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.

*Expanded version of the paper presented at the first International Static Analysis Symposium, Namur, Belgium, September 1994.

†Current address: Laboratoire d'Informatique, Ecole Polytechnique, 91128 Palaiseau cedex France. e-mail:banerjee@lix.polytechnique.fr. Phone: (+33)1.69.33.37.98, Fax: (+33)1.69.33.30.14

‡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 $\mathsf{begin\ D\ in\ U\ end}$ 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 evaluation of the body of the λ-abstraction produces the value $\lambda y.x$. This is the final result of evaluation, but the binding of x to 2 cannot be freed. Apparently one must abandon the stack environment for a heap environment.

A novel alternative was recently proposed by Tofte and Talpin [13], who apply an *effects analysis* to determine the extents of bindings. Then, they insert constructs of the form $\mathsf{letregion}\ \rho\ \mathsf{in}\ e\ \mathsf{end}$ into the program. These constructs control extents: ρ allocates storage in advance for the bindings made within e. The resulting programs are verbose, and the underlying implementation is a stack of heaps rather than a classical stack.

In this paper, rather than altering an arbitrary program and running it 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 [11] and the Krivine-machine [7]; 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 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*.

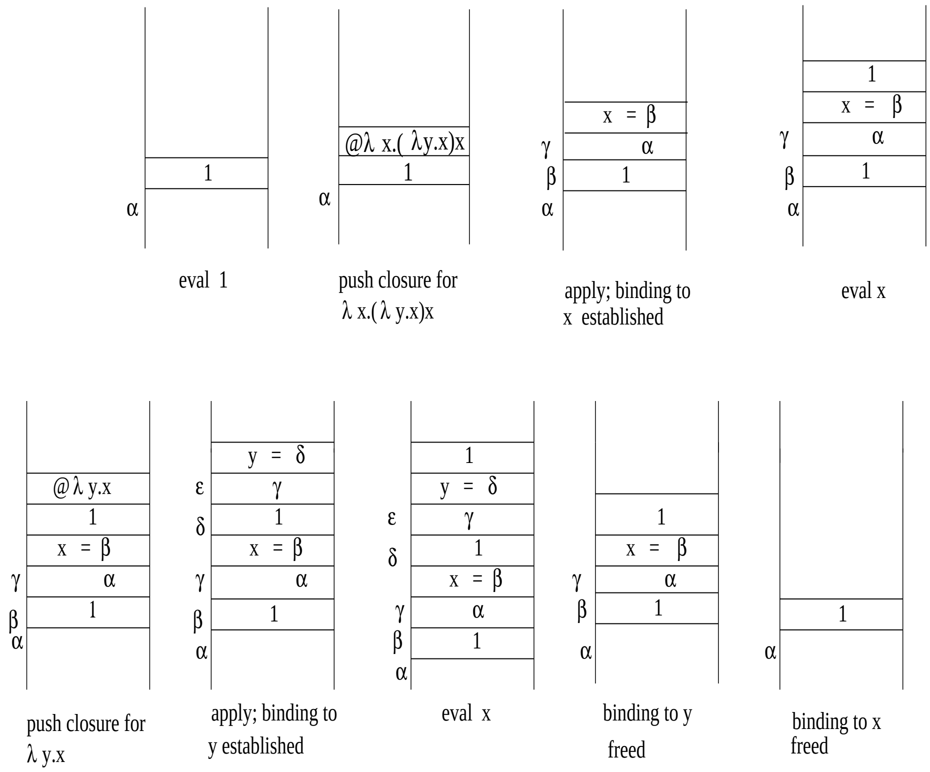


Figure 1: Snapshots of the environment stack for $(\lambda x.(\lambda y.x)x)1$

2 Simple Expressions and Partially-Applied Closures

The BNF and static semantics 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 $\tau_1 \to \tau_2 \to \cdots \to \tau_n$, where τ_n is a base type, should be read as an abbreviation for $\lambda x_1.\lambda x_2 \ldots \lambda x_n.e\ x_1\ x_2 \ldots x_n$, (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, ..., x_n are established, as a group, only when the body of the λ-abstraction, a phrase of base type, is 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 [10].

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 [3] noted that decurrying can be simulated 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
$(\lambda x_1.\lambda x_2 \ldots \lambda x_i.\lambda x_{i+1} \ldots \lambda x_n.e_0)e_1 e_2 \ldots e_i$ evaluates to
$\langle \lambda x_{i+1} \ldots \lambda x_n.e_0, (x_i = v_i, \ldots, x_2 = v_2, x_1 = v_1), \rho_0\rangle$, 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

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

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

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

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

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

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

Figure 3: Static Semantics

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. [2] 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 env-stackable, but Georgeff identified a class of expressions, called the *simple expressions*, which 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 base type. 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 $((\lambda\, x : int \,.\, \lambda\, f : int \to int \,.\, fx)0)(\lambda\, y : int \,.\, y)$. The turnstiles are numbered to indicate the order of evaluation steps.

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

[2]Note that there are two rules for identifier lookup. Identifiers which name recursive functions are bound to recursive closures of the form $\mu f.\langle \lambda x.e, \rho \rangle$ in the environment. Their evaluation entails one unfolding of the recursion.

$$
\begin{array}{lll}
c \in \text{BaseValue} & & \\
v \in \text{Value}_G & & v ::= c \mid \langle \lambda x.e, \rho_l, \rho_n \rangle \mid \mu f.\langle \lambda x.e, \rho \rangle \\
\rho \in \text{Env}_G & & \rho ::= <> \mid (i = v).\rho
\end{array}
$$

Figure 4: Semantic Objects for Georgeff-style Dynamic Semantics

$$\rho \vdash c \Rightarrow c \qquad\qquad \rho \vdash \lambda x.e \Rightarrow \langle \lambda x.e, (), \rho \rangle$$

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

$$\rho \vdash x \Rightarrow v \ \text{ if } (x = v) \in \rho, \ \text{otherwise}$$

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

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

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

Figure 5: Dynamic Semantics, Georgeff-style

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, one can 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 6, a static link to ρ and the bindings for f and x are established (pushed) onto the stack. 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
$((\lambda\, x : \mathit{int}\,.\,(\lambda\, f : \mathit{int} \rightarrow \mathit{int}\,.\,f)(\lambda\, y : \mathit{int}\,.\,x))5)6$:

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

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

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

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

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

Figure 6: Simple Expressions

(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 $(\lambda\, f : int \rightarrow int\, .\, (\lambda\, y : int \rightarrow int\, .\, f)f)(\lambda\, x : int\, .\, x)$, 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.

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.

$$p \in \text{Pointer} \qquad \bar{\rho} \in \text{Env}_P$$
$$\bar{\rho}_l \in \text{Local\_Env}_P \quad \bar{v} \in \text{Value}_P$$

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

Figure 7: Semantic Objects for Pointer Semantics

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

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

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

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

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

Figure 8: Pointer Semantics

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*:

$$\begin{array}{ll}
activeenv \;:\; \text{Env}_P \times \text{Env}_P \to \text{Env}_G & \\
activeenv(\bar{\rho}_0, <>) & = <> \\
activeenv(\bar{\rho}_0, (i = \bar{v}) \;.\; \bar{\rho}) & = (i = activeval(\bar{\rho}_0, \bar{v}))\;.\; activeenv(\bar{\rho}_0, \bar{\rho}) \\
activeenv(\bar{\rho}_0, p \;.\; \bar{\rho}_2) & = \text{if } p = ptr(\bar{\rho}_1) \text{ and } \bar{\rho}_1 \text{ is a subenvironment of } \bar{\rho}_0 \\
& \quad \text{then } activeenv(\bar{\rho}_0, \bar{\rho}_1) \\
& \quad \text{else undefined}
\end{array}$$

$$
\begin{array}{ll}
\mathit{activeval} \ : \ \mathrm{Env}_P \times \mathrm{Value}_P \rightarrow \mathrm{Value}_G & \\
\mathit{activeval}(\bar{\rho}_0, c) & = c \text{ where } c \text{ is a base value} \\
\mathit{activeval}(\bar{\rho}_0, \langle \lambda x.e, \bar{\rho}_l, p \rangle) & = \text{if } p = \mathit{ptr}(\bar{\rho}) \text{ and } \bar{\rho} \text{ is a subenvironment of } \bar{\rho}_0 \\
& \quad \text{then } \langle \lambda x.e, \mathit{activelocalenv}(\bar{\rho}_0, \bar{\rho}_l), \mathit{activeenv}(\bar{\rho}_0, \bar{\rho}) \rangle \\
& \quad \text{else undefined} \\
\\
\mathit{activeval}(\bar{\rho}_0, \mu f. \langle \lambda x.e, p \rangle) & = \text{if } p = \mathit{ptr}(\bar{\rho}) \text{ and } \bar{\rho} \text{ is a subenvironment of } \bar{\rho}_0 \\
& \quad \text{then } \mu f. \langle \lambda x.e, \mathit{activeenv}(\bar{\rho}_0, \bar{\rho}) \rangle \\
& \quad \text{else undefined}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{activelocalenv} \ : \ \mathrm{Env}_P \times \mathrm{Local\_Env}_P \rightarrow \mathrm{Env}_G & & \\
\mathit{activelocalenv}(\bar{\rho}_0, <>) & = & <> \\
\mathit{activelocalenv}(\bar{\rho}_0, (i = \bar{v}) \ . \ \bar{\rho}_l) & = & (i = \mathit{activeval}(\bar{\rho}_0, \bar{v})) \ . \ \mathit{activelocalenv}(\bar{\rho}_0, \bar{\rho}_l)
\end{array}
$$

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. For $\bar{\rho} \in \mathrm{Env}_P$, if $\rho = \mathit{activeenv}(\bar{\rho}, \bar{\rho})$, then there exists a pointer-semantics tree $\bar{\rho} \vdash_{ps} e \Rightarrow \bar{v}$ such that $v = \mathit{activeval}(\bar{\rho}, \bar{v})$.

Proof: By induction on the depth of inference in the dynamic-semantics tree. Assume $\rho = \mathit{activeenv}(\bar{\rho}, \bar{\rho})$. Then we have the following cases:

- $\rho \vdash c \Rightarrow c$: Consider $\bar{\rho} \vdash_{ps} c \Rightarrow c$. $c = \mathit{activeval}(\bar{\rho}, c)$, by definition.

- $\rho \vdash x \Rightarrow v$: It must be the case that $(x = v) \in \rho$. Since $\rho = \mathit{activeenv}(\bar{\rho}, \bar{\rho})$, it must be the case that there exists $(x = \bar{v}) \in \bar{\rho}$, such that $v = \mathit{activeval}(\bar{\rho}, \bar{v})$. Hence $\bar{\rho} \vdash_{ps} x \Rightarrow \bar{v}$.

- $\rho \vdash f \Rightarrow \langle \lambda x.e, (f = \mu f. \langle \lambda x.e', \rho' \rangle), \rho' \rangle$:
  It must be the case that $(f = \mu f. \langle \lambda x.e', \rho' \rangle) \in \rho$. Since $\rho = \mathit{activeenv}(\bar{\rho}, \bar{\rho})$, there must exist $\bar{v}$ such that $\mathit{activeval}(\bar{\rho}, \bar{v}) = \mu f. \langle \lambda x.e', \rho' \rangle$. Such a $\bar{v}$ is $\mu f. \langle \lambda x.e', \bar{\rho'} \rangle$, where $\rho' = \mathit{activeenv}(\bar{\rho}, \bar{\rho'})$. Hence,
  $\bar{\rho} \vdash_{ps} f \Rightarrow \langle \lambda x.e, (f = \mu f. \langle \lambda x.e', \bar{\rho'} \rangle), \bar{\rho'} \rangle$.

- $\rho \vdash \lambda x.e \Rightarrow \langle \lambda x.e, (), \rho \rangle$: Consider $\bar{\rho} \vdash_{ps} \lambda x.e \Rightarrow \langle \lambda x.e, (), \mathit{ptr}(\bar{\rho}) \rangle$.
  $\mathit{activeval}(\bar{\rho}, \langle \lambda x.e, (), \mathit{ptr}(\bar{\rho}) \rangle)$
  $= \langle \lambda x.e, (), \mathit{activeenv}(\bar{\rho}, \bar{\rho}) \rangle$
  $= \langle \lambda x.e, (), \rho \rangle$.

- $\rho \vdash \mathsf{rec}\ f.\lambda x.e \Rightarrow \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, \rho\rangle), \rho\rangle$:
  Consider $\bar{\rho} \vdash_{ps} \mathsf{rec}\ f.\lambda x.e \Rightarrow \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, ptr(\bar{\rho})\rangle), ptr(\bar{\rho})\rangle$.
  $activeval(\bar{\rho}, \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, ptr(\bar{\rho})\rangle), ptr(\bar{\rho})\rangle)$
  $= \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, activeenv(\bar{\rho}, ptr(\bar{\rho}))\rangle), activeenv(\bar{\rho}, ptr(\bar{\rho}))\rangle$
  $= \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, activeenv(\bar{\rho}, \bar{\rho})\rangle), activeenv(\bar{\rho}, \bar{\rho})\rangle$
  $= \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, \bar{\rho}\rangle), \bar{\rho}\rangle$

These complete the axiom cases. For the induction step we have the following cases:

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

  Since $\rho = activeenv(\bar{\rho}, \bar{\rho})$, by the induction hypothesis on the tree for e_1, there exists $\bar{\rho} \vdash_{ps} e_1 \Rightarrow \bar{v'}$ such that $\langle \lambda x.\lambda y.e, \rho_l, \rho_n\rangle = activeval(\bar{\rho}, \bar{v'})$. Clearly, $activeval(\bar{\rho}, \bar{v'})$ must have form $\langle \lambda x.\lambda y.e, \bar{\rho_l}, p\rangle$, where p has form $ptr(\bar{\rho_n})$ and $\bar{\rho_n}$ is a subenvironment of $\bar{\rho}$. Also, $\rho_l = activelocalenv(\bar{\rho}, \bar{\rho_l})$ and $\rho_n = activeenv(\bar{\rho}, p) = activeenv(\bar{\rho}, ptr(\bar{\rho_n})) = activeenv(\bar{\rho}, \bar{\rho_n})$.

  Similarly, by the induction hypothesis on the tree for e_2, there exists $\bar{\rho} \vdash_{ps} e_2 \Rightarrow \bar{v}$, such that $v = activeval(\bar{\rho}, \bar{v})$.

  Hence the derivation:

$$\frac{\bar{\rho} \vdash_{ps} e_1 \Rightarrow \langle \lambda x.\lambda y.e, \bar{\rho_l}, p\rangle \qquad \bar{\rho} \vdash_{ps} e_2 \Rightarrow \bar{v}}{\bar{\rho} \vdash_{ps} e_1 e_2 \Rightarrow \langle \lambda y.e, (x = \bar{v}).\bar{\rho_l}, p\rangle}$$

  exists in the pointer semantics.

  Now: $activelocalenv(\bar{\rho}, (x = \bar{v}).\bar{\rho_l})$
  $= (x = activeval(\bar{\rho}, \bar{v})).activelocalenv(\bar{\rho}, \bar{\rho_l})$
  $= (x = v).\rho_l$.

  Hence, $activeval(\bar{\rho}, \langle \lambda y.e, (x = \bar{v}).\bar{\rho_l}, p\rangle$
  $= activeval(\bar{\rho}, \langle \lambda y.e, (x = \bar{v}).\bar{\rho_l}, ptr(\bar{\rho_n})\rangle$
  $= \langle \lambda y.e, (x = v).\rho_l, \rho_n\rangle$.

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

  Since $\rho = activeenv(\bar{\rho}, \bar{\rho})$, therefore, by the induction hypothesis on the tree for e_1, there exists $\bar{\rho} \vdash_{ps} e_1 \Rightarrow \bar{v'}$ such that $\langle \lambda x.e, \rho_l, \rho_n\rangle = activeval(\bar{\rho}, \bar{v'})$. Clearly, $activeval(\bar{\rho}, \bar{v'})$ must have form $\langle \lambda x.e, \bar{\rho_l}, p\rangle$, where p has form $ptr(\bar{\rho_n})$ and $\bar{\rho_n}$ is a subenvironment of $\bar{\rho}$. Also, it must be the case that $\rho_l = activelocalenv(\bar{\rho}, \bar{\rho_l})$ and $\rho_n = activeenv(\bar{\rho}, p) = activeenv(\bar{\rho}, ptr(\bar{\rho_n})) = activeenv(\bar{\rho}, \bar{\rho_n})$.

  Similarly, by the induction hypothesis on the tree for e_2, there exists $\bar{\rho} \vdash_{ps} e_2 \Rightarrow \bar{v}$, such that $v = activeval(\bar{\rho}, \bar{v})$.

Now: $activeenv(\bar{\rho}, (x = \bar{v}).\bar{\rho}_l@p.\bar{\rho})$
$= activeenv(\bar{\rho}, (x = \bar{v}).\bar{\rho}_l@ptr(\bar{\rho}_n).\bar{\rho})$
$= (x = v).\rho_l@\rho_n$

Hence by the induction hypothesis on the tree for e, it must be the case that $(x = \bar{v}).\bar{\rho}_l\ @p.\bar{\rho} \vdash_{ps} e \Rightarrow \bar{w}$, where $w = activeval(\bar{\rho}, \bar{w})$. Hence there exists a pointer-semantics tree $\bar{\rho} \vdash_{ps} e_1e_2 \Rightarrow \bar{w}$ such that $w = activeval(\bar{\rho}, \bar{w})$. □

3.3 A Stronger Dynamic Criterion

It is difficult to arrive at a static criterion for env-stackability directly from Criterion 1. The criterion examines every path of the dynamic-semantics tree and checks whether every closure has downward static links. But a path in the dynamic-semantics tree says nothing about the context of creation and deletion of static links which is the crucial information required in formulating a static criterion. Hence we seek a dynamic criterion which would provide us with context information about where exactly a static link is created and deleted in the dynamic-semantics tree.

We 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* $\rho \vdash e_1e_2 \Rightarrow \langle \lambda y.e_3, \rho'_l, \rho'_n\rangle$ *that was built by the rule:*

$$\frac{\rho \vdash e_1 \Rightarrow \langle \lambda x.e, \rho_l, \rho_n\rangle \qquad \rho \vdash e_2 \Rightarrow v \qquad \rho' \vdash e \Rightarrow \langle \lambda y.e_3, \rho'_l, \rho'_n\rangle}{\rho \vdash e_1e_2 \Rightarrow \langle \lambda y.e_3, \rho'_l, \rho'_n\rangle}$$

where ρ' *is* $(x = v).\rho_l@\rho_n$*, it is the case that* $\rho' \neq \rho'_n$.[3]

It is surprising that we require merely that $\rho' \neq \rho'_n$, but this inequality forces ρ'_n to be a subenvironment of ρ. Let us see the intuitions behind the inequality. First note that in any dynamic-semantics tree built using the semantics in Figure 5, the non-local environment of a closure must be an environment that appears to the left of the turnstile somewhere in the tree. For the particular case of env-stackability, we demand that the non-local environment of a closure at a particular node in the tree be an ancestral node in the path from the node to the root of the tree. The only time env-stackability can be lost is when the environment is extended during the application of a fully-applied closure and the result of the application is a closure — as in the rule above. Clearly, ρ' *cannot* be equal to ρ'_n, because the bindings of ρ' will be discarded upon evaluation of e. The question is whether this inequality is sufficient to satisfy the dynamic criterion. Suppose all existing non-local environments (present in the trees of e_1 and e_2) satisfy the dynamic criterion. Suppose also that e itself satisfies the dynamic criterion. Now we want to insert the dynamic-semantics tree for e and want to make the dynamic criterion hold for the whole tree. So we have to force ρ'_n to be either ρ_n or

- if v is a closure, ρ'_n can be the non-local environment within v or

[3] When this work was presented at INRIA, Rennes, Daniel Le Métayer rightly observed that the inequality could be simplified to length$(\rho') \neq$ length(ρ'_n).

- for any $(x' = v') \in \rho_l$, if v' is a closure, ρ'_n can be the non-local environment within v' or

- ρ'_n can be any of the ancestral environments in the path from the root of the tree to the node for e.

In each of the above cases, note that $\rho' \neq \rho'_n$.

The non-env-stackable example in Section 2 fails the modified dynamic criterion at Node B.

We are now interested in formalizing the above intuitions thus showing that the modified dynamic criterion implies the dynamic criterion. Towards this goal, we define the following. Let $C[\ ]$ denote a context, *i.e.*, a natural-semantics tree with a hole in it. Also, we let $\rho \vdash e \Rightarrow v$ stand for the natural-semantics tree whose root is $\rho \vdash e \Rightarrow v$. Next, we define a predicate ndr ("no dangling references") such that $ndr(C[\ ], v)$ verifies that v contains no dangling references when used within a non-empty context $C[\ ]$. The definition of ndr mimics the dynamic criterion and is shown below:

$$
\begin{array}{lll}
ndr(C[\ ], c) & \text{iff} & true, \text{ where } c \text{ is a base value} \\
ndr(C[\ ], \langle \lambda x.e, \rho_l, \rho_n \rangle) & \text{iff} & \rho_n \text{ occurs in an ancestor node in the path} \\
 & & \text{from the hole, } [\ ], \text{ in } C[\ ] \text{ to the root} \\
 & & \textit{and } (\text{for all } (i = v_i) \in \rho_l,\ ndr(C[\ ], v_i)) \\
 & & \textit{and } (\text{for all } (j = v_j) \in \rho_n,\ ndr(C[\ ], v_j)) \\
ndr(C[\ ], \langle \mu f.\lambda x.e, \rho_n \rangle) & \text{iff} & \rho_n \text{ occurs in an ancestor node in the path} \\
 & & \text{from the hole, } [\ ], \text{ in } C[\ ] \text{ to the root} \\
 & & \textit{and } (\text{for all } (i = v_i) \in \rho_n,\ ndr(C[\ ], v_i))
\end{array}
$$

Given ndr, let $P(C[\ ], \rho \vdash e \Rightarrow v)$ stand for the assertion that ndr holds for those values v_i saved in ρ, and let $Q(C[\ ], \rho \vdash e \Rightarrow v)$ stand for the assertion that ndr holds for v:

$$
\begin{array}{lll}
P(C[\ ], \rho \vdash e \Rightarrow v) & \text{iff} & \text{for all } (i = v_i) \in \rho, ndr(C[\ ], v_i) \\
Q(C[\ ], \rho \vdash e \Rightarrow v) & \text{iff} & ndr(C[\ ], v)
\end{array}
$$

We intend to show that if a dynamic-semantics tree, $t = \rho \vdash e \Rightarrow v$, satisfies the modified dynamic criterion, then it satisfies the dynamic criterion. To this end, we first show that if in an arbitrary context $C[\ ]$, ρ does not have any dangling references and if plugging the hole in the context by t still makes $C[t]$ satisfy the modified dynamic criterion, then v does not contain any dangling references. This is formalized in the following lemma.

Lemma 1
For all trees, $t = \rho \vdash e \Rightarrow v$, if t satisfies the modified dynamic criterion, then for all contexts, $C[\ ]$, if $C[t]$ satisfies the modified dynamic criterion and $P(C[\ ], t)$ holds, then $Q(C[\ ], t)$ holds.

Proof: Consider any tree t. Assume that t satisfies the modified dynamic criterion. Consider an arbitrary context $C[\ ]$, such that $C[t]$ satisfies the modified dynamic criterion and assume $P(C[\ ], t)$ holds. Then the proof proceeds by induction on the structure of the derivation tree. The following are the two interesting cases:

(i) t is

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

This is the application rule for partially-applied closures.

Since $P(C[\,], t)$ holds, therefore, $P(C[K], t_1)$ holds for $t_1 = \rho \vdash e_1 \Rightarrow \langle \lambda x.\lambda y.e, \rho_l, \rho_n\rangle$, where $C[K]$ is the context:

$$\frac{[\,] \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, (x = v).\rho_l, \rho_n\rangle}$$

Hence by the induction hypothesis on t_1, $Q(C[K], t_1)$ holds.
Hence $ndr(C[K], \langle \lambda x.\lambda y.e, \rho_l, \rho_n\rangle)$ holds.
Does $ndr(C[\,], \langle \lambda x.\lambda y.e, \rho_l, \rho_n\rangle)$ hold?

- If $\rho_n \neq \rho$, it does, since $ndr(C[K], \langle \lambda x.\lambda y.e, \rho_l, \rho_n\rangle)$ holds.
- If $\rho_n = \rho$, then consider where $\rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, (x = v).\rho_l, \rho_n\rangle$ resides in $C[\,]$:
  If $C[\,]$ is non-empty, then the only case where $ndr(C[\,], \langle \lambda x.\lambda y.e, \rho_l, \rho_n\rangle)$ cannot hold is when $\rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, \rho_l, \rho_n\rangle$ is the rightmost subtree in the rule:

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

  where $\rho = (z = v').\rho_l' @ \rho_n'$.
  But this violates the modified dynamic criterion for $C[t]$, hence this case is impossible.
  Thus $ndr(C[\,], \langle \lambda x.\lambda y.e, \rho_l, \rho_n\rangle)$ holds.

Similarly, via the induction hypothesis and following the same reasoning as above, we can show $ndr(C[\,], v)$ holds.
Thus following the above reasoning once more, $ndr(C[\,], \langle \lambda y.e, (x = v).\rho_l, \rho_n\rangle)$ holds.
That is, $Q(C[\,], t)$ holds.

(ii) t is $\rho \vdash \lambda x.e \Rightarrow \langle \lambda x.e, (), \rho\rangle$.

Assume a non-empty context, $C[\,]$. Assume $P(C[\,], t)$. Then t can occur in an application context, where it could be:
(1) the operator part of an application of one of the two application rules, *i.e.*,

$$\frac{\rho \vdash \lambda x.e \Rightarrow \langle \lambda x.e, (), \rho\rangle \qquad \ldots}{\rho \vdash (\lambda x.e)e' \Rightarrow \ldots}$$

Note that it is the case that the environment of the operator and the environment of the application, its ancestor node, are identical. Now

$$\begin{array}{lll} Q(C[\,], t) & \textit{iff} & ndr(C[\,], \langle \lambda x.e, (), \rho \rangle) \\ & \textit{iff} & \rho \text{ occurs in an ancestor node in the path from the} \\ & & \text{hole, } [\,] \text{ in } C[\,] \text{ to the root} \\ & & and \text{ (for all } (i = v_i) \in (),\ ndr(C[\,], v_i) \\ & & \text{and (for all } (j = v_j) \in \rho,\ ndr(C[\,], v_j) \\ & \textit{iff} & \textit{true and true and true} \\ & \textit{iff} & \textit{true} \end{array}$$

(2) the operand part of an application of one of the two application rules, *i.e.*,

$$\frac{\rho \vdash e' \Rightarrow ... \qquad \rho \vdash \lambda x.e \Rightarrow \langle \lambda x.e, (), \rho \rangle \qquad ...}{\rho \vdash e'(\lambda x.e) \Rightarrow ...}$$

We can prove that $Q(C[\,], t)$ holds in a manner similar to Case (1).

(3) the substitution part of a fully-applied closure,

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

This case is impossible since it is disallowed by the inference rules for application. □

Corollary 1 (Modified Dynamic Criterion implies Dynamic Criterion)
If ρ_0 is an initial environment, i.e. has no bindings to closures, then if $t = \rho_0 \vdash e_0 \Rightarrow v_0$ satisfies the modified dynamic criterion, then t satisfies the dynamic criterion.

Proof: We have that $P(C[\,], t)$ holds since ρ_0 is initial. By induction on the structure of t, we can verify that, for all contexts $C'[\,]$, and for all subtrees $t' = \rho' \vdash e \Rightarrow v'$ of t, $P(C'[\,], t')$ holds and hence, by Lemma 1, $Q(C'[\,], t')$ holds. (Note that $t = C'[t']$, and by hypothesis, t satisfies the modified dynamic criterion). Hence v' cannot contain any dangling references. Hence the dynamic criterion is satisfied. □

4 Static Analysis of Stackability

The Dynamic Criterion and the Modified Dynamic Criterion both require the dynamic-semantics tree. For 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 which 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 [12]. The details, including a soundness theorem, are reviewed in Section 4.3. For now, we give a very brief notational summary and an intuitive explanation of soundness and conclude with an example.

4.1 Closure Analysis

We begin by labeling 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 program labels. Given a program e in the source language, closure analysis generates two *closure description functions*, ϕ, γ: $CDescription = Label \rightarrow \mathcal{P}(Label)$, 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*.

To help calculate ϕ and γ, 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

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

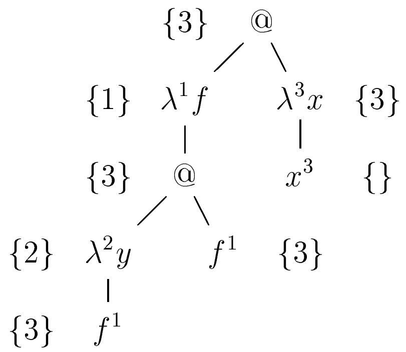
Therefore, we want a solution (for expression e_0) of the equation:

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

Before proceeding to a formal definition of $\mathcal{C}_a$ and $\mathcal{C}_p$, let us try to understand closure analysis informally. Suppose, for an environment ρ, and any identifier x^ℓ in its domain, if $\rho(x^\ell)$ evaluates to a closure, say, ℓ', then we want $\ell' \in \gamma\ell$ and 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.* $\ell \in \mathcal{C}_a[\![e]\!]\phi\gamma$. Let us clarify this with an example.

4.2 Example

Below is the labeled 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 Soundness of Closure Analysis

Here are the definitions of the functions $\mathcal{C}_a$ and $\mathcal{C}_p$, taken from Sestoft [12]:

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

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

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

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

We show that closure analysis is sound with respect to the dynamic semantics of Figure 5. We shall assume that all textual lambdas (and their binding identifiers) are labeled in the dynamic semantics. Towards the proof, we have the following definitions:
Let e be any given expression.

Definition (Consistency of Environments)
For closure description γ, type assignment π, and environment ρ, where $\models\; \rho : \pi^4$, 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) $\gamma m \supseteq \{\ell\}$

(ii) *ρ_l and ρ_n are consistent with respect to γ.*

- *For all $f^m \in \mathrm{Dom}(\rho)$, if $\rho(f^m) = \mu f.\langle \lambda^\ell x \,.\, e_\ell, \rho' \rangle$, then*

  (i) $\gamma m = \{\ell\}$

  (ii) *ρ' is consistent with respect to γ.*

Note that the definition of consistency is well-founded since it is defined inductively on the structure of environments and values.

Definition (Initial Environment)
Environment ρ is an initial environment iff:
$\rho = \{(x = v) \mid x \in \mathrm{Dom}(\rho)$ and v *is of ground type*$\}$

For closure description γ, type assignment π and environment ρ where $\models \rho : \pi$ and ρ is an initial environment, ρ is consistent with respect to γ.

Theorem 2 (Soundness)
For all (ϕ, γ), for all e_0 if (ϕ, γ) is a solution for e_0 of $\mathcal{C}_a$ and $\mathcal{C}_p$, and for all ρ if ρ is consistent with respect to γ, then for all (finite) derivation trees $\rho \vdash e_0 \Rightarrow v$, if v is a closure, say, $\langle \lambda^\ell x \,.\, e_\ell, \rho_l, \rho_n \rangle$, then $\ell \in \mathcal{C}_a[\![e_0]\!]\phi\gamma$

Proof: We prove a stronger result: In addition to showing $\ell \in \mathcal{C}_a[\![e_0]\!]\phi\gamma$, we show that ρ_l and ρ_n are consistent with respect to γ.
Assume for any given expression e_0, (ϕ, γ) is a solution for e_0 of $\mathcal{C}_a$ and $\mathcal{C}_p$.
Assume that ρ is consistent. Then the proof is by induction on the depth of inference of the inferred sentence.

The interesting case is that of application.

Case 1

$$\frac{\rho \vdash e_1 \Rightarrow \langle \lambda^\ell x \,.\, (\lambda^{\ell'} y \,.\, e_{\ell'})_\ell, \rho_l, \rho_n \rangle \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash (e_1\ e_2) \Rightarrow \langle \lambda^{\ell'} y \,.\, e_{\ell'}, (x = v).\rho_l, \rho_n \rangle}$$

(a) Let v not be a closure. By the induction hypothesis on the shorter proof tree of e_1,

$$\begin{array}{l} \ell \in \mathcal{C}_a[\![e_1]\!]\phi\gamma \text{ and} \\ \rho_l \text{ and } \rho_n \text{ are consistent with respect to } \gamma \end{array}$$

Hence $(x = v).\rho_l$ is consistent with respect to γ. Now

$$\begin{array}{rcl} \mathcal{C}_a[\![e_1 e_2]\!]\phi\gamma & = & \bigcup\{\phi\ell' \mid \ell' \in \mathcal{C}_a[\![e_1]\!]\phi\gamma\} \\ & = & \phi\ell, \text{ since } \ell \in \mathcal{C}_a[\![e_1]\!]\phi\gamma \\ & = & \mathcal{C}_a[\![\lambda^{\ell'} y \,.\, e_{\ell'}]\!]\phi\gamma, \text{ by hypothesis} \\ & = & \{\ell'\} \end{array}$$

[4]Read $\models \rho : \pi$ as ρ has type π, see Appendix A

Hence, $\ell' \in \mathcal{C}_a[\![e_1 e_2]\!]\phi\gamma$.
We conclude further that the local environment $(x = v).\rho_l$ is consistent with respect to γ as is ρ_n.

(b) Let v be a closure, say, $\langle \lambda^m\, y\,.\, e_m, \rho'_l, \rho'_g \rangle$. By the induction hypothesis on the shorter proof tree of e_1,

$$\begin{array}{l} \ell \in \mathcal{C}_a[\![e_1]\!]\phi\gamma \text{ and} \\ \rho_l \text{ and } \rho_n \text{ are consistent with respect to } \gamma \end{array}$$

By the induction hypothesis on the shorter proof tree of e_2,

$$\begin{array}{l} m \in \mathcal{C}_a[\![e_1]\!]\phi\gamma \text{ and} \\ \rho'_l \text{ and } \rho'_g \text{ are consistent with respect to } \gamma \end{array}$$

To show that $(x^\ell = \langle \lambda^m\, y\,.\, e_m, \rho'_l, \rho'_g \rangle).\rho_l$ is consistent with respect to γ, it suffices to show that $\gamma\ell \supseteq \{m\}$.
By hypothesis,

$$\begin{array}{rcl} \gamma\ell & = & \mathcal{C}_p[\![e_1 e_2]\!]\phi\gamma\ell \\ & = & \mathcal{C}_p[\![e_1]\!]\phi\gamma\ell \;\cup\; \mathcal{C}_p[\![e_2]\!]\phi\gamma\ell \;\cup\; \mathcal{C}_a[\![e_2]\!]\phi\gamma, \text{ since } \ell \in \mathcal{C}_a[\![e_1]\!]\phi\gamma \\ & \supseteq & \mathcal{C}_a[\![e_2]\!]\phi\gamma \\ & \supseteq & \{m\} \end{array}$$

Now

$$\begin{array}{lcl} \mathcal{C}_a[\![e_1 e_2]\!]\phi\gamma & = & \bigcup\{\phi\ell' \mid \ell' \in \mathcal{C}_a[\![e_1]\!]\phi\gamma\} \\ & \supseteq & \phi\ell, \text{ since } \ell \in \mathcal{C}_a[\![e_1]\!]\phi\gamma \\ i.e. & \supseteq & \mathcal{C}_a[\![\lambda^{\ell'}\, y\,.\, e_{\ell'}]\!]\phi\gamma, \text{ by hypothesis} \\ i.e. & \supseteq & \{\ell'\} \end{array}$$

Hence, $\ell' \in \mathcal{C}_a[\![e_1 e_2]\!]\phi\gamma$.

Case 2

$$\frac{\rho \vdash e_1 \Rightarrow \langle \lambda^\ell\, x\,.\, e_\ell, \rho_l, \rho_n \rangle \qquad \rho \vdash e_2 \Rightarrow v \qquad (x^\ell = v).\rho_l \;@\rho_n \vdash e_\ell \Rightarrow w}{\rho \vdash (e_1\ e_2) \Rightarrow w}$$

(a) w is not a closure. Trivial.
(b) w is a closure, say, $\langle \lambda^n\, z\,.\, e_n, \rho'_l, \rho'_g \rangle$. There are two subcases.

(i) Let v not be a closure. By the induction hypothesis on the shorter proof tree of e_1,

$$\begin{array}{l} \ell \in \mathcal{C}_a[\![e_1]\!]\phi\gamma \text{ and} \\ \rho_l \text{ and } \rho_n \text{ are consistent with respect to } \gamma \end{array}$$

Hence $(x^\ell = v).\rho_l$ is consistent with respect to γ.
Hence $((x^\ell = v).\rho_l)@\rho_n$ is consistent with respect to γ.

Therefore by the induction hypothesis on e_ℓ, $n \in \mathcal{C}_a[\![e_\ell]\!]\phi\gamma$ and ρ'_l, ρ'_g are consistent with respect to γ. Now

$$\begin{array}{rcl} \mathcal{C}_a[\![e_1 e_2]\!]\phi\gamma & = & \bigcup\{\phi\ell' \quad | \ \ell' \in \mathcal{C}_a[\![e_1]\!]\phi\gamma\} \\ & = & \phi\ell, \text{ since } \ell \in \mathcal{C}_a[\![e_1]\!]\phi\gamma \\ & = & \mathcal{C}_a[\![e_\ell]\!]\phi\gamma, \text{ by hypothesis} \\ & = & \{n\} \end{array}$$

Therefore $n \in \mathcal{C}_a[\![e_1 e_2]\!]\phi\gamma$.

(ii) Let v be a closure, say, $\langle \lambda^m\, y \,.\, e_m, \rho''_l, \rho''_g \rangle$. It suffices to show $(x^\ell = v).\rho_l$ is consistent with respect to γ, which we do in a manner similar to Case 1, part (b). The rest follows as in *(i)*. □

4.4 The Static Criterion for Env-Stackability

Our goal is to use closure analysis to predict when the modified dynamic criterion (Criterion 2, Section 3.3) 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

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

correctly predicts that $\models \rho'_l : \pi'_l$ and that $\models \rho'_n : \pi'_n$. Similarly, the typing derivation

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

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 labeled program $\pi_0 \vdash e_0 : \tau_0$ we define:

Definition (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 labeled 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*
> *$\pi_a \vdash \lambda^{\ell_1} x_1. \cdots \lambda^{\ell_i} x_i \lambda^{\ell'} y.e_{\ell'} : \tau_1' \to \tau_2'$, and $\pi_o \vdash \lambda^{\ell}\, x\,.\,e_{\ell} : \tau_1 \to \tau_2$,*
> *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.

4.5 Example

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

Consider the application $(\lambda^1 f\ : int \to int.\ (\lambda^2 y\ : int \to int.\ f^1) f^1)(\lambda^3 x\ : int.\ x^3)$. 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, $\lambda^1 f\ : int \to int.\ (\lambda^2 y\ : int \to int.\ f^1) f^1$. 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 $(f^1 : int \to int) \oplus \pi \neq \pi$. Thus the static criterion is satisfied for this application.

Consider the other application $(\lambda^2 y\ : int \to int.\ f^1) f^1$. 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, $\lambda^2 y\ : int \to int.\ f^1$. Closure analysis predicts that it must evaluate to the closure representing λ-abstraction 2. The type assignment for λ-abstraction 2 is $(f^1 : int \to int) \oplus \pi$. Note that $(y^2 : int \to int, f^1 : int \to int) \oplus \pi \neq \pi$. 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 Implementation

In this section, we present the implementation of env-stackability as presented in Figure 1. The BNF of the language is reproduced in Figure 9. It differs from the one in Figure 2 in that some arithmetic and boolean operations and the conditional have been added to get interesting programs.

$\iota \in$ BaseType	$\iota ::= \mathsf{nat} \mid \mathsf{bool}$
$\tau \in$ Type	$\tau ::= \iota \mid \tau_1 \rightarrow \tau_2$
$\pi \in$ TypeAssignment	$\pi ::= \{x_i : \tau_i\}_{i \geq 0}$
$c \in$ ConstExpression	
$x \in$ Identifier	
$e \in$ Expression	$e ::= c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 = e_2 \mid e_1 > e_2 \mid$ $\mathsf{succ} \mid \mathsf{pred} \mid \mathsf{iszero} \mid$ $\lambda x : \tau . e \mid e_1 e_2 \mid \mathsf{if}\ e_1\ e_2\ e_3 \mid \mathsf{rec}\ f.\lambda x : \tau . e$

Figure 9: BNF of the Simply-Typed, Call-by-value Lambda Calculus

We are interested in implementing the subset of the simply-typed call-by-value λ-calculus that is env-stackable. This implementation is described below. All values and static links exist on one data structure, namely, the stack.

Our implementation has two phases, a static (or compile-time) phase and a dynamic (or run-time) phase. In the static phase, an expression, e, is parsed and type-checked. It is then analyzed using the static criterion for env-stackability. If the expression is env-stackable, then we know by the safety of the static criterion that it can be run on a machine with a single, global, stack and no dangling references will ever arise. Accordingly, we compile (the abstract syntax tree of) e into stack-code. The implementation of this phase has been done in Standard ML of New Jersey.

The run-time machine interprets the compiled code for e. There is only one data structure that it manipulates: a single, global, stack. The stack not only contains all temporary values, but also environment frames (activation records) as found in a traditional compiler implementation [1]. There is a fixed number of registers in the machine and this can be determined at compile-time. Essentially, the number of registers is the length of the biggest closure that can possibly arise. The implementation of the run-time machine has been done in C.

5.1 Representation of Values

There are two types of run-time values: basic values and closures. Basic values have type nat and bool. Closures are values of function type. There are two types of closures: the "usual" ones are represented as a triple consisting of the code, the local environment and the non-local environment. Recursive closures, on the other hand, are represented as a pair consisting of the code and the environment.

We would like closures to be "boxed" whenever possible, *i.e.*, they would occupy a fixed number of words on the stack and references to them would be carried out via pointers to the boxes. This would prevent the copying of big chunks of the stack and provide faster access of non-local references. Below we give the layout of a closure:

address of code
size of closure
number of arguments
pointer to value$_0$
...
pointer to value$_k$
static link
value$_0$
...
value$_k$

The address-of-code field denotes the address where the code corresponding to the body of the λ-abstraction resides on the stack. The number-of-arguments field denotes the number of identifiers in the local environment of a closure. The local environment itself is represented by pointers to the values of its identifiers. The non-local environment of the closure is represented by a (downwards) static link. Immediately following the static link, appear (some of) the values in the local environment.

The size-of-closure field contains the sum of the following: one cell for the number-of-arguments, (n, say); one cell for the static link; n, for the n pointers to values; and the number of cells occupied by the values. The size of a basic value is 1.

As an example, consider the closure, $\langle \lambda y.x, (x = 1), \rho \rangle$, obtained as a result of evaluation of the application $(\lambda x.\lambda y.x)1$ in an environment ρ. The layout of this closure is as follows:

address to code for $\lambda y.x$
4
1
pointer to argument x
static link to ρ
1

Recursive closures are of the form $\mu f.\langle \lambda x.e, \rho \rangle$. Such closures are 2-celled and have the following layout:

address to code
static link

A recursive closure always appears within a "usual" closure, and does not have an independent existence of its own. When it is looked up in an environment, it is immediately expanded, as its natural semantics in Figure 3.5 illustrates.

As an example, consider the closure, $\langle \lambda x.2, (f = \mu f.\langle \lambda x.2, \rho \rangle), \rho \rangle$, obtained as a result of evaluating the expression rec $f.\lambda x.2$ in the environment ρ. Note that this closure contains the recursive closure, $\mu f.\langle \lambda x.2, \rho \rangle$. Here is its layout:

address to code for $\lambda x.2$
5
1
pointer to argument f
static link to ρ
pointer to code for $\lambda x.2$
static link to ρ

5.2 Benchmarks

To measure the performance improvements due to our optimizations, we performed benchmarks of standard programs on the stack machine and on a heap-based architecture with as similar of an instruction set as possible. The benchmarks are taken from [8] where they have been used to benchmark the ZINC Abstract Machine (ZAM) of CAML-light. Here are three of them:

```
(rec fib = lambda n. if (iszero n)
                       1
                      if (iszero (n-1))
                          1
                         fib(x-1) + fib(x-2))
26
```

```
(((rec tak = lambda x. lambda y. lambda z.
                if x > y
                   tak (tak (x-1) y z) (tak (y-1) z x) (tak (z-1) x y)
                   z)
18)12)6
```

```
(rec sum = lambda l.
              if (null l)
                  0
                  (hd l) + sum(tl l))
((rec interval = lambda n.
                     if (iszero n)
                         nil
                         cons n (interval (n-1)))
1000)
```

The `fib` function is the naive Fibonacci function and in this example it calculates the fibonacci of 26.

The `tak` function is highly recursive and has three arguments. It has been chosen because it tests partial application and partially-applied closures.

The last example computes the sum of the first 1000 natural numbers. It first builds the list containing the first 1000 natural numbers (this is what `interval` does) and then sums them up (this is what `sum` does. This tests recursive calls as well as the construction and destructuring of lists. All of these examples are found to be env-stackable by our analysis.

All run-time values exist on the stack. The parameters measured for the benchmarks are

- Maximum size of the execution stack
- Number of environment touches

Here are the results for the stack implementation on a SPARCstation 10/52 running Solaris:

Program	Max.stack size	Env. Touches
fib26	383	2206958
tak	388	381651
sum(interval...)	511575	9003

If we execute the programs on a traditional heap-machine, with a value stack and a heap-based environment and with no garbage collection, we obtain the following results:

Program	Max. heap size	Env. Touches
fib26	4413988	2206958
tak	763396	381651
sum(interval...)	512584	9003

Massive improvements are obtained for a stack-based implementation of `fib26` and `tak`. For `sum(interval...)`, the improvement is not so marked because our prototype stack architecture makes no effort to improve list storage management. We hope to include analyses for data structures that would improve performance in the future. It also remains to be seen whether the analysis and the implementation model allow the efficient compilation of tail recursive functions.

6 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 [4], 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, $\mathsf{letregion}\ \rho\ \mathsf{in}\ e\ \mathsf{end}$, on an arbitrary program [13]. 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.

7 Conclusion

We have developed a statically checkable criterion to detect stackability of environments for a call-by-value λ-calculus. We have also provided an implementation of env-stackability. All programs that satisfy the static criterion are implemented with a single, global, stack-based environment.

An interesting variation on the implementation would be to have a heap as well as a stack. Instead of rejecting functions that cannot be stack-allocated, we could detect which of their parameters are env-stackable and allocate them on the stack. All other parameters can be allocated on the heap.

It is simple to extend the modified dynamic criterion to find out whether a parameter of a function should be allocated on the heap or on the stack. First, let us recall that for the modified dynamic criterion to be satisfied, if an application, $e_{i1}e_{i2}$, 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 $(x^{\ell} = v).\rho_l @ \rho_n$, then we require that $\rho' \neq \rho'_n$.

For all nodes that satisfy the modified dynamic criterion, we can put the bindings $(x^{\ell} = v).\rho_l$ on the stack: at the end of e_{ℓ}'s evaluation they can be destroyed. Otherwise, they must be put on the heap — they cannot be destroyed as they maybe required in a future computation of $e_{\ell'}$.

It is easy to extend the static criterion to handle the case of heap-allocation and thus our analysis can be extended for the implementation described in [5].

The analysis has been adapted to handle storage allocation of variables on storage stacks for block-structured imperative languages. In fact, we can use the analysis to give a more liberal interpretation of blocks than that in Algol-like languages: a block controls the scope of a variable, but *not* its extent. It is the analysis that provides an algorithm for inferring "extent blocks" in the program. These control the extent of a variable while still maintaining a store-stack. The details can be found in [2].

Acknowledgements: 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. Thanks to Olivier Danvy for his comments on an earlier draft and to the anonymous referees for their helpful suggestions. Finally, warm thanks to Pascal Fradet and Daniel Le Métayer for their comments and hospitality and for arranging a presentation of the work at IRISA/INRIA, Rennes.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Anindya Banerjee. *The semantics and implementation of bindings in higher-order programming languages*. PhD thesis, Kansas State University, Manhattan, Kansas, U.S.A, July 1995.

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

[4] 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.

[5] John Hannan. A type-based analysis for stack allocation in functional languages. In A. Mycroft, editor, *Proceedings of the second International Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 172–188, Glasgow, U.K., September 1995.

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

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

[8] Xavier Leroy. The ZINC experiment: An economical implementation of the ML language. Rapports Techniques 117, INRIA, February 1990.

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

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

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

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

[13] 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.

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 $\pi \vdash \mathsf{rec}\ f.\lambda x.e : \tau$, $\text{Dom}(\pi) = \text{Dom}(\rho)$, 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 [6]) semantics for the call-by-value, simply-typed λ-calculus are shown in Figures 10 and 11.

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* [9], *rel*, that relates Figures 5 and 11. If $\rho \rhd e \Rightarrow v$ and $\rho' \vdash e \Rightarrow v'$, then $v\ rel\ v'$ must satisfy:

$\pi \in$ TypeEnv	$x \in$ Identifier	$\pi ::= \{x_i : \tau_i\}_{i \geq 0}$
$c \in$ BaseValue	$v \in$ Value	$v ::= c \mid \langle \lambda x.e, \rho \rangle \mid \mu f.\langle \lambda x.e, \rho \rangle$
$\rho \in$ Env		$\rho ::= <> \mid (i = v).\rho$

Figure 10: Semantic Objects for the Standard Dynamic Semantics

$$\rho \triangleright c \Rightarrow c \qquad\qquad \rho \triangleright \lambda x.e \Rightarrow \langle \lambda x.e, \rho \rangle$$

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

$$\rho \triangleright x \Rightarrow v \ \text{ if } (x = v) \in \rho, \text{ otherwise}$$

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

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

Figure 11: Standard Dynamic Semantics

(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 $\rho'(x)\ \mathit{rel}\ (\rho_l @ \rho_n)(x)$ 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 $\rho'(x)\ \mathit{rel}\ \rho''(x)$

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.