Selective Thunkification

Paul Steckler and Mitchell Wand*

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
{wand,steck}@ccs.neu.edu

Abstract. Recently, Amtoft presented an analysis and transformation for mapping typed call-by-name programs to call-by-value equivalents. Here, we present a comparable analysis and transformation for untyped programs using dataflow analysis. In the general case, the transformation generates thunks for call site operands of a call-by-name program. Using strictness information derived as part of a larger flow analysis, we can determine that some operands are necessarily evaluated under call-by-name, so the transformation does not need to generate thunks for them. The dataflow analysis is formulated as the solution to a set of constraints. We show that any solution to the constraints is sound, and that any such solution justifies the resulting transformation.

1 Introduction

In [1], Amtoft presented a strictness analysis for typed programs, and based on that analysis, developed an algorithm to transform call-by-name programs into call-by-value equivalents. The algorithm was correct in that the transformed programs were shown to satisfy a correctness predicate. Here, we adapt the flow analysis and transformation method developed in [12] and [14] for performing closure conversion to the call-by-name context, and incorporate a strictness analysis into that framework. We are then able to transform call-by-name programs written in an untyped language to call-by-value equivalents. Our transformation is correct in the same way as our closure conversion transformations: a transformed program evaluates to the transform of the value of the original program.

The paper is organized as follows:

- In Section 2, we provide motivation for the transformation, and discuss the problems addressed by the analysis.
- In Section 3, we present a source language and its call-by-name evaluator.
- In Section 4, we show how we distinguish occurrences of terms, and present an *occurrence evaluator*.

* Work supported by the National Science Foundation and DARPA under grants CCR-9002253 and CCR-9014603.

- In Section 5, we describe the annotations given to source programs. We describe the meanings of those annotations with a formal semantics, given as a series of satisfaction relations.
- In Section 6, we show how annotations may be derived as the solution to sets of constraints, presented as *local consistency conditions*, and that such solutions make the annotations true.
- In Section 7, we show that any solution to the constraints also justifies the selective thunkification transformation.
- We finish with a discussion of related work and conclusions about the current research and some possibilities for extending it.

2 Examples and discussion

In a call-by-name evaluation strategy, an argument to a procedure is not evaluated when the procedure is applied [10]. The formal parameter of the procedure becomes bound to the actual argument, not its value. If a free occurrence of the formal parameter in the procedure's body is referenced, then the actual argument is evaluated.

A call-by-name program can be transformed into a call-by-value program by turning all operands at all call sites into thunks. Let **loop** be a λ-term which diverges under call-by-value and call-by-name. Under call-by-name, the program

$$(\lambda x.c)\ \mathbf{loop}$$

evaluates to c. The operand is never evaluated, so it poses no problem. Under call-by-value, the same program will not terminate, because the attempt to evaluate the operand produces divergence. If we transform the program to

$$(\lambda x.c)\ (\mathbf{thunk\ loop})$$

then the transform will evaluate to c under call-by-value.

If a particular operand in a program is certain to be evaluated and divergence does not occur, thunkification may slow evaluation considerably. Consider:

$$(\lambda x.\ \ldots\ x\ \ldots\ x\ \ldots\ x\ \ldots)\ M$$

If we enclose the operand in a thunk, the transform is:

$$(\lambda x.\ \ldots\ (\mathbf{run}\ x)\ \ldots\ (\mathbf{run}\ x)\ \ldots\ (\mathbf{run}\ x)\ \ldots)\ (\mathbf{thunk}\ M)$$

We may have to run the thunk three times. If so, it is better to evaluate M just once, so that x becomes bound to M's value. Therefore, it is desirable to avoid thunkification when possible.

Our strictness analysis uses dataflow information to determine that some procedure arguments are certain to be evaluated, and so do not need to be made thunks by the transformation. Since only some operands are thunkified, the transformation is *selective*. For each occurrence in a program, we compute which of its free variables is certain to be evaluated. We also have a *closure analysis* to determine what procedures may be called at each call site. The key idea is: At a given call site, if every procedure which may be called is certain to evaluate a free occurrence of its formal parameter when the procedure's body is evaluated, then the argument at that call site is certain to be evaluated.

The transformation raises issues of protocol agreement. When a variable bound to a thunkified term is evaluated, the thunked term must be evaluated. If instead the variable is bound to an ordinary value, the value must not be treated as a thunk. Since we do not want to make that decision at run-time, the dataflow analysis assures that a given variable is bound only to thunks or only to values.

3 Languages; the term evaluator

Our source language, Λ_{in}, is an untyped λ-calculus with constants, boolean values, conditionals, and primitive operators, given by the following grammar:

$$\begin{aligned} M ::= {} & x \mid c \mid \mathbf{true} \mid \mathbf{false} \mid \lambda x.M \mid \mathit{PrimOp}\ M \mid MM \mid \\ & \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \end{aligned}$$

where

$$\mathit{PrimOp} ::= \mathbf{succ} \mid \mathbf{pred} \mid \mathbf{zero?}$$

The metavariable c ranges over a set of constants indexed by the integers. A *value* is any variable, constant, boolean value, or procedure in Λ_{in}. A *scalar value* is a constant or boolean value.

We write $M\{\rho\}$ for the term obtained by performing the substitution ρ on term M.

The call-by-name term evaluator is shown in Figure 1. We use the symbol $\underset{tn}{\Longrightarrow}$ for the relation on terms and their values.

The selective thunkification transformation produces terms in the output language Λ_{thunk}, given by the grammar:

$$\begin{aligned} M ::= {} & x \mid c \mid \mathbf{true} \mid \mathbf{false} \mid \lambda x.M \mid \mathit{PrimOp}\ M \mid MM \mid \\ & \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \mid \mathbf{thunk}\ M \mid \mathbf{run}\ M \end{aligned}$$

We will evaluate Λ_{thunk} terms call-by-value. There we use the symbol $\underset{t}{\Longrightarrow}$ for the relation on Λ_{thunk} terms and their values. Therefore, the term evaluation rules for Λ_{thunk} are the same as in Figure 1, except (1) we replace $\underset{tn}{\Longrightarrow}$ by $\underset{t}{\Longrightarrow}$, (2) the rule for applications becomes:

$$\frac{M \underset{t}{\Longrightarrow} \lambda x.Q \quad N \underset{t}{\Longrightarrow} N' \quad Q[N'/x] \underset{t}{\Longrightarrow} V}{MN \underset{t}{\Longrightarrow} V}$$

and (3), we add the rules

$$\mathbf{thunk}\ M \underset{t}{\Longrightarrow} \mathbf{thunk}\ M$$

$$\frac{M \underset{t}{\Longrightarrow} \mathbf{thunk}\ N \quad N \underset{t}{\Longrightarrow} P}{\mathbf{run}\ M \underset{t}{\Longrightarrow} P}$$

$$x \underset{tn}{\Longrightarrow} x \qquad c \underset{tn}{\Longrightarrow} c \qquad \lambda x.Q \underset{tn}{\Longrightarrow} \lambda x.Q$$

$$\frac{M \underset{tn}{\Longrightarrow} c_n}{\mathbf{succ}\ M \underset{tn}{\Longrightarrow} c_{n+1}} \quad \frac{M \underset{tn}{\Longrightarrow} c_n}{\mathbf{pred}\ M \underset{tn}{\Longrightarrow} c_{n-1}}$$

$$\frac{M \underset{tn}{\Longrightarrow} c_0}{\mathbf{zero?}\ M \underset{tn}{\Longrightarrow} \mathbf{true}} \quad \frac{M \underset{tn}{\Longrightarrow} c_n, \quad n \neq 0}{\mathbf{zero?}\ M \underset{tn}{\Longrightarrow} \mathbf{false}}$$

$$\frac{M \underset{tn}{\Longrightarrow} \lambda x.Q \quad Q[N/x] \underset{tn}{\Longrightarrow} V}{MN \underset{tn}{\Longrightarrow} V}$$

$$\frac{M \underset{tn}{\Longrightarrow} \mathbf{true} \quad N \underset{tn}{\Longrightarrow} Q}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P \underset{tn}{\Longrightarrow} Q} \quad \frac{M \underset{tn}{\Longrightarrow} \mathbf{false} \quad P \underset{tn}{\Longrightarrow} Q}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P \underset{tn}{\Longrightarrow} Q}$$

Fig. 1. Call-by-name term evaluator

4 Occurrences; occurrence evaluator

The call-by-name evaluator uses substitution, discarding information about the sources of terms. To track individual occurrences, we write an *occurrence evaluator* that represents terms as a substitution instances of occurrences in a program.

To refer to particular occurrences, we assign each occurrence an index which describes the path from the root of a program's parse tree to the occurrence. For source programs, an *occurrence index* is a finite string over the alphabet

$$\{bv, body, rator, rand, test, then, else\}$$

Therefore, the empty string represents the root of the parse tree, and $i.rator$ represents the operator of an ordinary or primitive operator application with index i, and so forth. We use $i.rand$ for the operand of an ordinary or primitive operator application with index i.

Terms that appear in a starting environment have their own identifying indices; we can suffix those indices with elements from the above alphabet to refer to their subterms. We write $[\![i]\!]$ for the term with index i. Constants are their own indices, so that $[\![c]\!] = c$. Let O be the set of all occurrence indices.

We use the predicates *Const*, *Var*, *Abs*, *PrimApp*, *App*, and *Cond* to test the syntactic category of occurrences of terms in Λ_{in}.

The following definitions describe data structures containing occurrences:

Definition 1. The following definitions are mutually referential.

1. An *occurrence closure* (i, ψ) is a pair consisting of the occurrence index of a term and an occurrence environment.
2. An *occurrence environment* ψ is a finite map from variables to occurrence closures.

This definition is well-founded, since it makes the empty map an occurrence environment. If (j, ψ') is in the range of an occurrence environment ψ, j may be the occurrence index of any term in Λ_{in}, not necessarily a value (cf. [14]). An occurrence environment ψ is a *scalar environment* iff $\forall x \in Dom(\psi),\ \psi(x) = (j, \psi')$ implies $[\![j]\!]$ is a scalar value.

An occurrence closure represents a term in Λ_{in}. We can retrieve the term represented by an occurrence closure by applying an unwinding function:

$$\mathcal{U}[\![(i, \psi)]\!] = [\![i]\!]\{\mathcal{U}[\![\cdot]\!] \circ \psi\}$$

This algorithm follows the inductive structure of Definition 1, so it terminates; $\psi = \emptyset$ serves as the base case.

We write $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ to indicate that (i, ψ) evaluates to (j, ψ'). We present the call-by-name occurrence evaluator in Figure 2. The occurrence closure evaluator simulates the operation of the term evaluator on unwound occurrence closures:

Theorem 2 (Simulation). *If* $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, *then* $\mathcal{U}[\![(i, \psi)]\!] \underset{tn}{\Longrightarrow} \mathcal{U}[\![(j, \psi')]\!]$

Proof. Induction on the definition of $\underset{ocn}{\Longrightarrow}$. ∎

Of course, the theorem would hold if we had defined $\underset{ocn}{\Longrightarrow}$ as the empty relation. The non-triviality of our definition of $\underset{ocn}{\Longrightarrow}$ is established by:

Theorem 3 (Adequacy). *If* $\mathcal{U}[\![(i, \psi)]\!] \underset{tn}{\Longrightarrow} M$ *then* $\exists j, \psi'$ *such that* $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ *and* $M = \mathcal{U}[\![(j, \psi')]\!]$.

Proof. Induction on the size of the derivation $\mathcal{U}[\![(i, \psi)]\!] \underset{tn}{\Longrightarrow} M$. ∎

5 Annotations and their semantics

Here we describe the language of program annotations and some definitions required to describe the semantics of annotations.

A *flow* describes the set of procedures to which an occurrence might evaluate. A *thunk tag* describes whether or not its associated occurrence in operand position should be thunkified.

More formally, we have:

Definition 4. The following definitions are mutually referential.

1. A *flow* ϕ is a finite set of abstract closures.

$$\frac{Var(i) \wedge [\![i]\!] \notin Dom(\psi)}{(i,\psi) \underset{ocn}{\Longrightarrow} (i,\emptyset)} \qquad \frac{\begin{array}{l} Var(i) \wedge [\![i]\!] \in Dom(\psi) \\ \psi([\![i]\!]) \underset{ocn}{\Longrightarrow} (j,\psi') \end{array}}{(i,\psi) \underset{ocn}{\Longrightarrow} (j,\psi')}$$

$$\frac{Const(i)}{(i,\psi) \underset{ocn}{\Longrightarrow} (i,\emptyset)} \qquad \frac{Abs(i)}{(i,\psi) \underset{ocn}{\Longrightarrow} (i,\psi)}$$

$$\frac{PrimApp(i) \wedge [\![i.rator]\!] = \mathbf{succ} \quad (i.rand,\psi) \underset{ocn}{\Longrightarrow} (j,\psi') \quad [\![j]\!] = c_n}{(i,\psi) \underset{ocn}{\Longrightarrow} (c_{n+1},\emptyset)}$$

$$\frac{PrimApp(i) \wedge [\![i.rator]\!] = \mathbf{pred} \quad (i.rand,\psi) \underset{ocn}{\Longrightarrow} (j,\psi') \quad [\![j]\!] = c_n}{(i,\psi) \underset{ocn}{\Longrightarrow} (c_{n-1},\emptyset)}$$

$$\frac{PrimApp(i) \wedge [\![i.rator]\!] = \mathbf{zero}? \quad (i.rand,\psi) \underset{ocn}{\Longrightarrow} (j,\psi') \quad [\![j]\!] = c_0}{(i,\psi) \underset{ocn}{\Longrightarrow} (\mathbf{true},\emptyset)}$$

$$\frac{PrimApp(i) \wedge [\![i.rator]\!] = \mathbf{zero}? \quad (i.rand,\psi) \underset{ocn}{\Longrightarrow} (j,\psi') \quad [\![j]\!] = c_n,\ n \neq 0}{(i,\psi) \underset{ocn}{\Longrightarrow} (\mathbf{false},\emptyset)}$$

$$\frac{\begin{array}{l} App(i) \\ (i.rator,\psi) \underset{ocn}{\Longrightarrow} (j,\psi'),\ Abs(j) \\ (j.body,\psi'[[\![j.bv]\!] \mapsto (i.rand,\psi)]) \underset{ocn}{\Longrightarrow} (k,\psi'') \end{array}}{(i,\psi) \underset{ocn}{\Longrightarrow} (k,\psi'')}$$

$$\frac{\begin{array}{l} Cond(i) \\ (i.test,\psi) \underset{ocn}{\Longrightarrow} (j,\psi'), \quad [\![j]\!] = \mathbf{true} \\ (i.then,\psi) \underset{ocn}{\Longrightarrow} (k,\psi'') \end{array}}{(i,\psi) \underset{ocn}{\Longrightarrow} (k,\psi'')}$$

$$\frac{\begin{array}{l} Cond(i) \\ (i.test,\psi) \underset{ocn}{\Longrightarrow} (j,\psi'), \quad [\![j]\!] = \mathbf{false} \\ (i.else,\psi) \underset{ocn}{\Longrightarrow} (k,\psi'') \end{array}}{(i,\psi) \underset{ocn}{\Longrightarrow} (k,\psi'')}$$

Fig. 2. Rules for the call-by-name occurrence evaluator

2. A *thunk tag* π is an element of $\{id, th\}$.
3. A *thunk tag assignment* Π is a map from occurrences to thunk tags.
4. An *abstract closure* (i, A) is a pair consisting of the occurrence index of an abstraction and an environment proposition.
5. A *value proposition* $\mathcal{P}$ is a pair consisting of a flow and a thunk tag.
6. An *environment proposition* is a map from variables to value propositions.

Value propositions are ordered as follows: $(\phi, \pi) \leq (\phi', \pi')$ iff $\phi \subseteq \phi'$ and $\pi = \pi'$.

We also have:

Definition 5. A *strictness proposition* $\mathcal{S}$ is a set of variables.

Informally, a strictness proposition is the set of free variables of an occurrence certain to be evaluated when the occurrence is evaluated.

Next, we give the formal semantics of annotations for the selective thunkification analysis.

1. An occurrence environment ψ satisfies an environment proposition A under a thunk tag assignment Π, as follows:

$$\psi \overset{\Pi}{\models}_{env} A$$

iff $\forall x \in Dom(A)$

(a) $x \in Dom(\psi)$ and $\psi(x) \overset{\Pi}{\models}_{thunk} A(x)$, and

(b) if $A(x) = (\phi, \pi)$ and $\psi(x) = (i, \psi')$, then $\Pi(i) = \pi$

2. An occurrence closure (i, ψ) satisfies a value proposition (ϕ, π) under a thunk tag assignment Π, as follows:

$$(i, \psi) \overset{\Pi}{\models}_{thunk} (\phi, \pi)$$

iff

(a) $(i, \psi) \overset{\Pi}{\models}_{val} (\phi, \pi)$ or

(b) $(i, \psi) \overset{\Pi}{\models}_{ocn} (\phi, \pi)$

3. Define a relation:

$$(i, \psi) \overset{\Pi}{\models}_{val} (\phi, \pi)$$

iff

(a) $Var(i) \wedge [\![i]\!] \notin Dom(\psi) \wedge \Pi(i) = \pi$, or

(b) $Const(i)$, or

(c) $Abs(i)$, and $\exists A$ such that $(i, A) \in \phi$ and $\psi \overset{\Pi}{\models}_{env} A$

4. Define a relation:

$$(i, \psi) \overset{\Pi}{\models}_{ocn} (\phi, \pi)$$

iff

(a) $(Var(i) \wedge [\![i]\!] \in Dom(\psi))$, or $PrimApp(i)$, or $App(i)$, or $Cond(i)$, and

(b) if $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ then $(j, \psi') \overset{\Pi}{\models}_{val} (\phi, \pi)$

5. An evaluation $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ satisfies a strictness proposition $\mathcal{S}$, as follows:

$$(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \models_{strict} \mathcal{S}$$

iff $\forall x \in \mathcal{S}$, $\exists k, m, \psi'', \psi'''$ such that
(a) k is a free occurrence of x in $[\![i]\!]$,
(b) $(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''')$ is a subproof of $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, and
(c) if $x \in Dom(\psi)$, then $x \in Dom(\psi'')$ and $\psi(x) = \psi''(x)$

These definitions are well-founded, since occurrence closures are finite.

For the $\overset{\Pi}{\models}_{env}$ relation, we require pointwise $\overset{\Pi}{\models}_{thunk}$-satisfaction, and that the environment proposition gives the right thunk tags across environment lookups.

For the $\overset{\Pi}{\models}_{thunk}$ relation on occurrence closures and value propositions, we have to concern ourselves whether the occurrence closure is a value or not.[2] For values, we delegate to the $\overset{\Pi}{\models}_{val}$ relation; for other occurrence closures, we delegate to the $\overset{\Pi}{\models}_{ocn}$ relation, which itself depends on $\overset{\Pi}{\models}_{val}$.

The definition of $\models_{strict}$ makes precise the notion that a strictness proposition is a set of variables certain to be referenced during the evaluation of an occurrence.

The ordering on value propositions gives us:

Lemma 6. *If $(i, \psi) \overset{\Pi}{\models}_{val} \mathcal{P}$ and $\mathcal{P} \leq \mathcal{P}'$ then $(i, \psi) \overset{\Pi}{\models}_{val} \mathcal{P}'$.*

Proof. By the definition of $\overset{\Pi}{\models}_{val}$. ∎

6 Local consistency conditions; soundness

In Figure 3, we give the local consistency conditions which support the selective thunkification transformation.

An annotation map Γ is a map from occurrence indices to triples consisting of an environment proposition, a strictness proposition, and a value proposition. Therefore, $\Gamma(i) = (A_i, \mathcal{S}_i, \mathcal{P}_i)$. If the constraints given in Figure 3 are satisfied at every program node, we say an annotation map is *locally-consistent*. We say that an annotation map is *monovariant* iff for every abstract closure (j, B), $B = A_j$.

The flow and strictness proposition constraints can be solved by an iterative algorithm. For each occurrence i, start with $\phi_i = \emptyset$ and $\mathcal{S}_i = FV([\![i]\!])$. With each iteration, apply the constraints in Figure 3 so that abstract closures are added to the flows and variables are removed from the strictness propositions until a fixed point is reached. Monovariance implies there are finitely many abstract closures, so the algorithm terminates. Alternatively, the flow and strictness proposition constraints are solvable by a graph-based algorithm that runs in time $O(n^3)$ (see [9] for a similar graph-based algorithm for closure analysis only). Once the

[2] More accurately, the concern is whether the occurrence closure may appear on the right-hand side of $\underset{ocn}{\Longrightarrow}$.

$$Var(i) \Longrightarrow \begin{cases} A_i([\![i]\!]) = \mathcal{P}_i, \text{ and} \\ \mathcal{S}_i \subseteq \{[\![i]\!]\} \end{cases}$$

$$Const(i) \Longrightarrow \mathcal{S}_i = \emptyset$$

$$Abs(i) \Longrightarrow \begin{cases} A_{i.body} = A_i[[\![i.bv]\!] \mapsto \mathcal{P}_{i.bv}], \\ \{(i, A_i)\} \subseteq \phi_i, \\ \pi_i = th \Longrightarrow \forall \text{ free var. occurrences } j \text{ in } [\![i]\!],\ \pi_j = th, \text{ and} \\ \mathcal{S}_i = \emptyset \end{cases}$$

$$PrimApp(i) \Longrightarrow \begin{cases} A_{i.rand} = A_i, \\ \pi_i = th \Longrightarrow \forall \text{ free var. occurrences } j \text{ in } [\![i]\!],\ \pi_j = th, \text{ and} \\ \mathcal{S}_i \subseteq \mathcal{S}_{i.rand} \end{cases}$$

$$App(i) \Longrightarrow \begin{cases} A_{i.rator} = A_{i.rand} = A_i, \\ \pi_i = th \Longrightarrow \forall \text{ free var. occurrences } j \text{ in } [\![i]\!],\ \pi_j = th, \text{ and} \\ \forall (j, B) \in \phi_{i.rator} \\ \quad \begin{cases} Abs(j), \\ \mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}, \\ \mathcal{P}_{j.body} \leq \mathcal{P}_i, \text{ and} \end{cases} \\ \text{if } \forall (j, B) \in \phi_{i.rator}, [\![j.bv]\!] \in \mathcal{S}_{j.body} \\ \text{then } \mathcal{S}_i \subseteq \mathcal{S}_{i.rator} \cup \mathcal{S}_{i.rand} \\ \text{else } \begin{cases} \mathcal{S}_i \subseteq \mathcal{S}_{i.rator} \text{ and} \\ \pi_{i.rand} = th \end{cases} \end{cases}$$

$$Cond(i) \Longrightarrow \begin{cases} A_{i.test} = A_{i.then} = A_{i.else} = A_i, \\ \mathcal{P}_{i.then} \leq \mathcal{P}_i, \\ \mathcal{P}_{i.else} \leq \mathcal{P}_i, \\ \pi_i = th \Longrightarrow \forall \text{ free var. occurrences } j \text{ in } [\![i]\!],\ \pi_j = th, \text{ and} \\ \mathcal{S}_i \subseteq \mathcal{S}_{i.test} \cup (\mathcal{S}_{i.then} \cap \mathcal{S}_{i.else}) \end{cases}$$

Fig. 3. Local consistency conditions

flows have been solved, the constraints on thunk tags are solved by building and merging equivalence classes using standard disjoint-set operations [13]. The entire algorithm runs in time $O(n^3)$.

The Soundness Theorem requires another definition:

Definition 7. Let Γ be an annotation map, and let Π be a thunk tag assignment. An occurrence closure (i, ψ) is *well-thunked* with respect to Γ and Π iff

1. $Var(i) \wedge [\![i]\!] \notin Dom(\psi)$, or $Const(i)$, or

2. $\psi \overset{\Pi}{\models}_{env} A_i$, and $\forall x \in FV([\![i]\!]) \cap Dom(\psi)$, $\psi(x)$ is well-thunked with respect to Γ and Π

Since occurrence environments are finitely deep, this definition is well-founded. Any occurrence closure containing the empty occurrence environment is well-thunked, and so serves as the base case. Where the context assumes a particular annotation map and thunk tag assignment, we will just say that an occurrence closure is well-thunked.

Now we can state our first main result:

Theorem 8 (Soundness). *Let Γ be a monovariant and locally-consistent annotation map, and let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Let (i, ψ) be an occurrence closure which is well-thunked with respect to Γ and Π. Suppose $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$. Then for any subproof $(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''')$ of this derivation:*

1. *(k, ψ'') is well-thunked,*
2. *(m, ψ''') is well-thunked,*
3. *$(m, \psi''') \overset{\Pi}{\models}_{val} (\phi_k, \pi_k)$, and*
4. *$(k, \psi'') \underset{ocn}{\Longrightarrow} (m, \psi''') \models_{strict} \mathcal{S}_k$*

Proof. Induction on the size of the derivation that $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$.

The most difficult case in this proof is when $App(i)$. Some details for this case are given in Appendix A; full details appear in [11]. ∎

7 Selective thunkification; correctness

We can now define the selective thunkification transformation:

$$\mathcal{T} : O \rightarrow \Lambda_{thunk}$$

shown in Figure 4. We can extend the transformation to occurrence closures:

$$\hat{\mathcal{T}}(i, \psi) = \mathcal{T}(i)\{\mathcal{T}' \circ \psi\}$$

where

$$\mathcal{T}'(i, \psi) = \begin{cases} \hat{\mathcal{T}}(j, \psi') & \text{if } \pi_i = id \text{ and } (i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \\ \textbf{thunk}\ \hat{\mathcal{T}}(i, \psi) & \text{if } \pi_i = th \end{cases}$$

Note that the $\mathcal{T}'$ transformation invokes the occurrence evaluator on occurrence closures which are tagged id. It is possible that the evaluation will fail because of divergence. Even if the evaluation succeeds, the $\hat{\mathcal{T}}$ transformation of the result may fail. For these reasons, we need to impose some restrictions on occurrence closures and their annotations.

To obtain these results, we introduce the notions of *thunk-safety* and *thunk-acceptability*.

$$Var(i) \wedge \pi_i = id \Longrightarrow \mathcal{T}(i) = [\![i]\!]$$

$$Var(i) \wedge \pi_i = th \Longrightarrow \mathcal{T}(i) = \textbf{run}\, [\![i]\!]$$

$$Const(i) \Longrightarrow \mathcal{T}(i) = [\![i]\!]$$

$$Abs(i) \Longrightarrow \mathcal{T}(i) = \lambda [\![i.bv]\!] . \mathcal{T}(i.body)$$

$$PrimApp(i) \Longrightarrow \mathcal{T}(i) = [\![i.rator]\!]\ \mathcal{T}(i.rand)$$

$$App(i) \wedge \pi_{i.rand} = id \Longrightarrow \mathcal{T}(i) = \mathcal{T}(i.rator)\ \mathcal{T}(i.rand)$$

$$App(i) \wedge \pi_{i.rand} = th \Longrightarrow \mathcal{T}(i) = \mathcal{T}(i.rator)\ (\textbf{thunk}\ \mathcal{T}(i.rand))$$

$$Cond(i) \Longrightarrow \mathcal{T}(i) = \textbf{if}\ \mathcal{T}(i.test)\ \textbf{then}\ \mathcal{T}(i.then)\ \textbf{else}\ \mathcal{T}(i.else)$$

Fig. 4. Definition of the selective thunkification transformation

Definition 9. An occurrence closure (i, ψ) is *thunk-safe* with respect to an annotation map Γ iff $\forall x \in FV([\![i]\!]) \cap Dom(\psi)$, $\psi(x) = (j, \psi')$ implies either

1. $Var(j) \wedge [\![j]\!] \notin Dom(\psi')$, or $Const(j)$, or
2. $\pi_j = th$, and (j, ψ') is thunk-safe with respect to Γ

This definition is well-founded because of the finite depth of occurrence environments.

We can show:

Lemma 10. *If an occurrence closure (i, ψ) is thunk-safe with respect to an annotation map, then $\hat{\mathcal{T}}(i, \psi)$ is well-defined.*

Proof. Induction on the depth of ψ. ∎

Less restrictive than thunk-safety is:

Definition 11. An occurrence closure (i, ψ) is *thunk-acceptable* with respect to an annotation map Γ iff $\forall x \in FV([\![i]\!]) \cap Dom(\psi)$, $\psi(x) = (j, \psi')$ implies either

1. $\pi_j = id$ and either
   - $x \in \mathcal{S}_i$, and if $\exists k, \psi''$ such that $(j, \psi') \underset{ocn}{\Longrightarrow} (k, \psi'')$, then (k, ψ'') is thunk-safe with respect to Γ, or
   - $Var(j) \wedge [\![j]\!] \notin Dom(\psi')$, or $Const(j)$, or
2. $\pi_j = th$ and (j, ψ') is thunk-safe with respect to Γ

Because this definition just relies on Definition 9, it is well-founded.

If we have thunk-acceptability, then $\hat{\mathcal{T}}$ does not introduce divergence:

Lemma 12. *Let Γ be a monovariant and locally-consistent annotation map, let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$, and let (i, ψ) be an occurrence closure which is well-thunked with respect to Γ and Π, and thunk-acceptable with respect to Γ. If $\exists j, \psi'$ such that $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then $\hat{\mathcal{T}}(i, \psi)$ is well-defined.*

Proof. By Theorem 8 and the definitions of $\hat{\mathcal{T}}$ and thunk-acceptability. Since $\hat{\mathcal{T}}(i, \psi) = \mathcal{T}(i)\{\mathcal{T}' \circ \psi\}$, and $\mathcal{T}(i)$ is well-defined, we only need to show that the substitution is well-defined. ∎

If we start with a thunk-acceptable occurrence closure, its evaluation result is thunk-safe:

Lemma 13. *Let Γ be a monovariant and locally-consistent annotation map, and let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Let (i, ψ) be an occurrence closure which is well-thunked with respect to Γ and Π, and thunk-acceptable with respect to Γ. If $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then (j, ψ') is thunk-safe with respect to Γ.*

Proof. Routine induction on the size of the derivation that $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$. ∎

Therefore:

Lemma 14. *Let Γ be a monovariant and locally-consistent annotation map, and let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Let (i, ψ) be an occurrence closure which is well-thunked with respect to Γ and Π, and thunk-acceptable with respect to Γ. If $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then $\hat{\mathcal{T}}(j, \psi')$ is well-defined.*

Proof. Immediately by Lemmas 10 and 13. ∎

We need one more definition for the Correctness Theorem:

Definition 15. An occurrence closure (i, ψ) is *open-safe* with respect to an annotation map Γ iff for all free variable occurrences j in $[\![i]\!]$:

1. if $[\![j]\!] \notin Dom(\psi)$, then $\pi_j = id$
2. if $[\![j]\!] \in Dom(\psi)$, then $\psi([\![j]\!])$ is open-safe with respect to Γ

As usual, the well-foundedness of this definition proceeds from the finiteness of occurrence closures.

We can produce an occurrence closure that is an example of the last several definitions:

Lemma 16. *Let Γ be an annotation map, let Π be a thunk tag assignment, and let i be an occurrence index. If ψ is a scalar environment such that $FV([\![i]\!]) \subseteq Dom(\psi)$, and $\psi \overset{\Pi}{\models}_{env} A_i$, then (i, ψ) is well-thunked, thunk-safe, thunk-acceptable, and open-safe with respect to Γ.*

Proof. By the relevant definitions. ∎

Our correctness theorem is identical in form to that in [14]. Note, however, that we have $\underset{ocn}{\Longrightarrow}$ in a premise of the theorem, and $\underset{t}{\Longrightarrow}$ in the consequent.

Theorem 17 (Correctness). *Let Γ be a monovariant and locally-consistent annotation map, and let Π be the thunk tag assignment defined by $\forall i,\ \Pi(i) = \pi_i$. Let (i, ψ) be an occurrence closure which is well-thunked with respect to Γ and Π, and thunk-acceptable and open-safe with respect to Γ. If $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$, then $\hat{\mathcal{T}}(i, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(j, \psi')$.*

Proof. Induction on the size of the derivation that $(i, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$.

We illustrate this theorem by the diagram:

$$\begin{array}{ccc} (i,\psi) & \underset{ocn}{\Longrightarrow} & (j,\psi') \\ \big\downarrow \hat{\mathcal{T}} & & \big\downarrow \hat{\mathcal{T}} \\ \hat{\mathcal{T}}(i,\psi) & \underset{t}{\Longrightarrow} & \hat{\mathcal{T}}(j,\psi') \end{array}$$

The details of the proof appear in [11]. We mention the application case: that is where we make use of strictness information. If the operand has a thunk tag of id, the transformation will not produce a thunk, so the transformed operand will be evaluated under call-by-value. The Soundness Theorem (Theorem 8) and local consistency conditions assure that $(i.rand, \psi)$ is also evaluated by the call-by-name occurrence evaluator, allowing us to invoke the induction hypothesis. ■

Corollary 1. Let Γ be a monovariant and locally-consistent annotation map and let Π be the thunk tag assignment defined by $\forall i, \Pi(i) = \pi_i$. Let i be an occurrence index and ψ a scalar environment such that $FV([\![i]\!]) = Dom(\psi)$ and $\psi \overset{\Pi}{\models}_{env} A_i$. If $(i, \psi) \underset{ocn}{\Longrightarrow} (k, \psi')$, and $Const(k)$, then $\hat{\mathcal{T}}(i, \psi) \underset{t}{\Longrightarrow} \hat{\mathcal{T}}(k, \psi') = [\![k]\!]$.

We may use these results to compile any Λ_{in} program, as follows: Use the algorithm described on p. 8 to find an annotation Γ of the program that satisfies the local consistency conditions of Figure 3. Then transform the program to call-by-value, using Γ and the transformation $\hat{\mathcal{T}}$. Corollary 1 then asserts that if the resulting program is run under call-by-value in an appropriate scalar environment, the result is the same as if it had been run under call-by-name. So standard call-by-value compilation techniques may then be used on the transformed program.

8 Related work

The problem of transforming programs with delayed evaluation into call-by-value equivalents was first studied by Mycroft [8], using a first-order language. Instead of considering full call-by-name programs, in which arguments may be evaluated an arbitrary number of times, Mycroft looked at call-by-need programs, in which arguments are evaluated at most once. His strictness analysis was based on an abstract interpretation.

Many authors since have studied strictness analysis. Burn, Hankin, and Abramsky proved that a strictness analysis for a higher-order typed language based on an abstract interpretation is correct with respect to a standard interpretation [3]. Jensen presented a lattice-theoretic framework for comparing strictness analyses based on abstract interpretation and type inference [6]. Kuo and Mishra use a type system with strictness annotations on base types [7]. Wright developed a type system in which all types are decorated with strictness information [15].

Danvy and Hatcliff presented a continuation-passing style (CPS) conversion algorithm for an untyped language using thunkification and relying on strictness information [4]. Burn and Le Métayer claim the correctness of a CPS transformation for a typed language that allows the evaluation of arguments to strict procedures, although they do not give their own strictness analysis [2]. Hudak and Young proved the correctness of a strictness analysis for an untyped λ-calculus [5]; unlike ours, their analysis was not guaranteed to terminate.

Closest to the current work is that of Amtoft [1], who uses a type system derived from Wright to prove the correctness of a transformation from call-by-name to call-by-value. Our strictness propositions are borrowed from Amtoft, whose inference system computes a set of "needed variables" for a typed expression.

9 Conclusions

As far as we know, our work is the first provably correct strictness analysis and call-by-name to call-by-value transformation for an untyped higher-order language.

We believe that the current result demonstrates the robustness of our method. Enough is different under call-by-name to have required significant modification to our analysis, yet we were able fairly easily to graft on a strictness analysis and end up with a statement of correctness of the same form as our earlier correctness theorems.

We have now shown that our monovariant analysis and transformation method succeeds for both call-by-value and call-by-name programs. Under monovariance, all environments are merged so a procedure is represented by a single abstract closure. A possible extension is to attempt a transformation based on a polyvariant analysis, either under call-by-name or call-by-value. In such an analysis, there would be several possible abstract closures that represent a given procedure, corresponding to several possible sets of closing environments for the procedure. In a polyvariant version of the analysis in this paper, we might discover that in some set of environments, a call site argument is certain to be evaluated, and then transform appropriately.

It would be a useful exercise to directly compare Amtoft's analysis with our own. Some of our untyped terms do not have a strictness type in Amtoft's system, so we may be able to avoid thunks for arguments in some programs that his system cannot analyze. On the other hand, our closure analysis is approximate, so we may forfeit some opportunities for avoiding thunks that are provided by Amtoft's type system.

References

1. Torben Amtoft. Minimal Thunkification. In Patrick Cousot et al., editors, *Proc. 3rd International Workshop on Static Analysis (WSA '93)*, number 724 in Lecture Notes in Computer Science, pages 218–29, Padova, Italy, 1993.
2. Geoffrey Burn and Daniel Le Métayer. Proving the Correctness of Compiler Optimisations Based on a Global Program Analysis: A Study of Strictness Analysis. Technical Report TR93-42, Department of Computing, Imperial College of Science, Technology and Medicine, August 1993.
3. Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness Analysis for Higher-order Functions. *Science of Computer Programming*, 7:249–78, 1986.
4. Olivier Danvy and John Hatcliff. CPS Transformation After Strictness Analysis. *ACM Letters on Programming Languages*, 1(3):195–212, 1993.
5. Paul Hudak and Jonathan Young. Higher-Order Strictness Analysis in Untyped Lambda Calculus. In *Conference Record of 13th ACM Symposium on Principles of Programming Languages*, pages 97–109, 1986.
6. Thomas P. Jensen. Strictness Analysis in Logical Form. In John Hughes, editor, *International Conf. on Functional Programming Languages and Computer Architecture*, pages 352–66, 1991.
7. Tsung-Min Kuo and Prateek Mishra. Strictness Analysis: A New Perspective. In *International Conf. on Functional Programming and Computer Architecture*, pages 260–72, 1989.
8. Alan Mycroft. The Theory and Practice of Transforming Call-by-Need Into Call-by-Value. In B. Robinet, editor, *International Symposium on Programming*, pages 269–81, 1980.
9. Jens Palsberg and Michael I. Schwartzbach. Safety Analysis vs. Type Inference. *Information and Computation*, to appear.
10. Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
11. Paul Steckler. *Correctness of Higher-Order Program Transformations*. PhD thesis, College of Computer Science, Northeastern University, July 1994.
12. Paul Steckler and Mitchell Wand. Tracking Available Values for Lightweight Closures (Summary). In Neil Jones and Carolyn Talcott, editors, *Proc. Atlantique Workshop on Semantics Based Program Manipulation*, pages 63–70, 1994. Available as DIKU Report No. 94/12, University of Copenhagen.
13. Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1983.
14. Mitchell Wand and Paul Steckler. Selective and Lightweight Closure Conversion. In *Conference Record of 21st ACM Symposium on Principles of Programming Languages*, pages 435–45, 1994.
15. David A. Wright. A New Technique for Strictness Analysis. In *Proc. TAPSOFT '91*, pages 235–58, 1991.

A Appendix: Proof of Theorem 8

Suppose $App(i)$ and:

$$\frac{(i.rator, \psi) \underset{ocn}{\Longrightarrow} (j, \psi'), \quad Abs(j) \qquad (j.body, \psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)]) \underset{ocn}{\Longrightarrow} (k, \psi'')}{(i, \psi) \underset{ocn}{\Longrightarrow} (k, \psi'')}$$

Then:

(1) $(i.rator, \psi)$ is well-thunked
[(i, ψ) is well-thunked]

(2) $(i.rand, \psi)$ is well-thunked
[(i, ψ) is well-thunked]

(3) $(j, \psi') \models^{\Pi}_{val} (\phi_{i.rator}, \pi_{i.rator})$
[IH re $\models^{\Pi}_{val}$ at $i.rator$]

(4) $(j, A_j) \in \phi_{i.rator}$
[(3), definition of $\models^{\Pi}_{val}$, monovariance]

(5) (j, ψ') is well-thunked
[IH re well-thunkedness at $i.rator$]

(6) $\psi' \models^{\Pi}_{env} A_j$
[(3), (4), definition of $\models^{\Pi}_{val}$]

(7) $\phi_{i.rand} \subseteq \phi_{j.bv}$
[$\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}$]

(8) $\pi_{i.rand} = \pi_{j.bv}$
[$\mathcal{P}_{i.rand} \leq \mathcal{P}_{j.bv}$]

(9) $\phi_{j.body} \subseteq \phi_i$
[$\mathcal{P}_{j.body} \leq \mathcal{P}_i$]

(10) $\pi_{j.body} = \pi_i$
[$\mathcal{P}_{j.body} \leq \mathcal{P}_i$]

(11) $(i.rand, \psi) \models^{\Pi}_{thunk} (\phi_{i.rand}, \pi_{i.rand})$
[by the definition of $\models^{\Pi}_{thunk}$]

(12) $(i.rand, \psi) \models^{\Pi}_{thunk} (\phi_{j.bv}, \pi_{j.bv})$
[by (7), (8), (11), Lemma 6]

(13) $\psi'[[\![j.bv]\!] \mapsto (i.rand, \psi)] \models^{\Pi}_{env} A_j[[\![j.bv]\!] \mapsto (\phi_{j.bv}, \pi_{j.bv})]$
[by (6), (8), (12)]

(14) $A_{j.body} = A_j[\llbracket j.bv \rrbracket \mapsto (\phi_{j.bv}, \pi_{j.bv})]$
[definition of locally-consistent annotation]

(15) $\psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)] \models^{\Pi}_{env} A_{j.body}$
[by (13), (14)]

(16) $(j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)])$ is well-thunked
[(2), $Abs(j)$, (5), (15)]

(17) $(k, \psi'') \models^{\Pi}_{val} (\phi_{j.body}, \pi_{j.body})$
[IH re $\models^{\Pi}_{val}$ at $j.body$]

(18) (k, ψ'') is well-thunked
[by IH re well-thunkedness at $j.body$]

(19) $(k, \psi'') \models^{\Pi}_{val} (\phi_i, \pi_i)$
[by (9), (10), (17)]

We now wish to show that the consequent involving strictness propositions holds. Any variable $x \in \mathcal{S}_i$ must also be an element of the strictness propositions for the operator or operand, according to the local consistency conditions.

Suppose $x \in \mathcal{S}_i \cap \mathcal{S}_{i.rator}$. By the induction hypothesis regarding strictness propositions, $(i.rator, \psi) \underset{ocn}{\Longrightarrow} (j, \psi') \models_{strict} \mathcal{S}_{i.rator}$. A free occurrence of x in the operator is also free in the application, and any subproof of $(i.rator, \psi) \underset{ocn}{\Longrightarrow} (j, \psi')$ is also a subproof of the evaluation of the application.

Now suppose $x \in \mathcal{S}_i \cap \mathcal{S}_{i.rand}$ but $x \notin \mathcal{S}_{i.rator}$. Hence, we must be dealing with the case in the local consistency conditions where $\forall (j, B) \in \phi_{i.rator}, \llbracket j.bv \rrbracket \in \mathcal{S}_{j.body}$. First, we note that any free occurrence of x in the operand is also free in the application. Next, we show that evaluating the application requires that the operand be evaluated.

By step (4) above, (j, A_j) in $\phi_{i.rator}$. Therefore, by the local consistency conditions, $\llbracket j.bv \rrbracket \in \mathcal{S}_{j.body}$. We showed above in step (16) that $(j.body, \psi'[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)])$ is well-thunked. So by the induction hypothesis at $j.body$, we have $(j.body, \psi[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)]) \underset{ocn}{\Longrightarrow} (k, \psi'') \models_{strict} \mathcal{S}_{j.body}$. By the definition of $\models_{strict}$, there are p, q, ψ_3, ψ_4, where p is a free occurrence of $\llbracket j.bv \rrbracket$ in $\llbracket j.body \rrbracket$, such that in the evaluation of $(j.body, \psi[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)])$, there is a subproof $(p, \psi_3) \underset{ocn}{\Longrightarrow} (q, \psi_4)$, and $\psi_3(\llbracket j.bv \rrbracket) = \psi[\llbracket j.bv \rrbracket \mapsto (i.rand, \psi)](\llbracket j.bv \rrbracket) = (i.rand, \psi)$. By the evaluation rules, that subproof has an immediate subproof $(i.rand, \psi) \underset{ocn}{\Longrightarrow} (q, \psi_4)$. By step (2), $(i.rand, \psi)$ is well-thunked, so by the induction hypothesis at $i.rand$ regarding strictness propositions, the evaluation of $(i.rand, \psi)$ satisfies $\mathcal{S}_{i.rand}$. Recall, we supposed $x \in \mathcal{S}_{i.rand}$. So by the definition of $\models_{strict}$, there are s, t, ψ_5, ψ_6 such that the evaluation of $(i.rand, \psi)$ has a subproof $(s, \psi_5) \underset{ocn}{\Longrightarrow} (t, \psi_6)$, where s is a free occurrence of x, and if $x \in Dom(\psi)$, then $x \in Dom(\psi_5)$, and $\psi(x) = \psi_5(x)$. But since that subproof is also a subproof of the evaluation of the procedure body, it is a subproof of the evaluation of the application.