Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices!

Jacob Matthews[1] and Amal Ahmed[2]

[1] University of Chicago
jacobm@cs.uchicago.edu
[2] Toyota Technological Institute at Chicago
amal@tti-c.org

Abstract. We show how to extend System F's parametricity guarantee to a Matthews-Findler-style multi-language system that combines System F with an untyped language by use of dynamic sealing. While the use of sealing for this purpose has been suggested before, it has never been proven to preserve parametricity. In this paper we prove that it does using step-indexed logical relations. Using this result we show a scheme for implementing parametric higher-order contracts in an untyped setting which corresponds to a translation given by Sumii and Pierce. These contracts satisfy rich enough guarantees that we can extract analogues to Wadler's free theorems that rely on run-time enforcement of dynamic seals.

1 Introduction

There have been two major strategies for hiding the implementation details of one part of a program from its other parts: the static approach and the dynamic approach.

The static approach can be summarized by the slogan "information hiding = parametric polymorphism." In it, the language's type system is equipped with a facility such as existential types so that it can reject programs in which one module makes unwarranted assumptions about the internal details of another, even if those assumptions happen to be true. This approach rests on Reynolds' notion of abstraction [1], later redubbed the "parametricity" theorem by Wadler [2].

The dynamic approach, which goes back to Morris [3], can be summarized by the alternate slogan "information hiding = local scope + generativity." Rather than statically rejecting programs that make unwarranted assumptions, the dynamic approach simply takes away programs' ability to see if those assumptions are correct. It allows a programmer to *dynamically seal* values by creating unique keys (*create-seal* $: \rightarrow key$) and using those keys with locking and unlocking operations (*seal* $: v \times key \rightarrow opaque$ and *unseal* $: opaque \times key \rightarrow v$ respectively). A value locked with a particular key is opaque to third parties: nothing can be done but unlock it with the same key. Here is a simple implementation written in Scheme, where **gensym** is a function that generates a new, completely unique symbol every time it is called:

```
(define (create-seal) (gensym))
(define (seal v s1) (λ (s2) (if (eq? s1 s2) v (error))))
(define (unseal sealed-v s) (sealed-v s))
```

S. Drossopoulou (Ed.): ESOP 2008, LNCS 4960, pp. 16–31, 2008.

Using this facility a module can hand out a particular value while hiding its representation by creating a fresh seal in its private lexical scope, sealing the value and hand the result to clients, and then unsealing it again whenever it returns. This is the primary information-hiding mechanism in many untyped languages. For instance PLT Scheme [4] uses generative `structs`, essentially a (much) more sophisticated version of seals, to build abstractions for a great variety of programming constructs such as an object system. Furthermore, the idea has seen some use recently even in languages whose primary information-hiding mechanism is static, as recounted by Sumii and Pierce [5].

Both of these strategies seem to match an intuitive understanding of what information-hiding ought to entail. So it is surprising that a fundamental question — what is the relationship between the guarantee provided by the static approach and the dynamic approach? — has not been answered in the literature.

In this paper we take a new perspective on the problem, posing it as a question of parametricity in a multi-language system [6]. After reviewing our previous work on multi-language systems and giving a multi-language system that combines System F (henceforth "ML") and an untyped call-by-value lambda calculus (henceforth "Scheme") (section 2), we use this vantage point to show two results. First, in section 3 we show that dynamic sealing preserves ML's parametricity guarantee even when interoperating with Scheme. For the proof, we define two step-indexed logical relations [7], one for ML (indexed by both types as well as, intuitively, the number of steps available for future evaluation) and one for Scheme (indexed only by steps since Scheme is untyped). The stratification provided by step-indexing is essential for modeling unbounded computation, available in Scheme due to the presence of what amounts to a recursive type, and available in ML via interaction with Scheme. Then we show the fundamental theorems of each relation. The novelty of this proof is its use of what we call the "bridge lemma," which states that if two terms are related in one language, then wrapping those terms in boundaries results in terms that are related in the other. The proof is otherwise essentially standard. Second, in section 4 we restrict our attention to Scheme programs that use boundaries with ML only to implement a contract system [8]. Appealing to the first parametricity result, we give a more useful, contract-indexed relation for dealing with these terms and prove that it relates contracted terms to themselves. In section 4.1 we show that our notion of contracts corresponds to Findler and Felleisen's, and to a translation given by Sumii and Pierce [5, section 8].

We have elided most proofs here. They can be found in this paper's companion technical report [9].

2 A Brief Introduction to Multi-language Systems

To make the present work self-contained, in this section we summarize some relevant material from earlier work [6].

The natural embedding. The natural embedding multi-language system, presented in figure 1 is a method of modeling the semantics of a minimal "ML" (simply-typed, call-by-value lambda calculus) with a minimal "Scheme" (untyped, call-by-value lambda calculus) such that both languages have natural access to foreign values. They receive

$$
\begin{array}{rl}
\mathbf{e} & = \mathbf{x} \mid \mathbf{v} \mid (\mathbf{e}\ \mathbf{e}) \mid (op\ \mathbf{e}\ \mathbf{e}) \mid (\mathbf{if0}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e}) \\
 & \quad \mid (\mathbf{cons}\ \mathbf{e}\ \mathbf{e}) \mid ({}^{\tau}MS\ \mathbf{e}) \\
\mathbf{v} & = \lambda \mathbf{x}:\tau.\mathbf{e} \mid \overline{n} \mid \mathbf{nil} \mid (\mathbf{cons}\ \mathbf{v}_1\ \mathbf{v}_2) \mid \mathbf{fst} \mid \mathbf{rst} \\
op & = + \mid - \\
\tau & = \mathbf{Nat} \mid \tau \to \tau \mid \tau^* \\
\mathbf{x} & = \text{ML variables} \\
\mathbf{E} & = [\,]_M \mid (\mathbf{E}\ \mathbf{e}) \mid (\mathbf{v}\ \mathbf{E}) \mid (op\ \mathbf{E}\ \mathbf{e}) \mid (op\ \mathbf{v}\ \mathbf{E}) \\
 & \quad \mid (\mathbf{if0}\ \mathbf{E}\ \mathbf{e}\ \mathbf{e}) \mid (\mathbf{cons}\ \mathbf{E}\ \mathbf{e}) \mid (\mathbf{cons}\ \mathbf{v}\ \mathbf{E}) \mid ({}^{\tau}MS\ \mathbf{E})
\end{array}
$$

$$\frac{}{\Gamma, \mathbf{x}:\tau \vdash_M \mathbf{x}:\tau} \qquad \frac{\Gamma, \mathbf{x}:\tau_1 \vdash_M \mathbf{e}:\tau_2}{\Gamma \vdash_M \lambda \mathbf{x}:\tau_1.\,\mathbf{e}:\tau_1 \to \tau_2}$$

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

$$\frac{}{\Gamma \vdash_M \mathbf{nil}:\tau^*} \qquad \frac{\Gamma \vdash_M \mathbf{e}_1:\tau \quad \Gamma \vdash_M \mathbf{e}_2:\tau^*}{\Gamma \vdash_M (\mathbf{cons}\ \mathbf{e}_1\ \mathbf{e}_2):\tau^*}$$

$$\frac{}{\Gamma \vdash_M \mathbf{rst}:\tau^* \to \tau^*} \qquad \frac{}{\Gamma \vdash_M \mathbf{fst}:\tau^* \to \tau}$$

$$\frac{}{\Gamma \vdash_M \overline{n}:\mathbf{Nat}} \qquad \frac{\Gamma \vdash_M \mathbf{e}_1:\mathbf{Nat} \quad \Gamma \vdash_M \mathbf{e}_2:\mathbf{Nat}}{\Gamma \vdash_M (op\ \mathbf{e}_1\ \mathbf{e}_2):\mathbf{Nat}}$$

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

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

$$
\begin{array}{ll}
\mathscr{E}[((\lambda \mathbf{x}:\tau.\ \mathbf{e})\ \mathbf{v})]_M & \mapsto \mathscr{E}[\mathbf{e}[\mathbf{v}/\mathbf{x}]] \\
\mathscr{E}[(+\ \overline{n_1}\ \overline{n_2})]_M & \mapsto \mathscr{E}[\overline{n_1+n_2}] \\
\mathscr{E}[(-\ \overline{n_1}\ \overline{n_2})]_M & \mapsto \mathscr{E}[\overline{max(n_1-n_2,0)}] \\
\mathscr{E}[(\mathbf{if0}\ \overline{0}\ \mathbf{e}_1\ \mathbf{e}_2)]_M & \mapsto \mathscr{E}[\mathbf{e}_1] \\
\mathscr{E}[(\mathbf{if0}\ \overline{n}\ \mathbf{e}_1\ \mathbf{e}_2)]_M & \mapsto \mathscr{E}[\mathbf{e}_2] \quad \text{where } n \neq 0 \\
\mathscr{E}[(\mathbf{fst}\ (\mathbf{cons}\ \mathbf{v}_1\ \mathbf{v}_2))]_M & \mapsto \mathscr{E}[\mathbf{v}_1] \\
\mathscr{E}[(\mathbf{fst}\ \mathbf{nil})]_M & \mapsto \mathbf{Error}\text{: nil} \\
\mathscr{E}[(\mathbf{rst}\ (\mathbf{cons}\ \mathbf{v}_1\ \mathbf{v}_2))]_M & \mapsto \mathscr{E}[\mathbf{v}_2] \\
\mathscr{E}[(\mathbf{rst}\ \mathbf{nil})]_M & \mapsto \mathbf{Error}\text{: nil} \\
 & \\
\mathscr{E}[({}^{\mathbf{Nat}}MS\ \overline{n})]_M & \mapsto \mathscr{E}[\overline{n}] \\
\mathscr{E}[({}^{\mathbf{Nat}}MS\ \mathbf{v})]_M & \mapsto \mathbf{Error}\text{: Non-num} \\
 & \quad \text{where } \mathbf{v} \neq \overline{n} \text{ for any } n \\
\mathscr{E}[({}^{\tau_1 \mapsto \tau_2}MS(\lambda \mathsf{x}.\mathsf{e}))]_M & \mapsto \\
 & \quad \mathscr{E}[(\lambda \mathbf{x}:\tau_1.({}^{\tau_2}MS\,((\lambda \mathsf{x}.\mathsf{e})\,(SM^{\tau_1}\ \mathbf{x}))))] \\
\mathscr{E}[({}^{\tau_1 \mapsto \tau_2}MS\ \mathbf{v})]_M & \mapsto \mathbf{Error}\text{: non-proc} \\
 & \quad \text{where } \mathbf{v} \neq \lambda \mathsf{x}.\mathsf{e} \text{ for any } \mathsf{x}, \mathsf{e} \\
\mathscr{E}[({}^{\tau^*}MS\ \mathbf{nil})]_M & \mapsto \mathscr{E}[\mathbf{nil}] \\
\mathscr{E}[({}^{\tau^*}MS\ (\mathbf{cons}\ \mathbf{v}_1\ \mathbf{v}_2))]_M & \mapsto \\
 & \quad \mathscr{E}[(\mathbf{cons}\ ({}^{\tau}MS\ \mathbf{v}_1)\ ({}^{\tau^*}MS\ \mathbf{v}_2))] \\
\mathscr{E}[({}^{\tau^*}MS\ \mathbf{v})]_M & \mapsto \mathbf{Error}\text{: Non-list} \\
 & \quad \text{where } \mathbf{v} \text{ is not a pair or } \mathbf{nil}
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{e} & = \mathsf{v} \mid (\mathsf{e}\ \mathsf{e}) \mid \mathsf{x} \mid (op\ \mathsf{e}\ \mathsf{e}) \mid (\mathbf{if0}\ \mathsf{e}\ \mathsf{e}\ \mathsf{e}) \\
 & \quad \mid (pd\ \mathsf{e}) \mid (\mathbf{cons}\ \mathsf{e}\ \mathsf{e}) \mid (SM^{\tau}\ \mathbf{e}) \\
\mathsf{v} & = (\lambda \mathsf{x}.\mathsf{e}) \mid \overline{n} \mid \mathbf{nil} \mid (\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2) \mid \mathbf{fst} \mid \mathbf{rst} \\
op & = + \mid - \\
pd & = \mathsf{proc?} \mid \mathsf{nat?} \mid \mathsf{nil?} \mid \mathsf{pair?} \\
\mathsf{x} & = \text{Scheme variables} \\
\mathsf{E} & = [\,]_S \mid (\mathsf{E}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{E}) \mid (op\ \mathsf{E}\ \mathsf{e}) \mid (op\ \mathsf{v}\ \mathsf{E}) \\
 & \quad \mid (\mathbf{if0}\ \mathsf{E}\ \mathsf{e}\ \mathsf{e}) \mid (pred\ \mathsf{E}) \mid (\mathbf{cons}\ \mathsf{E}\ \mathsf{e}) \\
 & \quad \mid (\mathbf{cons}\ \mathsf{v}\ \mathsf{E}) \mid (SM^{\tau}\ \mathbf{E})
\end{array}
$$

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

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

$$
\begin{array}{ll}
\mathscr{E}[((\lambda \mathsf{x}.\ \mathsf{e})\ \mathsf{v})]_S & \mapsto \mathscr{E}[\mathsf{e}[\mathsf{v}/\mathsf{x}]] \\
\mathscr{E}[(\mathsf{v}_1\ \mathsf{v}_2)]_S & \mapsto \mathbf{Error}\text{: non-proc} \\
 & \quad \mathsf{v}_1 \neq \lambda \mathsf{x}.\mathsf{e} \\
\mathscr{E}[(+\ \overline{n_1}\ \overline{n_2})]_S & \mapsto \mathscr{E}[\overline{n_1+n_2}] \\
\mathscr{E}[(-\ \overline{n_1}\ \overline{n_2})]_S & \mapsto \mathscr{E}[\overline{max(n_1-n_2,0)}] \\
\mathscr{E}[(op\ \mathsf{v}_1\ \mathsf{v}_2)]_S & \mapsto \mathbf{Error}\text{: non-num} \\
 & \quad \mathsf{v}_1 \neq \overline{n} \text{ or } \mathsf{v}_2 \neq \overline{n} \\
\mathscr{E}[(\mathbf{if0}\ \overline{0}\ \mathsf{e}_1\ \mathsf{e}_2)]_S & \mapsto \mathscr{E}[\mathsf{e}_1] \\
\mathscr{E}[(\mathbf{if0}\ \mathsf{v}\ \mathsf{e}_1\mathsf{e}_2)]_S & \mapsto \mathscr{E}[\mathsf{e}_2] \quad \mathsf{v} \neq \overline{0} \\
\mathscr{E}[(\mathsf{proc?}\ (\lambda \mathsf{x}.\ \mathsf{e}))]_S & \mapsto \mathscr{E}[\overline{0}] \\
\mathscr{E}[(\mathsf{proc?}\ \mathsf{v})]_S & \mapsto \mathscr{E}[\overline{1}] \\
 & \quad \mathsf{v} \neq (\lambda \mathsf{x}.\mathsf{e}) \text{ for any } \mathsf{x}, \mathsf{e} \\
\mathscr{E}[(\mathsf{nat?}\ \overline{n})]_S & \mapsto \mathscr{E}[\overline{0}] \\
\mathscr{E}[(\mathsf{nat?}\ \mathsf{v})]_S & \mapsto \mathscr{E}[\overline{1}] \\
 & \quad \mathsf{v} \neq \overline{n} \text{ for any } n \\
\mathscr{E}[(\mathsf{nil?}\ \mathbf{nil})]_S & \mapsto \mathscr{E}[\overline{0}] \\
\mathscr{E}[(\mathsf{nil?}\ \mathsf{v})]_S & \mapsto \mathscr{E}[\overline{1}] \quad \mathsf{v} \neq \mathbf{nil} \\
\mathscr{E}[(\mathsf{pair?}\ (\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2))]_S & \mapsto \mathscr{E}[\overline{0}] \\
\mathscr{E}[(\mathsf{pair?}\ \mathsf{v})]_S & \mapsto \mathscr{E}[\overline{1}] \\
 & \quad \mathsf{v} \neq (\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2) \text{ for any } \mathsf{v}_1, \mathsf{v}_2 \\
\mathscr{E}[(\mathbf{fst}\ (\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2))]_S & \mapsto \mathscr{E}[\mathsf{v}_1] \\
\mathscr{E}[(\mathbf{fst}\ \mathsf{v})]_S & \mapsto \mathbf{Error}\text{: non-pair} \\
 & \quad \mathsf{v} \neq (\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2) \text{ for any } \mathsf{v}_1, \mathsf{v}_2 \\
\mathscr{E}[(\mathbf{rst}\ (\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2))]_S & \mapsto \mathscr{E}[\mathsf{v}_2] \\
\mathscr{E}[(\mathbf{rst}\ \mathsf{v})]_S & \mapsto \mathbf{Error}\text{: non-pair} \\
 & \quad \mathsf{v} \neq (\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2) \text{ for any } \mathsf{v}_1, \mathsf{v}_2 \\
\mathscr{E}[(SM^{\mathbf{Nat}}\ \overline{n})]_S & \mapsto \mathscr{E}[\overline{n}] \\
\mathscr{E}[(SM^{\tau_1 \mapsto \tau_2}\ \mathbf{v})]_S & \mapsto \\
 & \quad \mathscr{E}[(\lambda \mathsf{x}.\,(SM^{\tau_2}(\mathbf{v}\,({}^{\tau_1}MS\,\mathsf{x}))))] \\
\mathscr{E}[(SM^{\tau^*}\ \mathbf{nil})]_S & \mapsto \mathscr{E}[\mathbf{nil}] \\
\mathscr{E}[(SM^{\tau^*}\ (\mathbf{cons}\ \mathbf{v}_1\ \mathbf{v}_2))]_S & \mapsto \\
 & \quad \mathscr{E}[(\mathbf{cons}\ (SM^{\tau}\mathbf{v}_1)\ (SM^{\tau^*}\ \mathbf{v}_2))]
\end{array}
$$

Fig. 1. Natural embedding of ML (left) and Scheme (right)

foreign numbers as native numbers, and they can call foreign functions as native functions. Note that throughout this paper 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. These font differences are semantically meaningful.

To the core languages we add new syntax, evaluation contexts, and reduction rules that define syntactic boundaries, written $^{\tau}MS$ and SM^{τ}, to allow cross-language communication. (For this paper we have chosen arbitrarily to make top-level programs be ML programs that optionally call into Scheme, and so we choose $\mathscr{E} = \mathbf{E}$; to make it the other way around we would let $\mathscr{E} = \mathsf{E}$ instead.) We assume we can translate numbers from one language to the other, and give reduction rules for boundary-crossing numbers based on that assumption:

$$\mathscr{E}[(SM^{\mathbf{Nat}}\ \overline{n})]_S \mapsto \mathscr{E}[\overline{n}] \qquad\qquad \mathscr{E}[(^{\mathbf{Nat}}MS\ \overline{n})]_M \mapsto \mathscr{E}[\overline{n}]$$

To convert procedures across languages, we use native proxy procedures. We represent a Scheme procedure in ML at type $\tau_1 \to \tau_2$ 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, $(^{\tau_1 \to \tau_2}MS\ \lambda \mathsf{x}.\mathsf{e})$ becomes $(\lambda \mathbf{x} : \tau_1 .\, ^{\tau_2}MS\ ((\lambda \mathsf{x}.\mathsf{e})\ (SM^{\tau_1}\ \mathbf{x})))$ and vice versa for Scheme to ML. Note that the boundary that converts the argument is an SM^{τ_1} boundary, not an $^{\tau_1}MS$ boundary—i.e., the direction of conversion reverses for function arguments. Whenever a Scheme value is converted to ML, we also check that value's first order properties: we check to see if a Scheme value is a number before converting it to an ML value of type **Nat** and that it is a procedure value before converting it to an ML value of arrow type (and signal an error if either check fails).

Theorem 1 (Natural embedding type safety [6]). *If* $\vdash_M \boldsymbol{e} : \tau$*, then either* $\boldsymbol{e} \mapsto^* \boldsymbol{v}$*,* $\boldsymbol{e} \mapsto^*$ **Error***: str, or* $\boldsymbol{e}$ *diverges.*

We showed in prior work that the dynamic checks in this system naturally give rise to higher-order contracts [8,10]; in section 4 of this work we show another way of arriving at the same conclusion, this time equating a contract enforcing that an untyped term e behave as a (closed) type specification τ (which we write e^{τ}) by converting it to and from ML at that type: to a first approximation, $\mathsf{e}^{\tau} = (SM^{\tau}\ (^{\tau}MS\ \mathsf{e}))$.

2.1 Polymorphism, Attempt One

An omission from the "ML" side of the natural embedding to this point is that it contains no polymorphism. We now extend it to support polymorphism by replacing the simply-typed lambda calculus with System F. When we do so, we immediately hit the question of how to properly handle boundaries. In this subsection, we make what we consider the most straightforward decision of how to handle boundaries and show that it results in a system that does not preserve System F's parametricity property; in the next subsection we refine our strategy using dynamic sealing techniques.

Figure 2 shows the extensions we need to make to figure 1 to support non-parametric polymorphism. To ML's syntax we add type abstractions ($\Lambda\, \alpha.\ \mathbf{e}$) and type application ($\mathbf{e}\langle\tau\rangle$); to its types we add $\forall \alpha.\ \tau$ and α. Our embedding converts Scheme functions that work polymorphically into polymorphic ML values, and converts ML type abstractions directly into plain Scheme functions that behave polymorphically. For example, ML

$$\mathbf{e} = \cdots \mid \Lambda\alpha.\mathbf{e} \mid \mathbf{e}\langle\tau\rangle$$
$$\mathbf{v} = \cdots \mid \Lambda\alpha.\mathbf{e} \mid ({}^{\mathbf{L}}MS\ \mathsf{v})$$
$$\tau = \cdots \mid \forall\alpha.\tau \mid \alpha \mid \mathbf{L}$$
$$\Delta = \bullet \mid \Delta, \tau$$
$$\mathbf{E} = \cdots \mid \mathbf{E}\langle\tau\rangle$$

$$\frac{\Delta, \alpha; \Gamma \vdash_M \mathbf{e} : \tau}{\Delta; \Gamma \vdash_M (\Lambda\alpha.\mathbf{e}) : \forall\alpha.\tau}$$

$$\frac{\Delta; \Gamma \vdash_M \mathbf{e} : \forall\alpha.\tau' \quad \Delta \vdash \tau}{\Delta; \Gamma \vdash_M \mathbf{e}\langle\tau\rangle : \tau'[\tau/\alpha]}$$

$$\mathscr{E}[(\Lambda\alpha.\mathbf{e})\langle\tau\rangle]_M \mapsto \mathscr{E}[\mathbf{e}[\tau/\alpha]]$$
$$\mathscr{E}[({}^{\forall\alpha.\tau}MS\ \mathsf{v})]_M \mapsto \mathscr{E}[(\Lambda\alpha.({}^{\tau}MS\ \mathsf{v}))]$$
$$\mathscr{E}[(SM^{\forall\alpha.\tau}\ \mathbf{v})]_S \mapsto \mathscr{E}[(SM^{\tau[\mathbf{L}/\alpha]}\ \mathbf{v}\langle\mathbf{L}\rangle)]$$
$$\mathscr{E}[(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \mathsf{v}))]_S \mapsto \mathscr{E}[\mathsf{v}]$$

Fig. 2. Extensions to figure 1 for non-parametric polymorphism

might receive the Scheme function $(\lambda\mathsf{x}.\mathsf{x})$ from a boundary with type $\forall\alpha.\alpha \to \alpha$ and use it successfully as an identity function, and Scheme might receive the ML type abstraction $(\Lambda\alpha.\lambda\mathbf{x} : \alpha.\mathbf{x})$ as a regular function that behaves as the identity function for any value Scheme gives it.

To support this behavior, the model must create a type abstraction from a regular Scheme value when converting from Scheme to ML, and must drop a type abstraction when converting from ML to Scheme. The former is straightforward: we reduce a redex of the form $({}^{\forall\alpha.\tau}MS\ \mathsf{v})$ by dropping the $\forall$ quantifier on the type in the boundary and binding the now-free type variable in τ by wrapping the entire expression in a Λ form, yielding $(\Lambda\alpha.\ ({}^{\tau}MS\ \mathsf{v}))$.

This works for ML, but making a dual of it in Scheme would be somewhat silly, since every Scheme value inhabits the same type so type abstraction and application forms would be useless. Instead, we would like to allow Scheme to use an ML value of type, say, $\forall\alpha.\alpha \to \alpha$ directly as a function. To make boundaries with universally-quantified types behave that way, when we convert a polymorphic ML value to a Scheme value we need to remove its initial type-abstraction by applying it to some type and then convert the resulting value according to the resulting type. As for which type to apply it to, we need a type to which we can reliably convert any Scheme value, though it must not expose any of those values' properties. In prior work, we used the "lump" type to represent arbitrary, opaque Scheme values in ML; we reuse it here as the argument to the ML type abstraction. More specifically, we add **L** as a new base type in ML and we add the cancellation rule for lumps to the set of reductions: these changes, along with all the other additions required to support polymorphism, are summarized in figure 2.

2.2 Polymorphism, Attempt Two

Although this embedding is type safe, the polymorphism is not parametric in the sense of Reynolds [1]. We can see this with an example: it is well-known that in System F, for which parametricity holds, the only value with type $\forall\alpha.\alpha \to \alpha$ is the polymorphic identity function. In the system we have built so far, though, the term

$$({}^{\forall\alpha.\alpha\to\alpha}MS(\lambda\mathsf{x}.(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{x})\ (+\ \mathsf{x}\ \overline{1})\ \mathsf{x})))$$

has type $\forall\alpha.\alpha \to \alpha$ but when applied to the type **Nat** evaluates to

$$(\lambda\mathbf{y}.({}^{\mathbf{Nat}}MS((\lambda\mathsf{x}.(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{x})\ (+\ \mathsf{x}\ \overline{1})\ \mathsf{x})(SM^{\mathbf{Nat}}\mathbf{y})))))$$

Since the argument to this function is always a number, this is equivalent to

$$(\lambda\mathbf{y}.({}^{\mathbf{Nat}}MS((\lambda\mathsf{x}.(+\ \mathsf{x}\ \overline{1}))(SM^{\mathbf{Nat}}\mathbf{y}))))$$

which is well-typed but is not the identity function.

The problem with the misbehaving $\forall\alpha.\alpha \to \alpha$ function above is that while the type system rules out ML fragments that try to treat values of type α non-generically, it still

$$\begin{array}{l}
\mathbf{e} = \cdots \mid \Lambda\alpha.\mathbf{e} \mid \mathbf{e}\langle\tau\rangle \mid ({}^{\kappa}MS\ \mathsf{e}) \\
\mathsf{e} = \cdots \mid (SM^{\kappa}\ \mathbf{e}) \\
\mathbf{v} = \cdots \mid \Lambda\alpha.\mathbf{e} \mid ({}^{\mathbf{L}}MS\ \mathsf{v}) \\
\mathsf{v} = \cdots \mid (SM^{\langle\beta;\tau\rangle}\ \mathbf{v}) \\
\tau = \cdots \mid \forall\alpha.\tau \mid \alpha \mid \mathbf{L} \\
\kappa = \mathbf{Nat} \mid \kappa_1 \to \kappa_2 \mid \kappa^* \mid \forall\alpha.\kappa \mid \alpha \mid \mathbf{L} \mid \langle\alpha;\tau\rangle
\end{array}$$

$$\frac{\Delta,\alpha;\Gamma \vdash_M \mathbf{e} : \tau}{\Delta;\Gamma \vdash_M (\Lambda\alpha.\mathbf{e}) : \forall\alpha.\tau} \qquad \frac{\Delta;\Gamma \vdash_M \mathbf{e} : \forall\alpha.\tau' \quad \Delta \vdash \tau}{\Delta;\Gamma \vdash_M \mathbf{e}\langle\tau\rangle : \tau'[\tau/\alpha]}$$

$$\frac{\Delta;\Gamma \vdash_S \mathsf{e} : \mathbf{TST} \quad \Delta \vdash \lfloor\kappa\rfloor}{\Delta;\Gamma \vdash_M ({}^{\kappa}MS\ \mathsf{e}) : \lfloor\kappa\rfloor} \qquad \frac{\Delta;\Gamma \vdash_M \mathbf{e} : \lfloor\kappa\rfloor \quad \Delta \vdash \lfloor\kappa\rfloor}{\Delta;\Gamma \vdash_S (SM^{\kappa}\ \mathbf{e}) : \mathbf{TST}}$$

$$\begin{array}{rl}
\mathscr{E}[(SM^{\forall\alpha.\tau}\ \mathbf{v})]_S &\mapsto \mathscr{E}[(SM^{\tau[\mathbf{L}/\alpha]}\ \mathbf{v}\langle\mathbf{L}\rangle)] \\
\mathscr{E}[({}^{\mathbf{L}}MS\ ({}^{\mathbf{L}}MS\ \mathsf{v}))]_S &\mapsto \mathscr{E}[\mathsf{v}] \\
\\
\mathscr{E}[(\Lambda\alpha.\mathbf{e})\langle\tau\rangle]_M &\mapsto \mathscr{E}[\mathbf{e}[\tau/\alpha]] \\
\mathscr{E}[({}^{\forall\alpha.\kappa}MS\ \mathsf{v})]_M &\mapsto \mathscr{E}[(\Lambda\alpha.({}^{\kappa}MS\ \mathsf{v}))] \\
\mathscr{E}[({}^{\langle\alpha;\tau\rangle}MS\ (SM^{\langle\alpha;\tau\rangle}\ \mathbf{v}))]_M &\mapsto \mathscr{E}[\mathbf{v}] \\
\mathscr{E}[({}^{\langle\alpha;\tau\rangle}MS\ \mathsf{v})]_M &\mapsto \mathbf{Error}\text{: bad value} \\
&\quad (\mathsf{v} \neq SM^{\langle\alpha;\tau\rangle}\ \mathbf{v} \text{ for any } \mathbf{v})
\end{array}$$

$$\begin{array}{rl}
\lfloor\ \rfloor : & \kappa \to \tau \\
\lfloor\mathbf{Nat}\rfloor &= \mathbf{Nat} \\
\lfloor\kappa_1 \to \kappa_2\rfloor &= \lfloor\kappa_1\rfloor \to \lfloor\kappa_2\rfloor \\
\lfloor\kappa^*\rfloor &= \lfloor\kappa\rfloor^* \\
\lfloor\forall\alpha.\kappa\rfloor &= \forall\alpha.\lfloor\kappa\rfloor \\
\lfloor\alpha\rfloor &= \alpha \\
\lfloor\mathbf{L}\rfloor &= \mathbf{L} \\
\lfloor\langle\alpha;\tau\rangle\rfloor &= \tau
\end{array}$$

Fig. 3. Extensions to figure 1 to support parametric polymorphism

allows Scheme programs to observe the concrete choice made for α and act accordingly. To restore parametricity, we use dynamic seals to protect ML values whose implementation should not be observed. When ML provides Scheme with a value whose original type was α, Scheme gets a sealed value; when Scheme returns a value to ML at a type that was originally α, ML unseals it or signals an error if it is not a sealed value with the appropriate key.

This means that we can no longer directly substitute types for free type variables on boundary annotations. Instead we introduce *seals* as type-like annotations of the form $\langle\alpha;\tau\rangle$ that indicate on a boundary's type annotation that a particular type is the instantiation of what was originally a type variable, and *conversion schemes* (indicated with metavariable κ) as types that may also contain seals; conversion schemes only appear as the annotations on boundaries. From a technical standpoint, seals are introduced into a reduction sequence by the type substitution in the type application rule. For a precise definition, a *type substitution* η is a partial function from type variables to closed types. We extend type substitutions to apply to types, conversion schemes, and terms as follows (we show the interesting cases, the rest are merely structural recursion):

$$\eta(\alpha) \stackrel{\text{def}}{=} \begin{cases} \tau \text{ if } \exists\eta'.\ \eta = \eta', \alpha : \tau \\ \alpha \text{ otherwise} \end{cases} \qquad \begin{array}{l} \eta({}^{\kappa}MS\ \mathsf{e}) \stackrel{\text{def}}{=} {}^{\mathbf{sl}(\eta,\kappa)}MS\ \eta(\mathsf{e}) \\ \eta(SM^{\kappa}\ \mathbf{e}) \stackrel{\text{def}}{=} SM^{\mathbf{sl}(\eta,\kappa)}\ \eta(\mathbf{e}) \end{array}$$

The boundary cases (which use the seal metafunction $\mathbf{sl}(\cdot,\cdot)$ defined below) are different from the regular type cases. When we close a type with respect to a type substitution η, we simply replace all occurrences of free variables with their mappings in η, but when we close a conversion scheme with respect to a type substitution we replace free variables with "sealed" instances of the types in η. The effect of this is that even when we have performed a type substitution, we can distinguish between a type that was concrete in the original program and a type that was abstract in the original program but has been substituted with a concrete type. The $\mathbf{sl}(\cdot,\cdot)$ metafunction maps a type τ (or more generally a conversion scheme κ) to an isomorphic conversion scheme κ where

each instance of each type variable that occurs free in τ is replaced by an appropriate sealing declaration, if the type variable is in the domain of η.

Definition 1 (sealing). *The metafunction* $\mathbf{sl}(\eta, \kappa)$ *is defined as follows:*

$$\begin{array}{lll}
\mathbf{sl}(\cdot,\cdot) & : & \eta \times \kappa \to \kappa \\
\mathbf{sl}(\eta,\alpha) & \stackrel{\text{def}}{=} & \begin{cases} \langle \alpha; \eta(\alpha) \rangle & \textit{if } \eta(\alpha) \textit{ is defined} \\ \alpha & \textit{otherwise} \end{cases} \\
\mathbf{sl}(\eta, \langle \alpha; \tau \rangle) & \stackrel{\text{def}}{=} & \langle \alpha; \tau \rangle \\
\mathbf{sl}(\eta, \boldsymbol{L}) & \stackrel{\text{def}}{=} & \boldsymbol{L}
\end{array}
\qquad
\begin{array}{ll}
\mathbf{sl}(\eta, \mathbf{Nat}) & \stackrel{\text{def}}{=} \mathbf{Nat} \\
\mathbf{sl}(\eta, \kappa_1 \to \kappa_2) & \stackrel{\text{def}}{=} \mathbf{sl}(\eta, \kappa_1) \to \mathbf{sl}(\eta, \kappa_2) \\
\mathbf{sl}(\eta, \forall \alpha.\kappa_1) & \stackrel{\text{def}}{=} \forall \alpha.\mathbf{sl}(\eta, \kappa_1) \\
\mathbf{sl}(\eta, \kappa^*) & \stackrel{\text{def}}{=} \mathbf{sl}(\eta, \kappa)^*
\end{array}$$

We use the *seal erasure* metafunction $\lfloor\ \rfloor$ to project conversion schemes to types. Figure 3 defines these changes precisely. One final subtlety not written in figure 3 is that we treat a seal $\langle \alpha; \tau \rangle$ as a free occurrence of α for the purposes of capture-avoiding substitution, and we treat boundaries that include $\forall \alpha.\tau$ types as though they were binding instances of α. In fact, the production of fresh names by capture-avoiding substitution corresponds exactly to the production of fresh seals for information hiding, and the system would be neither parametric nor even type-sound were we to omit this detail.

3 Parametricity

In this section we establish that the language of figure 3 is parametric, in the sense that all terms in the language map related environments to related results, using a syntactic logical relation. Our parametricity property does not establish the exact same equivalences that would hold for terms in plain System F, but only because the embedding we are considering gives terms the power to diverge and to signal errors. So, for example, we cannot show that any ML value of type $\forall \alpha.\alpha \to \alpha$ must be the identity function, but we *can* show that it must be either the identity function, the function that always diverges, or the function that always signals an error.

Our proof amounts to defining two logical relations, one for ML and one for Scheme (see figure 4) and proving that the ML (Scheme) relation relates each ML (Scheme) term to itself regardless of the interpretation of free type variables. Though logical relations in the literature are usually defined by induction on types, we cannot use a type-indexed relation for Scheme since Scheme has only one type. This means in particular that the arguments to function values have types that are as large as the type of the function values themselves; thus any relation that defines two functions to be related if the results are related for any pair of related arguments would not be well-founded. Instead we use a minor adaptation of the step-indexed logical relation for recursive types given by Ahmed [7]: our Scheme logical relation is indexed by the number of steps k available for computation. Intuitively, any two values are related for k steps if they cannot be distinguished by any computation running for no more than k steps.

Since we are interested in proving properties of ML terms that may contain Scheme subterms, the ML relation must also be step-indexed — if the Scheme subterms are only related for (say) 50 steps, then the ML terms cannot always be related for arbitrarily many steps. Thus, the ML relation is indexed by both types and steps (as in Ahmed [7]).

The definitions are largely independent (though we do make a few concessions on this front, in particular at the definition of the ML relation at type $\mathbf{L}$), but the

$$\mathbf{Rel}_{\tau_1,\tau_2} \;=\; \{\,\mathbf{R} \mid \forall (k,\mathbf{v}_1,\mathbf{v}_2) \in \mathbf{R}.\ \forall j \le k.\ (j,\mathbf{v}_1,\mathbf{v}_2) \in \mathbf{R} \text{ and } ;\vdash \mathbf{v}_1 : \tau_1 \text{ and } ;\vdash \mathbf{v}_2 : \tau_2\,\}$$

$$\Delta \vdash \delta \quad \stackrel{\text{def}}{=} \quad \Delta \subseteq \mathsf{dom}(\delta) \text{ and } \forall \alpha \in \Delta.\ \delta_R(\alpha) \in \mathbf{Rel}_{\delta_1(\alpha),\delta_2(\alpha)}$$

$$\begin{array}{lll}
\delta \vdash \gamma_M \le^k \gamma'_M : \Gamma_M & \stackrel{\text{def}}{=} & \forall (\mathbf{x}:\tau) \in \Gamma_M.\ \gamma_M(\mathbf{x}) = \mathbf{v}_1,\ \gamma'_M(\mathbf{x}) = \mathbf{v}_2 \text{ and } \delta \vdash \mathbf{v}_1 \lessapprox^k_M \mathbf{v}_2 : \tau \\
\delta \vdash \gamma_S \le^k \gamma'_S : \Gamma_S & \stackrel{\text{def}}{=} & \forall (\mathsf{x}:\mathbf{TST}) \in \Gamma_S.\ \gamma_S(\mathsf{x}) = \mathsf{v}_1,\ \gamma'_S(\mathsf{x}) = \mathsf{v}_2 \text{ and } \delta \vdash \mathsf{v}_1 \lessapprox^k_S \mathsf{v}_2 : \mathbf{TST} \\
\delta \vdash \gamma \le^k \gamma' : \Gamma & \stackrel{\text{def}}{=} & \Gamma = \Gamma_M \cup \Gamma_S,\ \gamma = \gamma_M \cup \gamma_S,\ \gamma' = \gamma'_M \cup \gamma'_S \text{ and} \\
& & \quad \delta \vdash \gamma_M \le^k \gamma'_M : \Gamma_M \text{ and } \delta \vdash \gamma_S \le^k \gamma'_S : \Gamma_S
\end{array}$$

$$\begin{array}{lll}
\Delta;\Gamma \vdash \mathbf{e}_1 \lesssim_M \mathbf{e}_2 : \tau & \stackrel{\text{def}}{=} & \forall k \ge 0.\ \forall \delta,\gamma_1,\gamma_2.\ \Delta \vdash \delta \text{ and } \delta \vdash \gamma_1 \le^k \gamma_2 : \Gamma \ \Rightarrow \\
& & \quad \delta \vdash \delta_1(\gamma_1(\mathbf{e}_1)) \lessapprox^k_M \delta_2(\gamma_2(\mathbf{e}_2)) : \tau \\
\delta \vdash \mathbf{e}_1 \lessapprox^k_M \mathbf{e}_2 : \tau & \stackrel{\text{def}}{=} & \forall j < k.\ (\mathbf{e}_1 \hookrightarrow^j \mathbf{Error}\text{: s} \ \Rightarrow\ \mathbf{e}_2 \hookrightarrow^* \mathbf{Error}\text{: s}) \text{ and} \\
& & \quad (\forall \mathbf{v}_1.\ \mathbf{e}_1 \hookrightarrow^j \mathbf{v}_1 \ \Rightarrow\ \exists \mathbf{v}_2.\ \mathbf{e}_2 \hookrightarrow^* \mathbf{v}_2 \text{ and } \delta \vdash \mathbf{v}_1 \lessapprox^{k-j}_M \mathbf{v}_2 : \tau)
\end{array}$$

$$\begin{array}{lll}
\delta \vdash \mathbf{v}_1 \lessapprox^k_M \mathbf{v}_2 : \alpha & \stackrel{\text{def}}{=} & (k,\mathbf{v}_1,\mathbf{v}_2) \in \delta_R(\alpha) \\
\delta \vdash {}^{\mathbf{L}}MS\ \mathsf{v}_1 \lessapprox^k_M {}^{\mathbf{L}}MS\ \mathsf{v}_2 : \mathbf{L} & \stackrel{\text{def}}{=} & \forall j < k.\ \delta \vdash \mathsf{v}_1 \lessapprox^j_S \mathsf{v}_2 : \mathbf{TST} \\
\delta \vdash \overline{n} \lessapprox^k_M \overline{n} : \mathbf{Nat} & & \textit{(unconditionally)} \\
\delta \vdash \lambda \mathbf{x} : \tau_1.\mathbf{e}_1 \lessapprox^k_M \lambda \mathbf{x} : \tau_1.\mathbf{e}_2 : \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & \forall j < k.\ \forall \mathbf{v}_1,\mathbf{v}_2.\ \delta \vdash \mathbf{v}_1 \lessapprox^j_M \mathbf{v}_2 : \tau_1 \ \Rightarrow \\
& & \quad \delta \vdash \mathbf{e}_1[\mathbf{v}_1/\mathbf{x}] \lessapprox^j_M \mathbf{e}_2[\mathbf{v}_2/\mathbf{x}] : \tau_2 \\
\delta \vdash \Lambda\alpha.\mathbf{e}_1 \lessapprox^k_M \Lambda\alpha.\mathbf{e}_2 : \forall\alpha.\tau & \stackrel{\text{def}}{=} & \forall j < k.\ \forall \text{ closed } \tau_1,\tau_2.\ \forall \mathbf{R} \in \mathbf{Rel}_{\tau_1,\tau_2}. \\
& & \quad \delta,\alpha : (\tau_1,\tau_2,\mathbf{R}) \vdash \mathbf{e}_1[\tau_1/\alpha] \lessapprox^j_M \mathbf{e}_2[\tau_2/\alpha] : \tau \\
\delta \vdash [\mathbf{v}_1,\cdots,\mathbf{v}_n] \lessapprox^k_M [\mathbf{v}'_1,\cdots,\mathbf{v}'_n] : \tau^* & \stackrel{\text{def}}{=} & \forall j < k.\ \forall i \in 1\ldots n.\ \delta \vdash \mathbf{v}_i \lessapprox^j_M \mathbf{v}'_i : \tau
\end{array}$$

$$\begin{array}{lll}
\Delta;\Gamma \vdash \mathsf{e}_1 \lesssim_S \mathsf{e}_2 : \mathbf{TST} & \stackrel{\text{def}}{=} & \forall k \ge 0.\ \forall \delta,\gamma_1,\gamma_2.\ \Delta \vdash \delta \text{ and } \delta \vdash \gamma_1 \le^k \gamma_2 : \Gamma \ \Rightarrow \\
& & \quad \delta \vdash \delta_1(\gamma_1(\mathsf{e}_1)) \lessapprox^k_S \delta_2(\gamma_2(\mathsf{e}_2)) : \mathbf{TST} \\
\delta \vdash \mathsf{e}_1 \lessapprox^k_S \mathsf{e}_2 : \mathbf{TST} & \stackrel{\text{def}}{=} & \forall j < k.\ (\mathsf{e}_1 \hookrightarrow^j \mathbf{Error}\text{: s} \ \Rightarrow\ \mathsf{e}_2 \hookrightarrow^* \mathbf{Error}\text{: s}) \text{ and} \\
& & \quad (\forall \mathsf{v}_1.\ \mathsf{e}_1 \hookrightarrow^j \mathsf{v}_1 \ \Rightarrow\ \exists \mathsf{v}_2.\ \mathsf{e}_2 \hookrightarrow^* \mathsf{v}_2 \text{ and } \delta \vdash \mathsf{v}_1 \lessapprox^{k-j}_S \mathsf{v}_2 : \mathbf{TST})
\end{array}$$

$$\begin{array}{lll}
\delta \vdash \overline{n} \lessapprox^k_S \overline{n} : \mathbf{TST} & & \textit{(unconditionally)} \\
\delta \vdash (SM^{\langle\alpha;\tau_1\rangle}\ \mathbf{v}_1) \lessapprox^k_S (SM^{\langle\alpha;\tau_2\rangle}\ \mathbf{v}_2) : \mathbf{TST} & \stackrel{\text{def}}{=} & (k,\mathbf{v}_1,\mathbf{v}_2) \in \delta_R(\alpha) \text{ and } \delta_1(\alpha) = \tau_1 \text{ and } \delta_2(\alpha) = \tau_2 \\
\delta \vdash \lambda \mathsf{x}.\mathsf{e}_1 \lessapprox^k_S \lambda \mathsf{x}.\mathsf{e}_2 : \mathbf{TST} & \stackrel{\text{def}}{=} & \forall j < k.\forall \mathsf{v}_1,\mathsf{v}_2.\ \delta \vdash \mathsf{v}_1 \lessapprox^j_S \mathsf{v}_2 : \mathbf{TST} \ \Rightarrow \\
& & \quad \delta \vdash \mathsf{e}_1[\mathsf{v}_1/\mathsf{x}] \lessapprox^j_S \mathsf{e}_2[\mathsf{v}_2/\mathsf{x}] : \mathbf{TST} \\
\delta \vdash \mathbf{nil} \lessapprox^k_S \mathbf{nil} : \mathbf{TST} & & \textit{(unconditionally)} \\
\delta \vdash (\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2) \lessapprox^k_S (\mathbf{cons}\ \mathsf{v}'_1\ \mathsf{v}'_2) : \mathbf{TST} & \stackrel{\text{def}}{=} & \forall j < k.\ \delta \vdash \mathsf{v}_1 \lessapprox^j_S \mathsf{v}'_1 : \mathbf{TST} \text{ and } \delta \vdash \mathsf{v}_2 \lessapprox^j_S \mathsf{v}'_2 : \mathbf{TST}
\end{array}$$

Fig. 4. Logical approximation for ML terms (middle) and Scheme terms (bottom)

proofs cannot be, since an ML term can have an embedded Scheme subexpression and vice versa. Instead, we prove the two claims by simultaneous induction and rely on a critical "bridge lemma" (lemma 1, see below) that lets us carry relatedness between languages.

Preliminaries. A *type relation* δ is a partial function from type variables to triples $(\tau_1, \tau_2, \mathbf{R})$, where τ_1 and τ_2 are closed types and R is a set of triples of the form $(k, \mathbf{v}_1, \mathbf{v}_2)$ (which intuitively means that $\mathbf{v}_1$ and $\mathbf{v}_2$ are related for k steps). We use the following notations: If $\delta(\alpha) = (\tau_1, \tau_2, \mathbf{R})$ then $\delta_1(\alpha) = \tau_1$, $\delta_2(\alpha) = \tau_2$, and $\delta_R(\alpha) = \mathbf{R}$. We also treat δ_1 and δ_2 as type substitutions. In the definition of the logical relation we only allow *downward closed* relations as choices for $\mathbf{R}$; *i.e.* relations that relate two values for k steps must also relate them for all $j < k$ steps. We make this restriction because downward closure is a critical property that would not otherwise hold.

A Scheme (ML) substitution γ_S (γ_M) is a partial map from Scheme (ML) variables to closed Scheme (ML) values, and a substitution $\gamma = \gamma_S \cup \gamma_M$ for some γ_S, γ_M. We say that $e \hookrightarrow v$ (or **Error**: s) if in all evaluation contexts $\mathscr{E}[e] \mapsto \mathscr{E}[v]$ (or **Error**: s).

Lemma 1 (bridge lemma). *For all $k \geq 0$, type environments Δ, type relations δ such that $\Delta \vdash \delta$, types τ such that $\Delta \vdash \tau$, both of the following hold:*

1. *For all e_1 and e_2, if $\delta \vdash \mathsf{e}_1 \lesssim_S^k \mathsf{e}_2 : \textbf{\textit{TST}}$ then $\delta \vdash (^{\mathbf{sl}(\delta_1,\tau)}MS\ \mathsf{e}_1) \lesssim_M^k (^{\mathbf{sl}(\delta_2,\tau)}MS\ \mathsf{e}_2) : \tau$.*
2. *For all $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, if $\delta \vdash \boldsymbol{e}_1 \lesssim_M^k \boldsymbol{e}_2 : \tau$, then $\delta \vdash (SM^{\mathbf{sl}(\delta_1,\tau)}\ \boldsymbol{e}_1) \lesssim_S^k (SM^{\mathbf{sl}(\delta_2,\tau)}\ \boldsymbol{e}_2) : \textbf{\textit{TST}}$.*

Proof. By induction on τ. All cases are straightforward given the induction hypotheses.

With the bridge lemma established, the fundamental theorem (and hence the fact that logical approximation implies contextual approximation) is essentially standard. We restrict the parametricity theorem to seal-free terms; otherwise we would have to show that any sealed value is related to itself at type α which is false. (A conversion strategy is seal-free if it contains no instances of $\langle \alpha; \tau \rangle$ for any α. A term is seal-free if it contains no conversion strategies with seals.) This restriction is purely technical, since the claim applies to open terms where seals may be introduced by closing environments.

Theorem 2 (parametricity / fundamental theorem). *For all seal-free terms $\boldsymbol{e}$ and e:*

1. *If $\Delta; \Gamma \vdash_M \boldsymbol{e} : \tau$, then $\Delta; \Gamma \vdash \boldsymbol{e} \lesssim_M \boldsymbol{e} : \tau$.*
2. *If $\Delta; \Gamma \vdash_S \mathsf{e} : \textbf{\textit{TST}}$, then $\Delta; \Gamma \vdash \mathsf{e} \lesssim_S \mathsf{e} : \textbf{\textit{TST}}$.*

Proof. By simultaneous induction on the derivations $\Delta; \Gamma \vdash_M \mathbf{e} : \tau$ and $\Delta; \Gamma \vdash_S \mathsf{e} : \mathbf{TST}$. The boundary cases both follow from lemma 1.

4 From Multi-language to Single-Language Sealing

Suppose that instead of reasoning about multi-language programs, we want to reason about Scheme terms but also to use a closed ML type τ as a behavioral specification for a Scheme term — **Nat** means the term must evaluate to a number, **Nat** → **Nat** means the term must evaluate to a function that returns a number under the promise that the context

$$\mathsf{Rel} \quad = \quad \{\mathsf{R} \mid \forall (k, \mathsf{v}_1, \mathsf{v}_2) \in \mathsf{R}.\ \forall j \leq k.\ (j, \mathsf{v}_1, \mathsf{v}_2) \in \mathsf{R}\}$$

$$\sigma \vdash \mathsf{e}_1 \leq^k \mathsf{e}_2 : \tau \quad \stackrel{\text{def}}{=} \quad \forall j < k.\ (\mathsf{e}_1 \hookrightarrow^j \textbf{Error}: \mathrm{s} \ \Rightarrow\ \mathsf{e}_2 \hookrightarrow^* \textbf{Error}: \mathrm{s}) \text{ and } (\forall \mathsf{v}_1.\ \mathsf{e}_1 \hookrightarrow^j \mathsf{v}_1 \ \Rightarrow\ \exists \mathsf{v}_2.\ \mathsf{e}_2 \hookrightarrow^* \mathsf{v}_2 \text{ and } \sigma \vdash \mathsf{v}_1 \leqq^{k-j} \mathsf{v}_2 : \tau)$$

$$\sigma \vdash \mathsf{v}_1 \leqq^k \mathsf{v}_2 : \alpha \quad \stackrel{\text{def}}{=} \quad (k, \mathsf{v}_1, \mathsf{v}_2) \in \sigma(\alpha)$$

$$\sigma \vdash \overline{n} \leqq^k \overline{n} : \textbf{Nat} \quad \textit{(unconditionally)}$$

$$\sigma \vdash \lambda \mathsf{x}.\mathsf{e}_1 \leqq^k \lambda \mathsf{x}.\mathsf{e}_2 : \tau_1 \to \tau_2 \quad \stackrel{\text{def}}{=} \quad \forall j < k.\ \forall \mathsf{v}_1, \mathsf{v}_2.\ \sigma \vdash \mathsf{v}_1 \leqq^j \mathsf{v}_2 : \tau_1 \ \Rightarrow\ \sigma \vdash \mathsf{e}_1[\mathsf{v}_1/\mathsf{x}] \leq^j \mathsf{e}_2[\mathsf{v}_2/\mathsf{x}] : \tau_2$$

$$\sigma \vdash [\mathsf{v}_1, \cdots, \mathsf{v}_n] \leqq^k [\mathsf{v}'_1, \cdots, \mathsf{v}'_n] : \tau^* \quad \stackrel{\text{def}}{=} \quad \forall j < k.\ \forall i \in 1 \ldots n.\ \sigma \vdash \mathsf{v}_i \leqq^j \mathsf{v}'_i : \tau$$

$$\sigma \vdash \mathsf{v}_1 \leqq^k \mathsf{v}_2 : \forall \alpha.\tau \quad \stackrel{\text{def}}{=} \quad \forall j < k.\ \forall \mathsf{R} \in \mathsf{Rel}.\ \sigma, \alpha : \mathsf{R} \vdash \mathsf{v}_1 \leqq^j \mathsf{v}_2 : \tau$$

Fig. 5. Behavioral specification for polymorphic contracts

will always provide it a number, and so on. We can implement this using boundaries with the program fragment $\mathsf{e}^\tau = \mathit{SM}^\tau\ (^\tau\mathit{MS}\ \mathsf{e})$.

It is easy to check that such terms are always well-typed as long as e itself is well-typed. Therefore, since we have defined a contract as just a particular usage pattern for boundaries, we have by virtue of theorem 2 that every contracted term corresponds to itself, so intuitively every contracted term of polymorphic type should behave parametrically. However, the logical relation we defined in the previous section is not particularly convenient for proving facts about contracted Scheme terms, so instead we give another relation in figure 5 that we think of as the "contracted-Scheme-terms" relation, which gives criteria for two Scheme terms being related at an ML type (which we now interpret as a behavioral contract). Here σ is an *untyped* mapping from type variables α to downward-closed relations R on Scheme values: that is, $\sigma = (\alpha_1 \mapsto \mathsf{R}_1, \cdots, \alpha_n \mapsto \mathsf{R}_n)$ where each $\mathsf{R}_i \in \mathsf{Rel}$ (see figure 5).

Our goal is to prove that closed, contracted terms are related to themselves under this relation. Proving this directly is intractable, but we can prove it by showing that boundaries "reflect their relations"; *i.e.* that if $\delta \vdash \mathbf{e}_1 \lesssim_M^k \mathbf{e}_2 : \tau$ then for some appropriate σ we have that $\sigma \vdash (\mathit{SM}^\tau\ \mathbf{e}_1) \leq^k (\mathit{SM}^\tau\ \mathbf{e}_2) : \tau$ and vice versa; this is the claim we show in lemma 2 (bridge lemma 2) below, and the result we want is an easy corollary when combined with theorem 2. Before we can precisely state the claim, though, we need some machinery for specifying what relationship between δ and σ we want to hold.

Definition 2 (hybrid environments). *An hybrid environment ϕ is a partial map from type variables to tuples of the form (S, R) or $(M, \tau_1, \tau_2, \boldsymbol{R})$.*

The intuition is that a hybrid environment is a tagged union of a Scheme environment σ (each element of which is tagged with S) and an ML environment δ (each element of which is tagged with M). Given such a hybrid environment, one can mechanically derive both a Scheme and an ML representation of it by keeping native elements as-is and wrapping foreign elements in the appropriate boundary:

Definition 3 (Scheme and ML projections of hybrid environments). *For a hybrid environment ϕ, if $\phi(\alpha) = (S, R)$, then:*

$$\sigma_\phi(\alpha) \stackrel{\text{def}}{=} R$$
$$\delta_\phi(\alpha) \stackrel{\text{def}}{=} (\boldsymbol{L}, \boldsymbol{L}, \{(k, ({}^{\boldsymbol{L}}MS\ v_1), ({}^{\boldsymbol{L}}MS\ v_2)) \mid (k, v_1, v_2) \in R\})$$

If $\phi(\alpha) = (M, \tau_1, \tau_2, \boldsymbol{R})$*, then:*

$$\sigma_\phi(\alpha) \stackrel{\text{def}}{=} \{(k, (SM^{\langle\alpha;\tau_1\rangle}\ v_1), (SM^{\langle\alpha;\tau_2\rangle}\ v_2)) \mid (k, v_1, v_2) \in \boldsymbol{R}\}$$
$$\delta_\phi(\alpha) \stackrel{\text{def}}{=} (\tau_1, \tau_2, \boldsymbol{R})$$

We say that $\Delta \vdash \phi$ if for all $\alpha \in \Delta$, $\phi(\alpha)$ is defined, and if $\phi(\alpha) = (\mathrm{S}, \mathsf{R})$ then $\mathsf{R} \in \mathsf{Rel}$, and if $\phi(\alpha) = (\mathrm{M}, \tau_1, \tau_2, \mathbf{R})$ then $\mathbf{R} \in \mathbf{Rel}_{\tau_1,\tau_2}$. We also define operations $c_1(\cdot,\cdot)$ and $c_2(\cdot,\cdot)$ (analogous to $\mathbf{sl}(\cdot,\cdot)$ defined earlier) from hybrid environments ϕ and types τ to conversion schemes κ:

Definition 4 (closing with respect to a hybrid environment). *For* $i \in \{1,2\}$*:*

$$c_i(\phi,\alpha) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{L} & \text{if } \phi(\alpha) = (S, R) \\ \langle\alpha;\tau_i\rangle & \text{if } \phi(\alpha) = (M, \tau_1, \tau_2, \boldsymbol{R}) \\ \alpha & \text{otherwise} \end{cases}$$

$$\begin{aligned} c_i(\phi, \mathbf{Nat}) &\stackrel{\text{def}}{=} \mathbf{Nat} \\ c_i(\phi, \boldsymbol{L}) &\stackrel{\text{def}}{=} \boldsymbol{L} \\ c_i(\phi, \tau_1 \to \tau_2) &\stackrel{\text{def}}{=} c_i(\phi,\tau_1) \to c_i(\phi,\tau_2) \\ c_i(\phi, \forall\alpha.\tau') &\stackrel{\text{def}}{=} \forall\alpha.c_i(\phi,\tau') \\ c_i(\phi, \tau^*) &\stackrel{\text{def}}{=} c_i(\phi,\tau)^* \end{aligned}$$

The interesting part of the definition is its action on type variables. Variables that ϕ maps to Scheme relations are converted to type $\mathbf{L}$, since when Scheme uses a polymorphic value in ML its free type variables are instantiated as $\mathbf{L}$. Similarly, variables that ϕ maps to ML relations are instantiated as seals because when ML uses a Scheme value as though it were polymorphic it uses dynamic seals to protect parametricity.

Now we can show that contracts respect the relation in figure 5 via a bridge lemma.

Lemma 2 (bridge lemma 2). *For all* $k \geq 0$*, type environments* Δ*, hybrid environments* ϕ *such that* $\Delta \vdash \phi$*,* τ *such that* $\Delta \vdash \tau$*, and for all terms* e_1*,* e_2*,* $\boldsymbol{e}_1$*,* $\boldsymbol{e}_2$*:*

1. *If* $\delta_\phi \vdash \boldsymbol{e}_1 \lesssim^k_M \boldsymbol{e}_2 : \tau$ *then* $\sigma_\phi \vdash (SM^{c_1(\phi,\tau)}\ \boldsymbol{e}_1) \leq^k (SM^{c_2(\phi,\tau)}\ \boldsymbol{e}_2) : \tau$*.*
2. *If* $\sigma_\phi \vdash \boldsymbol{e}_1 \leq^k \boldsymbol{e}_2 : \tau$ *then* $\delta_\phi \vdash {}^{c_1(\phi,\tau)}MS\ \boldsymbol{e}_1 \lesssim^k_M ({}^{c_2(\phi,\tau)}MS\ \boldsymbol{e}_2) : \tau$*.*

Proof. Induction on τ. All cases are easy applications of the induction hypotheses.

Theorem 3. *For any seal-free term* e *such that* $\vdash_S e : \boldsymbol{TST}$ *and for any closed type* τ*, we have that for all* $k \geq 0$*,* $\vdash e^\tau \leq^k e^\tau : \tau$*.*

Proof. By theorem 2, for all $k \geq 0$, $\vdash ({}^\tau MS\ e) \lesssim^k_M ({}^\tau MS\ e) : \tau$. Thus, by lemma 2, we have that for all $k \geq 0$, $\vdash (SM^\tau\ ({}^\tau MS\ e)) \leq^k (SM^\tau\ ({}^\tau MS\ e)) : \tau$.

Definition 5 (relational equality). *We write* $\sigma \vdash e_1 = e_2 : \tau$ *if for all* $k \geq 0$*,* $\sigma \vdash e_1 \leq^k e_2 : \tau$ *and* $\sigma \vdash e_2 \leq^k e_1 : \tau$*.*

Corollary 1. *For any seal-free term* e *such that* $\vdash_S \mathsf{e} : \mathbf{TST}$ *and for any closed type* τ*, we have that* $\vdash e^{\tau} = e^{\tau} : \tau$.

4.1 Dynamic Sealing Replaces Boundaries

The contract system of the previous section is a multi-language system, but just barely, since the only part of ML we make any use of is its boundary form to get back into Scheme. In this section we restrict our attention to Scheme plus boundaries used only for the purpose of implementing contracts, and we show an alternate implementation of contracts that uses dynamic sealing. Rather than the concrete implementation of dynamic seals we gave in the introduction, we opt to use (a slight restriction of) the more abstract constructs taken from Sumii and Pierce's $\lambda_{\mathtt{seal}}$ language [5]. Specifically, we use the following extension to our Scheme model:

$$
\begin{array}{lll}
\mathsf{e} & = \cdots \mid \nu \mathsf{sx}.\ \mathsf{e} \mid \{\mathsf{e}\}_{\mathsf{se}} \mid (\mathbf{let}\ \{\mathsf{x}\}_{\mathsf{se}} = \mathsf{e}\ \mathbf{in}\ \mathsf{e}) & \mathscr{E}[\nu \mathsf{sx}.\ \mathsf{e}]_S \mapsto \mathscr{E}[\mathsf{e}[\mathsf{sv}/\mathsf{sx}]] \\
\mathsf{v} & = \cdots \mid \{\mathsf{v}\}_{\mathsf{sv}} & \quad \text{where } \mathsf{sv} \text{ fresh} \\
\mathsf{se} & = \mathsf{sx} \mid \mathsf{sv} & \mathscr{E}[(\mathbf{let}\ \{\mathsf{x}\}_{\mathsf{sv}_1} = \{\mathsf{v}\}_{\mathsf{sv}_1}\ \mathbf{in}\ \mathsf{e})]_S \mapsto \mathscr{E}[\mathsf{e}_1[\mathsf{v}/\mathsf{x}]] \\
\mathsf{sx} & = [\text{variables distinct from } \mathsf{x}] & \mathscr{E}[(\mathbf{let}\ \{\mathsf{x}\}_{\mathsf{sv}_1} = \mathsf{v}\ \mathbf{in}\ \mathsf{e})]_S \mapsto \mathbf{Error}\text{: bad value} \\
\mathsf{sv} & = [\text{unspecified, unique brands}] & \quad \text{where } \mathsf{v} \neq \{\mathsf{v}'\}_{\mathsf{sv}_1} \text{ for any } \mathsf{v}' \\
\mathsf{E} & = \cdots \mid \{\mathsf{E}\}_{\mathsf{sv}} \mid (\mathbf{let}\ \{\mathsf{x}\}_{\mathsf{sv}} = \mathsf{E}\ \mathbf{in}\ \mathsf{e}) &
\end{array}
$$

We introduce a new set of seal variables sx that stand for seals (elements of sv) that will be computed at runtime. They are bound by $\nu \mathsf{sx}.\ \mathsf{e}$, which evaluates its body (e) with sx bound to a freshly-generated sv. Two operations make use of these seals. The first, $\{\mathsf{e}\}_{\mathsf{se}}$, evaluates e to a value and then itself becomes an opaque value sealed with the key to which se evaluates. The second, $(\mathbf{let}\ \{\mathsf{x}\}_{\mathsf{se}} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2)$, evaluates e_1 to a value; if that value is an opaque value sealed with the seal to which se evaluates, then the entire unsealing expression evaluates to e_2 with x bound to the value that was sealed, otherwise the expression signals an error.[1]

Using these additional constructs we can demonstrate that a translation essentially the same as the one given by Sumii and Pierce [5, figure 4] does in fact generate parametrically polymorphic type abstractions. Their translation essentially attaches a higher-order contract [8] τ to an expression of type τ (though they do not point this out). It extends Findler and Felleisen's notion of contracts, which does not include polymorphic types, by adding an environment ρ that maps a type variable to a tuple consisting of a seal and a symbol indicating the party (either $+$ or $-$ in Sumii and Pierce) that has the power to instantiate that type variable, and translating uses of type variable α in a contract to an appropriate seal or unseal based on the value of $\rho(\alpha)$. We define it as follows: when p and q are each parties ($+$ or $-$) and $p \neq q$,

[1] This presentation is a simplification of $\lambda_{\mathtt{seal}}$ in two ways. First, in $\lambda_{\mathtt{seal}}$ the key position for a sealed value or for an unseal statement may be an arbitrary expression, whereas here we syntactically restrict expressions that appear in those positions to be either seal variables or seal values. Second, in $\lambda_{\mathtt{seal}}$ an unseal expression has an "else" clause that allows the program to continue even if an unsealing operation fails; we do not allow those clauses.

$$
\begin{array}{lcl}
E^{p,q}_{\mathbf{Nat}}(\rho,\mathsf{e}) & = & (+\ \mathsf{e}\ 0) \\
E^{p,q}_{\tau^*}(\rho,\mathsf{e}) & = & (\mathbf{let}\ ((v\ e))\ (\mathbf{if0}\ (\mathsf{nil?}\ v) \\
 & & \qquad \mathsf{nil} \\
 & & \qquad (\mathbf{if0}\ (\mathsf{pair?}\ v) \\
 & & \qquad\quad (\mathsf{cons}\ E^{p,q}_{\tau}(\rho,(\mathsf{fst}\ \mathsf{v}))\ E^{p,q}_{\tau^*}(\rho,(\mathsf{rst}\ \mathsf{v}))) \\
 & & \qquad\quad (\mathsf{wrong}\ \texttt{"Non-list"}))))) \\
E^{p,q}_{\tau_1\to\tau_2}(\rho,\mathsf{e}) & = & (\mathbf{let}\ ((v\ e))\ (\mathbf{if0}\ (\mathsf{proc?}\ v) \\
 & & \qquad (\lambda\ x.\ E^{p,q}_{\tau_2}(\rho,(\mathsf{v}\ E^{q,p}_{\tau_1}(\rho,\mathsf{x})))) \\
 & & \qquad (\mathsf{wrong}\ \texttt{"Non-proc"}))) \\
E^{p,q}_{\forall\alpha.\tau'}(\rho,\mathsf{e}) & = & v\mathsf{sx}.\ E^{p,q}_{\mathsf{e}}(\rho,\alpha\mapsto(\mathsf{sx},q),\mathsf{e}) \\
E^{p,q}_{\alpha}(\rho,\alpha\mapsto(\mathsf{sx},p),\mathsf{e}) & = & \{\mathsf{e}\}_{\mathsf{sx}} \\
E^{p,q}_{\alpha}(\rho,\alpha\mapsto(\mathsf{sx},q),\mathsf{e}) & = & (\mathbf{let}\ \{\mathsf{x}\}_{\mathsf{sx}} = \mathsf{e}\ \mathbf{in}\ \mathsf{x})
\end{array}
$$

The differences between our translation and Sumii and Pierce's are as follows. First, we have mapped everything into our notation and adapted to our types (we omit booleans, tuples, and existential types and add numbers and lists). Second, our translations apply to arbitrary expressions rather than just variables. Third, because we are concerned with the expression violating parametricity as well as the context, we have to seal values provided by the context as well as by the expression, and our decision of whether to seal or unseal at a type variable is based on whether the party that instantiated the type variable is providing a value with that contract or expecting one. Fourth, we modify the result of $\forall\alpha.\tau$ so that it does not require application to a dummy value. (The reason we do this bears explanation. There are two components to a type abstraction in System F — abstracting over an interpretation of a variable and suspending a computation. Sumii and Pierce's system achieves the former by generating a fresh seal, and the latter by wrapping the computation in a lambda abstraction. In our variant, $\forall\alpha.\tau$ contracts still abstract over a free contract variable's interpretation, but they do not suspend computation; for that reason we retain fresh seal generation but eliminate the wrapper function.)

Definition 6 (boundary replacement). *$\mathscr{R}[\mathsf{e}]$ is defined as follows:*

$$\mathscr{R}[\mathsf{e}^{\tau}] = E^{+,-}_{\tau}(\bullet,\mathscr{R}[\mathsf{e}]) \qquad \mathscr{R}[(\mathsf{e}_1\ \mathsf{e}_2)] = (\mathscr{R}[\mathsf{e}_1]\ \mathscr{R}[\mathsf{e}_2]) \qquad \ldots$$

Theorem 4 (boundary replacement preserves termination). *If $;\vdash_S \mathsf{e} : \mathbf{TST}$, then $\mathsf{e}\mapsto^*\mathsf{v}_1 \Leftrightarrow \mathscr{R}[\mathsf{e}]\mapsto^*\mathsf{v}_2$, where $\mathsf{v}_1 = \overline{n} \Leftrightarrow \mathsf{v}_2 = \overline{n}$.*

This claim is a special case of a more general theorem that requires us to consider open contracts. The term $\mathsf{v}^{\forall\alpha.\alpha\to\alpha}$ where v is a procedure value reduces as follows:

$$
\begin{array}{rcl}
\mathsf{v}^{\forall\alpha.\alpha\to\alpha} & = & (SM^{\forall\alpha.\alpha\to\alpha}({}^{\forall\alpha.\alpha\to\alpha}MS\ \mathsf{v})) \\
 & \mapsto^3 & (SM^{\mathbf{L}\to\mathbf{L}}({}^{\langle\alpha;\mathbf{L}\rangle\to\langle\alpha;\mathbf{L}\rangle}MS\ \mathsf{v})) \\
 & \mapsto^2 & \lambda\mathsf{x}.(SM^{\mathbf{L}}\ ((\lambda\mathsf{y}:\mathbf{L}.\ ({}^{\langle\alpha;\mathbf{L}\rangle}MS\ (\mathsf{v}\ (SM^{\langle\alpha;\mathbf{L}\rangle}\ \mathsf{y}))))\ ({}^{\mathbf{L}}MS\ \mathsf{x}))) \\
 & = & \lambda\mathsf{x}.(SM^{\mathbf{L}}\ ({}^{\langle\alpha;\mathbf{L}\rangle}MS\ (\mathsf{v}\ (SM^{\langle\alpha;\mathbf{L}\rangle}\ ({}^{\mathbf{L}}MS\ \mathsf{x})))))
\end{array}
$$

Notice that the two closed occurrences of α in the original contracts become two different configurations of boundaries when they appear open in the final procedure. These correspond to the fact that negative and positive occurrences of a type variable with respect to its binder behave differently. Negative occurrences, of the form $(SM^{\langle\alpha;\mathbf{L}\rangle}\ ({}^{\mathbf{L}}MS\ldots))$, act as dynamic seals on their bodies. Positive occurrences, of the form $(SM^{\mathbf{L}}\ ({}^{\langle\alpha;\mathbf{L}\rangle}MS\ldots))$,

dynamically unseal the values their bodies produce. So, we write open contract variables as $\alpha-$ (for negative occurrences) and $\alpha+$ (for positive occurrences).

Now we are prepared to define another logical relation, this time between contracted Scheme terms and $\lambda_{\texttt{seal}}$ terms. We define it as follows, where p owns the given expressions, q is the other party, and ρ maps type variables to seals and owners:

$$
\begin{array}{lcl}
p;q;\rho \vdash \mathsf{e}_1 =^k_{seal} \mathsf{e}_2 & \stackrel{\text{def}}{=} & \forall j < k.\ (\mathsf{e}_1 \mapsto^j \textbf{Error}\colon \mathrm{s} \Rightarrow \mathsf{e}_2 \mapsto^* \textbf{Error}\colon \mathrm{s}) \text{ and} \\
& & \quad (\forall \mathsf{v}_1.\ \mathsf{e}_1 \mapsto^j \mathsf{v}_1 \Rightarrow \exists \mathsf{v}_2.\ \mathsf{e}_2 \mapsto^* \mathsf{v}_2 \text{ and } p;q;\rho \vdash \mathsf{v}_1 =^{k-j}_{seal} \mathsf{v}_2) \\
& & \forall j < k.\ (\mathsf{e}_2 \mapsto^j \textbf{Error}\colon \mathrm{s} \Rightarrow \mathsf{e}_1 \mapsto^* \textbf{Error}\colon \mathrm{s}) \text{ and} \\
& & \quad (\forall \mathsf{v}_1.\ \mathsf{e}_2 \mapsto^j \mathsf{v}_2 \Rightarrow \exists \mathsf{v}_2.\ \mathsf{e}_1 \mapsto^* \mathsf{v}_1 \text{ and } p;q;\rho \vdash \mathsf{v}_1 =^{k-j}_{seal} \mathsf{v}_2) \\
p;q;\rho \vdash \mathsf{v}_1{}^{\alpha-} =^k_{seal} \{\mathsf{v}_2\}_{\mathsf{sv}} & \stackrel{\text{def}}{=} & \rho(\alpha) = (\mathsf{sx}, q) \text{ and } \forall j < k.\ p;q;\rho \vdash \mathsf{v}_1 =^j_{seal} \mathsf{v}_2 \\
& \vdots & \\
p;q;\rho \vdash (\lambda \mathsf{x}.\mathsf{e}_1) =^k_{seal} (\lambda \mathsf{x}.\mathsf{e}_2) & \stackrel{\text{def}}{=} & \forall j < k, \mathsf{v}_1, \mathsf{v}_2.\ \ q;p;\rho \vdash \mathsf{v}_1 =^j_{seal} \mathsf{v}_2 \Rightarrow \\
& & \qquad\qquad p;q;\rho \vdash \mathsf{e}_1[\mathsf{v}_1/\mathsf{x}] =^j_{seal} \mathsf{e}_2[\mathsf{v}_2/\mathsf{x}]
\end{array}
$$

The rest of the cases are defined as in the Scheme relation of figure 4. An important subtlety above is that two sealed terms are related only if they are locked with a seal owned by the *other* party, and that the arguments to functions are owned by the party that does *not* own the function. The former point allows us to establish this lemma, after which we can build a new bridge lemma and then prove the theorem of interest:

Lemma 3. *If $p;q;\rho,\alpha : (\mathsf{sx}, p) \vdash \mathsf{e}_1 =^k_{seal} \mathsf{e}_2$ (and α not free in e_1), then $p;q;\rho \vdash \mathsf{e}_1 =^k_{seal} \mathsf{e}_2$. Similarly if $p;q;\rho \vdash \mathsf{e}_1 =^k_{seal} \mathsf{e}_2$, then $p;q;\rho,\alpha : (\mathsf{sx}, p) \vdash \mathsf{e}_1 =^k_{seal} \mathsf{e}_2$.*

Proof. We prove both claims simultaneously by induction on k.

Lemma 4. *For any two terms e_1 and e_2 such that e_1's open type variables (and their ownership information) occur in ρ, and so do the open type variables in τ, then if $(\forall k.p;q;\rho \vdash \mathsf{e}_1 =^k_{seal} \mathsf{e}_2)$ then $(\forall k.p;q;\rho \vdash \mathsf{e}_1{}^{\tau} =^k_{seal} E^{p,q}_{\tau}(\rho, \mathsf{e}_2))$.*

Proof. By induction on τ. The $\forall\alpha.\tau$ case requires the preceding lemma.

Theorem 5. *If $\rho \vdash \gamma_1 =_{seal} \gamma_2 : \Gamma$, e's open type variables occur in ρ, $\Delta;\Gamma \vdash_S \mathsf{e} : \textbf{\textit{TST}}$, and e only uses boundaries as contracts, then $\forall k.p;q;\rho \vdash \gamma_1(\mathsf{e}) =^k_{seal} \gamma_2(\mathscr{R}[\mathsf{e}])$.*

Proof. Induction on the derivation $\Delta;\Gamma \vdash_S \mathsf{e} : \textbf{TST}$. Contract cases appeal to lemma 4.

This theorem has two consequences: first, contracts as we have defined them in this paper can be implemented by a variant on Sumii and Pierce's translation, and thus due to our earlier development their translation preserves parametricity; and second, since Sumii and Pierce's translation is itself a variant on Findler-and-Felleisen-style contracts, our boundary-based contracts are actually contracts in that sense.

Finally, notice that if we choose $\mathscr{E} = \mathsf{E}$ then there is no trace of ML left in the language we are considering; it is pure Scheme with contracts. But, strangely, the contract system's parametricity theorem relies on the fact that parametricity holds in ML.

5 Related Work and Conclusions

We have mentioned Sumii and Pierce's investigation of dynamic sealing [11, 5] many times in this paper. Sumii and Pierce also investigate logical relations for encryption

[12], which is probably the most technically similar paper in their line of research to the present work. In that work, they develop a logical relation that tracks secret keys as a proof technique for establishing the equivalence of programs that use encryption to hide information. One can think of our development as a refinement of their relation that allows Turing-complete "attackers" (which in particular may not terminate) and that clarifies the fundamental connection between parametricity and dynamic sealing.

Zdancewic, Grossman, and Morrisett's notion of *principals* [13, 14] and their associated proof technique are also related. Compared to their work, the present proof technique establishes a much stronger property, but it is comparatively more difficult to scale to more sophisticated programming language features such as state or advanced control features. Rossberg [15,16] discusses the idea of preserving abstraction safety by the use of dynamically-generated types that are very similar to our $\langle \alpha; \tau \rangle$ sealed conversion schemes. The property we have proven here is much stronger than the abstraction properties established by Rossberg; however, his analysis considers a more complicated type system than we do. It is certainly worth investigating how well the multi-language technique presented here maps into Rossberg's setting, but we have not done so yet.

The thrust of this paper has been to demonstrate that the parametricity property of System F is preserved under a multi-language embedding with Scheme, provided we protect all values that arise from terms that had quantified types in the original program using dynamic seals. We think this fact is in itself interesting, and has the interesting consequence that polymorphic contracts are also parametric in a meaningful sense, in fact strong enough that we can derive "free theorems" about contracted Scheme terms (see the technical report [9] for examples). But it also suggests something broader. Rather than just knowing that parametricity continues to hold in System F after the extension, we would like the stronger property that the extension does not weaken System F's contextual equivalence relation at all; in other words to design an embedding such that $e_1 \simeq_{ctxt} e_2$ when considering only contexts without boundaries implies that $e_1 \simeq_{ctxt} e_2$ in contexts with boundaries. This may be a useful way to approach the full-abstraction question raised by Sumii and Pierce.

References

1. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress, pp. 513–523 (1983)
2. Wadler, P.: Theorems for free! In: Functional Programming Languages and Computer Architecture (FPCA), pp. 347–359 (1989)
3. Morris Jr., J.H.: Types are not sets. In: POPL (1973)
4. Flatt, M.: PLT MzScheme: Language manual. Technical Report TR97-280, Rice University (1997), `http://www.plt-scheme.org/software/mzscheme/`
5. Sumii, E., Pierce, B.: A bisimulation for dynamic sealing. In: POPL (2004)
6. Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. In: POPL (2007), Extended version: University of Chicago Technical Report TR-2007-8, under review
7. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: ESOP. pp. 69–83 (2006), Extended version: Harvard University Technical Report TR-01-06, `http://ttic.uchicago.edu/~amal/papers/lr-recquant-techrpt.pdf`
8. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ICFP (2002)

9. Matthews, J., Ahmed, A.: Parametric polymorphism through run-time sealing, or, theorems for low, low prices! (extended version). Technical Report TR-2008-01, University of Chicago (2008)
10. Findler, R.B., Blume, M.: Contracts as pairs of projections. In: FLOPS (2006)
11. Pierce, B., Sumii, E.: Relating cryptography and polymorphism. Unpublished manuscript (2000)
12. Sumii, E., Pierce, B.: Logical relations for encryption. Journal of Computer Security (JSC) 11(4), 521–554 (2003)
13. Zdancewic, S., Grossman, D., Morrisett, G.: Principals in programming languages. In: ICFP (1999)
14. Grossman, D., Morrisett, G., Zdancewic, S.: Syntactic type abstraction. ACM Transactions on Programming Languages and Systems 22, 1037–1080 (2000)
15. Rossberg, A.: Generativity and dynamic opacity for abstract types. In: Miller, D. (ed.) PADL, Uppsala, Sweden, ACM Press, New York (2003), Extended version: `http://www.ps.uni-sb.de/Papers/generativity-extended.html`
16. Rossberg, A.: Typed Open Programming – A higher-order, typed approach to dynamic modularity and distribution. Phd thesis, Universität des Saarlandes, Saarbrücken, Germany. Preliminary version (2007)