Stackability in the Simply-Typed Call-By-Value Lambda Calculus *

Anindya Banerjee and David A. Schmidt †
Department of Computing and Information Sciences
Kansas State University ‡
{banerjee, schmidt}@cis.ksu.edu

Abstract

This paper addresses two issues: (1) What it means for a higher-order, eager functional language to be implemented with a single, global, stack-based environment and (2) how the existence of such an environment can be predicted statically.

The central theme is the use of the λ-abstraction to control the *extent* or *lifetime* of bindings. All programs in a higher-order, call-by-name language can be implemented with a stack environment. The reason: soundness of η-expansion and decurrying for call-by-name. However, η-expansion is *unsound* for call-by-value. Hence, we must identify a subset of the simply-typed, call-by-value λ-calculus, where the λ-abstraction can serve as the block construct for a stack implementation.

The essence of environment stackability is that the shape of the environment remains the same before and after the execution of an expression. Thus if a closure is returned as a value, the environment trapped in it must be a subenvironment of the global environment. This yields a dynamic criterion for stackability — indeed, it is the downwards funargs criterion of the LISP community. A safe static criterion can now be found via closure analysis.

1 Introduction

1.1 What is Block-Structure?

A syntactic construct that admits local definitions is called a *block*, and a language with blocks is called *block-structured*. The prototypical block has the syntax begin D in U end, and its intended semantics is that the bindings, D, are visible only to the *body*, U. This defines the *scope* or *visibility* of bindings.

*A preliminary version of this work was presented at the first Atlantique Workshop on Semantics-Based Program Manipulation, Portland, Oregon, January 1994.

†Supported by NSF grant CCR-93-02962 and ONR grant N00014-94-1-0866

‡Manhattan, Kansas 66506, USA. Phone:(913)532-6350, Fax:(913)532-7353

Here, we examine a related notion, namely, the *extent* or *lifetime* of bindings. The lifetime of a binding is the period between the time the binding is established and the time the binding is freed. The scope of a binding can be statically determined, but the binding's extent is dynamic. For example, a function, f, might evaluate to a closure that contains a free identifier, v. The extent of the binding for v depends upon f: the binding to v cannot be freed until all uses of f are evaluated.

A significant feature of languages like Algol-68 is that both scope and extent of bindings are controlled by the begin D in U end block: the bindings, D, are established on the environment stack when the block is entered, and the bindings are freed from the stack when U evaluates to a value. (The danger that U's value references a binding in D is avoided by syntactic restrictions.)

In this paper we focus on extents for eager functional languages. The λ-abstraction is the obvious candidate for the block-structuring construct, but it controls only scope and not extent. To see this, consider the example $(\lambda x.(\lambda v.v)(\lambda y.x))2$: the binding of x to 2 is established, and the body of the λ-abstraction produces the value $\lambda y.x$. The binding of x to 2 cannot be freed, even though the λ-abstraction has finished. Apparently, one must abandon the stack environment for a heap environment.

A novel alternative was recently proposed by Tofte and Talpin [10], who apply an effects analysis to determine the extents of bindings. Then, they insert into the program constructs of the form letregion ρ in e end to control extents, where ρ allocates storage in advance for the bindings made within e. The resulting programs are verbose, and the underlyimg implementation is a stack of heaps rather than a classical stack.

In this paper, rather than altering an arbitrary program and forcing it to run on a heap-stack, we identify a subset of the simply-typed, call-by-value λ-calculus, whose programs can be run "as is" with a classical- stack environment. The λ-abstraction serves as the block construct for both scope and extent.

1.2 What is a Stack?

Since we intend to use an environment stack for bindings, we should explain what one is.

The environment stack is a data structure upon which bindings are *established* and then *freed*. The freeing of bindings must be done in a "disciplined" way so that stack space is re-used. A block construct supplies the discipline: bindings are established on block entry and are freed on block exit.

Our environment stack implements environment sharing. That is, downwards pointers ("static links") are used when programs with nested blocks are evaluated. Contrast this with the environment as a stack-of-binding-lists in the VEC-machine [8] and the Krivine-machine [5]; there, each block has its own binding list and much duplication arises. To prevent this duplication we legislate that, once established, a binding cannot be re-copied onto the top of the environment stack.

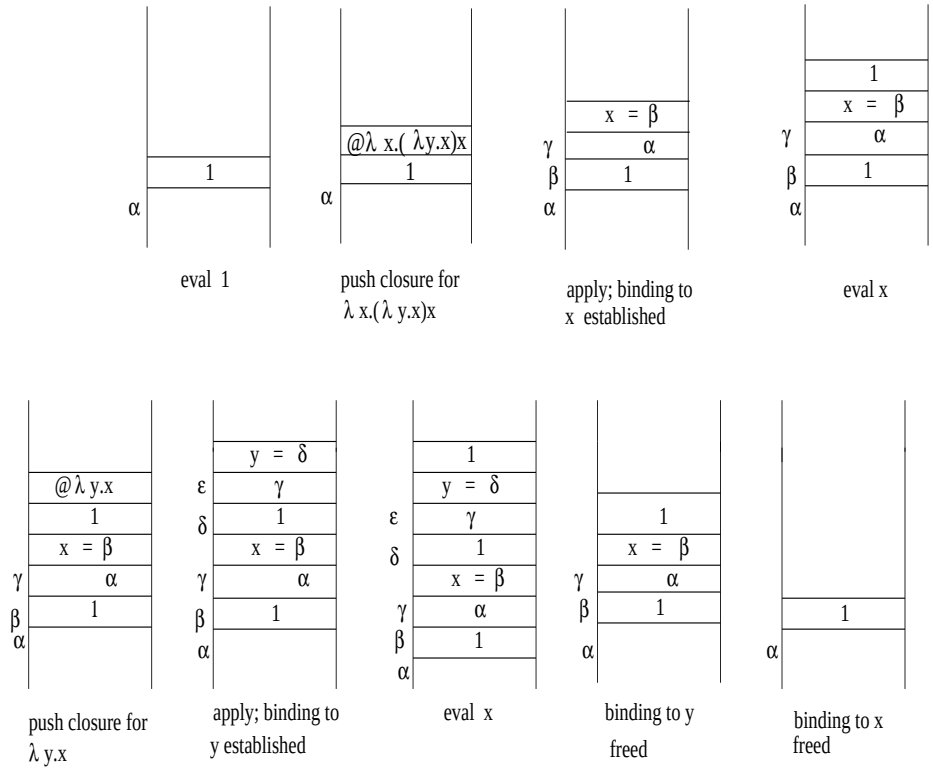


Figure 1: Snapshots of the environment stack for $(\lambda x.(\lambda y.x)x)1$

Figure 1 shows snapshots of the environment stack during the evaluation of the expression $(\lambda x.(\lambda y.x)x)1$. The static links are α, γ and ϵ, where α is the initial static link. (The dynamic links are not shown.) Temporaries are also pushed on the stack. An expression that can be evaluated in the style of Figure 1, where the λ-abstraction serves as the block construct for extents, is called *env-stackable*.

2 Simple Expressions and Partially-Applied Closures

The BNF of the λ-calculus we study are given in Figures 2 and 3.

In this paper we study call-by-value evaluation. But it is worthwhile to first review why all programs in the *call-by-name* λ-calculus are env-stackable. In the call-by-name language, an expression e of type $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n$, where τ_n is a base type, should be read as an abbreviation for $\lambda x_1.\lambda x_2.\cdots\lambda x_n.e\ x_1\ x_2 \cdots x_n$, (due to η-expansion), which is itself an abbreviation for $\lambda\langle x_1, x_2, \ldots, x_n\rangle.e\ x_1\ x_2 \ldots x_n$, (due to decurrying). This expression is env-stackable, because the bindings to x_1, x_2, $\cdots$, x_n are established, as a group, only when the body of the λ-abstraction, a phrase of ground type, is

$\iota \in \text{BaseType}$
$\tau \in \text{Type} \qquad \tau ::= \iota \mid \tau_1 \to \tau_2$
$\pi \in \text{TypeAssignment} \qquad \pi ::= \{x_i : \tau_i\}_{i \geq 0}$
$c \in \text{ConstExpression}$
$x \in \text{Identifier}$
$e \in \text{Expression} \qquad e ::= c \mid x \mid \lambda\, x : \tau\, .\, e \mid e_1 e_2 \mid \mathsf{if}\ e_1\ e_2\ e_3 \mid \mathsf{rec}\ f . \lambda\, x : \tau\, .\, e$

Figure 2: BNF of the Typed Call-by-value Lambda Calculus

$$\pi \vdash c : \iota \qquad\qquad \pi \vdash x : \tau \ \text{ if } (x : \tau) \in \pi$$

$$\frac{\{x : \tau_1\} \oplus \pi \vdash e : \tau_2}{\pi \vdash \lambda\, x : \tau_1\, .\, e : \tau_1 \to \tau_2} \qquad\qquad \frac{\pi \vdash e_1 : \tau_1 \to \tau_2 \qquad \pi \vdash e_2 : \tau_1}{\pi \vdash e_1 e_2 : \tau_2}$$

$$\frac{\pi \vdash e_1 : \mathit{bool} \qquad \pi \vdash e_2 : \tau \qquad \pi \vdash e_3 : \tau}{\pi \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau}$$

$$\frac{\{f : \tau_1 \to \tau_2\} \oplus \pi \vdash \lambda\, x : \tau_1\, .\, e : \tau_1 \to \tau_2}{\pi \vdash \mathsf{rec}\ f : \tau_1 \to \tau_2\, .\, \lambda\, x : \tau_1\, .\, e : \tau_1 \to \tau_2}$$

Figure 3: Static Semantics

evaluated. The value produced by the body will contain no unresolved reference to an x_i, hence the bindings to the x_i's can be freed on schedule. Thus the λ-abstraction can serve as the block construct for extents.[1] These ideas lie at the heart of the stack implementation of Idealized Algol [7].

Can this approach be adapted to the call-by-value λ-calculus that we study in this paper? Unfortunately, η-expansion is unsound. But Georgeff [2] noted that decurrying can be represented by a closure of the form $\langle \lambda x.e, \rho_l, \rho_n \rangle$, where ρ_l is a *local environment* and ρ_n is a *nonlocal environment*. An expression like $(\lambda x_1.\lambda x_2.\cdots\lambda x_i.\lambda x_{i+1}.\cdots\lambda x_n.e_0)e_1e_2\cdots e_i$ evaluates to $\langle \lambda x_{i+1}.\cdots\lambda x_n.e_0, (x_i = v_i, \cdots, x_2 = v_2, x_1 = v_1), \rho_0 \rangle$, where each e_j evaluates to value v_j, $1 \leq j \leq i$, and ρ_0 is the nonlocal environment. The local environment accumulates the bindings to the x_j's until all are collected; then, as a group, they are established (appended to ρ_0). A closure $\langle \lambda x.e, \rho_l, \rho_n \rangle$, is *partially applied* if e is a λ-abstraction; else the closure is *fully applied*.

The dynamic semantics of the call-by-value λ-calculus is presented in Figures 4 and 5; it shows how the closures are built and applied. Appendix A shows that this semantics is equivalent to the usual one.

The above representation of closures and the evaluation rules for application do not in themselves make all simply-typed, call-by-value programs stackable, but Georgeff identified a class of expressions, called the *simple expressions*, which

[1] η-expansion and decurrying can be applied to the call-by-need calculus, but they do not address the fundamental problem of updating an established binding.

$c \in \text{BaseValue}$

$v \in \text{Value}_G \qquad v ::= c \mid \langle \lambda x.e, \rho_l, \rho_n \rangle \mid \mu f.\langle \lambda x.e, \rho \rangle$

$\rho \in \text{Env}_G \qquad \rho ::= <> \mid (i = v).\rho$

Figure 4: Semantic Objects for Georgeff-style Dynamic Semantics

$$\rho \vdash c \Rightarrow c \qquad \rho \vdash x \Rightarrow v \text{ if } (x = v) \in \rho \qquad \rho \vdash \lambda x.e \Rightarrow \langle \lambda x.e, (), \rho \rangle$$

$$\rho \vdash f \Rightarrow \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, \rho' \rangle), \rho' \rangle \text{ if } (f = \mu f.\langle \lambda x.e, \rho' \rangle) \in \rho$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle \lambda x.e, \rho_l, \rho_n \rangle \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash e_1 e_2 \Rightarrow \langle e, (x = v).\rho_l, \rho_n \rangle} \text{ if } e \text{ is a } \lambda\text{-abstraction}$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle \lambda x.e, \rho_l, \rho_n \rangle \qquad \rho \vdash e_2 \Rightarrow v \qquad (x = v).\rho_l \ @\rho_n \vdash e \Rightarrow v'}{\rho \vdash e_1 e_2 \Rightarrow v'} \text{ otherwise}$$

$$\frac{\rho \vdash e_1 \Rightarrow \text{true} \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash \text{if } e_1 \ e_2 \ e_3 \Rightarrow v} \qquad \frac{\rho \vdash e_1 \Rightarrow \text{false} \qquad \rho \vdash e_3 \Rightarrow v}{\rho \vdash \text{if } e_1 \ e_2 \ e_3 \Rightarrow v}$$

$$\rho \vdash \text{rec } f.\lambda x.e \Rightarrow \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, \rho \rangle), \rho \rangle$$

Figure 5: Dynamic Semantics, Georgeff-style

are env-stackable. The typing rules of these expressions are shown in Figure 6.

Simply stated, an expression is simple if all its λ-abstractions have as their bodies either λ-abstractions or expressions of basic type. In essence, a simple expression is fully η-expanded. It is the simulation of decurrying and η-expansion that allows a call-by-value implementation to evaluate simple expressions with a global environment stack. Let us examine the dynamic semantics of the simple expression $((\lambda\, x : int\,.\, \lambda\, f : int \rightarrow int\,.\, fx)1)(\lambda\, y : int\,.\, y)$. The turnstiles are numbered to indicate the order of evaluation steps.

$$\dfrac{\dfrac{\dfrac{\rho \vdash_4 0 \Rightarrow 0}{\rho \vdash_3 \lambda x.\lambda f.fx \Rightarrow \langle \lambda x.\lambda f.fx, (), \rho \rangle}}{\rho \vdash_2 (\lambda x.\lambda f.fx)0 \Rightarrow \langle \lambda f.fx, (x = 0), \rho \rangle} \quad \rho \vdash_5 \lambda y.y \Rightarrow \langle \lambda y.y, (), \rho \rangle \quad \dfrac{\dfrac{\rho_y \vdash_9 y \Rightarrow 0}{\rho_{f,x} \vdash_8 x \Rightarrow 0} \quad \rho_{f,x} \vdash_7 f \Rightarrow \langle \lambda y.y, (), \rho \rangle}{\rho_{f,x} \vdash_6 fx \Rightarrow 0}}{\rho \vdash_1 ((\lambda x.\lambda f.fx)0)(\lambda y.y) \Rightarrow 0}$$

where $\rho_{f,x}$ is $(f = \langle \lambda y.y, (), \rho \rangle).(x = 0).\rho$ and ρ_y is $(y = 0).\rho$.

This dynamic-semantics tree can be implemented with a global environment stack, similar to the one in Figure 1. To see this, read the nodes of the tree as a left-to-right tree traversal, as indicated by the subscripts on the turnstiles. At the start of evaluation, at Node 1, the stack is just ρ. The stack is unchanged during evaluation of Nodes 2 through 5, but at the start of evaluation of Node

$$\pi \vdash c : \iota \qquad \pi \vdash x : \tau \text{ if } (x : \tau) \in \pi$$

$$\frac{\{x_1 : \tau_1\} \oplus \cdots \oplus \{x_n : \tau_n\} \oplus \pi \vdash e : \iota}{\pi \vdash \lambda x_1 : \tau_1 \cdots \lambda x_n : \tau_n . e : \tau_1 \to \cdots \to \tau_n \to \iota} \qquad \frac{\pi \vdash e_1 : \tau_1 \to \tau_2 \quad \pi \vdash e_2 : \tau_1}{\pi \vdash e_1 e_2 : \tau_2}$$

Figure 6: Simple Expressions

6, a static link to ρ and the bindings for f and x are established (pushed) onto the stack. They are used by Node 8. Evaluation of Node 9 establishes another static link to ρ and the binding for y. At the conclusion of evaluation of Node 9, the static link and binding to y are freed. The static link and bindings to f and x are freed at the conclusion of evaluation of Node 6, leaving us with ρ.

The key to why this implementation worked correctly is that the ρ_n component of $\langle \lambda x.e, \rho_l, \rho_n \rangle$ can always be implemented as a static link. To see how this might fail, consider the dynamic semantics of the nonsimple expression $((\lambda\, x : int\,.\,(\lambda\, f : int \to int\,.\,f)(\lambda\, y : int\,.\,x))5)6$:

$$\frac{\dots \quad \dots \quad \dfrac{\dfrac{\dots}{\rho_x \vdash_B (\lambda f.f)(\lambda y.x) \Rightarrow \langle \lambda y.x, (), \rho_x \rangle}}{\rho \vdash_A (\lambda x.\dots.)5 \Rightarrow \langle \lambda y.x, (), \rho_x \rangle} \qquad \rho \vdash 6 \Rightarrow 6 \quad \rho_{y,x} \vdash_C x \Rightarrow 5}{\rho \vdash ((\lambda x.(\lambda f.f)(\lambda y.x))5)6 \Rightarrow 5}$$

where ρ_x is $(x = 5).\rho$ and $\rho_{y,x}$ is $(y = 6).(x = 5).\rho$.

At the node marked C, the environment $\rho_{y,x}$ *cannot* be implemented by a static link to ρ_x and a binding to y, because the environment ρ_x no longer exists on the global stack — ρ_x (namely, the binding to x and the static link to ρ) was popped at the conclusion of evaluation of Node B.

3 Analyses for Stackability

Not all nonsimple expressions are env-stackable, as we saw in the previous example. But many are. One example is $(\lambda\, f : int \to int\,.\,(\lambda\, y : int \to int\,.\,f)f)(\lambda\, x : int\,.\,x)$, whose dynamic-semantics tree is:

$$\frac{\rho \vdash \lambda f.\cdots \Rightarrow \langle \lambda f.\cdots, (), \rho \rangle \quad \rho \vdash \lambda x.x \Rightarrow v \quad \dfrac{\rho_f \vdash \lambda y.f \Rightarrow \langle \lambda y.f, (), \rho_f \rangle \quad \rho_f \vdash f \Rightarrow v \quad \rho_{y,f} \vdash f \Rightarrow v}{\rho_f \vdash (\lambda y.f)f \Rightarrow v}}{\rho \vdash (\lambda f.(\lambda y.f)f)(\lambda x.x) \Rightarrow v}$$

where v is the value $\langle \lambda x.x, (), \rho \rangle$, ρ_f is $(f = v).\rho$, and $\rho_{y,f}$ is $(y = v).(f = v).\rho$

Again, we see that the nonlocal environments can be implemented by static links. This clearly indicates that env-stackability is a deeper notion than just simple expressions—indeed, it is the "downwards funargs" property of the LISP community.

3.1 The Dynamic Criterion for Env-Stackability

We now state a criterion that ensures when a dynamic-semantics tree can be implemented by a global environment stack.

$p \in \text{Pointer}$ $\quad \bar{\rho} \in \text{Env}_P$

$\bar{\rho}_l \in \text{Local_Env}_P$ $\quad \bar{v} \in \text{Value}_P$

$$\bar{\rho}_l ::= <> \mid (i = \bar{v}).\bar{\rho}_l$$
$$\bar{\rho} ::= <> \mid (i = \bar{v}).\bar{\rho} \mid p.\bar{\rho}$$
$$v ::= n \mid b \mid \langle \lambda x.e, \bar{\rho}_l, p \rangle \mid \langle \mu f.\lambda x.e, p \rangle$$

Figure 7: Semantic Objects for Pointer Semantics

Criterion 1 (Dynamic Criterion)
A dynamic-semantics tree has the **dynamic criterion**, *if for every node in the tree of the form* $\rho \vdash e \Rightarrow \langle \lambda x.e', \rho_l, \rho_n \rangle$, *there is an ancestor node of the form* $\rho_n \vdash e_1 \Rightarrow v_1$.

It is easy to verify that the dynamic criterion holds for the two env-stackable examples seen earlier. Note also that the example that was non-stackable fails the criterion at Nodes A and B, which predict the problem at Node C.

To prove that a dynamic-semantics tree can be implemented on a machine with a global stack environment, we must formalize the latter. Figures 7 and 8 formalize this semantics, called the *pointer semantics*. The pointer semantics is so called because of the rule for λ-abstraction: when a closure is built, a "pointer" is saved in the closure instead of the nonlocal environment. We use the notation $ptr(\bar{\rho})$ to denote a pointer to $\bar{\rho}$. When the body of a fully-applied closure is evaluated, the pointer in the closure establishes the static link. The semantics rule for identifier lookup uses a *lookup* function that performs a top-down stack search that traverses static links. Of course, this explanation makes sense only when there are no "dangling" pointers in the evaluation.

3.2 The Simulation Theorem

To relate Figures 5 and 8, we must relate a pointer-semantics environment to a dynamic-semantics environment. This is done by the function *activeenv*:

$activeenv : \text{Env}_P \times \text{Env}_P \rightarrow \text{Env}_G$

$activeenv(\bar{\rho}_0, <>) = <>$

$activeenv(\bar{\rho}_0, (i = \bar{v}) \,.\, \bar{\rho}) = (i = activeval(\bar{\rho}_0, \bar{v})) \,.\, activeenv(\bar{\rho}_0, \bar{\rho})$

$activeenv(\bar{\rho}_0, p \,.\, \bar{\rho}_2) =$ if $p = ptr(\bar{\rho}_1)$ and $\bar{\rho}_1$ is a subenvironment of $\bar{\rho}_0$
then $activeenv(\bar{\rho}_0, \bar{\rho}_1)$
else undefined

$activeval : \text{Env}_P \times \text{Value}_P \rightarrow \text{Value}_G$

$activeval(\bar{\rho}_0, c) =$ c where c is a base value

$activeval(\bar{\rho}_0, \langle \lambda x.e, \bar{\rho}_l, p \rangle) =$ if $p = ptr(\bar{\rho})$ and $\bar{\rho}$ is a subenvironment of $\bar{\rho}_0$
then $\langle \lambda x.e,\ activelocalenv(\bar{\rho}_0, \bar{\rho}_l),\ activeenv(\bar{\rho}_0, \bar{\rho}) \rangle$
else undefined

$activeval(\bar{\rho}_0, \mu f.\langle \lambda x.e, p \rangle) =$ if $p = ptr(\bar{\rho})$ and $\bar{\rho}$ is a subenvironment of $\bar{\rho}_0$
then $\mu f.\langle \lambda x.e, activeenv(\bar{\rho}_0, \bar{\rho}) \rangle$
else undefined

$$\bar{\rho} \vdash_{ps} c \Rightarrow c \qquad \bar{\rho} \vdash_{ps} x \Rightarrow \bar{v} \text{ if } lookup(x, \bar{\rho}) = \bar{v} \qquad \bar{\rho} \vdash_{ps} \lambda x.e \Rightarrow \langle \lambda x.e, (), ptr(\bar{\rho}) \rangle$$

$$\bar{\rho} \vdash_{ps} f \Rightarrow \langle \lambda x.e, (f = \langle \mu f.\lambda x.e, p \rangle), p \rangle \text{ if } (f = \langle \mu f.\lambda x.e, p \rangle) \in \bar{\rho}$$

$$\frac{\bar{\rho} \vdash_{ps} e_1 \Rightarrow \langle \lambda x.e, \bar{\rho}_l, p \rangle \qquad \bar{\rho} \vdash_{ps} e_2 \Rightarrow \bar{v}}{\bar{\rho} \vdash_{ps} e_1 e_2 \Rightarrow \langle e, (x = \bar{v}).\bar{\rho}_l, p \rangle} \text{ if } e \text{ is a } \lambda\text{-abstraction}$$

$$\frac{\bar{\rho} \vdash_{ps} e_1 \Rightarrow \langle \lambda x.e, \bar{\rho}_l, p \rangle \qquad \bar{\rho} \vdash_{ps} e_2 \Rightarrow \bar{v} \qquad (x = \bar{v}).\bar{\rho}_l \ @p.\bar{\rho} \vdash_{ps} e \Rightarrow \bar{w}}{\bar{\rho} \vdash_{ps} e_1 e_2 \Rightarrow \bar{w}} \text{ otherwise}$$

$$\frac{\bar{\rho} \vdash_{ps} e_1 \Rightarrow \text{true} \qquad \bar{\rho} \vdash_{ps} e_2 \Rightarrow \bar{v}}{\bar{\rho} \vdash_{ps} \text{if } e_1 \ e_2 \ e_3 \Rightarrow \bar{v}} \qquad \frac{\bar{\rho} \vdash_{ps} e_1 \Rightarrow \text{false} \qquad \bar{\rho} \vdash_{ps} e_3 \Rightarrow \bar{v}}{\bar{\rho} \vdash_{ps} \text{if } e_1 \ e_2 \ e_3 \Rightarrow \bar{v}}$$

$$\bar{\rho} \vdash_{ps} \text{rec } f.\lambda x.e \Rightarrow \langle \lambda x.e, (f = \langle \mu f.\lambda x.e, ptr(\bar{\rho}) \rangle), ptr(\bar{\rho}) \rangle$$

Figure 8: Pointer Semantics

$$\begin{array}{lll} activelocalenv : \text{Env}_P \times \text{Local_Env}_P \rightharpoonup \text{Env}_G & & \\ activelocalenv\,(\bar{\rho}_0, <>) & = & <> \\ activelocalenv\,(\bar{\rho}_0, (i = \bar{v}) \ . \ \bar{\rho}_l) & = & (i = activeval(\bar{\rho}_0, \bar{v})) \ . \ activelocalenv\,(\bar{\rho}_0, \bar{\rho}_l) \end{array}$$

Essentially, *activeenv* and *activelocalenv* transform an environment with static links into one without.

If a program satisfies the dynamic criterion, then its evaluation with the dynamic semantics in Figure 5 can be replicated by the pointer semantics of Figure 8 and no dangling references arise. This intuition is captured in the following theorem.

Theorem 1
Let $\rho \vdash e \Rightarrow v$ be the root of a dynamic-semantics tree that satisfies the dynamic criterion. If $\rho = activeenv(\bar{\rho}, \bar{\rho})$, then there exists a pointer-semantics tree $\bar{\rho} \vdash_{ps} e \Rightarrow \bar{v}$ such that $v = activeval(\bar{\rho}, \bar{v})$.

Proofs of this and subsequent theorems are found in [1].

3.3 A Stronger Dynamic Criterion

It is a bit surprising to discover that the dynamic criterion is implied by a criterion that examines only those closures built at certain application nodes.

Criterion 2 (Modified Dynamic Criterion)
A dynamic-semantics tree has the **modified dynamic criterion** *if for every node of the form $\rho \vdash e'e'' \Rightarrow \langle \lambda y.e''', \rho'_l, \rho'_n \rangle$ that was built by the rule:*

$$\frac{\rho \vdash e' \Rightarrow \langle \lambda x.e, \rho_l, \rho_n \rangle \qquad \rho \vdash e'' \Rightarrow v \qquad \rho' \vdash e \Rightarrow \langle \lambda y.e''', \rho'_l, \rho'_n \rangle}{\rho \vdash e'e'' \Rightarrow \langle \lambda y.e''', \rho'_l, \rho'_n \rangle}$$

where ρ' is $(x = v).\rho_l @ \rho_n$, it is the case that $\rho' \neq \rho'_n$.

It is also surprising that we require merely that $\rho' \neq \rho'_n$, but this inequality forces ρ'_n to be a subenvironment of ρ. The non-env-stackable example in Section 2 fails the modified dynamic criterion at Node B.

Theorem 2 (Modified Dynamic Criterion implies Dynamic Criterion) *If ρ_0 is an initial environment, i.e., ρ_0 has no bindings to closures, then if the dynamic-semantics tree $\rho_0 \vdash e_0 \Rightarrow v_0$ satisfies the modified dynamic criterion, then it satisfies the dynamic criterion.*

4 Static Analysis of Stackability

The Dynamic Criterion and the Modified Dynamic Criterion both require the dynamic-semantics tree. When we do a static analysis, we have only the syntax tree, so we must predict what environments will be created at run time. Since env-stackability may be lost only when an application expression evaluates to a closure, (*cf.* the modified dynamic criterion) we concentrate upon learning what set of closures may be generated by the subexpressions in the syntax tree. This is done by closure analysis, which is a static program analysis that approximates the set of textual lambdas that a program point can evaluate to. The approximation is conservative in that the actual evaluation of the expression will yield only a subset of the lambdas that the analysis predicts. We use the closure analysis algorithm developed by Sestoft in [9]. The details, including a soundness theorem, are reviewed in Appendix B. For the purposes of this section, we give a very brief notational summary and an intuitive explanation of soundness.

4.1 Closure Analysis

We begin by labelling all lambdas and variables in the source program so that $\lambda^\ell x$ binds occurrences of x^ℓ. We write $\lambda^\ell x\,.\,e_\ell$ so that e_ℓ is easily identified as the body of the λ-abstraction $\lambda^\ell x$. Also, rec will be treated like λ: write $\mathsf{rec}^{\ell_1} f.\lambda^{\ell_2} x\,.\,e_{\ell_2}$. Finally, we assume that the initial run-time environment cannot contain closures.

Let *Label* denote the set of labels. Given a program e in the source language, we use two *closure descriptions*, ϕ, γ: $CDescription = Label \to \mathcal{P}(Label)$, with the following intended meanings:

$\phi\ell$ is the set of closures that e_ℓ in $\lambda^\ell x\,.\,e_\ell$ can evaluate to.

$\gamma\ell$ is the set of closures that $\lambda^\ell x\,.\,e_\ell$ can be applied to,*i.e.*, the set of closures that $\lambda^\ell x$ can be bound to. ϕ is called the *result closure description* and γ is called the *argument closure description.*

Next, we define two analysis functions, $\mathcal{C}_a$ and $\mathcal{C}_p$, with the following intended meanings:

$\mathcal{C}_a[\![e]\!]\phi\gamma$ is the set of closures that expression e can evaluate to.

$\mathcal{C}_p[\![e]\!]\phi\gamma\ell'$ is the set of closures that $\lambda^{\ell'} x$ can bind to in e.

$\mathcal{C}_a$ is called the *closure analysis function* and $\mathcal{C}_p$ is called the *closure propagation function.*

Let e_0 be the overall expression. We want a solution $(\ \phi\ ,\gamma\ )$ to satisfy the equations

$$(*)\begin{cases} \phi\ell = \mathcal{C}_a[\![e_\ell]\!]\phi\gamma,\ \forall\ell \in Label. \\ \gamma\ell = \mathcal{C}_p[\![e_0]\!]\phi\gamma\ell,\ \forall\ell \in Label. \end{cases}$$

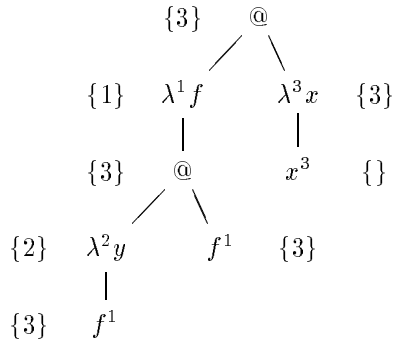
Therefore, what we want is a solution (for expression e_0) of the equation:

$$(\ \phi\ ,\gamma\ ) = (\ \lambda\ell.\mathcal{C}_a[\![e_\ell]\!]\phi\gamma\ ,\mathcal{C}_p[\![e_0]\!]\phi\gamma\ )$$

The definitions of $\mathcal{C}_a$ and $\mathcal{C}_p$ can be found in Appendix B. Suppose, for an environment ρ, and any identifier x^ℓ in its domain, if $\rho(x^\ell)$ evaluates to a closure, say, ℓ', then $\ell' \in \gamma\ell$. Then we say that ρ *is consistent with respect to* γ. With this notion of consistency, we can show that the closure analysis is *sound* with respect to the dynamic semantics of Figure 5. This means that in an environment ρ which is consistent with respect to γ, if an expression e evaluates to a closure corresponding to the λ-abstraction ℓ, then ℓ must have been predicted by the closure analysis, *i.e.* $\ell \in \mathcal{C}_a[\![e]\!]\phi\gamma$. The formal definition of consistency and the soundness theorem appear in Appendix B.

4.2 Example

Below is the labelled syntax tree for the example in Section 3 with its nodes annotated with the closure analysis information calculated by Sestoft's algorithm. The annotation at each node denotes the set of λ-abstractions that the node might produce during its actual evaluation.



For instance, the application $(\lambda^2\, y\, .\, f^1)f^1$ can evaluate to λ-abstraction 3, *i.e.*, to $\lambda^3\, x\, .\, x^3$.

4.3 The Static Criterion for Env-Stackability

Our goal is to use closure analysis to predict when the modified dynamic criterion holds for an evaluation. Since the modified dynamic criterion focuses on the evaluation of applications, we use closure analysis to predict the behavior of the application expressions in a program.

For a program $\pi_0 \vdash e_0 : \tau_0$, its closure analysis (ϕ, γ), and for each application subexpression $e_{i1}e_{i2}$ within e_0, we define these sets:

$$\mathcal{A}_i = \mathcal{C}_a[\![e_{i1}e_{i2}]\!]\phi\gamma$$
$$\mathcal{O}_i = \mathcal{C}_a[\![e_{i1}]\!]\phi\gamma$$

$\mathcal{A}_i$ denotes the set of labels of the λ-abstractions that the application $e_{i1}e_{i2}$ can evaluate to, and $\mathcal{O}_i$ denotes the set of labels of the λ-abstractions that the operator e_{i1} can evaluate to.

Say that $e_{i1}e_{i2}$ evaluates as follows:

$$\frac{\rho \vdash e_{i1} \Rightarrow \langle \lambda^{\ell}\, x\, .\, e_{\ell}, \rho_l, \rho_n\rangle \qquad \rho \vdash e_{i2} \Rightarrow v \qquad \rho' \vdash e_{\ell} \Rightarrow \langle \lambda^{\ell'}\, y\, .\, e_{\ell'}, \rho'_l, \rho'_n\rangle}{\rho \vdash e_{i1}e_{i2} \Rightarrow \langle \lambda^{\ell'}\, y\, .\, e_{\ell'}, \rho'_l, \rho'_n\rangle}$$

where ρ' is $(x^{\ell} = v).\rho_l@\rho_n$.

By the soundness of closure analysis, $\ell' \in \mathcal{A}_i$ and $\ell \in \mathcal{O}_i$. The subject reduction property (*cf.* Appendix A) ensures us that the typing derivation

$$\pi'_l \oplus \pi'_n \vdash \lambda^{\ell'}\, y\, .\, e_{\ell'} : \tau'_1 \rightarrow \tau'_2$$

correctly predicts that $\models \rho'_l : \pi'_l$ and that $\models \rho'_n : \pi'_n$. (Read $\models \rho : \pi$ as "ρ has type π". See Appendix A.) Similarly, the typing derivation

$$\pi_l \oplus \pi_n \vdash \lambda^{\ell}\, x\, .\, e_{\ell} : \tau_1 \rightarrow \tau_2$$

correctly predicts that $\models \rho_l : \pi_l$ and that $\models \rho_n : \pi_n$. Since the modified dynamic criterion merely demands that we verify $(x^{\ell} = v).\rho_l@\rho_n \neq \rho'_n$, we can do this with τ, π_l, π_n and π'_n:

$$(x^{\ell} : \tau) \oplus \pi_l \oplus \pi_n \neq \pi'_n$$

This inequality implies the modified dynamic criterion. Therefore, we can use closure analysis to enforce a sufficient static criterion that implies the modified dynamic criterion.

Now we formalize the above. First, given a labelled program $\pi_0 \vdash e_0 : \tau_0$ we define:

Definition 1 (Maximal λ-abstraction)
A λ-abstraction $\lambda^{\ell_1}x_1 \cdots \lambda^{\ell_n}x_n.e_{\ell_n}$ is **maximal** *in e_0 if there does not exist an expression $\lambda^{\ell_0}x_0.\lambda^{\ell_1}x_1 \cdots \lambda^{\ell_n}x_n.e_{\ell_n}$ in e_0.*

Let $abs(\ell)$ denote the abstract syntax tree for the λ-abstraction labelled ℓ, namely, $\lambda^{\ell}\, x\, .\, e_{\ell}$. Let $max(abs(\ell))$ denote the maximal λ-abstraction containing $abs(\ell)$, namely, $\lambda^{\ell_1}x_1.\cdots\lambda^{\ell_i}x_i\lambda^{\ell}x.e_{\ell}$. As above, for every application $e_{i1}e_{i2}$ in e_0, define $\mathcal{A}_i = \mathcal{C}_a[\![e_{i1}e_{i2}]\!]\phi\gamma$ and $\mathcal{O}_i = \mathcal{C}_a[\![e_{i1}]\!]\phi\gamma$.

Criterion 3 (Static Criterion)
For all expressions $\pi \vdash e : \tau$, for closure analysis (ϕ, γ), for $\models \rho : \pi$, and for ρ consistent with respect to γ, e is **statically stackable** *if for every application, $e_{i1}e_{i2}$, in e and its associated sets $\mathcal{A}_i$, $\mathcal{O}_i$, for all $\ell' \in \mathcal{A}_i$, for all $\ell \in \mathcal{O}_i$,*

if $\lambda^{\ell_1} x_1. \cdots \lambda^{\ell_i} x_i \lambda^{\ell'} y.e_{\ell'} = max(abs(\ell'))$, *and* $\lambda^{\ell} x\,.\,e_{\ell} = abs(\ell)$, *and* $\pi_a \vdash \lambda^{\ell_1} x_1. \cdots \lambda^{\ell_i} x_i \lambda^{\ell'} y.e_{\ell'} : \tau_1' \rightarrow \tau_2'$, *and* $\pi_o \vdash \lambda^{\ell} x\,.\,e_{\ell} : \tau_1 \rightarrow \tau_2$, *then* $(x^{\ell} : \tau) \oplus \pi_o \neq \pi_a$.

This criterion restates the ideas explained earlier: closure analysis predicts what the components of an application evaluate to, and by soundness, the type assignment predicts the environments that will appear during evaluation. Therefore, a statically-stackable expression must satisfy the modified dynamic criterion.

4.4 Example

Consider again the example in Section 3. Let its initial type assignment be π. Its labelled abstract syntax tree annotated with closure analysis information, is shown in Section 4.2.

Consider the application $(\lambda^1 f\ : int \rightarrow int.\ (\lambda^2 y\ : int \rightarrow int.\ f^1) f^1)(\lambda^3 x\ : int.\ x^3)$. Closure analysis predicts that the application must evaluate to the closure representing λ-abstraction 3. This λ-abstraction has type assignment π and is maximal. Consider now the operator part of the application, namely, $\lambda^1 f\ : int \rightarrow int.\ (\lambda^2 y\ : int \rightarrow int.\ f^1) f^1$. Closure analysis predicts that it must evaluate to the closure representing λ-abstraction 1. The type assignment for λ-abstraction 1 is also π. Note that $(f^1 : int \rightarrow int) \oplus \pi \neq \pi$. Thus the static criterion is satisfied for this application.

Consider the other application $(\lambda^2 y\ : int \rightarrow int.\ f^1) f^1$. Closure analysis predicts that the application must evaluate to the closure representing λ-abstraction 3. This λ-abstraction has type assignment π and is maximal. Consider now the operator part of the application, namely, $\lambda^2 y\ : int \rightarrow int.\ f^1$. Closure analysis predicts that it must evaluate to the closure representing λ-abstraction 2. The type assignment for λ-abstraction 2 is $(f^1 : int \rightarrow int) \oplus \pi$. Note that $(y^2 : int \rightarrow int, f^1 : int \rightarrow int) \oplus \pi \neq \pi$. Thus the static criterion is satisfied for this application. Since the static criterion is satisfied for all applications, it holds for the overall expression.

Theorem 3 (Safety of Static Analysis)
For all $\pi_0 \vdash e_0 : \tau_0$, *for closure analysis* (ϕ, γ), *for an environment* ρ_0 *such that* $\models \rho_0 : \pi_0$ *and* ρ_0 *is consistent with respect to* γ, *if* $\pi_0 \vdash e_0 : \tau_0$ *is statically-stackable, then it satisfies the modified dynamic criterion.*

Proof:
Let $\pi_0 \vdash e_0 : \tau_0$ and let (ϕ, γ) be its closure analysis. Let $\models \rho_0 : \pi_0$, and let ρ_0 be consistent with respect to γ and $\rho_0 \vdash e_0 \Rightarrow v_0$.

For the modified dynamic criterion to be satisfied by $\rho_0 \vdash e_0 \Rightarrow v_0$, we must examine every subtree of the form

$$\frac{\rho \vdash e_{i1} \Rightarrow \langle \lambda^{\ell} x\,.\,e_{\ell}, \rho_l, \rho_n \rangle \quad \rho \vdash e_{i2} \Rightarrow v \quad \rho' \vdash e_{\ell} \Rightarrow \langle \lambda^{\ell'} y\,.\,e_{\ell'}, \rho_l', \rho_n' \rangle}{\rho \vdash e_{i1} e_{i2} \Rightarrow \langle \lambda^{\ell'} y\,.\,e_{\ell'}, \rho_l', \rho_n' \rangle}$$

in $\rho_0 \vdash e_0 \Rightarrow v_0$, where $\rho' = (x = v).\rho_l @ \rho_n$, and must ensure that $\rho' \neq \rho_n'$.

Since $\rho \vdash e_{i1} \Rightarrow \langle \lambda^{\ell}\, x\, .\, e_{\ell}, \rho_l, \rho_n \rangle$, therefore, by soundness of closure analysis, $\ell \in \mathcal{O}_i$. By subject reduction we know that there exist type assignments π_l and π_n such that $\pi_l \oplus \pi_n \vdash \lambda^{\ell}\, x\, .\, e_{\ell} : \tau_1 \rightarrow \tau_2$, $\mathrm{Dom}(\pi_l) = \mathrm{Dom}(\rho_l)$ and $\mathrm{Dom}(\pi_n) = \mathrm{Dom}(\rho_n)$. Let $\pi = \pi_l \oplus \pi_n$. Further, since $\rho \vdash e_{i1} e_{i2} \Rightarrow \langle \lambda^{\ell'}\, y\, .\, e_{\ell'}, \rho'_l, \rho'_n \rangle$, therefore, $\ell' \in \mathcal{A}_i$.
Again by subject reduction we know that there exist type assignments π'_l and π'_n such that $\pi'_l \oplus \pi'_n \vdash \lambda^{\ell'}\, y\, .\, e_{\ell'} : \tau'_1 \rightarrow \tau'_2$, $\mathrm{Dom}(\pi'_l) = \mathrm{Dom}(\rho'_l)$ and $\mathrm{Dom}(\pi'_n) = \mathrm{Dom}(\rho'_n)$. Consider $max(abs(\ell')) = \lambda^{\ell_1} x_1 . \cdots \lambda^{\ell_i} x_i \lambda^{\ell'} y . e'_{\ell}$. Therefore, $\pi'_n \vdash \lambda^{\ell_1} x_1 . \cdots \lambda^{\ell_i} x_i \lambda^{\ell'} y . e'_{\ell} : \tau$, where $\{x_1^{\ell_1} : \tau_1, \cdots x_i^{\ell_i} : \tau_i\} = \pi'_l$.
By the static criterion we know that $(x^{\ell} : \tau) \oplus \pi \neq \pi'_n$. Hence $\rho' \neq \rho'_n$. □

5 Related Work

As has already been mentioned, Georgeff's approach seems to be the first in trying to syntactically characterize stackable expressions.

Another line of research, most notably that of Goldberg and Park [3], uses abstract interpretation-based escape analysis to detect stackability of environments. However, not all simple expressions are detected as stackable in their approach.

As noted in Section 1.1, Tofte and Talpin have used effect inference (type inference, region inference) to overlay block-structured extents, `letregion` ρ `in` e `end`, on an arbitrary program [10]. This allows them to translate the source language into a region-annotated target language at compile-time. In spirit, the target programs are env-stackable, but the run-time stack is a stack of heap regions, and static links can be dangling. (A soundness theorem ensures that dangling links are never traversed.) Finally, note that scope is controlled by λ, and extent is controlled by `letregion`. For these reasons our approach and Tofte-Talpin's are not readily comparable.

6 Conclusion

We have developed a statically checkable criterion to detect stackability of environments for a call-by-value λ-calculus. Next, we wish to consider the extension of our ideas to higher-order, call-by-value imperative languages and explore the relationship between stack-based environments and stack-based stores as arises in a compiler-based implementation.

Acknowledgement: We wish to thank Allen Stoughton for stimulating discussions. Thanks also to Mads Tofte, Mitch Wand, and other attendees of the Atlantique workshop on Semantics-Based Program Manipulation for their questions, comments and interest. Finally, thanks to Olivier Danvy and Pascal Fradet for their comments on an earlier draft.

References

[1] Anindya Banerjee and David A. Schmidt. Stackability in the Typed Call-by-value Lambda Calculus. Technical Report TR-94-6, Kansas State University, Manhattan, Kansas, U.S.A., 1994.

[2] Michael Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Transactions on Programming Languages and Systems*, 6(4):603–631, October 1984.

[3] Benjamin Goldberg and Young Gil Park. Higher order escape analysis:optimizing stack allocation in functional program implementations. In N.Jones, editor, *Proceedings of the Third European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 152–160, Copenhagen, Denmark, May 1990.

[4] Gilles Kahn. Natural semantics. Technical Report 601, INRIA, Sophia Antipolis, France, February 1987.

[5] Jean-Louis Krivine. *Lambda Calculus, Types, and Models.* Ellis-Horwood, Chichester, 1993.

[6] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 17:209–220, 1992.

[7] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1988.

[8] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[9] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, Copenhagen, Denmark, October 1991. Rapport Nr. 92/6.

[10] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-value λ-calculus using a Stack of Regions. In *Proceedings of the Twentyfirst Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

$\pi \in \text{TypeEnv}$ $\quad x \in \text{Identifier}$ $\quad \pi ::= \{x_i : \tau_i\}_{i \geq 0}$

$c \in \text{BaseValue}$ $\quad v \in \text{Value}$ $\quad v ::= c \mid \langle \lambda x.e, \rho \rangle \mid \mu f.\langle \lambda x.e, \rho \rangle$

$\rho \in \text{Env}$ $\quad \rho ::= <> \mid (i = v).\rho$

Figure 9: Semantic Objects for the Standard Dynamic Semantics

$$\rho \triangleright c \Rightarrow c \qquad \rho \triangleright x \Rightarrow v \text{ if } (x = v) \in \rho \qquad \rho \triangleright \lambda x.e \Rightarrow \langle \lambda x.e, \rho \rangle$$

$$\rho \triangleright f \Rightarrow \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, \rho' \rangle).\rho' \rangle, \text{ if } (f = \mu f.\langle \lambda x.e, \rho' \rangle) \in \rho$$

$$\frac{\rho \triangleright e_1 \Rightarrow \langle \lambda x.e, \rho' \rangle \qquad \rho \triangleright e_2 \Rightarrow v \qquad (x = v).\rho' \triangleright e \Rightarrow w}{\rho \triangleright e_1 e_2 \Rightarrow w}$$

$$\frac{\rho \triangleright e_1 \Rightarrow \text{true} \qquad \rho \triangleright e_2 \Rightarrow v}{\rho \triangleright \text{if } e_1\ e_2\ e_3 \Rightarrow v} \qquad \frac{\rho \triangleright e_1 \Rightarrow \text{false} \qquad \rho \triangleright e_3 \Rightarrow v}{\rho \triangleright \text{if } e_1\ e_2\ e_3 \Rightarrow v}$$

$$\rho \triangleright \text{rec } f.\lambda x.e \Rightarrow \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, \rho \rangle).\rho \rangle$$

Figure 10: Standard Dynamic Semantics

Appendix

A Subject Reduction and the Equivalence of Georgeff's Dynamic Semantics and the Standard Dynamic Semantics

We can show that subject reduction holds for the Georgeff-style dynamic semantics. We want a relation $\models \subseteq \text{Value}_G \times \text{Type}$ that satisfies $\models v : \tau$ iff:

(i) if v is c then $\models v : \tau$.

(ii) if v is $\langle \lambda x.e, \rho_l, \rho_n \rangle$ then there exist type assignments π_l and π_n such that $\pi_l \oplus \pi_n \vdash \lambda x.e : \tau$, $\text{Dom}(\pi_i) = \text{Dom}(\rho_i)$ and $\models \rho_i(x) : \pi_i(x)$ for all $x \in \text{Dom}(\rho_i)$, for $i \in \{l, n\}$.

(iii) if v is $\mu f.\langle \lambda x.e, \rho \rangle$ then there exists a type assignment π such that $\pi \vdash \text{rec } f.\lambda x.e : \tau$, $\text{Dom}(\pi) = \text{Dom}(\rho)$, and $\models \rho(x) : \pi(x)$ for all $x \in \text{Dom}(\rho)$.

We choose $\models$ to be the least fixpoint of the corresponding monotonic functional. Then we have the following theorem:

Theorem 4 (Subject Reduction Property for Georgeff-style Semantics) *If* $\pi \vdash e : \tau$, $\models \rho : \pi$, *and* $\rho \vdash e \Rightarrow v$, *then* $\models v : \tau$.

The semantic objects and the standard dynamic (natural [4]) semantics for the call-by-value, simply-typed λ-calculus are shown in Figures 9 and 10.

To show that it suffices to work with the Georgeff-style dynamic semantics exclusively, we prove that it is equivalent to the standard dynamic semantics.

We create a *correspondence relation* [6], *rel*, that relates Figures 5 and 10. If $\rho \triangleright e \Rightarrow v$ and $\rho' \vdash e \Rightarrow v'$, then v *rel* v' must satisfy:

(i) if $\models v : \iota$ then $v = v'$.

(ii) if $v = \langle \lambda x.e, \rho \rangle$ then there exist environments ρ_l and ρ_n such that $v' = \langle \lambda x.e, \rho_l, \rho_n \rangle$, $\mathrm{Dom}(\rho) = \mathrm{Dom}(\rho_l @ \rho_n)$ and $\rho(x)$ *rel* $(\rho_l @ \rho_n)(x)$ for all $x \in \mathrm{Dom}(\rho)$.

(iii) if $v = \mu f.\langle \lambda x.e, \rho \rangle$ then there exists an environment ρ' such that $v' = \mu f.\langle \lambda x.e, \rho' \rangle$, $\mathrm{Dom}(\rho) = \mathrm{Dom}(\rho')$ and $\rho(x)$ *rel* $\rho'(x)$

Note that *rel* as stated above is a property, rather than a definition, because of the existential quantification. We define *rel* as the least fixpoint of the monotonic operator induced by clauses (i) to (iii).

Theorem 5
If ρ rel ρ' then $\rho \triangleright e \Rightarrow v$ iff $\rho' \vdash e \Rightarrow v'$, where v rel v'.

Thus it suffices to work with the Georgeff-style semantics.

B Closure Analysis

The functions $\mathcal{C}_a$ and $\mathcal{C}_p$ mentioned in Section 4.1 are defined as follows:

$\mathcal{C}_a : \mathit{Expression} \rightarrow \mathit{CDescription} \rightarrow \mathit{CDescription} \rightarrow \mathcal{P}(\mathit{Label})$

$$
\begin{array}{lcl}
\mathcal{C}_a[\![c]\!]\phi\gamma & = & \{\} \\
\mathcal{C}_a[\![x^{\ell}]\!]\phi\gamma & = & \gamma\ell \\
\mathcal{C}_a[\![\lambda^{\ell} x.e_{\ell}]\!]\phi\gamma & = & \{\ell\} \\
\mathcal{C}_a[\![e_1 e_2]\!]\phi\gamma & = & \bigcup_{\ell \in \mathcal{C}_a[\![e_1]\!]\phi\gamma} \phi\ell \\
\mathcal{C}_a[\![\mathsf{if}\ e_1\ e_2\ e_3]\!]\phi\gamma & = & \mathcal{C}_a[\![e_2]\!]\phi\gamma \ \cup \ \mathcal{C}_a[\![e_3]\!]\phi\gamma \\
\mathcal{C}_a[\![\mathsf{rec}^{\ell_1}\ f.\ \lambda^{\ell_2}\ x\ .\ e_{\ell_2}]\!]\phi\gamma & = & \{\ell_2\}
\end{array}
$$

$\mathcal{C}_p : \mathit{Expression} \rightarrow \mathit{CDescription} \rightarrow \mathit{CDescription} \rightarrow \mathit{Label} \rightarrow \mathcal{P}(\mathit{Label})$

$$
\begin{array}{lcl}
\mathcal{C}_p[\![c]\!]\phi\gamma\ell' & = & \{\} \\
\mathcal{C}_p[\![x^{\ell}]\!]\phi\gamma\ell' & = & \{\} \\
\mathcal{C}_p[\![\lambda^{\ell}\ x\ .\ e_{\ell}]\!]\phi\gamma\ell' & = & \mathcal{C}_p[\![e_{\ell}]\!]\phi\gamma\ell' \\
\mathcal{C}_p[\![e_1 e_2]\!]\phi\gamma\ell' & = & \begin{cases} \mathcal{C}_p[\![e_1]\!]\phi\gamma\ell' \ \cup \ \mathcal{C}_p[\![e_2]\!]\phi\gamma\ell', & \text{if } \ell' \notin \mathcal{C}_a[\![e_1]\!]\phi\gamma \\ \mathcal{C}_p[\![e_1]\!]\phi\gamma\ell' \ \cup \ \mathcal{C}_p[\![e_2]\!]\phi\gamma\ell' \ \cup \mathcal{C}_a[\![e_2]\!]\phi\gamma, & \text{otherwise} \end{cases} \\
\mathcal{C}_p[\![\mathsf{if}\ e_1\ e_2\ e_3]\!]\phi\gamma\ell' & = & \mathcal{C}_p[\![e_1]\!]\phi\gamma\ell' \ \cup \ \mathcal{C}_p[\![e_2]\!]\phi\gamma\ell' \ \cup \ \mathcal{C}_p[\![e_3]\!]\phi\gamma\ell' \\
\mathcal{C}_p[\![\mathsf{rec}^{\ell_1}\ f.\ \lambda^{\ell_2}\ x\ .\ e_{\ell_2}]\!]\phi\gamma\ell' & = & \mathcal{C}_p[\![e_{\ell_2}]\!]\phi\gamma\ell'
\end{array}
$$

Definition 2 (Consistency of Environments)
For closure description γ, type assignment π, and environment ρ, where $\models \rho : \pi$, ρ is consistent with respect to γ iff for all $z^m \in \mathrm{Dom}(\rho)$. if $\rho(z^m) = \langle \lambda^{\ell} x\ .\ e_{\ell}, \rho_l, \rho_n \rangle$, then (i)$\gamma m \supseteq \{\ell\}$ and (ii) ρ_l and ρ_n are consistent with respect to γ.

Theorem 6 (Soundness)
For all expressions e, for all (ϕ, γ), where (ϕ, γ) is a solution of the equation $(\phi, \gamma) = (\lambda\ell.\mathcal{C}_a[\![e_{\ell}]\!]\phi\gamma, \mathcal{C}_p[\![e]\!]\phi\gamma)$, for all ρ, if ρ is consistent with respect to γ, then for all natural semantics trees $\rho \vdash e \Rightarrow v$, if v is a closure, say, $\langle \lambda^{\ell} x\ .\ e_{\ell}, \rho_l, \rho_n \rangle$, then $\ell \in \mathcal{C}_a[\![e]\!]\phi\gamma$.