semantics to future work.

Sumii *et al.*'s *environmental* bisimulations (aka "relation-sets bisimulations") are perhaps the most powerful form of bisimulation yet developed for ML-like languages [27, 36, 19, 33, 35]. As the latter name suggests, these bisimulations are not term relations, but *sets* $\mathcal{X}$ whose elements are themselves term relations R (possibly paired with some additional environmental information, such as knowledge about the state of the heap). In essence, each $R \in \mathcal{X}$ defines some piece of "local knowledge" (following our terminology) about program equivalence. In order to show $\mathcal{X}$ to be a bisimulation, one must check that for all $R \in \mathcal{X}$, uses of terms related by R will never result in observably different outcomes and will always produce values that are related by some $R' \in \mathcal{X}$ s.t. $R' \supseteq R$.

Viewed in terms of RTSs, one can understand an environmental bisimulation $\mathcal{X}$ as effectively defining an abstract state space, with each $R \in \mathcal{X}$ as a distinct state. However, the accessibility (transition) relation between these states is essentially baked in: roughly speaking, a term relation R' is (publicly) accessible from another term relation R if $R' \supseteq R$. Thus, environmental bisimulations provide less control over the structure of the transition system than RTSs do, and they do not support anything directly analogous to the distinction between public and private transitions.

As a consequence, environmental bisimulations are most effective at proving equivalences that require transition systems with only public transitions (*e.g.,* the twin abstraction example), and their proofs for examples where private transitions are required (*e.g.,* the well-bracketed state change example) are comparatively "brute-force". It is an open question whether environmental bisimulations can be generalized to support the full power of RTSs with both public and private transitions.

Our approach to reasoning about parametricity of ADTs, by populating the local knowledge of a world with relations at abstract *type names*, is inspired directly by Sumii and Pierce [36].

Large vs. Small Worlds While RTSs build very closely on the state transition systems in Dreyer *et al.*'s KLRs [5, 12], there is a big difference between them, which we like to think of in terms of *large* vs. *small* worlds.

Under Dreyer *et al.*'s approach, in order to demonstrate the equivalence of functions f_1 and f_2 under a "possible world" W, one proves that they behave the same when passed arguments that are related under any "future world" W' of W, which may contain arbitrary new invariants concerning the local state of other modules in the program. One can really think of the "future world" relation (*i.e.,* the Kripke structure) as defining its own transition system (or *large* world), with the possible worlds W as its states.

In contrast, our RTSs rely only on *small* worlds. For us, worlds W are static entities that contain only the local invariants relevant to the module we are reasoning about, and nothing about any invariants for other parts of the program. In proving equivalence of functions f_1 and f_2 under W, we never quantify over any future worlds that extend W. Of course, in order to support compositional reasoning—*i.e.,* in order to show that consistency of worlds is preserved under separating conjunction—we must show that f_1 and f_2 behave the same when applied to arguments drawn from some larger relation than just W's local knowledge; but for that purpose we quantify over the global knowledge G, which is not a world, but rather an arbitrary extension of W's local knowledge.

These different accounts of worlds are strongly reminiscent of the different techniques that have been proposed for modeling resource invariants in logics of storable locks. Gotsman *et al.* [15] and Hobor *et al.* [16] presented, roughly contemporaneously, two different models of a concurrent separation logic for local reasoning about programs that dynamically allocate locks and store them in the heap. The central challenge in developing such a model is in dealing with the semantic circularity that arises when accounting for locks whose resource invariants are essentially recursive.

Gotsman *et al.* deal with this circularity syntactically, by assuming a static set of named "sorts" of resource invariants, which includes not all possible invariants, but only those needed for reasoning about a particular program. In contrast, Hobor *et al.* (and more recently, Buisse *et al.* [10]) deal with the circularity head-on, defining once and for all what recursive resource invariants *mean* using step-indexing. The latter is analogous to Dreyer *et al.*'s "large worlds" approach, which defines the space of all possible heap invariants, while the former is analogous to our "small world" approach of defining only the heap invariants needed within the module we are reasoning about.

Which is better? It is hard to say. Our small-world relations seem easier to compose transitively, precisely because we make no assumption whatsoever about the relatedness of functions defined outside of whatever module we are reasoning about. That is, the global knowledge G that we quantify over (*e.g.,* when proving world consistency) could include complete garbage, and the transitivity proof sketched in Section 8 relies in a fundamental way on the surgical insertion of contentful garbage into the global knowledge. On the other hand, it is also possible that this approach is what leads our model not to validate η-equivalence.

The Trouble with η-Equivalence One limitation of RTSs is that they do not validate the η-equivalence rules for function and universal types. To see why, suppose the η rule for functions were true (a similar argument applies to universals):

$$f : \tau_1 \to \tau_2 \vdash f \sim \lambda x.\, f\ x : \tau_1 \to \tau_2$$

Then, by definition, there would exist a consistent world W s.t. for all $G \in \mathrm{GK}(W)$, and for all functions f_1 and f_2 related by G, we would have f_1 and $\lambda x.\, f_2\ x$ related by G as well. (We're glossing over the role of states and transitions here because they're orthogonal.) The problem is that it is easy to construct an "uncivilized" G that contains (f_1, f_2) but not $(f_1, \lambda x.\, f_2\ x)$. Ironically, the same uncivilized G's that make our proof of transitivity (Section 8) possible also cause η to fail.

As a result, there are certain examples that have appeared in the literature on relational reasoning for ML-like languages [36, 22, 11], which our method cannot handle, precisely because they depend fundamentally on η-equivalence. The best-known one is the *syntactic minimal invariance* example [30], which demonstrates that the "infinite η-expansion" at a general recursive type (*e.g.,* $\mu\alpha.\mathsf{unit} + (\alpha \to \alpha)$) is equivalent to the identity function.

The lack of η in our model makes a lot of sense because our proofs make no assumptions about whether unknown functions are even λ-expressions, let alone whether they obey η. In this respect, RTSs are again similar to normal form bisimulations [21, 34], which are sometimes easier to prove congruent in their non-η-supporting formulations. There are known ways to close normal form bisimulations over η-equivalence by complicating the definition of consistency, and it is possible that we could adapt such techniques to work for RTSs.

However, our concern, based at this point solely on intuition, is that there may be a fundamental tradeoff between supporting η and supporting transitive composition of RTS equivalence proofs. In that case, we would consider transitivity a more important desideratum, at least in the context of reasoning about multi-phase compiler correctness. Moreover, our lack of support for η may in fact render our method applicable to reasoning about higher-order languages with more restricted equational theories (*e.g.,* OCaml with its equality tests on function pointers). We leave this matter to be explored further in future work.

10. Conclusion and Future Work

We have developed a novel method—*relation transition systems*—for proving equivalence of ML-like programs, combining some of the best aspects of Kripke logical relations, environmental bisimulations, and normal form bisimulations. In addition to providing a useful synthesis of the complementary advantages of its ancestors, our method shows promise as a way of reasoning about equivalences between different languages, thanks to (1) our avoidance of "syntactic" devices that would preclude inter-language reasoning, and (2) the transitive composability of RTS equivalence proofs. We have briefly sketched the proof of transitivity here in a simplified setting, and we intend to report on the full transitivity result in a forthcoming paper. We have mechanized the metatheory of RTSs in Coq and made the proofs available at the link below.

There are several exciting directions for future work. First and foremost, we aim to concretely demonstrate the suitability of RTSs for inter-language reasoning. For example, we would like to take Hur and Dreyer's recent work on Kripke logical relations between ML and assembly [17], replace the logical relations with RTSs, and then apply the technique to reasoning about compositional correctness of multi-phase compilation.

Second, following Dreyer *et al.*'s work on Kripke logical relations [12], we would like to explore how well our account of RTSs can be adapted to handle the introduction of control effects (call/cc, exceptions) into the language and/or the restriction of the language to first-order state. In principle, we believe it should be possible to employ techniques similar to theirs, but we have not yet tried.

Lastly, we have recently discovered what appears to be a deep connection between our technique of global vs. local knowledge and *Mendler-style recursion* [24, 38], in particular the notion of a *robustly postfixed-point (rpofp)*. L is defined to be a rpofp of an endofunction F if $\forall G \geq L.\ L \leq F(G)$. This bears a striking resemblance to our definition of consistency for local knowledges L, at least in the pure setting of Section 4. One interesting feature of rpofps is that they enjoy a "robust" version of Tarski's fixed-point theorem, which applies even when the endofunction F is not monotone. Indeed, in our scenario, F is *not* monotone, due to the quantification over related function arguments, and this is precisely what motivated our parameterization over G. We intend to explore this connection further in future work.

Online Appendix

Details of the RTS metatheory (in PDF and Coq) are available at:

`http://www.mpi-sws.org/~dreyer/papers/marriage`

Acknowledgments

We thank Deepak Garg, Eijiro Sumii, and the anonymous reviewers for very helpful feedback on the paper, as well as Andrew Pitts for alerting us to the connection with Mendler-style recursion.

References

[1] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. 1990.

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

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

[4] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *ICFP*, 2011.

[5] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

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

[7] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[8] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. In *LICS*, 2011.

[9] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *LMCS*, 2(5:1), 2006.

[10] A. Buisse, L. Birkedal, and K. Støvring. A step-indexed Kripke model of separation logic for storable locks. In *MFPS*, 2011.

[11] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *LMCS*, 7(2:16):1–37, June 2011.

[12] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, 2010.

[13] D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, 2010.

[14] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[15] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning about storable locks and threads. In *APLAS*, 2007.

[16] A. Hobor, A. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[17] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *POPL*, 2011.

[18] V. Koutavas, P. B. Levy, and E. Sumii. From applicative to environmental bisimulation. In *MFPS*, 2011.

[19] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, 2006.

[20] J. Laird. A fully abstract trace semantics for general references. In *ICALP*, 2007.

[21] S. Lassen. Eager normal form bisimulation. In *LICS*, 2005.

[22] S. B. Lassen and P. B. Levy. Typed normal form bisimulation. In *CSL*, 2007.

[23] S. B. Lassen and P. B. Levy. Typed normal form bisimulation for parametric polymorphism. In *LICS*, 2008.

[24] N. P. Mendler. Inductive types and type constraints in the second-order lambda-calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[25] A. S. Murawski and N. Tzevelekos. Game semantics for good general references. In *LICS*, 2011.

[26] G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. *JFP*, 21(4&5):497–562, 2011.

[27] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–586, 2000.

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

[29] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[30] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

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

[32] D. Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994.

[33] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *LICS*, 2007.

[34] K. Støvring and S. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *POPL*, 2007.

[35] E. Sumii. A complete characterization of observational equivalence in polymorphic λ-calculus with general references. In *CSL*, 2009.

[36] E. Sumii and B. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54(5):1–43, 2007.

[37] J. Thamsborg and L. Birkedal. A Kripke logical relation for effect-based program transformations. In *ICFP*, 2011.

[38] T. Uustalu and V. Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, 1999.

[39] V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, 2011.

[40] P. Wadler. Theorems for free! In *FPCA*, 1989.