A Simple Proof Technique for Certain Parametricity Results*

Karl Crary

Carnegie Mellon University

Abstract

Many properties of parametric, polymorphic functions can be determined simply by inspection of their types. Such results are usually proven using Reynolds's parametricity theorem. However, Reynolds's theorem can be difficult to show in some settings, particularly ones involving computational effects. I present an alternative technique for proving some parametricity results. This technique is considerably simpler and easily generalizes to effectful settings. It works by instantiating polymorphic functions with singleton types that fully specify the behavior of the functions. Using this technique, I show that callers' stacks are protected from corruption during function calls in Typed Assembly Language programs.

1 Introduction

A polymorphic function can be termed *parametric* [11] if it always uses the same algorithm, regardless of the type at which it is applied. In particular, a parametric, polymorphic function can neither branch on nor otherwise analyze its type argument. In many type theories (such as the polymorphic lambda calculus) all polymorphic functions must be parametric.

It has long been recognized [10, 12, 8, 3] that in type theories in which all polymorphism is parametric, many properties of polymorphic functions can be determined solely by inspection of their types. For instance, in the polymorphic lambda calculus, any function having the type $\forall\alpha.\,\alpha\to\alpha$ must be the identity function. In an extension of the polymorphic lambda calculus with a uniquely inhabited `unit` type,[1] a simple intuitive argument for this fact can be made by appealing to parametricity:

> We write the application of a polymorphic function f to a type argument τ as $f[\tau]$. Suppose f has type $\forall\alpha.\,\alpha\to\alpha$. Then $f[\texttt{unit}]\texttt{()}$ has type `unit` and is therefore equal to `()`. Consequently, $f[\texttt{unit}]$ is either the constant `()` function or the identity function. By parametricity, at any type f must be either the constant `()` function or the identity function. The former is ill-typed, so f must be the identity function.

Results such as this can be of practical importance. For example, Typed Assembly Language with stacks (STAL) [6, 5], contains a notion of stack types, and allows polymorphism over stack types as well as ordinary types. In STAL, a function returning an integer would be seen to have type $\forall\rho.\{\texttt{sp}:\rho,\texttt{ra}:\{\texttt{sp}{:}\rho,\texttt{r1}{:}\texttt{int}\}\}$. This type is read, "For any stack type ρ, the function may be called so long as the stack has type ρ, and the register `ra` contains a return address that can be called when the stack has type ρ and `r1` contains an integer." The type indicates that the function returns the stack with the same type (ρ) as it received it. This property is certainly necessary for proper functioning of the caller, but a stronger property is also desirable: that the callee returns not only a stack with the same shape, but the *exact* same stack, without meddling with it in any way.

This property is exactly the sort of *abstraction* property that Reynolds envisioned in his seminal paper on parametricity [10]. Polymorphism in STAL is parametric, so we can give an intuitive argument similar to the previous one:

> We write the empty stack and its type as `nil`. Suppose g has type $\forall\rho.\{\texttt{sp}:\rho,\texttt{ra}:\{\texttt{sp}{:}\rho,\texttt{r1}{:}\texttt{int}\}\}$. Suppose further that the

*This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

To appear in 1999 International Conference on Functional Programming, Paris, France, September 1999.

[1] Without a primitive `unit` type, although the result is still true, this simple argument is circular, as it requires identifying some type with a unique element, for which the usual candidate is the type in question, $\forall\alpha.\,\alpha\to\alpha$.

> stack is empty and h is a continuation function accepting an empty stack and an integer in `r1`. Then g[`nil`] can be called after moving a pointer to h into `ra`. If and when g transfers control back to the continuation h, the stack must be empty. Consequently, g[`nil`] transfers control back to its return address[2] with either an empty stack or the same stack. By parametricity, at any stack type, g returns with either an empty stack or the same stack. The former possibility is barred by the type, so g must always return the same stack.

These sorts of results are usually formalized using Reynolds's parametricity theorem [10] (he called it the abstraction theorem). The parametricity theorem states (in part) that when given related arguments, a function returns related results. The identity function result is then shown, for any prospective argument, by choosing the relation to be the one-point relation that only relates the prospective argument to itself. That relation obviously relates the argument to itself, so it also relates the result to itself. Hence, by choice of the relation, the argument and result are equal.

This sort of result only scratches the surface of the power of the parametricity theorem. The theorem is capable of concluding far-ranging results about the behavior of expressions from no information other than their types [12]. This is a pleasant situation, provided that one is working in a context for which the theorem has been proved. There has been considerable work in the direction of extending the parametricity theorem past the polymorphic lambda calculus into more expressive type theories [12, 1, 4, 9, 2].

Unfortunately, the parametricity theorem has been prohibitively difficult to prove for some type theories, particularly those with computational effects.[3] Consequently, many appealing parametricity conjectures remain unproven. For instance, STAL certainly supports computational effects, so the intuitive argument for stack preservation has heretofore been unsubstantiated by formal proof.

An Alternative In this paper I present an alternative proof technique for certain parametricity results. This technique is limited in scope; it suffices for considerably fewer results than the parametricity theorem. However, for those results to which it does apply, it has the advantage that it is very simple and easily extends to richer type systems, including those with computational effects.

The essence of the technique is to show that the type theory in question may be soundly extended with a notion of singleton types (types containing only a single particular term). This extension is done only by adding new rules, so any judgements derivable in the original type system are derivable in the extended system. In the extended system we may instantiate polymorphic functions at singleton types, and any result value that consequently belongs to the singleton type has its identity determined thereby. Finally, since the function is parametric, we can conclude that the same value is obtained when the function is instantiated at an ordinary type. In essence, this argument specializes the parametricity theorem to a one-point relation, where it is much easier to prove.

Informally, we use this technique to show the identity function result as follows:

> We write singleton types as $\mathfrak{S}(v{:}\tau)$. Suppose f has type $\forall\alpha.\,\alpha\to\alpha$ and v has type τ, and suppose we wish to show that $f[\tau]v = v$. In the extended system, $f[\mathfrak{S}(v{:}\tau)]$ has type $\mathfrak{S}(v{:}\tau)\to\mathfrak{S}(v{:}\tau)$ and v has type $\mathfrak{S}(v{:}\tau)$, so $f[\mathfrak{S}(v{:}\tau)]v$ has type $\mathfrak{S}(v{:}\tau)$. Therefore $f[\mathfrak{S}(v)]v = v$. Since f is parametric, $f[\tau]v$ is also equal to v.

In the remainder of this paper, I formalize this technique for the polymorphic lambda calculus and apply it to some examples. Then, to show how it generalizes to richer type systems, I use it to show the stack preservation property in STAL.

2 The Proof Technique

The treatment of the polymorphic lambda calculus I use here is standard. For ease of reference, its syntax, and its static and dynamic semantics are given in Figure 1. Evaluation is call-by-value and is given by a small-step relation $e \mapsto e'$. We write the capture-avoiding substitution of E for X in E' as $E'[E/X]$. As usual, alpha-equivalent expressions (written $E \equiv E'$) are considered identical.

The first step in the proof technique is to augment the original language (in this case the polymorphic lambda calculus) with singleton types. We will perform this augmentation only by adding rules, so any judgement derivable in the original language will also be derivable in the augmented language. We will write judgements derivable in the original language with the symbol $\vdash$ and in the augmented language with the symbol $\vdash_{\mathfrak{s}}$. The rules governing the dynamic semantics will be entirely unchanged.

We write singleton types as $\mathfrak{S}(v{:}\tau)$. Two rules govern their usage:

$$\frac{\Delta,\ \vdash_{\mathfrak{s}} v : \tau}{\Delta,\ \vdash_{\mathfrak{s}} \mathfrak{S}(v{:}\tau)\ \text{type}} \qquad \frac{\Delta,\ \vdash_{\mathfrak{s}} v : \tau}{\Delta,\ \vdash_{\mathfrak{s}} v : \mathfrak{S}(v{:}\tau)}$$

There are several important remarks to make about this formulation of singleton types:

[2] This assumes that g has no other access to h and ignores possible variation in behavior resulting from side-effects. A more careful proof appears in Section 3.

[3] Although parametricity with effects has been problematic in lambda-calculus-like languages, it has been successfully used with computational effects in Algol-like languages [7].

Syntax

$$\begin{array}{llll} \textit{types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \\ \textit{terms} & e & ::= & x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau] \\ \textit{values} & v & ::= & x \mid \lambda x{:}\tau.e \mid \Lambda\alpha.e \\ \textit{contexts} & , & ::= & \epsilon \mid , , \alpha \mid , , x{:}\tau \end{array}$$

Static semantics

$$\frac{}{, \vdash \alpha \text{ type}}\ (\alpha \in ,) \qquad \frac{, \vdash \tau_1 \text{ type} \quad , \vdash \tau_2 \text{ type}}{, \vdash \tau_1 \to \tau_2 \text{ type}} \qquad \frac{, , \alpha \vdash \tau \text{ type}}{, \vdash \forall\alpha.\tau \text{ type}}\ (\alpha \notin ,)$$

$$\frac{}{, \vdash x : \tau}\ (x{:}\tau \in ,)$$

$$\frac{, \vdash \tau \text{ type} \quad , \vdash \tau' \text{ type} \quad , , x{:}\tau \vdash e : \tau'}{, \vdash \lambda x{:}\tau.e : \tau \to \tau'}\ (x \notin ,) \qquad \frac{, \vdash e_1 : \tau \to \tau' \quad , \vdash e_2 : \tau}{, \vdash e_1 e_2 : \tau'}$$

$$\frac{, , \alpha \vdash e : \tau}{, \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ (\alpha \notin ,) \qquad \frac{, \vdash e : \forall\alpha.\tau' \quad , \vdash \tau \text{ type}}{, \vdash e[\tau] : \tau'[\tau/\alpha]}$$

Dynamic semantics

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2} \qquad \frac{e_2 \mapsto e_2'}{v\, e_2 \mapsto v\, e_2'} \qquad \frac{}{(\lambda x{:}\tau.e) v \mapsto e[v/x]}$$

$$\frac{e \mapsto e'}{e[\tau] \mapsto e'[\tau]} \qquad \frac{}{(\Lambda\alpha.e)[\tau] \mapsto e[\tau/\alpha]}$$

Figure 1: The Polymorphic Lambda Calculus

1. This formulation is far short of a useful programming language with singleton types. In particular, no elimination rule for singleton types is provided. No such rule is required for the proofs we are interested in; instead, elimination will be performed externally by a canonical forms lemma.
2. The notion of equality inherent in this axiomatization of singleton types is very strong: the type $\mathfrak{S}(v{:}\tau)$ will contain a value v' only if v and v' are *identical*, and will only contain a term e only if $e \mapsto^* v$. This strong formulation is by no means essential to the proof technique (weaker notions such as observational equivalence would work), but it provides the strongest possible parametricity result.
3. In anticipation of our interest in extending this technique to effectful computation, singleton types are restricted to values, thereby avoiding the dubious notion of an effectful type.
4. In the augmented system, types can contain free occurrences of term variables, and we will find it necessary to substitute for those variables. Such substitutions are syntactically well-formed only for substitutends that are values. Since evaluation is call-by-value, this will not be problematic.

The main proof burden is to show that the addition of singleton types is a sound extension. Ordinarily this requires the proof of Subject Reduction and Progress lemmas [13], but for our purposes we only require Subject Reduction:

Lemma 1 (Subject Reduction) *If* $\vdash_{\mathfrak{s}} e : \tau$ *and* $e \mapsto e'$ *then* $\vdash_{\mathfrak{s}} e' : \tau$.

As usual, the Subject Reduction lemma is proven by induction on typing derivations, using substitution lemmas for types and values:

Lemma 2 (Type Substitution) *If* $, \vdash_{\mathfrak{s}} \tau$ type *then:*

1. *If* $, , \alpha \vdash_{\mathfrak{s}} \tau'$ type *then* $, \vdash_{\mathfrak{s}} \tau'[\tau/\alpha]$ type, *and*
2. *If* $, , \alpha \vdash_{\mathfrak{s}} e : \tau'$ *then* $, \vdash_{\mathfrak{s}} e[\tau/\alpha] : \tau'[\tau/\alpha]$.

Lemma 3 (Value Substitution) *If* $, \vdash_{\mathfrak{s}} v : \tau$ *then:*

1. *If* $, , x{:}\tau \vdash_{\mathfrak{s}} \tau'$ type *then* $, \vdash_{\mathfrak{s}} \tau'[v/x]$ type, *and*
2. *If* $, , x{:}\tau \vdash_{\mathfrak{s}} e : \tau'$ *then* $, \vdash_{\mathfrak{s}} e[v/x] : \tau'[v/x]$.

With these obligations fulfilled, we can now prove most of the identity function result: Suppose $\vdash f : \forall\alpha.\, \alpha \to \alpha$

and $\vdash v : \tau$. Then $\vdash_{\mathfrak{s}} f[\mathfrak{S}(v{:}\tau)]v : \mathfrak{S}(v{:}\tau)$. Suppose also that $f[\mathfrak{S}(v{:}\tau)]v \mapsto^* v'$. By Subject Reduction, $\vdash_{\mathfrak{s}} v' : \mathfrak{S}(v{:}\tau)$.

From $\vdash_{\mathfrak{s}} v' : \mathfrak{S}(v{:}\tau)$ we wish to deduce that $v \equiv v'$. This is apparent in an inspection of the typing rules, since the only rules that can assign a value a singleton type are the singleton introduction rule and the variable rule, and v' cannot be a variable since the context is empty. To summarize formally:

Lemma 4 (Canonical Singleton Forms) *If* $, \vdash_{\mathfrak{s}} v : \mathfrak{S}(v'{:}\tau)$ *then either* $v \equiv v'$ *or (for some* x*)* $v \equiv x$ *and* $(x{:}\mathfrak{S}(v'{:}\tau)) \in , .$

Corollary 5 *If* $\vdash_{\mathfrak{s}} v : \mathfrak{S}(v'{:}\tau)$ *then* $v \equiv v'$.

Thus, from our initial suppositions, we can deduce that $f[\mathfrak{S}(v{:}\tau)]v \mapsto^* v'$ implies $v \equiv v'$. To conclude, we invoke parametricity to show that $f[\tau]v$ computes to the same value as $f[\mathfrak{S}(v{:}\tau)]v$. To do so we use the following definition and lemma, which together state that terms that differ only in their types compute in the same manner:

Definition 6 *Two terms* t *and* t' *are* identical modulo type annotations *(written* $t =_0 t'$*) if they differ only in their component type expressions. Formally,* $=_0$ *is the least congruence such that (for any* e, τ *and* τ'*)* $\lambda x{:}\tau.e =_0 \lambda x{:}\tau'.e$ *and* $e[\tau] =_0 e[\tau']$.

Lemma 7 (Parametricity) *If* $e_1 =_0 e_2$ *then:*

1. *If* $e_1 \mapsto^* v_1$ *then there exists* v_2 *such that* $e_2 \mapsto^* v_2$ *and* $v_1 =_0 v_2$.
2. *If* $e_1 \mapsto^* v_1$ *and* $e_2 \mapsto^* v_2$ *then* $v_1 =_0 v_2$.

Since $f[\mathfrak{S}(v{:}\tau)]v =_0 f[\tau]v$ and $f[\mathfrak{S}(v{:}\tau)]v \mapsto v$, we may conclude by Lemma 7 that the result of $f[\tau]v$ is identical modulo type annotations to v. The full argument is summarized as Theorem 8:

Theorem 8 *If* $\vdash f : \forall\alpha.\,\alpha \to \alpha$ *and* $\vdash v : \tau$ *and* $f[\tau]v \mapsto^* v'$ *then* $v' =_0 v$.

Proof

Since augmentation preserves derivability, $\vdash_{\mathfrak{s}} f : \forall\alpha.\,\alpha \to \alpha$ and $\vdash_{\mathfrak{s}} v : \tau$. Therefore $\vdash_{\mathfrak{s}} v : \mathfrak{S}(v{:}\tau)$ so $\vdash_{\mathfrak{s}} f[\mathfrak{S}(v{:}\tau)]v : \mathfrak{S}(v{:}\tau)$. Since $f[\tau]v \mapsto^* v'$, by Parametricity there exists v'' such that $f[\mathfrak{S}(v{:}\tau)]v \mapsto^* v''$ and $v' =_0 v''$. By Subject Reduction, $\vdash_{\mathfrak{s}} v'' : \mathfrak{S}(v{:}\tau)$. By Corollary 5, $v'' \equiv v$. Hence $v' =_0 v$. □

Although Theorem 8 used a empty context, we can easily prove similar results for nonempty contexts analogously. Lemmas 2 and 3 allow working beneath a substitution, and since an unaugmented context will contain no singleton bindings, we can always be in Lemma 4's first case.

Similar arguments can show results determining the behavior of functions with a variety of different types, such as:

- Functions with type $\forall\alpha.\forall\beta.\,\alpha \to \beta \to \alpha$ must be extensionally equivalent to the K combinator.
- Functions with type $\forall\alpha.\,(\forall\beta.\,(\alpha{\to}\beta){\to}\beta){\to}\alpha$ must be extensionally equivalent to $\Lambda\alpha.\,\lambda f.\,f[\alpha]id$.
- Functions with type $\forall\alpha.\forall\beta.\,(\alpha \times \beta) \to (\beta \times \alpha)$ must be extensionally equivalent to the swap function (in an extension supporting product types).

2.1 Multiplicate Results

The technique as presented so far suffices to determine behaviors for a variety of types that imply a single-valued result, but it does not help in cases where a function can have multiple possible results. For example, a function with type $\forall\alpha.\,\alpha \to \alpha \to \alpha$ always returns one of its two arguments.

To extend the technique to this sort of problem, we want to use, instead of singleton types, a more general type that specifies a fixed finite set of members. We can build such types from singletons using union types. The union type $\tau_1 \vee \tau_2$ will contain all members of τ_1 and all members of τ_2:

$$\frac{, \vdash \tau_1 \text{ type} \quad , \vdash \tau_2 \text{ type}}{, \vdash \tau_1 \vee \tau_2 \text{ type}}$$

$$\frac{, \vdash e : \tau_1 \quad , \vdash \tau_2 \text{ type}}{, \vdash e : \tau_1 \vee \tau_2} \qquad \frac{, \vdash e : \tau_2 \quad , \vdash \tau_1 \text{ type}}{, \vdash e : \tau_1 \vee \tau_2}$$

Again, the primary burden is to show that the addition of union types is a sound extension. The Subject Reduction lemma remains true, and is proven in exactly the same way, as are the other lemmas, so I will not repeat them here. The only additional tool necessary is a canonical forms lemma for union types:

Lemma 9 (Canonical Union Forms) *If* $, \vdash_{\mathfrak{s}} v : \tau_1 \vee \tau_2$ *then either* $, \vdash_{\mathfrak{s}} v : \tau_i$ *(for* $i = 1$ or 2*) or (for some* x*)* $v \equiv x$ *and* $(x{:}(\tau_1 \vee \tau_2)) \in , .$

Corollary 10 *If* $\vdash_{\mathfrak{s}} v : (\mathfrak{S}(v_1{:}\tau) \vee \cdots \vee \mathfrak{S}(v_n{:}\tau))$ *then* $v \equiv v_i$ *for some* $1 \le i \le n$.

Theorem 11 *If* $\vdash f : \forall\alpha.\,\alpha \to \alpha \to \alpha$ *and* $\vdash v_1 : \tau$ *and* $\vdash v_2 : \tau$ *and* $f[\tau]v_1v_2 \mapsto^* v'$ *then either* $v' =_0 v_1$ *or* $v' =_0 v_2$.

Proof

Since augmentation preserves derivability, $\vdash_{\mathfrak{s}} f : \forall\alpha.\,\alpha \to \alpha \to \alpha$ and $\vdash_{\mathfrak{s}} v_1 : \tau$ and $\vdash_{\mathfrak{s}} v_2 : \tau$. Therefore

$\vdash_s v_1 : \mathfrak{S}(v_1{:}\tau) \vee \mathfrak{S}(v_2{:}\tau)$ and $\vdash_s v_2 : \mathfrak{S}(v_1{:}\tau) \vee \mathfrak{S}(v_2{:}\tau)$. Hence $f[\mathfrak{S}(v_1{:}\tau) \vee \mathfrak{S}(v_2{:}\tau)]v_1 v_2 : \mathfrak{S}(v_1{:}\tau) \vee \mathfrak{S}(v_2{:}\tau)$. Since $f[\tau]v_1 v_2 \mapsto^* v'$, by Parametricity there exists v'' such that $f[\mathfrak{S}(v_1{:}\tau) \vee \mathfrak{S}(v_2{:}\tau)]v_1 v_2 \mapsto^* v''$ and $v' =_0 v''$. By Subject Reduction, $\vdash_s v'' : \mathfrak{S}(v_1{:}\tau) \vee \mathfrak{S}(v_2{:}\tau)$. By Corollary 10, $v'' \equiv v_1$ or $v'' \equiv v_2$. Hence $v' =_0 v_1$ or $v' =_0 v_2$. □

3 Stack Preservation

The original motivation for this work was to prove the stack preservation property discussed in Section 1. Because state affecting operations are a fundamental part of assembly language, proving stack preservation using an analog of Reynolds's theorem is prohibitively difficult. However, the result easily succumbs to the technique presented in this paper.

3.1 STAL Overview

The static and dynamic semantics of STAL are long and somewhat complicated, and I will not reproduce them here. Instead, I will reprise the syntax and discuss the semantics at an intuitive level. Full formal details appear in Morrisett *et al.* [5]. Readers familiar with STAL can skip immediately to Section 3.2.

The main syntactic constructs of STAL are given in Figure 2. A machine state (or, program) in the STAL abstract machine is a triple, consisting of a heap, a register file, and a sequence of instructions. A register file is a finite mapping of register names to word-sized values, such as integers and pointers. A heap is a finite mapping of labels (ℓ) to larger-than-word values, that is, executable code (discussed below) and tuples. A third sort of value, a *small* value, is either a register name or a word value; this distinction is drawn because a register's contents cannot be a register name. Also, in addition to mapping registers to word values, a register file also maps a special distinguished register, the stack pointer (`sp`), to the current machine stack, which is a `nil`-terminated list of word values.

The STAL type system contains several notions of type. Ordinary data (word, heap, and small value) are given ordinary types (τ); such as type variables (α), `int`, or code types ($\forall[\Delta].,$, discussed below). Stacks are given stack types (σ); such as stack type variables[4] (ρ), the empty stack type `nil` (which specifies an empty stack), or cons types $\tau::\sigma$ (which specify a stack $w::S$ where w has type τ and S has type σ). Finally, heaps and register files each have a notion of type; heap types assign a type to every label, and register file types assign a stack type to the stack pointer and an ordinary type to each other register. I will often write $M\{X{:}E\}$ or $M\{X \mapsto E\}$ as notion for map update.

Code blocks, written $\texttt{code}[\Delta].I$, are sequences of instructions I, that abstract a set of type and stack type variables (Δ), and provide a type $(,)$ for the incoming register file. The code's register file type serves as a precondition that must hold of the register file before the code block may be called. Accordingly, the type of a code block ($\forall[\Delta].,$) indicates its type and stack type arguments and its register file precondition. When a code block abstracts no variables, we often omit the prefix $\forall[]$ in its type. A code type does not specify any sort of postcondition because code blocks do not return *per se;* rather, they transfer control by calling some other code (often given by a return address argument) and satisfy that code's precondition.

For example, code having the type discussed in Section 1,

$$\forall[\rho].\{\texttt{sp} : \rho, \texttt{ra} : \{\texttt{sp}{:}\rho, \texttt{r1}{:}\texttt{int}\}\}$$

can be called, after instantiation ρ with some σ, when the stack has type σ and when `ra` contains a return address that itself may be called when the stack has type σ and `r1` contains an integer. This type does not guarantee that the code will transfer control back to the address in `ra`, or even that it will terminate at all, but the precondition $\{\texttt{sp}{:}\sigma, \texttt{r1}{:}\texttt{int}\}$ will be satisfied if it does.

Semantics The dynamic semantics of STAL is given by a small-step evaluation relation (written $P \mapsto P'$) that maps a machine state (H, R, I) to a new machine state in a manner determined by the first instruction of I. The static semantics is given by several judgements. All of them are certainly relevant to the Subject Reduction proof, but we need be directly concerned with only a few of them here:

Judgement	Meaning
$\vdash P$	Program P is a well-formed.
$\vdash H : \Psi$	Heap H has type Ψ.
$\Psi \vdash R : ,$	Register file R has type $,$.
$\Psi \vdash S : \sigma$	Stack S has type σ.
$\Psi \vdash h : \tau\ \mathit{hval}$	Heap value h has type τ.
$\Psi; \Delta \vdash w : \tau\ \mathit{wval}$	Word value w has type τ.
$\Psi; \Delta; , \vdash v : \tau$	Small value v has type τ.
$\Psi; \Delta; , \vdash I$	Instruction sequence I is well-formed.
$\Delta \vdash \tau$ type	Type τ is well-formed.
$\Delta \vdash \sigma$ stype	Stack type σ is well-formed.
Where, as appropriate, Ψ is the type of the heap, Δ specifies free type variables, and $,$ is the type of the register file.	

A few relevant inference rules are given in Figure 3.

Polymorphism The mechanism for abstracting code to take type (and stack type) arguments was discussed above. Type application is performed by instantiating a value with a type, written (for example) $w[\tau]$. This operation attaches type arguments to a code label. (Types are erased at run time, so no action need be taken at run time to do this.) When control is transferred to instantiated code, the type arguments are substituted

[4] The importance of stack type variables is made clear shortly.

$$
\begin{array}{lrcl}
\textit{programs} & P & ::= & (H, R, I) \\
\textit{heaps} & H & ::= & \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \\
\textit{register files} & R & ::= & \{\mathtt{sp} \mapsto S, r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} \\
\textit{stacks} & S & ::= & \mathtt{nil} \mid w{::}S \\
\textit{instruction sequences} & I & ::= & \mathtt{add}\ r_d, r_s, v; I \mid \mathtt{ld}\ r_d, r_s(i); I \mid \mathtt{mov}\ r_d, v; I \mid \cdots \\
 & & & \\
\textit{registers} & r & ::= & \mathtt{r1} \mid \mathtt{r2} \mid \cdots \\
\textit{word values} & w & ::= & \ell \mid i \mid w[\tau] \mid w[\sigma] \mid \cdots \\
\textit{heap values} & h & ::= & \mathtt{code}[\Delta].\ I \mid \langle w_1, \ldots, w_n \rangle \\
\textit{small values} & v & ::= & r \mid w \mid v[\tau] \mid v[\sigma] \mid \cdots \\
 & & & \\
\textit{types} & \tau & ::= & \alpha \mid \mathtt{int} \mid \forall[\Delta]., \ \mid \cdots \\
\textit{stack types} & \sigma & ::= & \rho \mid \mathtt{nil} \mid \tau{::}\sigma \mid \cdots \\
\textit{heap types} & \Psi & ::= & \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\} \\
\textit{register file types} & , & ::= & \{\mathtt{sp}{:}\sigma, r_1{:}\tau_1, \ldots, r_n{:}\tau_n\} \\
\textit{contexts} & \Delta & ::= & \epsilon \mid \Delta, \alpha \mid \Delta, \rho
\end{array}
$$

Figure 2: STAL Syntax (Abridged)

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : , \quad \Psi; \epsilon; , \ \vdash I}{\vdash (H, R, I)}$$

$$\frac{\Psi \vdash S : \sigma \quad \Psi; \epsilon \vdash w_i : \tau_i \ \ wval}{\Psi \vdash \{\mathtt{sp} \mapsto S, r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} : \{\mathtt{sp}{:}\sigma, r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}}$$

$$\frac{}{\Psi \vdash \mathtt{nil} : \mathtt{nil}} \qquad \frac{\Psi; \epsilon \vdash w : \tau \ \ wval \quad \Psi \vdash S : \sigma}{\Psi \vdash w{::}S : \tau{::}\sigma}$$

$$\frac{}{\Delta \vdash \mathtt{nil}\ \text{stype}} \qquad \frac{\Delta \vdash \tau\ \text{type} \quad \Delta \vdash \sigma\ \text{stype}}{\Delta \vdash \tau{::}\sigma\ \text{stype}}$$

$$\frac{\Psi; \Delta; , \ \vdash v : \{\mathtt{sp}{:}\sigma, r_1{:}\tau_1, \ldots, r_n{:}\tau_n\} \quad , \ = \{\mathtt{sp}{:}\sigma, r_1{:}\tau_1, \ldots, r_m{:}\tau_m\}}{\Psi; \Delta; , \ \vdash \mathtt{jmp}\ v} \ (n \leq m)$$

Figure 3: STAL Static Semantics (Abridged)

for the formal arguments in the corresponding code; for example:

$$(H, R, \mathtt{jmp}\ \mathtt{r1}[\sigma]) \mapsto (H, R, I[\sigma/\rho])$$
when
$R(\mathtt{r1}) = \ell$
$H(\ell) = \mathtt{code}[\rho].\ I$

3.2 Stack Preservation Proof

As before, we begin by augmenting the language, in this case with singleton stack types. We will again write judgements in the augmented language with the symbol $\vdash_s$. Two rules govern the usage of singleton stack types:[5]

$$\frac{}{\Delta \vdash_s \mathfrak{S}(S)\ \text{stype}} \qquad \frac{}{\Psi \vdash_s S : \mathfrak{S}(S)}$$

Again, the primary burden is to prove that this is a sound extension:

Lemma 12 (Augmented Subject Reduction)

[5] In contrast to the singleton types of Section 2, these singleton stack types do not constrain the stack in question to be well-formed. This simplifies the argument slightly: the stack formation judgement is relative to a heap type, so constraining singleton stack types to be well-formed would require the stack type formation judgement to be relative to a heap type as well. Since, in that case, the judgement forms of $\vdash$ and $\vdash_s$ would be different, a nontrivial argument would be required to show that augmentation preserves derivability. Not constraining the stacks in question to be well-formed removes the need for such an argument, and poses no other problems.

If $\vdash_{\mathfrak{s}} P$ *and* $P \mapsto P'$ *then* $\vdash_{\mathfrak{s}} P'$.

The proof is rather involved, but not much more so than the proof for the unaugmented language, which closely follows the proof in Morrisett *et al.* [6]. We may also easily show Canonical Forms and Parametricity lemmas analogous to Lemmas 4 and 7:

Lemma 13 (Canonical Singleton Forms) *If* $\Psi \vdash S : \mathfrak{S}(S')$ *then* $S \equiv S'$.

Definition 14 *Two expressions* E *and* E' *are* identical modulo type annotations *(written* $E =_0 E'$*) if they differ only in their component type expressions.*

Lemma 15 (Parametricity) *If* $P_1 =_0 P_2$ *then:*

1. *If* $P_1 \mapsto^* P_1'$ *then there exists* P_2' *such that* $P_2 \mapsto^* P_2'$ *and* $P_1' =_0 P_2'$.
2. *If* $P_1 \mapsto^* P_1'$ *and* $P_2 \mapsto^* P_2'$ *then* $P_1' =_0 P_2'$.

The last thing we must do before we can prove the stack preservation property is to define what is meant by return from a function. The naive idea is that a function has returned when control enters the code block pointed to be the return address. This naive idea does not work because control might return to the return code other than through the interface provided; such as by a global jump, by a recursive jump to the caller, or by some other indirect means. We need the definition to identify as a return only invocations of the return code that pass through the particular interface provided. This is done by constructing a clone of the return address that is not accessible in any other way and running the program until the clone is used. Then the clone is deleted to ensure that a return state is a state actually reached by computation (Proposition 17).

Definition 16 *The state of* (H_1, R_1, I_1) *on return through register* r *is* (H_2, R_2, I_2) *if:*

- $R_1(r) = \ell[\psi_1]\cdots[\psi_n]$ *(where each* ψ_i *is either a type or a stack type), and*
- ℓ' *is fresh, and*
- $(H_1\{\ell' \mapsto H_1(\ell)\}, R_1\{r \mapsto \ell'[\psi_1]\cdots[\psi_n]\}, I_1) \mapsto^* (H', R', I')$, *and*
- *the next evaluation step of* (H', R', I') *is to call* $\ell'[\psi_1]\cdots[\psi_n]$ *(that is,* I' *begins with either a jump or a successful conditional branch to* $\ell'[\psi_1]\cdots[\psi_n]$*), and*
- $H_2 \equiv (H' \setminus \ell')[\ell/\ell']$ *and* $R_2 \equiv R'[\ell/\ell']$ *and* $I_2 = I'[\ell/\ell']$.

Proposition 17 *If the state of* P *on return through any register is* P'*, then* $P \mapsto^* P'$.

Theorem 18 (Stack Preservation) *Suppose* $\vdash H_1 : \Psi$ *and* $\Psi \vdash R_1 : \,,$ *and* $\,, (\mathtt{sp}) = \sigma$. *If* $\Psi; \epsilon; \, \vdash v : \forall[\rho].\{\mathtt{sp} : \rho, \mathtt{ra} : \{\mathtt{sp}{:}\rho, \mathtt{r1}{:}\mathtt{int}\}\}$ *and the state of* $(H_1, R_1, \mathtt{jmp}\ v[\sigma])$ *on return through* `ra` *is* (H_2, R_2, I_2), *then* $R_1(\mathtt{sp}) =_0 R_2(\mathtt{sp})$.

Proof

Let H', R', I', ℓ, ℓ' and ψ_i be defined according to Definition 16. Let $H_1' = H_1\{\ell' \mapsto H_1(\ell)\}$ and let $R_1' = R_1\{\mathtt{ra} \mapsto \ell'[\psi_1]\cdots[\psi_n]\}$. By definition, $(H_1', R_1', \mathtt{jmp}\ v[\sigma]) \mapsto^* (H', R', I')$ and the next evaluation step of (H', R', I') is to call $\ell'[\psi_1]\cdots[\psi_n]$.

Using Parametricity, we will construct another computation $P \mapsto^* P'$, where $P =_0 (H_1', R_1', \mathtt{jmp}\ v[\sigma])$, where $P' =_0 (H', R', I')$, and where the next evaluation step of P' is also to call $\ell'[\psi_1]\cdots[\psi_n]$. However, the type of ℓ' in P and P' will use a singleton stack type to constrain the stack to be the desired stack, $R_1(\mathtt{sp})$. Since P' will immediately call ℓ', and since P' will be well-formed, it will follow that the stack of P' is $R_1(\mathtt{sp})$, and therefore that $R_2(\mathtt{sp}) =_0 R_1(\mathtt{sp})$.

Let $S = R_1(\mathtt{sp})$ and let $h_\ell = H_1(\ell)$. The typing assumptions ensure that h_ℓ must be a code value, since the current instruction is a jump to $v[\sigma]$ and the precondition of $v[\sigma]$ says that the contents of `ra` (that is, $\ell[\psi_1]\cdots[\psi_n]$) are a function. Thus, let $h_\ell = \mathtt{code}[\Delta_\ell]\,,_\ell.I_\ell$. At the start of our new computation, instead of binding ℓ' to h_ℓ (as in H_1'), we will bind it to a variant of h_ℓ that demands a singleton stack type.

Let $h_\ell' = \mathtt{code}[\Delta_\ell](\,,_\ell\{\mathtt{sp}{:}\mathfrak{S}(S)\}).I_\ell$. Let $H_1'' = H_1\{\ell' \mapsto h_\ell'\}$. Observe that $H_1' =_0 H_1''$ and $\mathtt{jmp}\ v[\sigma] =_0 \mathtt{jmp}\ v[\mathfrak{S}(S)]$. Therefore $(H_1', R_1', \mathtt{jmp}\ v[\sigma]) =_0 (H_1'', R_1', \mathtt{jmp}\ v[\mathfrak{S}(S)])$. Let P be the latter. By Parametricity, $P \mapsto^* P'$ for some P' such that $P' \equiv (H'', R'', I'')$, $H' =_0 H''$, $R' =_0 R''$ and $I' =_0 I''$. There exist no transitions in the operational semantics that alter a code value in the heap, so $H''(\ell') \equiv H_1''(\ell') \equiv h_\ell'$.

Since augmentation preserves derivability, all the assumed judgements are derivable in the augmented language, where we also have $\epsilon \vdash_{\mathfrak{s}} \mathfrak{S}(S)$ stype and $\Psi \vdash_{\mathfrak{s}} S : \mathfrak{S}(S)$. It is then easy to show that $\vdash_{\mathfrak{s}} (H_1'', R_1', \mathtt{jmp}\ v[\mathfrak{S}(S)])$. Therefore, by Subject Reduction, $\vdash_{\mathfrak{s}} (H'', R'', I'')$. By inspection of the typing rules, $\vdash_{\mathfrak{s}} H'' : \Psi'$ and $\Psi' \vdash_{\mathfrak{s}} R'' : \,,'$, for some Ψ' and $\,,'$.

Recall the next evaluation step of (H', R', I') is to call $\ell'[\psi_1]\cdots[\psi_n]$. Since $(H', R', I') =_0 (H'', R'', I'')$, the latter's next evaluation step must be to call $\ell'[\psi_1']\cdots[\psi_n']$, for some $\psi_i' =_0 \psi_i$. Therefore, $\,,'$ must match the precondition on $\ell'[\psi_1']\cdots[\psi_n']$, which is $(\,,_\ell\{\mathtt{sp}{:}\mathfrak{S}(S)\})[\vec{\psi}/\Delta]$. In particular, $\,,'(\mathtt{sp}) \equiv \mathfrak{S}(S)[\vec{\psi}/\Delta]$. Since S has no free variables (recall it typechecks in an empty context), $\,,'(\mathtt{sp}) \equiv \mathfrak{S}(S)$. Since $\Psi' \vdash_{\mathfrak{s}} R'' : \,,'$, we conclude by Lemma 13 that $R''(\mathtt{sp}) \equiv S$.

Now recall that $R' =_0 R''$, so $R'(\mathtt{sp}) =_0 S$. The label ℓ' was fresh, so it could not appear in S and thus $S[\ell/\ell'] \equiv S$. Consequently, $R_2(\mathtt{sp}) \equiv (R'(\mathtt{sp}))[\ell/\ell'] =_0 S[\ell/\ell'] \equiv S$. □

4 Conclusions

Syntactic proof techniques, where applicable, have often been found to be simpler than semantic proof techniques for similar results. This has been particularly true in setting that include computational effects, where non-syntactic techniques have often been prohibitively challenging. In this paper I propose a syntactic approach to proving certain parametricity results. This technique is effective for results in which a function's output (or, more generally, an expression's result) is shown to be a simple variant of one of its inputs.

The technique reveals no fundamental new insights into parametricity, but rather frames the standard observations in a particularly direct way. It is that directness that leads to the technique's simplicity, but also that limits its applicability. To see the directness of connection, recall the intuitive parametricity arguments from Section 1. In each argument, a conclusion was drawn regarding the behavior of general instances of a polymorphic function by analogy with a particular instance; in one case the `unit` instance, and in the other case the `nil` instance. The common property of each of these types is that they are *singleton* types. Each contained only one member and therefore fully specified its function's results. In this paper I lift such arguments to nontrivial types by adding singleton subtypes of other types.

The primary proof obligation of this technique is to demonstrate Subject Reduction in the presence of singleton types. Subject Reduction arguments are usually not hampered severely by computational effects, making it tractable to generalize this technique to effectful settings. This is illustrated by the main new result of this paper, a proof of the Stack Preservation property for STAL. This property provides an important security guarantee for STAL programs, that their stacks are protected from corruption during calls to untrusted code.

References

[1] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In *Theoretical Aspects of Computer Software 1991*, volume 526 of *Lecture Notes in Computer Science*, pages 750–770, Sendai, Japan, 1991. Springer-Verlag.

[2] Andrew J. Kennedy. Relational parametricity and units of measure. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 442–455, Paris, January 1997.

[3] QingMing Ma. Parametricity as subtyping. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 281–292, Albuquerque, New Mexico, January 1992.

[4] QingMing Ma and John C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In *Seventh Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40, Pittsburgh, Pennsylvania, March 1991. Springer-Verlag.

[5] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1998. Extended version published as CMU technical report CMU-CS-98-178.

[6] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 1999. To appear. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[7] Peter W. O'Hearn and Robert D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.

[8] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In *International Conference on Typed Lambda Calculi and Applications*, pages 361–375, 1993.

[9] Gordon Plotkin, Martín Abadi, and Luca Cardelli. Subtyping and parametricity. In *Ninth IEEE Symposium on Logic in Computer Science*, pages 310–319, July 1994.

[10] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[11] Chistopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967.

[12] Philip Wadler. Theorems for free! In *Fourth Conference on Functional Programming Languages and Computer Architecture*, London, September 1989.

[13] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.