Stackability in the simply-typed call-by-value lambda calculus

Anindya Banerjee a, *, David A. Schmidt b,1,2

a Department of Computer Science, Stevens Institute of Technology, Hoboken, NJ 07030, USA
b Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA

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 $\lambda$-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 $\eta$-expansion and decurrying for call-by-name. However, $\eta$-expansion is unsound for call-by-value. Hence, we must identify a subset of the simply typed, call-by-value $\lambda$-calculus, where the $\lambda$-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. © 1998 Published by Elsevier Science B.V. All rights reserved.

Keywords: Block-structure; Closure analysis; Env-stackability; Extent; Lambda calculus

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 \texttt{begin} D \texttt{in} U \texttt{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.
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 \( \lambda \)-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 \( \lambda \)-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 `letregion \( \rho \) in \( e \) end` into the program. These constructs control extents: \( \rho \) 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 \( \lambda \)-calculus, whose programs can be run “as is” with a classical-stack environment. The \( \lambda \)-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.
Fig. 1 shows snapshots of the environment stack during the evaluation of the expression \((\lambda x.(\lambda y.x)x)l\). The static links are \(\alpha\), \(\gamma\) and \(\varepsilon\), where \(\alpha\) 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 Fig. 1, where the \(\lambda\)-abstraction serves as the block construct for extents, is called env-stackable.

2. Simple expressions and partially applied closures

The BNF and static semantics of the \(\lambda\)-calculus we study are given in Figs. 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 \(\lambda\)-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 \(\tau_n\) 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 \(\eta\)-expansion), which is itself an abbreviation for \(\lambda(x_1,x_2,\ldots,x_n).e\ x_1\ x_2\ldots x_n\) (due to decurring). This expression is env-stackable, because the bindings to \(x_1, x_2, \ldots, x_n\) are established, as a group, only when the body of the \(\lambda\)-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 \( \lambda \)-abstraction can serve as the block construct for extents. These ideas lie at the heart of the stack implementation of Idealized Algol [10].

Can this approach be adapted to the call-by-value \( \lambda \)-calculus that we study in this paper? Unfortunately, \( \eta \)-expansion is unsound. But Georgeff [3] noted that decurrying can be simulated by a closure of the form \((\lambda x.e, \rho_l, \rho_n)\), where \( \rho_l \) is a local environment and \( \rho_n \) is a nonlocal environment. An expression like \((\lambda x_1.\lambda x_2 \ldots \lambda x_i.\lambda x_{i+1} \ldots \lambda x_n.e_0)e_1e_2 \ldots e_i\) evaluates to \((\lambda x_1 \ldots \lambda x_n.e_0,(x_i = v_i, \ldots, x_2 = v_2, x_1 = v_1), \rho_0)\), where each \( e_j \) evaluates to value \( v_j, 1 \leq j \leq i \), and \( \rho_0 \) 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 \( \rho_0 \)). A closure \((\lambda x.e, \rho_l, \rho_n)\) is partially applied if \( e \) is a \( \lambda \)-abstraction; else the closure is fully applied.

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

---

3 \( \eta \)-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.

4 Note that there are two rules for identifier lookup. Identifiers which name recursive functions are bound to recursive closures of the form \( \mu f.(\lambda x.e, \rho) \) in the environment. Their evaluation entails one unfolding of the recursion.
c ∈ BaseValue

v ∈ ValueG

\[ v := c \mid \langle \lambda x.e, \rho_1, \rho_n \rangle \mid \mu f.\langle \lambda x.e, \rho \rangle \]

\( \rho \in \text{Env}_G \)

\[ \rho := \langle \rangle \mid (i = v).\rho \]

Fig. 4. Semantic objects for Georgeff-style dynamic semantics.

\[ \rho \vdash c \Rightarrow c \quad \rho \vdash \lambda x.e \Rightarrow \langle \lambda x.e(), \rho \rangle \]

\[ \rho \vdash f \Rightarrow \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, \rho \rangle), \rho \rangle \] if \((f = \mu f.\langle \lambda x.e, \rho \rangle) \in \rho \)

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

\[ \rho \vdash e_1 \Rightarrow \langle \lambda x.e, \rho_1, \rho_n \rangle \quad \rho \vdash e_2 \Rightarrow v \]

\[ \rho \vdash e_1 e_2 \Rightarrow \langle e_1, (x = v).\rho_1, \rho_n \rangle \] if \(e\) is a \(\lambda\)-abstraction

\[ \rho \vdash e_1 \Rightarrow \langle \lambda x.e, \rho_1, \rho_n \rangle \quad \rho \vdash e_2 \Rightarrow v \]

\[ \rho \vdash e_1 e_2 \Rightarrow v' \text{ otherwise} \]

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

Fig. 5. Dynamic semantics, Georgeff style.

\[ \pi \vdash c : \iota \quad \pi \vdash x : \tau \text{ if } (x : \tau) \in \pi \]

\[ \pi \vdash x_1 : \tau_1 \cdot \cdots \cdot x_1 : \tau_n \cdot e : \iota \quad \pi \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \pi \vdash e_2 : \tau_2 \]

Fig. 6. Simple expressions.

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 Fig. 6.

Simply stated, an expression is simple if all its \(\lambda\)-abstractions have as their bodies either \(\lambda\)-abstractions or expressions of base type. A simple expression is "fully \(\eta\)-expanded". It is the simulation of decurrying and \(\eta\)-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 \(((\lambda x : \text{int} . f : \text{int} \rightarrow \text{int} . f x)0)(\lambda y : \text{int} . y)\). The turnstiles are numbered to indicate the order of evaluation steps:

\[
\begin{align*}
\rho \vdash 4 \Rightarrow 0 \\
\rho \vdash \lambda x.f.x \Rightarrow \langle \lambda x.f.x, (\lambda y.y), \rho \rangle \\
\rho \vdash (\lambda x.f.x)0 \Rightarrow (\lambda f.x, (x = 0), \rho) \\
\rho \vdash 5 y \Rightarrow (\lambda y.y, (\lambda y.y), \rho) \\
\rho \vdash (\lambda x.f.x)0 \Rightarrow (\lambda y.y, (\lambda y.y), \rho) \\
\end{align*}
\]

where \(\rho_{f,x}\) is \((f = (\lambda y.y, (\lambda y.y), (x = 0), \rho))\) and \(\rho_y\) is \((y = 0), \rho)\).
This dynamic-semantics tree can be implemented with a global environment stack, similar to the one in Fig. 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 $\rho$. The stack is unchanged during evaluation of Nodes 2–5, but at the start of evaluation of Node 6, a static link to $\rho$ and the bindings for $f$ and $x$ are established (pushed) onto the stack. Evaluation of Node 9 establishes another static link to $\rho$ 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 $\rho$.

The key to why this implementation worked correctly is that the $\rho_1$ component of $(\langle x.e, \rho_1, \rho_1 \rangle)$ can always be implemented as a static link. To see how this might fail, consider the dynamic semantics of the nonsimple expression $((\lambda x : \text{int}.(\lambda f : \text{int} \rightarrow \text{int}.f)(\lambda y : \text{int}.x))5):(\ldots)5:(\ldots)5$:

$$\begin{align*}
\vdash_f (\lambda f. f)(\lambda y.x) & \Rightarrow (\lambda y.x(\cdot), \rho_y) \\
\vdash_A (\lambda x..)5 & \Rightarrow (\lambda y.x(\cdot), \rho_y) \\
\vdash ((\lambda x. (\lambda f. f)(\lambda y.x))5)6 & \Rightarrow 5
\end{align*}$$

where $\rho_x$ 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 $\rho_x$ and a binding to $y$, because the environment $\rho_x$ no longer exists on the global stack: $\rho_x$ (namely, the binding to $x$ and the static link to $\rho$) 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 $(\lambda f : \text{int} \rightarrow \text{int}.(\lambda y : \text{int} \rightarrow \text{int}.f)f)(\lambda x : \text{int}.x)$, whose dynamic-semantics tree is

$$\begin{align*}
\vdash_f (\lambda f..) & \Rightarrow (\lambda f..(\cdot), \rho_f) \\
\vdash_A (\lambda x..) & \Rightarrow v \\
\vdash_f (\lambda y.f)(\lambda x..) & \Rightarrow v \\
\vdash (\lambda y.(\lambda f..(\lambda x..))f)(\lambda x..) & \Rightarrow v
\end{align*}$$

where $v$ is the value $(\lambda x.x(\cdot), \rho_f)$, $\rho_f$ 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.
\[
p \in \text{Pointer} \quad \bar{\rho} \in \text{Env}_P \\
\bar{\rho}_l \in \text{Local}_\text{Env}_P \quad \bar{v} \in \text{Value}_P \\
\bar{\rho}_l ::= <> \mid (i = \bar{v}).\bar{\rho}_l \\
\bar{\rho} ::= <> \mid (i = \bar{v}).\bar{\rho} \mid p.\bar{\rho} \\
\bar{v} ::= c \mid \langle \lambda x.e, \bar{\rho}_l, p \rangle \mid \langle \mu f.\lambda x.e, p \rangle
\]

Fig. 7. Semantic objects for pointer semantics.

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

\[
\frac{\bar{\rho} \vdash_{ps} e_1 \Rightarrow \langle \lambda x.e, \bar{\rho}_l, p \rangle \quad \bar{\rho} \vdash_{ps} e_2 \Rightarrow \bar{v}}{
\bar{\rho} \vdash_{ps} e_1 e_2 \Rightarrow \langle e, (x = \bar{v}), \bar{\rho}_l, p \rangle} \quad \text{if } e \text{ is a } \lambda \text{ abstraction} \\
\frac{\bar{\rho} \vdash_{ps} e_1 \Rightarrow \langle \lambda x.e, \bar{\rho}_l, p \rangle \quad \rho \vdash_{ps} e_2 \Rightarrow \bar{v} \quad (x = \bar{v}).\bar{\rho}_l \otimes p.\bar{\rho} \vdash_{ps} e \Rightarrow \bar{w}}{
\bar{\rho} \vdash_{ps} e_1 e_2 \Rightarrow \bar{w}} \quad \text{otherwise} \\
\]

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

Fig. 8. Pointer semantics.

**Criterion 1** (Dynamic criterion). *A dynamic-semantics tree has the dynamic criterion, if for every node in the tree of the form \( \rho \vdash e \Rightarrow \langle \lambda x.e', \rho_1, \rho_n \rangle \), 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. Figs. 7 and 8 formalize this semantics, called the *pointer semantics*. The pointer semantics is so called because of the rule for \( \lambda \)-abstraction: when a closure is built, a "pointer" is saved in the closure instead of the nonlocal environment. We use the notation \( \text{ptr} (\bar{\rho}) \) 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 Figs. 5 and 8, we must relate a pointer-semantics environment to a dynamic-semantics environment. This is done by the function \textit{activeenv}:

\[
\text{activeenv} : \text{Env}_P \times \text{Env}_P \rightarrow \text{Env}_G \\
\text{activeenv}(\tilde{\rho}_0, \langle \rangle) = \langle \rangle \\
\text{activeenv}(\tilde{\rho}_0, (i = \tilde{v}) \cdot \tilde{\rho}) = (i = \text{activeval}(\tilde{\rho}_0, \tilde{v})) \cdot \text{activeenv}(\tilde{\rho}_0, \tilde{\rho}) \\
\text{activeenv}(\tilde{\rho}_0, p \cdot \tilde{\rho}_1) = \begin{cases} 
\text{if } p = \text{ptr}(\tilde{\rho}_1) \text{ and } \tilde{\rho}_1 \text{ is a subenvironment of } \tilde{\rho}_0 \\
\text{then } \text{activeenv}(\tilde{\rho}_0, \tilde{\rho}_1) \\
\text{else undefined} 
\end{cases}
\]

\[
\text{activeval} : \text{Env}_P \times \text{Value}_P \rightarrow \text{Value}_G \\
\text{activeval}(\tilde{\rho}_0, c) = c \text{ where } c \text{ is a base value} \\
\text{activeval}(\tilde{\rho}_0, (\lambda x. e, \rho_1, p)) = \begin{cases} 
\text{if } p = \text{ptr}(\tilde{\rho}) \text{ and } \tilde{\rho} \text{ is a subenvironment of } \tilde{\rho}_0 \\
\text{then } (\lambda x. e, \text{activeval}(\tilde{\rho}_0, \tilde{\rho}_1), \text{activeval}(\tilde{\rho}_0, \tilde{\rho})) \\
\text{else undefined} 
\end{cases}
\]

\[
\text{activeval}(\tilde{\rho}_0, \mu f. (\lambda x. e, p)) = \begin{cases} 
\text{if } p = \text{ptr}(\tilde{\rho}) \text{ and } \tilde{\rho} \text{ is a subenvironment of } \tilde{\rho}_0 \\
\text{then } \mu f. (\lambda x. e, \text{activeval}(\tilde{\rho}_0, \tilde{\rho})) \\
\text{else undefined} 
\end{cases}
\]

\[
\text{activelocalenv} : \text{Env}_P \times \text{LocalEnv}_P \rightarrow \text{Env}_G \\
\text{activelocalenv}(\tilde{\rho}_0, \langle \rangle) = \langle \rangle \\
\text{activelocalenv}(\tilde{\rho}_0, (i = \tilde{v}) \cdot \tilde{\rho}_1) = (i = \text{activeval}(\tilde{\rho}_0, \tilde{v})) \cdot \text{activelocalenv}(\tilde{\rho}_0, \tilde{\rho}_1)
\]

Essentially, \textit{activeenv} and \textit{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 Fig. 5 can be replicated by the pointer semantics of Fig. 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 \( \tilde{\rho} \in \text{Env}_P \), if \( \rho = \text{activeenv}(\tilde{\rho}, \tilde{\rho}) \), then there exists a pointer-semantics tree \( \tilde{\rho} \vdash_{ps} e \Rightarrow \tilde{v} \) such that \( v = \text{activeval}(\tilde{\rho}, \tilde{v}) \).

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

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

- $\rho \vdash \lambda x.e \Rightarrow \langle \lambda x.e, (\cdot), \rho \rangle$: Consider $\tilde{\rho} \vdash_{ps} \lambda x.e \Rightarrow \langle \lambda x.e, (\cdot), ptr(\tilde{\rho}) \rangle$:

  $activeval(\tilde{\rho}, \langle \lambda x.e, (\cdot), ptr(\tilde{\rho}) \rangle)$

  $= \langle \lambda x.e, (\cdot), activeenv(\tilde{\rho}, \tilde{\rho}) \rangle$

  $= \langle \lambda x.e, (\cdot), \rho \rangle$.

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

  $activeval(\tilde{\rho}, \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, ptr(\tilde{\rho}) \rangle), ptr(\tilde{\rho}) \rangle)$

  $= \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, activeenv(\tilde{\rho}, ptr(\tilde{\rho})) \rangle), activeenv(\tilde{\rho}, ptr(\tilde{\rho})) \rangle$

  $= \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, activeenv(\tilde{\rho}, \tilde{\rho}) \rangle), activeenv(\tilde{\rho}, \tilde{\rho}) \rangle$

  $= \langle \lambda x.e, (f = \mu f.\langle \lambda x.e, \tilde{\rho} \rangle), \tilde{\rho} \rangle$.

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

- **Case I:**

  \[
  \rho \vdash e_1 \Rightarrow \langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle \quad \rho \vdash e_2 \Rightarrow v
  \]

  Since $\rho = activeenv(\tilde{\rho}, \tilde{\rho})$, by the induction hypothesis on the tree for $e_1$, there exists $\tilde{\rho} \vdash_{ps} e_1 \Rightarrow \tilde{v}$ such that $\langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle = activeval(\tilde{\rho}, \tilde{v})$. Clearly, $activeval(\tilde{\rho}, \tilde{v})$ must have form $\langle \lambda x.\lambda y.e, \tilde{\rho}_1, p \rangle$, where $p$ has form $ptr(\tilde{\rho}_n)$ and $\tilde{\rho}_n$ is a subenvironment of $\tilde{\rho}$. Also, $\rho_1 = activevalenv(\tilde{\rho}, \tilde{\rho}_1)$ and $\rho_n = activeenv(\tilde{\rho}, \tilde{\rho}) = activeenv(\tilde{\rho}, ptr(\tilde{\rho}_n)) = activeenv(\tilde{\rho}, \tilde{\rho}_n)$.

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

  Hence, the derivation

  \[
  \rho \vdash_{ps} e_1 \Rightarrow \langle \lambda x.\lambda y.e, \tilde{\rho}_1, p \rangle \quad \rho \vdash_{ps} e_2 \Rightarrow \tilde{v}
  \]

  exists in the pointer semantics.

  Now

  $activevalenv(\tilde{\rho}, (x = \tilde{v}), \tilde{\rho}_1)$

  $= (x = activeval(\tilde{\rho}, \tilde{v})), activevalenv(\tilde{\rho}, \tilde{\rho}_1)$

  $= (x = v), \rho_1$.

  Hence,

  $activeval(\tilde{\rho}, \langle \lambda y.e, (x = \tilde{v}), \tilde{\rho}_1, p \rangle)$

  $= activeval(\tilde{\rho}, \langle \lambda y.e, (x = \tilde{v}), \tilde{\rho}_1, ptr(\tilde{\rho}_n) \rangle)$

  $= \langle \lambda y.e, (x = v), \rho_1, \rho_n \rangle$. 


• Case 2:

\[
\rho \vdash e_1 \Rightarrow (\lambda x.e, \rho_1, \rho_n) \quad \rho \vdash e_2 \Rightarrow v \quad (x = v).\rho_1 \mathbin{@}\rho_n \vdash e \Rightarrow w
\]

\[
\rho \vdash e_1 e_2 \Rightarrow w
\]

Since \( \rho = \text{activeenv}(\tilde{\rho}, \tilde{\rho}) \), therefore, by the induction hypothesis on the tree for \( e_1 \), there exists \( \tilde{\rho} \vdash_p e_1 \Rightarrow \tilde{v} \) such that \( (\lambda x.e, \rho_1, \rho_n) = \text{activeval}(\tilde{\rho}, \tilde{v}) \). Clearly, \( \text{activeval}(\tilde{\rho}, \tilde{v}) \) must have form \( (\lambda x.e, \rho_1, p) \), where \( p \) has form \( \text{ptr}(\tilde{\rho}_n) \) and \( \tilde{\rho}_n \) is a subenvironment of \( \tilde{\rho} \). Also, it must be the case that \( \rho_1 = \text{activelocalenv}(\tilde{\rho}, \tilde{\rho}_1) \) and \( \rho_n = \text{activeenv}(\tilde{\rho}, p) = \text{activeenv}(\tilde{\rho}, \text{ptr}(\tilde{\rho}_n)) = \text{activeenv}(\tilde{\rho}, \tilde{\rho}_n) \).

Similarly, by the induction hypothesis on the tree for \( e_2 \), there exists \( \tilde{\rho} \vdash_p e_2 \Rightarrow \tilde{v} \), such that \( v = \text{activeval}(\tilde{\rho}, \tilde{v}) \).

Now

\[
\text{activeenv}(\tilde{\rho}, (x = \tilde{v}).\rho_1 \mathbin{@} p.\tilde{\rho}) = \text{activeenv}(\tilde{\rho}, (x = \tilde{v}).\rho_1 \mathbin{@} \text{ptr}(\tilde{\rho}_n).\tilde{\rho}) = (x = v).\rho_1 \mathbin{@} \rho_n
\]

Hence by the induction hypothesis on the tree for \( e \), it must be the case that \( (x = \tilde{v}).\rho_1 \mathbin{@} p.\tilde{\rho} \vdash_p e \Rightarrow \tilde{w} \), where \( w = \text{activeval}(\tilde{\rho}, \tilde{w}) \). Hence, there exists a pointer-semantics tree \( \tilde{\rho} \vdash_p e_1 e_2 \Rightarrow \tilde{w} \) such that \( w = \text{activeval}(\tilde{\rho}, \tilde{w}) \). \( \square \)

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 \( \rho \vdash e_1 e_2 \Rightarrow (\lambda y.e_3, \rho'_1, \rho'_n) \) that was built by the rule

\[
\rho \vdash e_1 \Rightarrow (\lambda x.e, \rho_1, \rho_n) \quad \rho \vdash e_2 \Rightarrow v \quad \rho' \vdash e \Rightarrow (\lambda y.e_3, \rho'_1, \rho'_n)
\]

\[
\rho \vdash e_1 e_2 \Rightarrow (\lambda y.e_3, \rho'_1, \rho'_n)
\]

where \( \rho' \) is \( (x = v).\rho_1 \mathbin{@} \rho_n \), it is the case that \( \rho' \neq \rho'_n \). 5

5 When this work was presented at INRIA, Rennes, Daniel Le Métayer rightly observed that the inequality could be simplified to \( \text{length}(\rho') \neq \text{length}(\rho'_n) \).
It is surprising that we require merely that $\rho' \neq \rho_n$, but this inequality forces $\rho_n$ to be a subenvironment of $\rho$. Let us see the intuitions behind the inequality. First note that in any dynamic-semantics tree built using the semantics in Fig. 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, $\rho'$ cannot be equal to $\rho_n$, because the bindings of $\rho'$ 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 $\rho_n'$ to be either $\rho_n$ or

- if $v$ is a closure, $\rho_n'$ can be the non-local environment within $v$ or
- for any $(x' = v') \in \rho_l$, if $v'$ is a closure, $\rho_n'$ can be the non-local environment within $v'$ or
- $\rho_n'$ 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 $\text{ndr}$ ("no dangling references") such that $\text{ndr}(C[\ ], v)$ verifies that $v$ contains no dangling references when used within a non-empty context $C[\ ]$. The definition of $\text{ndr}$ mimics the dynamic criterion and is shown below:

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

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

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

$$Q(C[ ], \rho \vdash e \Rightarrow v) \text{ iff } \text{ndr}(C[ ], v)$$

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 $C[ ]$, $\rho$ 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

$$\frac{\rho \vdash e_1 \Rightarrow \langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle \quad \rho \vdash e_2 \Rightarrow v}{\rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, (x = v), \rho_1, \rho_n \rangle}$$

This is the application rule for partially applied closures.

Since $P(C[ ], t)$ holds, therefore, $P(C[K], t_1)$ holds for $t_1 = \rho \vdash e_1 \Rightarrow \langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle$, where $C[K]$ is the context:

$$\frac{[ ]}{\rho \vdash e_2 \Rightarrow v}$$

$$\frac{\rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, (x = v), \rho_1, \rho_n \rangle}{\rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, (x = v), \rho_1, \rho_n \rangle}$$

Hence by the induction hypothesis on $t_1$, $Q(C[K], t_1)$ holds. Hence ndr($C[K], \langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle$) holds. Does ndr($C[ ], \langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle$) hold?

- If $\rho_n \neq \rho$, it does, since ndr($C[K], \langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle$) holds.
- If $\rho_n = \rho$, then consider where $\rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, (x = v), \rho_1, \rho_n \rangle$ resides in $C[ ]$:
  - If $C[ ]$ is non-empty, then the only case where ndr($C[ ], \langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle$) cannot hold is when $\rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, \rho_1, \rho_n \rangle$ is the rightmost subtree in the rule:

$$\frac{\rho' \vdash e'_1 \Rightarrow \langle \lambda z.e_1 e_2, \rho'_1, \rho'_n \rangle \quad \rho' \vdash e'_2 \Rightarrow v' \quad \rho \vdash e_1 e_2 \Rightarrow \langle \lambda y.e, (x = v), \rho_1, \rho \rangle}{\rho \vdash e'_1 e'_2 \Rightarrow \langle \lambda y.e, (x = v), \rho_1, \rho \rangle}$$

where $\rho = (z = v').\rho'_1 @ \rho'_n$.

But this violates the modified dynamic criterion for $C[t]$, hence this case is impossible.

Thus ndr($C[ ], \langle \lambda x.\lambda y.e, \rho_1, \rho_n \rangle$) holds.
Similarly, via the induction hypothesis and following the same reasoning as above, we can show \( \text{ndr}(C[ ], v) \) holds. Thus, following the above reasoning once more, \( \text{ndr}(C[ ], \langle \lambda y.e, (x = v).\rho_1, \rho_2 \rangle) \) holds. That is, \( Q(C[ ], t) \) holds.

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

\[
\begin{align*}
\rho \vdash \lambda x.e & \Rightarrow \langle \lambda x.e, (\cdot), \rho \rangle, \\
\rho \vdash (\lambda x.e)e' & \Rightarrow \\
\end{align*}
\]

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

\[
Q(C[ ], t) \iff \text{ndr}(C[ ], \langle \lambda x.e, (\cdot), \rho \rangle)
\]

\[
\iff \rho \text{ occurs in an ancestor node in the path from the hole, } [ ] \text{ in } C[ ] \text{ to the root}
\]

\[
\text{and (for all } (i = v_i) \in (\cdot), \ \text{ndr}(C[ ], v_i))
\]

\[
\text{and (for all } (j = v_j) \in \rho, \ \text{ndr}(C[ ], v_j))
\]

\[
\iff \text{true and true and true}
\]

\[
\iff \text{true}
\]

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

\[
\begin{align*}
\rho \vdash e' & \Rightarrow \\
\rho \vdash \lambda x.e & \Rightarrow \langle \lambda x.e, (\cdot), \rho \rangle
\end{align*}
\]

\[
\begin{align*}
\rho \vdash e' (\lambda x.e) & \Rightarrow \\
\end{align*}
\]

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

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

\[
\begin{align*}
\rho \vdash e_1 & \Rightarrow \langle \lambda y.\lambda x.e, \rho_1, \rho_n \rangle, \\
\rho \vdash e_2 & \Rightarrow v, \ (y = v), \rho_1(\rho_2) \vdash \lambda x.e \Rightarrow \\
\rho \vdash e_1 e_2 & \Rightarrow \\
\end{align*}
\]

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

**Corollary 1** (Modified dynamic criterion implies dynamic criterion). If \( \rho_0 \) 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 \( \rho_0 \) 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 $\lambda' x$ binds occurrences of $x'$. We write $\lambda' x . e_{fi}$ so that $e_{fi}$ is easily identified as the body of the $\lambda$-abstraction $\lambda' x$. Also, $\text{rec}$ will be treated like $\lambda$: write $\text{rec'} f . \lambda' x . e_{fi}$. 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, $\phi, \gamma : CDescription = \text{Label} \rightarrow \mathcal{P}(\text{Label})$, with the following intended meanings:

- $\phi e$ is the set of closures that $e_{fi}$ in $\lambda' x . e_{fi}$ can evaluate to.
- $\gamma e$ is the set of closures that $\lambda' x . e_{fi}$ can be applied to, i.e., the set of closures that $\lambda' x$ can be bound to. $\phi$ is called the result closure description and $\gamma$ is called the argument closure description.

To help calculate $\phi$ and $\gamma$, we define two analysis functions, $\mathcal{C}_a$ and $\mathcal{C}_p$, with the following intended meanings:

- $\mathcal{C}_a[e]\phi e$ is the set of closures that expression $e$ can evaluate to.
- $\mathcal{C}_p[e]\psi e'$ is the set of closures that $\lambda' x . e_{fi}$ 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

$$
\begin{cases}
\phi e = \mathcal{C}_a[e]\phi e, \forall e \in \text{Label}, \\
\gamma e = \mathcal{C}_p[e_0]\psi e, \forall e \in \text{Label}.
\end{cases}
$$

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

$$
(\phi, \gamma) = (\lambda' \mathcal{C}_a[e_{fi}]\phi e, \mathcal{C}_p[e_0]\psi e)
$$
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 $\rho$, and any identifier $x'$ in its domain, if $\rho(x')$ evaluates to a closure, say, $\ell'$, then we want $\ell' \in \gamma\ell$ and we say that $\rho$ is consistent with respect to $\gamma$. With this notion of consistency, we can show that the closure analysis is sound with respect to the dynamic semantics of Fig. 5. This means that in an environment $\rho$ which is consistent with respect to $\gamma$, if an expression $e$ evaluates to a closure corresponding to the $\lambda$-abstraction $\ell$, then $\ell$ 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 $\lambda$-abstractions that the node might produce during its actual evaluation.

```
{3} @
  / \\
{1} \lambda^1f \quad \lambda^3x \quad {3}
     |     |
{3} @ x^3 { }
```

{2} $\lambda^2y f^1 \quad {3}

{3} $f^1$

For instance, the application $(\lambda^2 y . f^1)f^1$ can evaluate to $\lambda$-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]:

$$
\begin{align*}
\mathcal{C}_a & : \text{Expression} \rightarrow C\text{Description} \\
\mathcal{C}_p & : C\text{Description} \rightarrow P(\text{Label})
\end{align*}
$$

$$
\begin{align*}
\mathcal{C}_a[e]\phi\gamma & = \{\} \\
\mathcal{C}_a[x']\phi\gamma & = \gamma\ell \\
\mathcal{C}_a[\lambda^x e]\phi\gamma & = \{\ell\} \\
\mathcal{C}_a[e_1 e_2]\phi\gamma & = \bigcup_{\ell \in \mathcal{C}_a[e_1]\phi\gamma} \phi\ell \\
\mathcal{C}_a[\text{rec}\ell f. \lambda^{x_2} e]\phi\gamma & = \{\ell_2\}
\end{align*}
$$
\[ \mathcal{C}_\rho : \text{Expression} \rightarrow \text{CDescription} \rightarrow \text{CDescription} \rightarrow \text{Label} \rightarrow \mathcal{P}(\text{Label}) \]

\[ \begin{align*}
\mathcal{C}_\rho [x] &\phi \gamma \ell' = \{ \} \\
\mathcal{C}_\rho [x^f] &\phi \gamma \ell' = \{ \} \\
\mathcal{C}_\rho [\lambda^t x. e_f] &\phi \gamma \ell' = \mathcal{C}_\rho [e_f] \phi \gamma \ell' \\
\mathcal{C}_\rho [e_1 e_2] &\phi \gamma \ell' = \begin{cases} 
\mathcal{C}_\rho [e_1] \phi \gamma \ell' \cup \mathcal{C}_\rho [e_2] \phi \gamma \ell', & \text{if } \ell' \notin \mathcal{C}_\rho [e_1] \phi \gamma \\
\mathcal{C}_\rho [e_1] \phi \gamma \ell' \cup \mathcal{C}_\rho [e_2] \phi \gamma \ell' \cup \mathcal{C}_\rho [e_2] \phi \gamma, & \text{otherwise}
\end{cases} \\
\mathcal{C}_\rho [\text{rec} f. \lambda^t x. e_f] &\phi \gamma \ell' = \mathcal{C}_\rho [e_f] \phi \gamma \ell' \cup \{ \ell_2 \mid \ell' = \ell_1 \}
\end{align*} \]

We show that closure analysis is sound with respect to the dynamic semantics of Fig. 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 \( \gamma \), type assignment \( \pi \), and environment \( \rho \), where \( \models \rho : \pi^d \), \( \rho \) is consistent with respect to \( \gamma \) iff:

- For all \( z^m \in \text{Dom}(\rho) \), if \( \rho(z^m) = (\lambda^t x. e_f, \rho_l, \rho_n) \), then
  
  (i) \( ym \supseteq \{ \ell \} \),
  
  (ii) \( \rho_l \) and \( \rho_n \) are consistent with respect to \( \gamma \).
- For all \( f^m \in \text{Dom}(\rho) \), if \( \rho(f^m) = \mu f.(\lambda^t x. e_f, \rho') \), then
  
  (i) \( ym = \{ \ell \} \),
  
  (ii) \( \rho' \) is consistent with respect to \( \gamma \).

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 \( \rho \) is an initial environment iff: \( \rho = \{(x = v) \mid x \in \text{Dom}(\rho) \text{ and } v \text{ is of ground type}\} \) For closure description \( \gamma \), type assignment \( \pi \) and environment \( \rho \) where \( \models \rho : \pi \) and \( \rho \) is an initial environment, \( \rho \) is consistent with respect to \( \gamma \).

**Theorem 2 (Soundness).** For all \( (\phi, \gamma) \), for all \( e_0 \) if \( (\phi, \gamma) \) is a solution for \( e_0 \) of \( \mathcal{C}_d \) and \( \mathcal{C}_p \), and for all \( \rho \) if \( \rho \) is consistent with respect to \( \gamma \), then for all (finite) derivation trees \( \rho \vdash e_0 \Rightarrow v \), if \( v \) is a closure, say, \( (\lambda^t x. e_f, \rho_l, \rho_n) \), then \( \ell \in \mathcal{C}_d[e_0] \phi \gamma \).

**Proof.** We prove a stronger result: In addition to showing \( \ell \in \mathcal{C}_d[e_0] \phi \gamma \), we show that \( \rho_l \) and \( \rho_n \) are consistent with respect to \( \gamma \). Assume for any given expression \( e_0 \), \( (\phi, \gamma) \) is a solution for \( e_0 \) of \( \mathcal{C}_d \) and \( \mathcal{C}_p \). Also assume that \( \rho \) is consistent. Then the proof is by induction on the depth of inference of the inferred sentence.

---

\(^6\) Read \( \models \rho : \pi \) as \( \rho \) has type \( \pi \), see Appendix A.
The interesting case is that of application.

**Case 1:**

\[
\rho \vdash e_1 \Rightarrow (\lambda x. (\lambda y. e_{\ell'})(\ell_1, \rho_1, \rho_n)) \quad \rho \vdash e_2 \Rightarrow v
\]

\[
\rho \vdash (e_1, e_2) \Rightarrow (\lambda y. e_{\ell'})(x = v), \rho_1, \rho_n)
\]

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

\( \ell \in \mathbb{C}_a[e_1] \phi \gamma \) and \( \rho_1 \) and \( \rho_n \) are consistent with respect to \( \gamma \),

Hence, \((x = v), \rho_1 \) is consistent with respect to \( \gamma \). Now

\[
\begin{align*}
\mathbb{C}_a[e_1 e_2] \phi \gamma &= \bigcup \{ \phi \ell' \mid \ell' \in \mathbb{C}_a[e_1] \phi \gamma \} \\
&= \phi \ell', \text{ since } \ell \in \mathbb{C}_a[e_1] \phi \gamma \\
&= \mathbb{C}_a[\lambda y. e_{\ell'}] \phi \gamma \text{ by hypothesis} \\
&= \{ \ell' \}
\end{align*}
\]

Hence, \( \ell' \in \mathbb{C}_a[e_1 e_2] \phi \gamma \). We conclude further that the local environment \((x = v), \rho_1 \) is consistent with respect to \( \gamma \) as is \( \rho_n \).

(b) Let \( v \) be a closure, say, \( (\lambda m. y. e_m, \rho_1', \rho_2') \). By the induction hypothesis on the shorter proof tree of \( e_1 \),

\( \ell \in \mathbb{C}_a[e_1 e_2] \phi \gamma \) and \( \rho_1 \) and \( \rho_n \) are consistent with respect to \( \gamma \).

By the induction hypothesis on the shorter proof tree of \( e_2 \),

\( m \in \mathbb{C}_a[e_1] \phi \gamma \) and \( \rho_1' \) and \( \rho_2' \) are consistent with respect to \( \gamma \).

To show that \((x^\ell = (\lambda m. y. e_m, \rho_1', \rho_2'))), \rho_1 \) is consistent with respect to \( \gamma \), it suffices to show that \( \gamma \cup \{ m \} \).

By hypothesis,

\[
\begin{align*}
\gamma \cup \ell' &= \mathbb{C}_a[e_1 e_2] \phi \gamma \\
&= \mathbb{C}_a[e_1] \phi \gamma \ell' \cup \mathbb{C}_a[e_2] \phi \gamma \ell' \cup \mathbb{C}_a[e_2] \phi \gamma \\
&\supseteq \mathbb{C}_a[e_2] \phi \gamma \\
&\supseteq \{ m \}
\end{align*}
\]

Now

\[
\begin{align*}
\mathbb{C}_a[e_1 e_2] \phi \gamma &= \bigcup \{ \phi \ell' \mid \ell' \in \mathbb{C}_a[e_1] \phi \gamma \} \\
&\supseteq \phi \ell', \text{ since } \ell \in \mathbb{C}_a[e_1] \phi \gamma \\
i.e.
&\supseteq \mathbb{C}_a[\lambda y. e_{\ell'}] \phi \gamma \text{ by hypothesis} \\
i.e.
&\supseteq \{ \ell' \}
\end{align*}
\]

Hence, \( \ell' \in \mathbb{C}_a[e_1 e_2] \phi \gamma \).
Case 2:

\[ \rho \vdash e_1 \Rightarrow (\lambda x. e_{\ell}, \rho_1, \rho_n) \quad \rho \vdash e_2 \Rightarrow v \quad (x^\prime = v), \rho_1 \@ \rho_n \vdash e_{\ell} \Rightarrow w \]

\[ \rho \vdash (e_1 e_2) \Rightarrow w \]

(a) \( w \) is not a closure. Trivial.

(b) \( w \) is a closure, say, \( (\lambda z. e_n, \rho'_1, \rho'_g) \). There are two subcases.

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

\[ \ell \in \mathcal{A}_a[[e_1]]_{\phi \gamma} \] and \( \rho_1 \) and \( \rho_n \) are consistent with respect to \( \gamma \).

Hence, \( (x^\prime = v), \rho_1 \) is consistent with respect to \( \gamma \). Hence \( ((x^\prime = v), \rho_1)@\rho_n \) is consistent with respect to \( \gamma \).

Therefore, by the induction hypothesis on \( e_{\ell} \), \( n \in \mathcal{A}_a[[e_{\ell}]]_{\phi \gamma} \) and \( \rho'_1, \rho'_g \) are consistent with respect to \( \gamma \). Now

\[ \mathcal{A}_a[[e_1 e_2]]_{\phi \gamma} = \bigcup \{ \phi \ell' \mid \ell' \in \mathcal{A}_a[[e_1]]_{\phi \gamma} \} = \phi \ell', \text{ since } \ell \in \mathcal{A}_a[[e_1]]_{\phi \gamma} = \mathcal{A}_a[[e_{\ell}]]_{\phi \gamma} \text{ by hypothesis} = \{n\} \]

Therefore, \( n \in \mathcal{A}_a[[e_1 e_2]]_{\phi \gamma} \).

(ii) Let \( v \) be a closure, say, \( (\lambda y. e_m, \rho''_1, \rho''_g) \). It suffices to show \( (x^\prime = v), \rho_1 \) is consistent with respect to \( \gamma \), which we do in a manner similar to Case 1, part (b). The rest follows as in (i). \( \square \)

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 \( (\phi, \gamma) \), and for each application subexpression \( e_{11} e_{12} \) within \( e_0 \), we define the sets

\[ \mathcal{A}_i = \mathcal{A}_a[[e_{11} e_{12}]]_{\phi \gamma}, \quad \mathcal{O}_i = \mathcal{O}_a[[e_{11}]]_{\phi \gamma} \]

\( \mathcal{A}_i \) denotes the set of labels of the \( \lambda \)-abstractions that the application \( e_{11} e_{12} \) can evaluate to, and \( \mathcal{O}_i \) denotes the set of labels of the \( \lambda \)-abstractions that the operator \( e_{11} \) can evaluate to.

Say that \( e_{11} e_{12} \) evaluates as follows:

\[ \rho \vdash e_{11} \Rightarrow (\lambda x. e_{\ell}, \rho_1, \rho_n) \quad \rho \vdash e_{12} \Rightarrow v \quad \rho' \vdash e_{\ell} \Rightarrow (\lambda y. e_{\ell'}, \rho'_1, \rho'_n) \quad \rho \vdash e_{11} e_{12} \Rightarrow (\lambda y. e_{\ell'}, \rho'_1, \rho'_n) \]

where \( \rho' \) is \( (x^\prime = v), \rho_1 @ \rho_n \).
By the soundness of closure analysis, \( \ell' \in \mathcal{A}_i \) and \( \ell \in \mathcal{C}_i \). 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 \( \models \rho'_n : \pi'_n \). 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 \( (x' = v), \rho_l @ \rho_n \neq \rho'_n \), we can do this with \( \tau, \pi_l, \pi_n \) and \( \pi'_n \):

\[
(x' : \tau) \oplus \pi_l \oplus \pi_n \neq \pi'_n
\]

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 \( \lambda \)-abstraction). A \( \lambda \)-abstraction \( \lambda^{\ell'} x_1 \cdots \lambda^{\ell_i} x_n . e_{\ell} \) is maximal in \( \pi_0 \) if there does not exist an expression \( \lambda^{\ell_0} x_0 . \lambda^{\ell_1} x_1 \cdots \lambda^{\ell_i} x_n . e_{\ell} \) in \( \pi_0 \). Let \( \text{abs}(\ell) \) denote the abstract syntax tree for the \( \lambda \)-abstraction labeled \( \ell \), namely, \( \lambda^{\ell'} x . e_{\ell} \). Let \( \text{max}(\text{abs}(\ell)) \) denote the maximal \( \lambda \)-abstraction containing \( \text{abs}(\ell) \), namely, \( \lambda^{\ell_0} x_0 . \cdots . \lambda^{\ell_i} x_n . \lambda^{\ell'} x . e_{\ell} \). As above, for every application \( e_1 e_2 \) in \( \pi_0 \), define

\[
\mathcal{A}_i = \mathcal{C}_a [e_1 e_2] \phi \gamma \text{ and } \mathcal{C}_i = \mathcal{C}_a [e_1] \phi \gamma.
\]

**Criterion 3** (Static criterion). For all expressions \( \pi \vdash e : \tau \), for closure analysis \((\phi, \gamma)\), for \( \models \rho : \pi \), and for \( \rho \) consistent with respect to \( \gamma \), \( e \) is statically stackable if for every application, \( e_1 e_2 \), in \( e \) and its associated sets \( \mathcal{A}_i, \mathcal{C}_i \), for all \( \ell' \in \mathcal{A}_i \), for all \( \ell \in \mathcal{C}_i \),

- if \( \lambda^{\ell'} x_1 . \cdots . \lambda^{\ell'} x_i . \lambda^{\ell'} y . e_{\ell'} = \text{max}(\text{abs}(\ell')) \), and \( \lambda^{\ell'} x . e_{\ell} = \text{abs}(\ell) \), and
- \( \pi_a \vdash \lambda^{\ell'} x_1 . \cdots . \lambda^{\ell'} x_i . \lambda^{\ell'} y . e_{\ell'} : \tau'_1 \rightarrow \tau'_2 \), and \( \pi_a \vdash \lambda^{\ell'} x . e_{\ell} : \tau_1 \rightarrow \tau_2 \),

then \( (x' : \tau) \oplus \pi_a \neq \pi_a \).

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 \( \pi \). Its labeled abstract syntax tree annotated with closure analysis information, is shown in Section 4.2.
Consider the application \((\lambda^1 f : \text{int} \rightarrow \text{int}.(\lambda^2 y : \text{int} \rightarrow \text{int}. f^1))f^1\). Closure analysis predicts that the application must evaluate to the closure representing \(\lambda\)-abstraction 3. This \(\lambda\)-abstraction has type assignment \(\pi\) and is maximal. Consider now the operator part of the application, namely, \(\lambda^1 f : \text{int} \rightarrow \text{int}.(\lambda^2 y : \text{int} \rightarrow \text{int}. f^1)f^1\). Closure analysis predicts that it must evaluate to the closure representing \(\lambda\)-abstraction 1. The type assignment for \(\lambda\)-abstraction 1 is also \(\pi\). Note that \((f^1 : \text{int} \rightarrow \text{int}) \oplus \pi \neq \pi\). Thus, the static criterion is satisfied for this application.

Consider the other application \((\lambda^2 y : \text{int} \rightarrow \text{int}. f^1)f^1\). Closure analysis predicts that the application must evaluate to the closure representing \(\lambda\)-abstraction 3. This \(\lambda\)-abstraction has type assignment \(\pi\) and is maximal. Consider now the operator part of the application, namely, \(\lambda^2 y : \text{int} \rightarrow \text{int}. f^1\). Closure analysis predicts that it must evaluate to the closure representing \(\lambda\)-abstraction 2. The type assignment for \(\lambda\)-abstraction 2 is \((f^1 : \text{int} \rightarrow \text{int}) \oplus \pi\). Note that \((\lambda^2 : \text{int} \rightarrow \text{int}, f^1 : \text{int} \rightarrow \text{int}) \oplus \pi \neq \pi\). 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 \(n \in \mathbb{N}\), for closure analysis \((\phi, \gamma)\), for an environment \(\rho_0\) such that \(\models \rho_0 : n \in \pi_0\) and \(\rho_0\) is consistent with respect to \(\gamma\), if \(n \in \rho_0 : \tau_0\) is statically-stackable, then it satisfies the modified dynamic criterion.

**Proof.** Let \(n \in \rho_0 : \tau_0\) and let \((\phi, \gamma)\) be its closure analysis. Let \(\models \rho_0 : n \in \pi_0\), and let \(\rho_0\) be consistent with respect to \(\gamma\) and \(\rho_0 \models e_0 \rightarrow v_0\).

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

\[
\rho \vdash e_{i1} \Rightarrow (\lambda^\ell_x e_{i1}, \rho_{i1}, \rho_n) \quad \rho \vdash e_{i2} \Rightarrow v \quad \rho' \vdash e_{i} \Rightarrow (\lambda^{\ell'} y e_{i'}, \rho_{i'}, \rho_n')
\]

\[
\rho \models e_{i1} e_{i2} \Rightarrow (\lambda^{\ell''} y e_{i''}, \rho_{i''}, \rho_n')
\]

in \(\rho_0 \models e_0 \Rightarrow v_0\), where \(\rho' = (x = v) \rho_{i1} \rho_{i2}\), and must ensure that \(\rho' \neq \rho_{n'}\).

Since \(\rho \models e_{i1} \Rightarrow (\lambda^\ell x e_{i1}, \rho_{i1}, \rho_n)\), therefore, by soundness of closure analysis, \(\ell \in \mathcal{A}_i\). By subject reduction we know that there exist type assignments \(\pi_{i1}\) and \(\pi_{n}\) such that \(\pi_{i1} \oplus \pi_{n} \vdash \lambda^\ell x e_{i1} : \tau_1 \rightarrow \tau_2\), \(\text{Dom}(\pi_{i1}) = \text{Dom}(\rho_{i1})\) and \(\text{Dom}(\pi_{n}) = \text{Dom}(\rho_{n})\). Let \(\pi = \pi_{i1} \oplus \pi_{n}\). Further, since \(\rho \models e_{i1} e_{i2} \Rightarrow (\lambda^{\ell''} y e_{i''}, \rho_{i''}, \rho_n')\), therefore, \(\ell'' \in \mathcal{A}_i\). Again by subject reduction we know that there exist type assignments \(\pi_{i1}'\) and \(\pi_{n}\) such that \(\pi_{i1}' \oplus \pi_{n} \vdash \lambda^{\ell''} y e_{i''} : \tau_1 \rightarrow \tau_2\), \(\text{Dom}(\pi_{i1}') = \text{Dom}(\rho_{i1}')\) and \(\text{Dom}(\pi_{n}) = \text{Dom}(\rho_{n})\). Consider \(\max(\text{abs}(\ell'')) = \ell''_1 x_1 \cdots \ell''_{i1} x_{i1} \ell''_y y e_{i1}\). Therefore, \(\pi_{n} \vdash \lambda^{\ell''_1} x_{i1} \cdots \lambda^{\ell''_{i1}} x_{i1} \ell''_y y e_{i1} : \tau, \{x_{i1} : \tau_1, \cdots, x_{i1} : \tau_i\} = \pi_{i1}'.\) By the static criterion we know that \((\lambda^{\ell''} : \tau) \oplus \pi \neq \pi_{n}'\). Hence \(\rho' \neq \rho_{n}'\).

5. Implementation

In this section, we present the implementation of env-stackability as presented in Fig. 1. The BNF of the language is reproduced in Fig. 9. It differs from the one in
\[\begin{align*}
&\ell \in \text{BaseType} & \ell ::= \text{nat} \mid \text{bool} \\
&\tau \in \text{Type} & \tau ::= \ell \mid \tau_1 \rightarrow \tau_2 \\
&\pi \in \text{TypeAssignment} & \pi ::= \{\tau_i : \tau_i \}_{i \geq 0} \\
&c \in \text{ConstExpression} \\
&x \in \text{Identifier} \\
&e \in \text{Expression} & e ::= c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \ast e_2 \mid e_1 = e_2 \mid e_1 > e_2 \mid \text{succ} \mid \text{pred} \mid \text{izero} \mid \\
& & \lambda x : \tau . e \mid e_1 e_2 \mid \text{if } e_1 e_2 e_3 \mid \text{rec } f . \lambda x : \tau . e
\end{align*}\]

Fig. 9. BNF of the simply typed, call-by-value lambda calculus.

Fig. 2 in that some arithmetic and boolean operations and the conditional have been added to get interesting programs.

We are interested in implementing the subset of the simply typed call-by-value \(\lambda\)-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:

<table>
<thead>
<tr>
<th>Address of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of closure</td>
</tr>
<tr>
<td>Number of arguments</td>
</tr>
<tr>
<td>Pointer to value₀</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Pointer to valueₖ</td>
</tr>
<tr>
<td>Static link</td>
</tr>
<tr>
<td>value₀</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>valueₖ</td>
</tr>
</tbody>
</table>

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 \(\rho\). The layout of this closure is as follows:

<table>
<thead>
<tr>
<th>Address to code for (\lambda y.x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Pointer to argument (x)</td>
</tr>
<tr>
<td>Static link to (\rho)</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Recursive closures are of the form \(\mu f.(\lambda x.e, \rho)\). 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 Fig. 3.5 illustrates.

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

\[
\text{fib} \text{ function is the naive Fibonacci function and in this example it calculates the fibonacci of 26.}
\]

```
(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 \text{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:

<table>
<thead>
<tr>
<th>Program</th>
<th>Max. stack size</th>
<th>Env. Touches</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fib26</code></td>
<td>383</td>
<td>2206958</td>
</tr>
<tr>
<td><code>tak</code></td>
<td>388</td>
<td>381651</td>
</tr>
<tr>
<td><code>sum(interval...)</code></td>
<td>511575</td>
<td>9003</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Program</th>
<th>Max. heap size</th>
<th>Env. Touches</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fib26</code></td>
<td>4413988</td>
<td>2206958</td>
</tr>
<tr>
<td><code>tak</code></td>
<td>763396</td>
<td>381651</td>
</tr>
<tr>
<td><code>sum(interval...)</code></td>
<td>512584</td>
<td>9003</td>
</tr>
</tbody>
</table>

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, `letregion ρ in e 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 $\lambda$, and extent is controlled by $\text{letregion}$. 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 envi-
ronments for a call-by-value $\lambda$-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, $e_1 e_2$, evaluates as follows:

$$
\rho \vdash e_1 \Rightarrow \langle \lambda x\cdot e_1, \rho_1, \rho_n \rangle \quad \rho \vdash e_2 \Rightarrow v \quad \rho' \vdash e_3 \Rightarrow \langle \lambda' y\cdot e_3, \rho'_1, \rho'_n \rangle
$$

$$
\rho \vdash e_1 e_2 \Rightarrow \langle \lambda'' y\cdot e_3, \rho'_1, \rho''_n \rangle
$$

where $\rho'$ is $(x' = v), \rho_1 \odot \rho_n$, then we require that $\rho' \neq \rho''_n$.

For all nodes that satisfy the modified dynamic criterion, we can put the bindings
$(x' = v), \rho_1$ on the stack: at the end of $e_3$'s evaluation they can be destroyed. Otherwise,
they must be put on the heap – they cannot be destroyed as they may be required in
a future computation of $e_3'$.

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 first Atlantique workshop on
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 $\Gamma \vdash c : \text{Value}_G \times \text{Type}$ that satisfies $\Gamma \vdash v : \tau$ iff:

(i) if $u$ is $c$ then $\Gamma \vdash u : \tau$.

(ii) if $u$ is $(\lambda x.e, \rho_1, \rho_n)$ then there exist type assignments $\pi_l$ and $\pi_n$ such that $\pi_l \equiv \pi_n \vdash \lambda x.e : \tau$, $\text{Dom}(\pi_l) = \text{Dom}(\rho_l)$ and $\Gamma \vdash \rho_l(x) : \pi_l(x)$ for all $x \in \text{Dom}(\rho_l)$, for $i \in \{l,n\}$.

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

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

Theorem A.1 (Subject reduction property for Georgeff-style semantics). If $\pi \vdash e : \tau$, $\Gamma \vdash \rho : \pi$, and $\rho \vdash e \Rightarrow v$, then $\Gamma \vdash v : \tau$.

The semantic objects and the standard dynamic (natural [6]) semantics for the call-by-value, simply typed $\lambda$-calculus are shown in Figs. 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 Figs. 5 and 11. If $\rho \triangleright e \Rightarrow v$ and $\rho' \vdash e \Rightarrow v'$, then $v \text{ rel } v'$ must satisfy

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

(ii) if $v = (\lambda x.e', \rho')$ then there exist environments $\rho_l$ and $\rho_n$ such that $v' = (\lambda x.e', \rho_l, \rho_n)$,

$\text{Dom}(\rho') = \text{Dom}(\rho_l@\rho_n)$ and $\rho'(x) \text{ rel } (\rho_l@\rho_n)(x)$ for all $x \in \text{Dom}(\rho')$.

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

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

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

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

Fig. 10. Semantic objects for the standard dynamic semantics.
\[ \rho \triangleright c \Rightarrow c \quad \rho \triangleright \lambda x.e \Rightarrow (\lambda x.e, \rho) \]
\[ \rho \triangleright f \Rightarrow (\lambda x.e, \langle f = \mu f. (\lambda x.e, \rho') \rangle, \rho'), \text{if } \langle f = \mu f. (\lambda x.e, \rho') \rangle \in \rho \]
\[ \rho \triangleright x \Rightarrow v \text{ if } (x = v) \in \rho, \text{ otherwise} \]
\[ \rho \triangleright e_1 \Rightarrow (\lambda x.e, \rho') \quad \rho \triangleright e_2 \Rightarrow v \quad (x = v), \rho' \triangleright e \Rightarrow w \]
\[ \rho \triangleright e_1 e_2 \Rightarrow w \]
\[ \rho \triangleright \text{rec } f. \lambda x.e \Rightarrow (\lambda x.e, \langle f = \mu f. (\lambda x.e, \rho) \rangle, \rho) \]

Fig. 11. Standard dynamic semantics.

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)-(iii).

**Theorem A.2.** If \( \rho \triangleright e \Rightarrow v \text{ iff } \rho' \triangleright e \Rightarrow v' \), where \( v \triangleright rel \ v' \).

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

**References**