Operational Semantics for Multi-Language Programs

Jacob Matthews Robert Bruce Findler
University of Chicago
{jacobm, robby}@cs.uchicago.edu

Abstract

Inter-language interoperability is big business, as the success of Microsoft's .NET and COM and Sun's JVM show. Programming language designers are designing programming languages that reflect that fact — SML#, Mondrian, and Scala, to name just a few examples, all treat interoperability with other languages as a central design feature. Still, current multi-language research tends not to focus on the semantics of interoperation features, but only on how to implement them efficiently. In this paper, we take first steps toward higher-level models of interoperating systems. Our technique abstracts away the low-level details of interoperability like garbage collection and representation coherence, and lets us focus on semantic properties like type-safety and observable equivalence.

Beyond giving simple expressive models that are natural compositions of single-language models, our studies have uncovered several interesting facts about interoperability. For example, higher-order contracts naturally emerge as the glue to ensure that interoperating languages respect each other's type systems. While we present our results in an abstract setting, they shed light on real multi-language systems and tools such as the JNI, SWIG, and Haskell's stable pointers.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages, theory

Keywords Interoperability, multi-language systems, operational semantics

1. Introduction

A modern large-scale software system is likely written in a variety of languages: its core might be written in Java, while it has specialized system interaction routines written in C and a web-based user interface written in PHP. And even academic languages have caught multi-language programming fever, perhaps due to temptingly large numbers of pre-existing libraries written in other languages. This has prompted language implementors to target COM [18, 41], Java Virtual Machine bytecode [7, 27, 34], and most recently Microsoft's Common Language Runtime [8, 32, 36]. Furthermore, where foreign function interfaces have historically been used in practice to allow high-level safe languages to call libraries written in low-level unsafe languages like C (as was the motivation for the popular wrapper generator SWIG [5]), these new foreign function interfaces are built to allow high-level, safe languages to interoperate with other high-level, safe languages, such as Python with Scheme [33] and Lua with OCaml [39].

Since these embeddings are driven by practical concerns, the research that accompanies them rightly focuses on the bits and bytes of interoperability — how to represent data in memory, how to call a foreign function efficiently, and so on. But an important theoretical problem arises, independent of these implementation-level concerns: how can we reason formally about multi-language programs? This is a particularly important question for systems that involve typed languages, because we have to show that the embeddings respect their constituents' type systems.

In this paper we present a simple method for giving operational semantics to multi-language systems. Our models are rich enough to support a wide variety of multi-language embedding strategies, and powerful enough that we have been able to use them for type soundness and contextual equivalence proofs. Our technique is based on simple constructs we call *boundaries*, cross-language casts that regulate both control flow and value conversion between languages. We introduce boundaries through series of operational semantics in which we combine a simple ML-like language with a simple Scheme-like language.

In section 2, we introduce those two constituent languages formally and connect them using a primitive embedding where values in one language are opaque to the other. In section 3, we enrich that embedding so that boundaries use type information to convert one language's values into their counterparts in the other, and we show that this embedding naturally leads to higher-order contracts. Section 4 shows a surprising relationship between the expressive power of these two embeddings, and section 5 shows how the system scales beyond purely type-directed conversion.

2. The lump embedding

To begin, we pick two languages, give them formal models, and then tie those formal models together. In the interest of focusing on interoperation rather than the special features of particular languages, we have chosen two simple calculi: an extended model of the untyped call-by-value lambda calculus, which we use as a stand-in for Scheme, and an extended model of the simply-typed lambda calculus, which we use as a stand-in for ML (though it more closely resembles Plotkin's PCF without fixpoint operators [37]). Figure 1 presents these languages in an abstract manner that we instantiate multiple ways to model different forms of interoperability. One goal of this section is to explain that figure's peculiarities, but for now notice that aside from unusual subscripts and font choices, the two language models look pretty much as they would in a normal Felleisen-and-Hieb-style presentation [15].

To make the preparation more concrete, as we explain our presentation of the core models we also simultaneously develop our first interoperation model, which we call the lump embedding. In the lump embedding, ML values can appear in Scheme and

POPL'07 January 17–19, 2007, Nice, France.

$$
\begin{array}{lcl}
\mathbf{e} & = & \mathbf{x} \mid \mathbf{v} \mid (\mathbf{e}\ \mathbf{e}) \mid (\mathit{op}\ \mathbf{e}\ \mathbf{e}) \mid (\texttt{if0}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e}) \\
\mathbf{v} & = & (\lambda\, \mathbf{x} : \tau\ .\ \mathbf{e}) \mid \overline{n} \\
\mathit{op} & = & + \mid - \\
\tau & = & \iota \mid \tau \to \tau \\
\mathbf{x} & = & \text{ML variables (distinct from Scheme variables)} \\
\mathbf{E} & = & [\,]_M \mid (\mathbf{E}\ \mathbf{e}) \mid (\mathbf{v}\ \mathbf{E}) \mid (\mathit{op}\ \mathbf{E}\ \mathbf{e}) \mid (\mathit{op}\ \mathbf{v}\ \mathbf{E}) \mid (\texttt{if0}\ \mathbf{E}\ \mathbf{e}\ \mathbf{e})
\end{array}
$$

$$
\frac{}{\Gamma, \mathbf{x} : \tau \vdash_M \mathbf{x} : \tau} \quad \frac{\Gamma, \mathbf{x} : \tau_1 \vdash_M \mathbf{e} : \tau_2}{\Gamma \vdash_M (\lambda\, \mathbf{x} : \tau_1\ .\ \mathbf{e}) : \tau_1 \to \tau_2} \quad \frac{}{\Gamma \vdash_M \overline{n} : \iota}
$$

$$
\frac{\Gamma \vdash_M \mathbf{e}_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_M \mathbf{e}_2 : \tau_1}{\Gamma \vdash_M (\mathbf{e}_1\ \mathbf{e}_2) : \tau_2} \quad \frac{\Gamma \vdash_M \mathbf{e}_1 : \iota \quad \Gamma \vdash_M \mathbf{e}_2 : \iota}{\Gamma \vdash_M (\mathit{op}\ \mathbf{e}_1\ \mathbf{e}_2) : \iota}
$$

$$
\frac{\Gamma \vdash_M \mathbf{e}_1 : \iota \quad \Gamma \vdash_M \mathbf{e}_2 : \tau \quad \Gamma \vdash_M \mathbf{e}_3 : \tau}{\Gamma \vdash_M (\texttt{if0}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3) : \tau}
$$

$$
\begin{array}{lcl}
\mathcal{E}[((\lambda\, \mathbf{x} : \tau\ .\ \mathbf{e})\ \mathbf{v})]_M & \to & \mathcal{E}[\mathbf{e}[\mathbf{x} \,/\, \mathbf{v}]] \\
\mathcal{E}[(+\ \overline{n_1}\ \overline{n_2})]_M & \to & \mathcal{E}[\overline{n_1 + n_2}] \\
\mathcal{E}[(-\ \overline{n_1}\ \overline{n_2})]_M & \to & \mathcal{E}[\overline{max(n_1 - n_2, 0)}] \\
\mathcal{E}[(\texttt{if0}\ \overline{0}\ \mathbf{e}_1\ \mathbf{e}_2)]_M & \to & \mathcal{E}[\mathbf{e}_1] \\
\mathcal{E}[(\texttt{if0}\ \overline{n}\ \mathbf{e}_1\ \mathbf{e}_2)]_M & \to & \mathcal{E}[\mathbf{e}_2] \text{ (where } n \neq 0)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{e} & = & \mathsf{x} \mid \mathsf{v} \mid (\mathsf{e}\ \mathsf{e}) \mid (\mathit{op}\ \mathsf{e}\ \mathsf{e}) \mid (\texttt{if0}\ \mathsf{e}\ \mathsf{e}\ \mathsf{e}) \mid (\mathit{pr}\ \mathsf{e}) \mid (\texttt{wrong}\ \mathit{str}) \\
\mathsf{v} & = & (\lambda\, \mathsf{x}\ .\ \mathsf{e}) \mid \overline{n} \\
\mathit{op} & = & + \mid - \\
\mathit{pr} & = & \texttt{proc?} \mid \texttt{nat?} \\
\mathsf{x} & = & \text{Scheme variables (distinct from ML variables)} \\
\mathsf{E} & = & [\,]_S \mid (\mathsf{E}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{E}) \mid (\mathit{op}\ \mathsf{E}\ \mathsf{e}) \mid (\mathit{op}\ \mathsf{v}\ \mathsf{E}) \mid (\texttt{if0}\ \mathsf{E}\ \mathsf{e}\ \mathsf{e}) \mid (\mathit{pr}\ \mathsf{E})
\end{array}
$$

$$
\frac{}{\Gamma, \mathsf{x} : \mathbf{TST} \vdash_S \mathsf{x} : \mathbf{TST}} \quad \frac{\Gamma, \mathsf{x} : \mathbf{TST} \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_S (\lambda\, \mathsf{x}\ .\ \mathsf{e}) : \mathbf{TST}} \quad \cdots
$$

$$
\begin{array}{lcl}
\mathcal{E}[((\lambda\, \mathsf{x}\ .\ \mathsf{e})\ \mathsf{v})]_S & \to & \mathcal{E}[\mathsf{e}[\mathsf{x} \,/\, \mathsf{v}]] \\
\mathcal{E}[\mathsf{v}_1\ \mathsf{v}_2]_S & \to & \mathcal{E}[\texttt{wrong}\ \text{“non-proc”}]\ (\mathsf{v}_1 \neq \lambda \mathsf{x}.\mathsf{e}) \\
\mathcal{E}[(+\ \overline{n_1}\ \overline{n_2})]_S & \to & \mathcal{E}[\overline{n_1 + n_2}] \\
\mathcal{E}[(-\ \overline{n_1}\ \overline{n_2})]_S & \to & \mathcal{E}[\overline{max(n_1 - n_2, 0)}] \\
\mathcal{E}[(\mathit{op}\ \mathsf{v}_1\ \mathsf{v}_2)]_S & \to & \mathcal{E}[\texttt{wrong}\ \text{“non-num”}]\ (\mathsf{v}_1 \neq \overline{n} \text{ or } \mathsf{v}_2 \neq \overline{n}) \\
\mathcal{E}[(\texttt{if0}\ \overline{0}\ \mathsf{e}_1\ \mathsf{e}_2)]_S & \to & \mathcal{E}[\mathsf{e}_1] \\
\mathcal{E}[(\texttt{if0}\ \mathsf{v}\ \mathsf{e}_1\ \mathsf{e}_2)]_S & \to & \mathcal{E}[\mathsf{e}_2]\ (\mathsf{v} \neq \overline{0}) \\
\mathcal{E}[(\texttt{proc?}\ (\lambda\, \mathsf{x}\ .\ \mathsf{e}))]_S & \to & \mathcal{E}[\overline{0}] \\
\mathcal{E}[(\texttt{proc?}\ \mathsf{v})]_S & \to & \mathcal{E}[\overline{1}]\ (\mathsf{v} \neq (\lambda \mathsf{x}.\mathsf{e})) \\
\mathcal{E}[(\texttt{nat?}\ \overline{n})]_S & \to & \mathcal{E}[\overline{0}] \\
\mathcal{E}[(\texttt{nat?}\ \mathsf{v})]_S & \to & \mathcal{E}[\overline{1}]\ (\mathsf{v} \neq \overline{n}) \\
\mathcal{E}[(\texttt{wrong}\ \mathit{str})]_S & \to & \textbf{Error}: \mathit{str}
\end{array}
$$

Figure 1. Models of ML (left) and Scheme (right), primed for interoperability

Scheme values can appear in ML. However, ML treats Scheme values as opaque lumps that cannot be used directly, only returned to Scheme; likewise ML values are opaque lumps to Scheme. For instance, we allow ML to pass a function to Scheme and then use it again as a function if Scheme returns it; but we do *not* allow Scheme to use that same value as a function directly or vice versa.

The lump embedding is a conveniently simple example, but it is worth attention for other reasons as well. First, it represents a particularly easy-to-implement useful multi-language system, achievable more or less automatically for any pair of programming languages so long as both languages have some notion of expressions that yield values. Second, it corresponds to real multi-language systems that can be found "in the wild": many foreign function interfaces give C programs access to native values only as pointers that C can only use by returning to the host language. For instance this is how stable pointers in the Haskell foreign function interface behave [12].

Where possible, we have typeset the nonterminals of our ML language using a **bold font with serifs**, and those of our Scheme language with a light sans-serif font. For instance, **e** means the ML expression nonterminal and e means the Scheme expression nonterminal. These distinctions are meaningful. Occasionally we use a subscript instead of a font distinction in cases where the font difference would be too subtle.

2.1 Syntax

The syntaxes of the two languages we use as our starting point are shown in figure 1. On the ML side, we have taken the explicitly-typed lambda calculus syntax and added numbers (where $\overline{n}$ indicates the syntactic term representing the number n) and a few built-in primitives including an `if0` form. On the Scheme side, we have taken an untyped lambda calculus syntax and added the same extensions plus some useful predicates and a `wrong` form that takes a literal error message string.

To extend that base syntax with the ability to interoperate, we introduce syntactic boundaries between ML and Scheme, a kind of cross-language cast that indicate a switch of languages. The extension is shown in figure 2.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathbf{e} & = & \cdots \mid ({}^{\tau}\mathit{MS}\ \mathsf{e}) & \mathsf{e} & = & \cdots \mid (\mathit{SM}^{\tau}\ \mathbf{e}) \\
\mathbf{v} & = & \cdots \mid ({}^{\mathbf{L}}\mathit{MS}\ \mathsf{v}) & \mathsf{v} & = & \cdots \mid (\mathit{SM}^{\tau}\ \mathbf{v}) \\
\tau & = & \cdots \mid \mathbf{L} & & & \text{where } \tau \neq \mathbf{L} \\
\mathbf{E} & = & \cdots \mid ({}^{\tau}\mathit{MS}\ \mathsf{E}) & \mathsf{E} & = & \cdots \mid (\mathit{SM}^{\tau}\ \mathbf{E}) \\
\multicolumn{6}{c}{\mathcal{E} = \mathbf{E}}
\end{array}
$$

$$
\frac{\Gamma \ \vdash_S \ \mathsf{e} : \mathbf{TST}}{\Gamma \ \vdash_M \ ({}^{\tau}\mathit{MS}\ \mathsf{e}) : \tau} \quad \frac{\Gamma \ \vdash_M \ \mathbf{e} : \tau}{\Gamma \ \vdash_S \ (\mathit{SM}^{\tau}\ \mathbf{e}) : \mathbf{TST}}
$$

$$
\begin{array}{lcl}
\mathcal{E}[({}^{\tau}\mathit{MS}(\mathit{SM}^{\tau}\mathbf{v}))]_M & \to & \mathcal{E}[\mathbf{v}] \\
\mathcal{E}[{}^{\tau}\mathit{MS}\ \mathsf{v}]_M & \to & \mathcal{E}[({}^{\tau}\mathit{MS}\ (\texttt{wrong}\ \text{“Bad value”}))] \\
 & & \text{if } \mathsf{v} \neq (\mathit{SM}^{\tau}\mathbf{v}) \text{ and } \tau \neq \mathbf{L} \\
\mathcal{E}[(\mathit{SM}^{\mathbf{L}}\ ({}^{\mathbf{L}}\mathit{MS}\ \mathsf{v}))]_S & \to & \mathcal{E}[\mathsf{v}]
\end{array}
$$

Figure 2. Extensions to figure 1 to form the lump embedding

Concretely we add boundaries as a new kind of expression in each language. In ML, we extend **e** to also produce $({}^{\tau}\mathit{MS}\ \mathsf{e})$ (think of MS as "ML-outside, Scheme-inside") and we extend Scheme's e to also produce $(\mathit{SM}^{\tau}\ \mathbf{e})$ ("Scheme outside, ML inside") where the τ on the ML side of each boundary indicates the type ML will consider the expression on its side of the boundary to be.

2.2 Typing rules

In figure 1, ML has a standard type system with the typing judgment $\vdash_M$ where numbers have type ι and functions have arrow types. Scheme has a type system with the judgment $\vdash_S$ that gives all closed terms the type **TST** ("the Scheme type"). We have omitted several typing rules from the Scheme side; every Scheme expression has a rule that gives it type **TST** if its subparts have type **TST**.

In our lump embedding extension, we add a new type **L** (for "lump") to ML and we add two new typing rules, one for each new syntactic form. The new Scheme judgment says that an $(\mathit{SM}^{\tau}\ \mathbf{e})$ boundary is well-typed if ML's type system proves **e** has type τ — that is, a Scheme program type-checks if it is closed and all its ML subterms have the types the program claims they have. The new ML judgment says that $({}^{\tau}\mathit{MS}\ \mathsf{e})$ has type τ if e type-checks under

Scheme's typing system. In both cases, τ can be any type, not just $\mathbf{L}$ as one might expect. If $\tau = \mathbf{L}$ we are sending a native Scheme value across the boundary (which will be a lump in ML); if $\tau \neq \mathbf{L}$ we are sending an ML value across the boundary (which will be a lump in Scheme).

In these typing judgments, Scheme rules use the same type environment that ML does. We do that to allow ML expressions to refer to variables bound by ML (or vice versa) even if they are separated by a sequence of boundaries. This is necessary to give types to functions that use foreign arguments, as we shall see.

2.3 Operational semantics

We use Felleisen and Hieb-style reduction semantics to specify operational semantics for our systems. In figure 1, we define an evaluation context for ML ($\mathbf{E}$) and one for Scheme (E), and we use a third, unspecified evaluation context ($\mathcal{E}$) to represent the top-level context. In examples in this paper, we will assume $\mathcal{E} = \mathbf{E}$ and thus that the top-level program is written in ML; we could set $\mathcal{E} = \mathsf{E}$ to reverse that assumption. To allow Scheme expressions to evaluate inside ML expressions and vice versa, we define evaluation contexts mutually recursively at boundaries: $\mathbf{E}$ produces $({}^{\tau}MS\ \mathsf{E})$ and E produces $(SM^{\tau}\ \mathbf{E})$.

The reduction rules in the core model are all reasonably standard, with a few peculiarities. On the ML side, we allow subtraction of two numbers but floor all results at zero. The Scheme side has a bit more going on dynamically. Since Scheme's type system does not protect it, we add dynamic checks to every appropriate form that reduce to `wrong` if they receive an illegal value. The reduction rule for `wrong` itself discards the entire program context and aborts the program with an error message.

To combine the languages, we might hope to just merge their sets of reductions together. That does not quite work. For instance, the ML term $((\lambda \mathbf{x} : \iota.\mathbf{x})\ (\overline{1}\ \overline{1}))$ would reduce to (`wrong` "Non-procedure"), rather than getting stuck, since Scheme's reduction for a non-procedure application would apply to $(\overline{1}\ \overline{1})$. To remedy this, we extend Felleisen and Hieb's context-sensitive rewriting framework by differentiating $[\,]_M$ holes generated by ML evaluation contexts from $[\,]_S$ holes generated by Scheme evaluation contexts. Scheme's rewriting rules only apply to evaluation contexts with $[\,]_S$ holes, (and similarly for ML); we indicate that restriction by writing an S (or M) subscript next to the bracket on each evaluation context on the left-hand side of Scheme's (or ML's) rewriting rules.[1]

With this extension, the example above decomposes into the context $((\lambda \mathbf{x} : \iota.\mathbf{x})[\,]_M)$ but not $((\lambda \mathbf{x} : \iota.\mathbf{x})[\,]_S)$. Because the hole is named $[\,]_M$ and the Scheme application rule only applies to holes named $[\,]_S$, the term remains stuck. However, if instead we had written $((\lambda \mathbf{x} : \iota.\mathbf{x})\ ({}^{\iota}MS\ (\overline{1}\ \overline{1})))$ it would decompose into an ML context with a Scheme hole: $((\lambda \mathbf{x} : \iota.\mathbf{x})\ ({}^{\iota}MS\ [\,]_S))$ with the erroneous application $(\overline{1}\ \overline{1})$ inside. Since that redex is in a Scheme hole it would reduce to an error.

To finish the lump embedding, all that remains is to specify the reduction rules and values for the boundaries between languages. If an MS boundary of type $\mathbf{L}$ has a Scheme value inside, then the boundary is an ML value. Similarly, when an SM boundary of a non-lump type has an ML value inside, then it is a Scheme value. However, MS boundaries with a non-lump type that contain Scheme values and SM boundaries of type $\mathbf{L}$ that contain ML values should reduce, since they represent foreign values returning to a native context. We do that by cancelling matching boundaries, as the reduction rules in figure 2 show.

ML's typing rules guarantee that values that appear inside $(SM^{\mathbf{L}}\ \mathbf{v})$ expressions will in fact be lump values, so the $SM^{\mathbf{L}}$ reduction can safely restrict its attention to values of the correct form. Scheme offers no such guarantee, so the rule for eliminating an ${}^{\tau}MS$ boundary must apply whenever the Scheme expression is a value at all.

These additions give us a precise account of the behavior for lump embedding we described at the beginning of this section. To get a sense of how the calculus works, consider this example:

$$\begin{array}{rl} & ((\lambda\ \mathbf{fa} : \mathbf{L} \to \mathbf{L} \to \mathbf{L}.\ ((\mathbf{fa}\ ({}^{\mathbf{L}}MS\ (\lambda\, \times\, .\ (+\ \times\ \overline{1}))))\\ & \qquad\qquad ({}^{\mathbf{L}}MS\ \overline{3})))\\ & \quad (\lambda\ \mathbf{f} : \mathbf{L}.\ \lambda\ \mathbf{x} : \mathbf{L}.\ ({}^{\mathbf{L}}MS\ ((SM^{\mathbf{L}}\ \mathbf{f})\ (SM^{\mathbf{L}}\ \mathbf{x})))))\\ \to^2 & ({}^{\mathbf{L}}MS\ ((SM^{\mathbf{L}}({}^{\mathbf{L}}MS(\lambda\times.(+\ \times\ \overline{1})))(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \overline{3})))))\\ \to^2 & ({}^{\mathbf{L}}MS\ ((\lambda\times.(+\ \times\ \overline{1}))\ \overline{3})) \to^2 ({}^{\mathbf{L}}MS\ \overline{4}) \end{array}$$

In the initial term of this reduction sequence, we use a "left-left-lambda" encoding of **let** to bind the name **fa** (for "foreign-apply") to a curried ML function that takes two foreign values, applies the first to the second, and returns the result as another foreign value. The program uses **fa** to apply a Scheme add-one function to the Scheme number $\overline{3}$. In two computation steps, we plug in the Scheme function and its argument into the body of **fa**. In that term there are two instances of $(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \mathsf{v}))$ subterms, both of which are cancelled in the next two computation steps. After those cancellations, the term is just a Scheme application of the add-one function to $\overline{3}$, which reduces to the Scheme value $\overline{4}$.

If we try to apply the ML add-one function to the Scheme number $\overline{3}$ instead (and adjust **fa**'s type to make that possible), we will end up with an intermediate term like this:

$$({}^{\iota}MS\ ((SM^{\iota\to\iota}(\lambda\mathbf{x} : \iota.(+\ \mathbf{x}\ \overline{1})))\ \overline{3})) \to^2 \textbf{Error}\text{: non-procedure}$$

Here, Scheme tries to apply the ML function directly, which leads to a runtime error since it is illegal for Scheme to apply ML functions. We cannot make the analogous mistake and try to apply a Scheme function in ML, since terms like $(({}^{\mathbf{L}}MS\ (\lambda\times.(+\ \times\ \overline{1})))\ \overline{3})$ are ill-typed.

The formulation of the lump embedding in figure 2 allows us to prove type soundness in the standard way, by establishing preservation and progress lemmas. Notice that because of the way we have combined the two languages, type soundness entails that both languages are type-sound with respect to *their own* type systems — in other words, that both single-language type soundness proofs are special cases of the soundness theorem for the entire system.

Theorem 1 (Lump type soundness)**.** *If* $\Gamma \vdash_M \boldsymbol{e} : \tau$*, then either* $\boldsymbol{e} \to^* \boldsymbol{v}$*,* $\boldsymbol{e} \to^*$ **Error***: str, or* $\boldsymbol{e}$ *diverges.*

Proof. (Sketch) A mutually-recursive variation on the standard preservation and progress lemmas. □

Proofs of this and the other theorems in this paper are available in this paper's companion technical report, University of Chicago CS TR-2006-10.

3. The natural embedding

The lump embedding is a useful starting point, but many multi-language systems offer richer cross-language communication primitives. A more natural way to pass values between our Scheme and ML models, suggested many times in the literature (*e.g.*, [6,35,39]) is to use a type-directed strategy to convert ML numbers to equivalent Scheme numbers and ML functions to equivalent Scheme functions (for some suitable notion of equivalence) and vice versa. We call this the natural embedding.

[1] We could also have introduced two different notations for Scheme and ML application, but we find it inelegant; it suggests that a multi-language implementation would decide how to evaluate each term by inspecting it, when real systems decide how to evaluate a term based on the language in which the term is being evaluated — *i.e.*, its context. Also, this is the same extension we made in earlier work to model Scheme's multiple values [30].

We can quickly get at the essence of this strategy by extending the core calculi from figure 1, just as we did before to form the lump embedding. Again, we add new syntax and reduction rules to figure 1. In this section we will add ${}^{\tau}MS_N$ and SM_N^{τ} boundaries, adding the subscript N (for "natural") to distinguish these new boundaries from lump boundaries from section 2.

We will assume we can translate numbers from one language to the other, and give reduction rules for boundary-crossing numbers based on that assumption:

$$\begin{array}{rcl} \mathcal{E}[(SM_N^{\iota}\,\overline{n})]_S & \rightarrow & \mathcal{E}[\overline{n}] \\ \mathcal{E}[({}^{\iota}MS_N\,\overline{n})]_M & \rightarrow & \mathcal{E}[\overline{n}] \end{array}$$

In some multi-language settings, differing byte representations might complicate this task. Worse, some languages may have more expansive notions of numbers than others — for instance, the actual Scheme language treats many different kinds of numbers uniformly (*e.g.*, integers, floating-point numbers, arbitrary precision rationals, and complex numbers are all operated on by the same operators), whereas the actual ML language imposes much more structure on its number representations. More sophisticated versions of the above rules would address these problems straightforwardly.

We must be more careful with procedures, though. We cannot get away with just moving the text of a Scheme procedure into ML or vice versa; aside from the obvious problem that their grammars generate different sets of terms, ML does not even have a reasonable equivalent for every Scheme procedure. Instead, for this embedding we represent a foreign procedure with a proxy. We represent a Scheme procedure in ML at type $\tau_1 \rightarrow \tau_2$ by a new procedure that takes an argument of type τ_1, converts it to a Scheme equivalent, runs the original Scheme procedure on that value, and then converts the result back to ML at type τ_2. For example, the scheme function $\mathbf{f}$ becomes $(\lambda \mathbf{x} : \tau_1.({}^{\tau_2}MS_N\,(\mathbf{f}\,(SM_N^{\tau_1}\,\mathbf{x}))))$ when embedded in ML. Since function arguments flow in the opposite direction from function results, the boundary that converts the argument to the Scheme function must be an $SM_N^{\tau_1}$ boundary, not an ${}^{\tau_1}MS_N$ boundary.

This would complete the natural embedding, but for one important problem: the system has stuck states, since a boundary might receive a value of an inappropriate shape. Stuck states violate type-soundness, and in an implementation they might correspond to segmentation faults or other undesirable behavior. As it turns out, higher-order contracts [16, 17] arise naturally as the checks required to protect against these stuck states. We show that in the next three sections: first we add dynamic guards directly to boundaries to provide a baseline, then show how to separate them, and finally observe that these separated guards are precisely contracts between ML and Scheme, and that since ML statically guarantees that it always lives up to its contracts, we can eliminate their associated dynamic checks.

3.1 A simple method for adding error transitions

In the lump embedding, we can always make a single, immediate check that would tell us if the value Scheme provided to ML was consistent with the type ML ascribed to it. This is no longer possible, since we cannot know if a Scheme function always produces a value that can be converted to the appropriate type. Still, we can perform an optimistic check that preserves ML's type safety: when a Scheme value crosses a boundary, we only check its first-order qualities — *i.e.*, whether it is a number or a procedure. If it has the appropriate first-order behavior, we assume the type ascription is correct and perform the conversion, distributing into domain and range conversions as before. If it does not, we immediately signal an error. This method works to catch all errors that would lead to stuck states. Although it only checks first-order properties, the program can only reach a stuck state if a value is used in such a way that it does not have the appropriate first-order properties anyway.

To model this method, rather than adding the SM_N^{τ} and ${}^{\tau}MS_N$ constructs to our core languages from figure 1, we instead add "guarded" versions GSM^{τ} and MSG^{τ} shown in figure 3. These rules translate values in the same way that SM_N^{τ} and ${}^{\tau}MS_N$ did before, but also detect concrete, first-order witnesses to an invalid type ascription (*i.e.*, numbers for procedures or procedures for numbers) and abort the program if one is found. We call the language formed by these rules the simple natural embedding. We give its rules in figure 3, but it may be easier to understand how it works by reconsidering the examples we gave at the end of section 2. Each of those examples, modified to use the natural embedding rather than the lump embedding, successfully evaluates to the ML number $\overline{4}$. Here is the reduction sequence produced by the last of those examples, which was ill-typed before:

$$\begin{array}{rl} & ((MSG^{\iota\rightarrow\iota}\,(\lambda \mathsf{x}.(+\,\mathsf{x}\,1)))\,\overline{3}) \\ \rightarrow & ((\lambda \mathbf{x}' : \iota.(MSG^{\iota}\,((\lambda \mathsf{x}.(+\,\mathsf{x}\,1))\,(GSM^{\iota}\,\mathbf{x}'))))\,\overline{3}) \\ \rightarrow & (MSG^{\iota}\,((\lambda \mathsf{x}.(+\,\mathsf{x}\,1))\,(GSM^{\iota}\,\overline{3}))) \\ \rightarrow & (MSG^{\iota}\,((\lambda \mathsf{x}.(+\,\mathsf{x}\,1))\,\overline{3})) \\ \rightarrow^2 & (MSG^{\iota}\,\overline{4}) \rightarrow \overline{4} \end{array}$$

ML converts the Scheme add-one function to an ML function with type $\iota \rightarrow \iota$ by replacing it with a function that converts its argument to a Scheme number, feeds that number to the original Scheme function, and then converts the result back to an ML number. Then it applies this new function to the ML number $\overline{3}$, which gets converted to the Scheme number $\overline{3}$, run through the Scheme function, and finally converted back to the ML number $\overline{4}$, which is the program's final answer.

The method works at higher-order types because it applies type conversions recursively. Consider this expression:

$$({}^{(\iota\rightarrow\iota)\rightarrow\iota}MS_N(\lambda \mathsf{f}.\,(\mathtt{if0}\,(\mathbf{f}\,\overline{1})\,\overline{2}\,\mathsf{f}))))$$

Depending on the behavior of its arguments, the Scheme procedure may or may not always produce numbers. ML treats it as though it definitely had type $(\iota \rightarrow \iota) \rightarrow \iota$, and wraps it to the ML value

$$\lambda \mathbf{x} : \iota \rightarrow \iota.({}^{\iota}MS_N((\lambda \mathsf{f}.\mathtt{if0}\,(\mathsf{f}\,\overline{1})\,\overline{2}\,\mathsf{f})(SM_N^{\iota\rightarrow}\mathbf{x})))$$

Whenever this value is applied to a function, that function is converted to a Scheme value at type $\iota \rightarrow \iota$ and the result is converted from Scheme to ML at type ι. Thus, intuitively, conversion in either direction works at a given type if it works in *both* directions at all smaller types.

Theorem 2. *If* $\vdash_M \boldsymbol{e} : \tau$*, then either* $\boldsymbol{e} \rightarrow^* \boldsymbol{v}$*,* $\boldsymbol{e} \rightarrow^*$ **Error***: str, or* $\boldsymbol{e}$ *diverges.*

3.2 A refinement: guards

Adding dynamic checks to boundaries themselves is an expedient way to ensure type soundness, but we find it a little troublesome. For one thing, boundaries are necessarily the core of any multi-language system, so they should be as small and simple as possible. For another, coupling the task of value conversion with the conceptually unrelated task of detecting and signalling errors means that changing the method of signalling errors requires modifying the internals of value conversion.

To decouple error-handling from value conversion, we separate the guarded boundaries of the previous subsection into their constituent parts: boundaries and guards. These separated boundaries have the semantics of the ${}^{\tau}MS_N$ and SM_N^{τ} boundaries we introduced at the beginning of this section. Guards will be new expressions of the form $(\mathcal{G}^{\tau}\,\mathtt{e})$ that perform all dynamic checks necessary to ensure that their arguments behave as τ in the sense of the previous subsection. In all initial terms, we will wrap every boundary

$$\begin{array}{rcl|l} \mathbf{e} & = & \cdots \mid (MSG^{\tau}\ \mathsf{e}) & \dfrac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_M (MSG^{\tau}\ \mathsf{e}) : \tau} \\ \mathsf{e} & = & \cdots \mid (GSM^{\tau}\ \mathbf{e}) & \\ & & & \\ \mathbf{E} & = & \cdots \mid (MSG^{\tau}\ \mathsf{E}) & \dfrac{\Gamma \vdash_M \mathbf{e} : \tau}{\Gamma \vdash_S (GSM^{\tau}\ \mathbf{e}) : \mathbf{TST}} \\ \mathsf{E} & = & \cdots \mid (GSM^{\tau}\ \mathbf{E}) & \end{array}$$

$$\begin{array}{rl} \mathcal{E}[MSG^{\iota}\ \overline{n}]_M & \to \mathcal{E}[\overline{n}] \\ \mathcal{E}[MSG^{\iota}\ \mathsf{v}]_M & \to \mathcal{E}[MSG^{\iota}\ (\texttt{wrong}\ \text{“Non-number”})] \quad \mathsf{v} \neq \overline{n} \\ \mathcal{E}[MSG^{\tau_1 \to \tau_2}\ \lambda \mathsf{x}.\mathsf{e}]_M & \to \mathcal{E}[\lambda \mathbf{x} : \tau_1.MSG^{\tau_2}\ ((\lambda \mathsf{x}.\mathsf{e})\ (GSM^{\tau_1}\ \mathbf{x}))] \quad \mathbf{x} \text{ not free in } \mathsf{e} \\ \mathcal{E}[MSG^{\tau_1 \to \tau_2}\ \mathsf{v}]_M & \to \mathcal{E}[MSG^{\tau_1 \to \tau_2}\ \texttt{wrong}\ \text{“Non-procedure”}] \quad \mathsf{v} \neq \lambda \mathsf{x}.\mathsf{e} \\ \mathcal{E}[(GSM^{\iota}\ \overline{n})]_S & \to \mathcal{E}[\overline{n}] \\ \mathcal{E}[(GSM^{\tau_1 \to \tau_2}\ \mathbf{v})]_S & \to \mathcal{E}[(\lambda \mathsf{x}.\ (GSM^{\tau_2}(\mathbf{v}\ (MSG^{\tau_1}\ \mathsf{x}))))] \end{array}$$

Figure 3. Extensions to figure 1 to form the simple natural embedding

$$\begin{array}{rcl} \mathbf{e} & = & \cdots \mid ({}^{\tau}MS_N\ \mathsf{e}) \\ \mathsf{e} & = & \cdots \mid (\mathcal{G}^{\tau}\ \mathsf{e}) \mid (SM_N^{\tau}\ \mathbf{e}) \\ & & \\ \mathbf{E} & = & \cdots \mid ({}^{\tau}MS_N\ \mathsf{E}) \\ \mathsf{E} & = & \cdots \mid (\mathcal{G}^{\tau}\ \mathsf{E}) \mid (SM_N^{\tau}\ \mathbf{E}) \end{array}$$

$$\frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_S (\mathcal{G}^{\tau}\ \mathsf{e}) : \mathbf{TST}} \quad \frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_M ({}^{\tau}MS_N\ \mathsf{e}) : \tau} \quad \frac{\Gamma \vdash_M \mathbf{e} : \tau}{\Gamma \vdash_S (SM_N^{\tau}\ \mathbf{e}) : \tau}$$

$$\begin{array}{rcl} \mathcal{E}[{}^{\iota}MS_N\ \overline{n}]_M & \to & \mathcal{E}[\overline{n}] \\ \mathcal{E}[SM_N^{\iota}\ \overline{n}]_S & \to & \mathcal{E}[\overline{n}] \\ \mathcal{E}[{}^{\tau_1 \to \tau_2}MS_N\ \lambda \mathsf{x}.\mathsf{e}]_M & \to & \mathcal{E}[\lambda \mathbf{x} : \tau_1.SM_N^{\tau_2}((\lambda \mathsf{x}.\mathsf{e})\ (SM_N^{\tau_1}\ \mathbf{x}))] \\ \mathcal{E}[(SM_N^{\tau_1 \to \tau_2}\ \mathbf{v})]_S & \to & \mathcal{E}[(\lambda \mathsf{x}.(SM_N^{\tau_2}\ (\mathbf{v}\ ({}^{\tau_1}MS_N\ \mathsf{x}))))] \\ \mathcal{E}[(\mathcal{G}^{\iota}\ \overline{n})]_S & \to & \mathcal{E}[\overline{n}] \\ \mathcal{E}[(\mathcal{G}^{\iota}\mathsf{v})]_S & \to & \mathcal{E}[\texttt{wrong}\ \text{“Non-number”}]\ (\mathsf{v} \neq \overline{n}) \\ \mathcal{E}[(\mathcal{G}^{\tau_1 \to \tau_2}(\lambda \mathsf{x}.\mathsf{e}))]_S & \to & \mathcal{E}[(\lambda \mathsf{x}'.(\mathcal{G}^{\tau_2}((\lambda \mathsf{x}.\mathsf{e})(\mathcal{G}^{\tau_1}\mathsf{x}'))))] \\ \mathcal{E}[(\mathcal{G}^{\tau_1 \to \tau_2}\mathsf{v})]_S & \to & \mathcal{E}[\texttt{wrong}\ \text{“Non-procedure”}]\ (\mathsf{v} \neq \lambda \mathsf{x}.\mathsf{e}) \end{array}$$

Figure 4. Extensions to figure 1 to form the separated-guards natural embedding

with an appropriate guard: $({}^{\tau}MS_N\ (\mathcal{G}^{\tau}\ \mathsf{e}))$ instead of $(MSG^{\tau}\ \mathsf{e})$ and $(\mathcal{G}^{\tau}\ (SM^{\tau}\ \mathbf{e}))$ instead of $(GSM^{\tau}\ \mathbf{e})$.

Figure 4 shows the rules for guards. An ι guard applied to a number reduces to that number, and the same guard applied to a procedure aborts the program. A $\tau_1 \to \tau_2$ guard aborts the program if given a number, and if given a procedure reduces to a new procedure that applies a τ_1 guard to its input, runs the original procedure on that value, and then applies a τ_2 guard to the original procedure's result. This is just like the strategy we use to convert functions in the first place, but it doesn't perform any foreign-language translation by itself; it just distributes the guards in preparation for conversion later on.

The guard distribution rules for functions can move a guard arbitrarily far away from the boundary it protects; if this motion ever gave a value the opportunity to get to a boundary without first being blessed by the appropriate guard, the system would get stuck. We can prove this never happens by defining a combined language that has *both* guarded boundaries GSM^{τ} and MSG^{τ} and unguarded boundaries with separated guards ${}^{\tau}MS_N$, SM_N^{τ}, and $\mathcal{G}^{\tau}$; *i.e.* the language formed by combining figure 1 with figures 3 and 4. In this combined language, an argument by induction shows that guarded boundaries are observably equivalent to guards combined with unguarded boundaries.

Theorem 3. *For all Scheme expressions* e *and ML expressions* $\mathbf{e}$*, the following propositions hold:*

$$\begin{array}{rrcl} (1) & (MSG^{\tau}\mathbf{e}) & \simeq & ({}^{\tau}MS_N(\mathcal{G}^{\tau}\mathsf{e})) \\ (2) & (GSM^{\tau}\mathbf{e}) & \simeq & (\mathcal{G}^{\tau}(SM_N^{\tau}\mathbf{e})) \end{array}$$

where $\simeq$ *is observable equivalence in the combined language.*

In other words, we can replace any or all MSG^{τ} and GSM^{τ} boundaries with their separated versions without affecting the program, and therefore a program with *only* MSG^{τ} and GSM^{τ} boundaries is equivalent to the same program with *only* separated guards and unchecked boundaries; and therefore the language of figure 3 is the same as the language of figure 4.

3.3 A further refinement: contracts

While the guard strategy of the last subsection works, an implementation based on it would perform many unnecessary dynamic checks. For instance, the term $({}^{\iota \to \iota}MS_N\ (\mathcal{G}^{\iota \to \iota}\ (\lambda \mathsf{x}.(\mathsf{x}))))$ is equivalent to

$$(\lambda \mathbf{x} : \iota.({}^{\iota}MS_N(\mathcal{G}^{\iota}\ ((\lambda \mathsf{x}.\mathsf{x})\ (\mathcal{G}^{\iota}\ (SM_N^{\iota}\ \mathbf{x}))))))$$

The check performed by the leftmost guard is necessary, but the check performed by the rightmost guard could be omitted — since the value is coming directly from ML, ML's type system guarantees that the conversion will succeed.

We can refine our guarding strategy to eliminate those unnecessary checks. We split guards into two varieties: positive guards, written $\mathcal{G}_+^{\tau}$, that apply to values going from Scheme to ML, and negative guards, written $\mathcal{G}_-^{\tau}$, that apply to values going from ML to Scheme. Their reduction rules are:

$$\begin{array}{rcll} \mathcal{E}[(\mathcal{G}_+^{\iota}\ \overline{n})]_S & \to & \mathcal{E}[\overline{n}] & \\ \mathcal{E}[(\mathcal{G}_+^{\iota}\mathsf{v})]_S & \to & \mathcal{E}[(\texttt{wrong}\ \text{“Non-number”})] & (\mathsf{v} \neq \overline{n}) \\ \mathcal{E}[(\mathcal{G}_+^{\tau_1 \to \tau_2}\mathsf{v})]_S & \to & \mathcal{E}[(\mathcal{G}_+^{\tau_2}\ (\mathsf{v}\ (\mathcal{G}_-^{\tau_1}\ \mathsf{x})))] & (\mathsf{v} = \lambda \mathsf{x}.\mathsf{e}) \\ \mathcal{E}[(\mathcal{G}_+^{\tau_1 \to \tau_2}\mathsf{v})]_S & \to & \mathcal{E}[(\texttt{wrong}\ \text{“Non-function”})] & (\mathsf{v} \neq \lambda \mathsf{x}.\mathsf{e}) \\ \mathcal{E}[(\mathcal{G}_-^{\iota}\mathsf{v})]_S & \to & \mathcal{E}[\mathsf{v}] & \\ \mathcal{E}[(\mathcal{G}_-^{\tau_1 \to \tau_2}\mathsf{v})]_S & \to & \mathcal{E}[(\lambda \mathsf{x}.(\mathcal{G}_-^{\tau_2}(\mathsf{v}(\mathcal{G}_+^{\tau_1}\mathsf{x}))))] & \end{array}$$

The function cases are the interesting rules here. Since functions that result from positive guards are bound for ML, we check the inputs that ML will supply them using a negative guard; since the result of those functions will be Scheme values going to ML, they must be guarded with positive guards. Negative guards never directly signal an error; they exist only to protect ML functions from erroneous Scheme inputs. They put positive guards on the arguments to ML functions but use negative guards on their results because those values will have come from ML.

This new system eliminates half of the first-order checks associated with the first separated-guard system, but maintains equivalence with the simple natural embedding system.

Theorem 4. *For all ML expressions* $\mathbf{e}$ *and Scheme expressions* e *in the language that combines simple natural boundaries and positive and negative guards, both of the following propositions hold:*

$$\begin{array}{rrcl} (1) & (MSG^{\tau}\mathsf{e}) & \simeq & ({}^{\tau}MS_N\ (\mathcal{G}_+^{\tau}\ \mathsf{e})) \\ (2) & (GSM^{\tau}\mathbf{e}) & \simeq & (\mathcal{G}_-^{\tau}\ (SM_N^{\tau}\mathbf{e})) \end{array}$$

As it happens, we do not need to add $\mathcal{G}_+^{\tau}$ and $\mathcal{G}_-^{\tau}$ as extensions, because they can be implemented directly in the core Scheme language of figure 1. In fact, they are exactly contracts, or more specifically pairs of projections in exactly the sense of Findler and Blume [16] where the two parties are + (Scheme) and − (ML). Since ML's type system guarantees that − never breaks its contract, only + will ever be blamed for a violation. Practically, this means we can safely use contracts to protect type invariants in foreign function

$$\mathcal{T}_M^{\tau_1\to\tau_2} \stackrel{\text{def}}{=} \lambda \mathbf{x} : \mathbf{L}.\lambda \mathbf{y} : \tau_1.(\mathcal{T}_M^{\tau_2}({}^{\mathbf{L}}MS_L((SM_L^{\mathbf{L}}\mathbf{x})\mathcal{T}_S^{\tau_1}(SM_L^{\tau_1}\mathbf{y}))))$$
$$\mathcal{T}_S^{\tau_1\to\tau_2} \stackrel{\text{def}}{=} \lambda \mathsf{x}.\lambda \mathsf{y}.\mathcal{T}_S^{\tau_2}(SM_L^{\tau_2}((^{\tau_1\to\tau_2}MS_L\mathsf{x})(\mathcal{T}_M^{\tau_1}({}^{\mathbf{L}}MS_L\mathsf{y})))))$$
$$\mathcal{T}_M^{\iota} \stackrel{\text{def}}{=} (\lambda\, \mathbf{x} : \mathbf{L}.\ {}^{\iota}MS_L\ (\mathsf{Y}\ (\lambda\, \mathsf{f}\ .\ \lambda\, \mathsf{n}\ .\ (\mathtt{if0}\ \mathsf{n}\ (SM_L^{\iota}\ \overline{0})\ (SM_L^{\iota}\ (+\ \overline{1}\ ({}^{\iota}MS_L\ (\mathsf{f}\ (-\ \mathsf{n}\ \overline{1}))))))))\ (SM_L^{\mathbf{L}}\ \mathbf{x})))$$
$$\mathcal{T}_S^{\iota} \stackrel{\text{def}}{=} (\lambda\, \mathsf{x}\ .\ (\mathsf{Y}\ (\lambda\, \mathsf{f}\ .\ \lambda\, \mathsf{n}\ .\ (SM_L^{\mathbf{L}}\ (\mathtt{if0}\ ({}^{\iota}MS_L\ \mathsf{n})\ ({}^{\mathbf{L}}MS_L\ \overline{0})\ ({}^{\mathbf{L}}MS_L\ (+\ \overline{1}\ (\mathsf{f}\ (SM_L^{\iota}\ (-\ ({}^{\iota}MS_L\ \mathsf{n})\ \overline{1}))))))))\ \mathsf{x}))$$

Figure 5. Translation functions for lump values

interfaces. This adds to our confidence in Gray *et al*'s fine-grained interoperability scheme [23], for example. More theoretically, it means we can use the simple system of section 3.1 for our models and be more confident that our soundness results apply to actual multi-language embeddings that we write with contracts. From the contract perspective, it also shows one way of using mechanized reasoning to statically eliminate dynamic assertions from annotated programs. In this light it can be seen as a hybrid type system [20].

4. What have we gained?

The natural embedding fits our intuitive notion of what a useful interoperability system ought to look like much more than the lump embedding does, so it seems like it should give us the power to write more programs. However, that is not the case: ${}^{\tau}MS_N$ and SM_N^{τ} boundaries are macro-expressible in the lump embedding (in the sense of Felleisen [14]), meaning that any natural-embedding program can be translated using local transformations into a lump-embedding program.

To show that, we define two type-indexed functions, $\mathcal{T}_S^{\tau}$ and $\mathcal{T}_M^{\tau}$, that can be written in the lump embedding. These functions translate values whose ML type is τ from Scheme to ML and from ML to Scheme, respectively; they are shown in figure 5. (for clarity, in that figure and below we use the notation ${}^{\tau}MS_L$ and SM_L^{τ} rather than ${}^{\tau}MS$ and SM^{τ} to refer to the lump embedding's boundaries.) The translation functions for higher-order types use essentially the same method we presented in section 3 for converting a procedure value — they translate a procedure by left- and right-composing appropriate translators for its input and output types. That leaves us with nothing but the base types, in our case numbers.

Those require more work. The key insight is that we can use a two-language equivalent of Church-encoded numbers to send a number across a boundary, and both translators do that the same way: the receiver gives the sender the opaque number to be translated, a zero value, and a successor function. The sender then performs a foreign application of the successor function to the zero value n times where n is the number to be sent. The result of that operation is a foreign value representing the native number that was to be sent. (The two translators are superficially different from each other, but only because we restrict the Y fixed-point combinator to Scheme, where it can be written directly.)

Theorem 5. *In the language formed by extending the language of figure 1 with both figure 2 and the language of section 3.3, both of the following propositions hold:*

$$(\mathcal{G}_-^{\tau}\ (SM_N^{\tau}\boldsymbol{e})) \simeq (\mathcal{G}_-^{\tau}\ (\mathcal{T}_S^{\tau}\ (SM_L^{\tau}\boldsymbol{e})))$$
$$({}^{\tau}MS_N\ (\mathcal{G}_+^{\tau}\,\mathsf{e})) \simeq (\mathcal{T}_M^{\tau}\ ({}^{\tau}MS_L\ (\mathcal{G}_+^{\tau}\,\mathsf{e})))$$

In other words, we can run any natural-embedding program in the lump embedding by converting all of its natural-embedding boundaries into lump boundaries wrapped in the appropriate guard and conversion functions. Since each translation in figure 5 can be applied independently, since each one is a local transformation, and since guards are expressible in the core Scheme language, it follows that the SM_N^{τ} and ${}^{\tau}MS_N$ boundaries are macro-expressible in the lump embedding.

Based on these translation rules, we were able to implement a program that communicated numbers between PLT Scheme and C using PLT Scheme's foreign interface [3] but maintaining a strict lump discipline.

While the translations of figure 5 suffice for our purposes here, is worth mentioning that they could be made faster. Rather than having the sender transmit "add-one" and "stop" signals, the receiver could give the sender representations of 0 and 1 and let the sender send successive bits of the number to be converted. This approach would run in time proportional to the log of the converted number, assuming efficient bit-shift operations. This is likely to be linear in the size of the representation of integers, though this is still suboptimal since such conversions are likely to be either simple bit operations or even `noop`s at the level of the underlying machine.

5. From type-directed to type-mapped conversion

In the previous sections, we have gotten double duty out of the type annotations on boundaries: statically, they have indicated the type of each boundary, and dynamically they have indicated how to convert any value that appears at the boundary. That overloading is fine (even desirable) for some multi-language systems, but it is insufficient for others. For instance, since C does not have an exception mechanism, many C functions (e.g., `malloc`, `fopen`, `sqrt`) return a normal value on success and a sentinel value on error. A foreign function interface might automatically convert error indicators from such functions into exceptions, while converting non-errors as normal. To model such a conversion, we must generalize boundaries so that instead of containing a *type*, they contain a *conversion strategy* from which a type can be derived.

In the numbers-for-errors example, we can consider a variant of our earlier multi-language systems in which we expect ML to use the zero for error convention for some functions, and Scheme has a very simple exception mechanism in which (`wrong` *str*) raises an exception, and (`handle` e_1 e_2) tries to evaluate e_2 unless it raises an exception, in which case it abandons e_2 and instead evaluates the handler expression e_1. If a Scheme exception reaches ML, it aborts the program unless the ML boundary it reaches expects a value following the zero-for-error convention, in which case it becomes $\overline{0}$. Similarly if $\overline{0}$ flows from ML to Scheme in a context where the zero-for-error convention is in place, it becomes a Scheme exception.

We give a model of this system in figure 6. The Scheme contexts identify two different layers: those in which no boundary or exception handler appears (H), and those in which boundaries and handlers may appear (E). This layering is a simplified variant of the exception model presented by Wright and Felleisen [43].

The core of the system is a conversion strategy κ that replaces the type on all boundaries, and an associated $\lfloor\cdot\rfloor$ metafunction from κ to τ that takes a conversion strategy "down to" a type. We use the conversion strategy to add a single conversion, $\iota!$, indicating places where $\overline{0}$ indicates an error. Typing judgments are as before, except that boundary rules must apply the $\lfloor\cdot\rfloor$ function as necessary. Reducing a boundary is also as before, with additions corresponding

$$
\begin{array}{rcl}
\mathbf{e} & = & \cdots \mid (MSG^{\kappa}\ \mathsf{e}) \\
\mathsf{e} & = & \cdots \mid (\texttt{handle}\ \mathsf{e}\ \mathsf{e}) \mid (GSM^{\kappa}\ \mathbf{e}) \mid (\texttt{handle}\ \mathsf{e}\ \mathsf{e}) \\
\kappa & = & \iota \mid \iota! \mid \kappa \to \kappa
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{E} & = & \cdots \mid \mathbf{E}[(MSG^{\kappa}\ \mathsf{E})]_M \\
\mathsf{H} & = & [\,]_S \mid (\mathsf{H}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{H}) \mid (op\ \mathsf{H}\ \mathsf{e}) \mid (op\ \mathsf{v}\ \mathsf{H}) \mid (\texttt{if0}\ \mathsf{H}\ \mathsf{e}\ \mathsf{e}) \mid (pr\ \mathsf{H}) \\
\mathsf{E} & = & \mathsf{H} \mid \mathsf{H}[(GSM^{\kappa}\ \mathbf{E})]_S \mid \mathsf{H}[(\texttt{handle}\ \mathsf{e}\ \mathsf{E})]_S
\end{array}
$$

$$
\frac{\Gamma\ \vdash_S\ \mathsf{e} : \mathbf{TST}}{\Gamma\ \vdash_M\ (MSG^{\kappa}\ \mathsf{e}) : \lfloor\kappa\rfloor} \qquad \frac{\Gamma\ \vdash_M\ \mathbf{e} : \lfloor\kappa\rfloor}{\Gamma\ \vdash_S\ (GSM^{\kappa}\ \mathbf{e}) : \mathbf{TST}}
$$

$$
\begin{array}{rcl}
\lfloor\iota\rfloor & = & \iota \\
\lfloor\iota!\rfloor & = & \iota \\
\lfloor\kappa_1 \to \kappa_2\rfloor & = & \lfloor\kappa_1\rfloor \to \lfloor\kappa_2\rfloor
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}[(MSG^{\iota!}\ \overline{n})]_M & \to & \mathcal{E}[\overline{n}] \\
\mathcal{E}[(MSG^{\iota!}\ \mathsf{H}[\texttt{wrong}\ str])]_M & \to & \mathcal{E}[\overline{0}] \\
\mathcal{E}[(MSG^{\kappa}\ \mathsf{H}[\texttt{wrong}\ str])]_M & \to & \textbf{Error}\text{: } str \text{ (if } \kappa \neq \iota!) \\
\mathcal{E}[(GSM^{\iota!}\ \overline{0})]_S & \to & \mathcal{E}[(\texttt{wrong}\ \text{“zero”})] \\
\mathcal{E}[(GSM^{\iota!}\ \overline{n})]_S & \to & \mathcal{E}[\overline{n}]\ (\text{if } n \neq 0) \\
\mathcal{E}[(\texttt{handle}\ \mathsf{e}_1\ \mathsf{H}[(\texttt{wrong}\ str)])]_S & \to & \mathcal{E}[\mathsf{e}_1]
\end{array}
$$

Figure 6. Extensions to figure 3 for mapped embedding

to the ML-to-Scheme and Scheme-to-ML conversions for values at $\iota!$ boundaries.

Theorem 6. *The language of figure 6 is type-sound.*

This example demonstrates a larger point: although we have used a boundary's type as its conversion strategy for most of the systems in this paper, they are separate ideas. Decoupling them has a number of pleasant effects: first, it allows us to use non-type-directed conversions, as we have shown. Second, it illustrates that boundaries depend on conversion strategies, not type information *per se*, which suggests that type erasure (but not conversion strategy erasure) remains possible in our framework. Finally, the separation makes it easier to understand the connection between these formal systems and tools like SWIG, in particular SWIG's type-map feature [4]: from this perspective, SWIG is a tool that automatically generates boundaries that pull C++ values into Python (or another high-level language), and type-maps allow the user to write a new conversion strategy and specify the circumstances under which it should be used.

6. Related work

The work most directly related to ours is Benton's "Embedded Interpreters" [6], which lays out a method for embedding interpreters into statically typed languages using type-indexed embeddings and projections (the same method was also discovered independently by Ramsey [39]). His work considers only the asymmetric case where a typed host language embeds an untyped language, and focuses on implementation rather than formal techniques. Still, readers will find that the flavor of his work is quite similar to this work. Zdancewic, Grossman, and Morrisett's work [44] is also similar to ours in that it introduces two-agent calculi and boundaries; their work, however, focuses on information-hiding properties and does not allow different languages to interoperate.

Other work on the semantics of interoperability tends to focus on the properties of multi-language runtime systems. This includes a pair of formalisms for the semantics of COM, the first by Ibrahim and Szyperski [26] and the second by Pucella [38] and also Gordon and Syme's formalization of a type-safe intermediate language designed specifically for multi-language interoperation [22]. Kennedy [28] pointed out that in multi-language systems, observations in one language can break equations in the other and that this is a practical problem. Our system is one way to reason about these problems precisely. Trifonov and Shao have developed an abstract intermediate language for multi-language programs that aids reasoning about interactions between effects in the two source languages [42]. While our framework can also addresses effects, it does not address their implementation. Instead our work focuses on their semantics as seen by the source languages, a topic Trifonov and Shao do not discuss. Finally, Furr and Foster have built a system for verifying certain safety properties of the OCaml foreign-function interface by analyzing C code for problematic uses of OCaml values [21].

On the issue of combining typed and untyped code, Henglein and Rehof [24, 25] have done work on translating Scheme into ML, inserting ML equivalents of our guards to simulate Scheme's dynamic checks. Some languages have introduced ways of mixing typed and untyped code using a dynamic type [1], similar to our boundaries and lumps; for instance Cardelli's Amber [10], Chambers *et al*'s Cecil [13] or Gray, Findler and Flatt's ProfessorJ [23].

There has been far too much implementation work connecting high-level languages to list it all here. In addition to the projects we have already mentioned, there are dozens of compilers that target the JVM, the .NET CLR, or COM. There have also been more exotic embeddings; two somewhat recent examples are an embedding of Alice (an SML extension) into the Oz programming language [29] and the LazyScheme educational embedding of a lazy variant of Scheme into strict-evaluation-order Scheme [2]. There has been even more work on connecting high-level languages to low-level languages; in addition to the venerable SWIG [5] there are many more systems that try to sanitize the task [3, 9, 11, 19, 40].

7. Conclusion

We have shown how to give operational semantics to multi-language systems and still use the same formal techniques that apply to single languages. This work has focused on two points in the design space for interoperating languages: the lump and natural embeddings. We see aspects of these two points in many real foreign function interfaces: for instance, SML.NET translates flat values and .NET objects, but forces lump-like behavior for higher-order ML functions. The Java Native Interface provides all foreign values as lumps, but provides a large set of constant functions for manipulating them, similar to our **fa** ("foreign-apply") function from section 2. The lump embedding's ease of implementation and natural embedding's ease of use both pull on language design, and most real multi-language systems lie in between.

Furthermore, we have talked mostly about foreign function interfaces in this paper, but our technique scales to other kinds of multi-language systems. Any situation in which two logically different languages interact is a candidate for this treatment — *e.g.* embedded domain-specific languages, some uses of metaprogramming, and even contract systems, viewing each party to the contract as a separate language. By treating these uniformly as multi-language systems, we might be able to connect them in fruitful ways. Even with the limited scope considered in this paper, we have discovered connections between contracts, foreign function interfaces, and hybrid type systems (as discussed in section 3.3). Widening the net to include a broader interpretation of interoperating systems suggests that this work barely scratches the surface.

Acknowledgments The authors would like to thank Cormac Flanagan for his suggestion of the phrase "cross-language cast"; to Matthias Felleisen, Dave MacQueen, and the anonymous reviewers for their comments on earlier versions of this paper; and to David Walker, for shepherding this paper from its original twelve-

page form to the eight-page version presented here. This work was funded in part by the National Science Foundation.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] E. Barzilay and J. Clements. Laziness without all the hard work. In *Workshop on Functional and Declarative Programming in Education (FDPE)*, 2005.

[3] E. Barzilay and D. Orlovsky. Foreign interface for PLT Scheme. In *Workshop on Scheme and Functional Programming*, 2004.

[4] D. Beazley. Pointers, constraints, and typemaps. In SWIG 1.1 Users Manual. Available online: `http://www.swig.org/Doc1.1/HTML/Typemaps.html`.

[5] D. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *4th Tcl/Tk Workshop*, pages 129–139, 1996. Available online: `http://www.swig.org/papers/Tcl96/tcl96.html`.

[6] N. Benton. Embedded interpreters. *Journal of Functional Programming*, 15:503–542, July 2005.

[7] N. Benton and A. Kennedy. Interlanguage working without tears: Blending SML with Java. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 126–137, 1999.

[8] N. Benton, A. Kennedy, and C. V. Russo. Adventures in interoperability: the SML.NET experience. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 215–226, 2004.

[9] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

[10] L. Cardelli. Amber. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and functional programming languages*, volume 242. Springer-Verlag, 1986.

[11] M. M. T. Chakravarty. C→HASKELL, or yet another interfacing tool. In *International Workshop on Implementation of Functional Languages (IFL)*, 1999.

[12] M. M. T. Chakravarty. The Haskell 98 foreign function interface 1.0, 2002. Available online: `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`.

[13] C. Chambers and The Cecil Group. The Cecil language: Specification and rationale, version 3.2. Technical report, Department of Computer Science and Engineering, University of Washington, February 2004. Available online: `http://www.cs.washington.edu/research/projects/cecil/pubs/cecil-spec.html`.

[14] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[15] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992. Original version in: Technical Report 89-100, Rice University, June 1989.

[16] R. B. Findler and M. Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, 2006.

[17] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002.

[18] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 114–125, 1999.

[19] K. Fisher, R. Pucella, and J. Reppy. A framework for interoperability. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

[20] C. Flanagan. Hybrid type checking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006.

[21] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 62–72, 2005.

[22] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 248–260, 2001.

[23] K. E. Gray, R. B. Findler, and M. Flatt. Fine grained interoperability through mirrors and contracts. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2005.

[24] F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

[25] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1995.

[26] R. Ibrahim and C. Szyperski. The COMEL language. Technical Report FIT-TR-97-06, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 1997.

[27] Jython. `http://www.jython.org/`.

[28] A. Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, To appear. `http://research.microsoft.com/~akenn/sec/`.

[29] L. Kornstaedt. Alice in the land of Oz - an interoperability-based implementation of a functional language on top of a relational language. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

[30] J. Matthews and R. B. Findler. An operational semantics for R5RS Scheme. In *Workshop on Scheme and Functional Programming*, 2005.

[31] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA)*, 2004.

[32] E. Meijer, N. Perry, and A. van Yzendoorn. Scripting .NET using Mondrian. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 150–164, London, UK, 2001. Springer-Verlag.

[33] P. Meunier and D. Silva. From Python to PLT Scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 24–29, 2003.

[34] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Introduction to Scala. `http://scala.epfl.ch/docu/files/ScalaIntro.pdf`, 2005.

[35] A. Ohori and K. Kato. Semantics for communication primitives in an polymorphic language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 99–112, 1993.

[36] P. Pinto. Dot-Scheme: A PLT Scheme FFI for the .NET framework. In *Workshop on Scheme and Functional Programming*, November 2003.

[37] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, pages 223–255, 1977.

[38] R. Pucella. Towards a formalization for COM, part I: The primitive calculus. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2002.

[39] N. Ramsey. Embedding an interpreted language using higher-order functions and types. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 6–14, 2003.

[40] J. Reppy and C. Song. Application-specific foreign-interface generation. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2006.

[41] P. Steckler. MysterX: A Scheme toolkit for building interactive applications with COM. In *Technology of Object-Oriented Languages and Systems (TOOL)*, pages 364–373, 1999. Available online: `citeseer.ist.psu.edu/steckler99mysterx.html`.

[42] V. Trifonov and Z. Shao. Safe and principled language interoperation. In *European Symposium on Programming (ESOP)*, pages 128–146, 1999.

[43] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.

[44] S. Zdancewic, D. Grossman, and G. Morrisett. Principals in programming languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1999.

PLT Redex [31] implementations of all the formal systems presented here and the companion technical report in both color and black and white are available online at `http://www.cs.uchicago.edu/~jacobm/papers/multilang/`.