In-place Updates in the Presence of Control Operators

Sandip K. Biswas*
Department of CIS
University of Pennsylvania
Philadelphia, PA 19104

Abstract

This paper presents a formal account of the concept of in-place updates in purely functional languages. In purely functional languages, updates of abstract objects involve creating duplicates of these objects. This paper reviews static conditions, which, if satisfied by λ-terms, guarantee that, even if updates are performed in-place, the purely functional semantics is retained. These static conditions, however, fail to guarantee the requisite safety in the presence of control operators like *callcc* and *throw*. The conditions are hence augmented by another condition which is defined on the operational semantics. Here we statically verify the satisfiability of a conservative approximation of this condition by data-flow analysis on *CPS*-terms. Also a significant class of programs is identified for which the condition holds even without data-flow analysis.

1 Introduction

Operational Semantics for functional languages can be defined without reference to a store of any kind. However these languages are implemented on von-Neumann machines, which have store-based architectures. If the language contains abstract data-types with update operations defined on such data-types, the actual implementations of the language in fact return a new copy of an abstract object on an update operation: there are no mutating operations performed in the memory. If the abstract object is something like an array, then this copy operation becomes very expensive. But if we can specify static conditions that guarantee that generating a new copy on an update and updating the object in the memory itself always produce the same result, then we have effectively changed the cost of the run-time update operation from the size of the object to a constant.

Such conditions were presented in [8] in context of denotational semantics. In the Scott-Strachey [10] denotational definition of a programming language, the denotation of a program is defined as a function from *Store* to *Store*. In [8] Schmidt proves an implementation of the denotational definition need not pass the *store* around as a parameter, instead it can be replaced by a single global variable, which could be updated in-place. A predicate on programs, *single-threadedness*, was defined. Programs satisfying this predicate have their denotations preserved under the implementation.

This paper re-phrases the above theorem in the language of λ-calculus, stating much more explicitly the meaning of in-place updates. The predicate *single-threadedness* is exactly the one Schmidt states. It is shown that, in the presence of control operators, *callcc* and *throw*, satisfiability of *single-threadedness* does not imply that in-place updates are safe. An additional condition required to ensure the safety of in-place updates without being overtly restrictive is defined on the operational semantics, and its correctness is proved. The technique of data-flow analysis is used to verify statically that a conservative approximation of this condition is satisfied.

Like Schmidt, this paper addresses programs that contain a single array/abstract data-type initially and the program performs a multitude of update and selection operations on this array in the course of its execution. In the terminology of [7], it is the single-*ls* pebbling situation. This is not a restrictive assumption: it allows a large number of programs of interest such as various sorting programs and programs that use an array as a global variable.

The fundamental principle involved in safe in-place updates is simple: when an abstract value is being updated, there must be no other reference to this value. This is exactly the way the problem is addressed in [3]. Hudak presents a finite abstraction for a first order language: it is not clear what a finite abstraction should be for a higher-order language. In the presence of *callcc* and *throw*, the complexity of the problem increases, as a user-level program represents a continuation as a variable that is bound to the rest of the computation at run-time. So the approach chosen in this paper is a more syntactic one. If the user-program satisfies the initial conditions, then, at an update at run-time, the reference count of the abstract value is indeed one.

2 Syntax and Semantics of the Functional Language

2.1 Syntax of the Language

The simply-typed λ-calculus along with a *mutable abstract data-type* τ is considered as the functional language. The definition given below follows [4]:

*This research was supported by ONR grant number N00014-88-K-0557

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

Definition: A *mutable abstract data-type* τ is one in which each operation can be classified as either a *generator* of type $X_1 \rightarrow \tau$, a *mutator* of type $X_2 \rightarrow \tau \rightarrow \tau$ or a *selector* of type $X_3 \rightarrow \tau \rightarrow X_4$, where $X_i \not\equiv \tau$ are types in the language.

Our language does not contain tuple types. Hence all functions are curried versions: the types X_i, in the definition above, need to be 'curried' in the language. As shown in Figure 1, a curried version of *int array* in SML/NJ, may be considered as a *mutable abstract data-type*.[1]

Considering *int array* as the prototype, it is assumed (without loss of generality) that τ has three operations: the generator $\mathcal{G} : int \rightarrow int \rightarrow \tau$, the mutator $\mathcal{M} : int \rightarrow int \rightarrow \tau \rightarrow \tau$ and the selector $\mathcal{S} : int \rightarrow \tau \rightarrow int$. This paper considers a typed language with the standard typing rules: [2] is a standard reference.

The functional language of interest is specified by the following grammar:

$$M ::= i \mid x \mid A \mid \mathcal{M} \mid \mathcal{S} \mid \lambda x : \eta.\ M \mid MM$$

$$V ::= i \mid x \mid A \mid \mathcal{M} \mid \mathcal{S} \mid \lambda x : \eta.\ M \mid \mathcal{M}i \mid \mathcal{M}ij \mid \mathcal{S}i$$

where i is a variable representing the integers and V is a subset of the terms M, representing the *values* in the language. A is a variable representing a value of the *abstract data-type* τ. Symbols A , A' , A'' all represent abstract values of type τ. Note that the language of terms M does not contain the generator for the type τ: this is because we are interested in the set of terms that contain a single value of the type τ initially, and only update and selection operations are performed on this value. The terms in the language are explicitly typed. When types are not relevant, or are obvious from the context, they are dropped.

2.2 Operational Semantics

As τ is an abstract type, the concrete structure of a value of type τ is not known. The operational semantics for operations on abstract values of type τ is defined with the aid of an interpretor function δ. For example, $(\mathcal{M}\ i\ j\ A)$ returns a new value $\delta(\mathcal{M}, i, j, A)$ of type τ. Since the actual semantics of the abstract type is not relevant here, we merely assume that the interpretor function δ returns a value of the appropriate type.

The operational semantics for this language is given in the style of [11]. This involves defining a set of redexes, and an evaluation strategy: how to locate the next redex in a term. The redexes in the language are $((\lambda x.M)V)$, $(\mathcal{M}\ i\ j\ A)$ and $(\mathcal{S}\ i\ A)$. The evaluation strategy is given by an evaluation context $E[\ ]$, which is a term with a 'hole' $[\ ]$ in it. $E[\ ]$ is specified by the following grammar:

$$E[\ ] ::= [\ ] \mid (E[\ ])M \mid V(E[\ ])$$

The operational semantics defined below is deterministic. This is because any type-checked term M which is not a value can be partitioned uniquely into an evaluation context $E[\ ]$ and a redex R such that $M \equiv E[R]$. The transition semantics is as follows:

$$E[(\lambda x.M)V] \longmapsto E[M\{x/V\}]$$

$$E[\mathcal{M}\ i\ j\ A] \longmapsto E[A']$$

where $A' = \delta(\mathcal{M}, i, j, A)$

$$E[\mathcal{S}\ i\ A] \longmapsto E[j]$$

where $j = \delta(\mathcal{S}, i, A)$

[1] In fact, *array* operations in SML/NJ actually perform mutations in the memory

2.3 Single-Threadedness

The evaluation context $E[\ ]$ defines a specific evaluation strategy for call-by-value. We define a more general context $F[\ ]$ as,

$$F[\ ] ::= [\ ] \mid (F[\ ])M \mid M(F[\ ])$$

Definition: N is an **active** sub-term of a term M, if N is not contained under a λ in M, i.e. $M \equiv F[N]$.

Definition: N_1 and N_2 are **disjoint** sub-terms of a term M, if it is the case that neither N_1 nor N_2 is contained in the other.

Definition: A term E is **single-threaded iff** all sub-terms N of E possess the following properties:

A. Non-Interference

1. If N is of type τ then if N contains disjoint active sub-terms N_1 and N_2 of type τ, then $N_1 \equiv N_2 \equiv A$ or $N_1 \equiv N_2 \equiv v$ for some variable v.
2. If N is not τ-typed, all active sub-terms of type τ in N are identical to a particular abstract value or variable.

B. Immediate Evaluation

1. If $N \equiv \lambda x.M : \tau \rightarrow \eta$, then all active variables of type τ in M are occurrences of x. There are no active τ-typed abstract values in M.
2. If $N \equiv \lambda x.M : \eta_1 \rightarrow \eta_2$, where $\eta_1 \not\equiv \tau$, then M contains no active terms of type τ.

The definition of single-threaded terms applies irrespective of the constants in the language.

The following important properties of single-threaded terms are expressed as lemmas. The proofs in most cases are trivial.

Lemma 2.1 *If a single-threaded term $M \equiv E[N]$, has a redex N of type τ, then M does not contain any other active sub-term of type τ, disjoint from N.*

Lemma 2.2 *If $(\lambda x.\ N)$ is a sub-term of a single-threaded term M, then it does not contain free variables of type τ nor does it contain any abstract value of type τ.*

Lemma 2.3 *In any closed single-threaded term M, all occurrences of abstract values of type τ are active. If M contains disjoint active sub-terms M_1 and M_2 of type τ then $M_1 \equiv M_2 \equiv A$.*

3 In-place Updates are Safe

Let $\overset{n}{\longmapsto}$ denote the n-fold composition of the transition relation $\longmapsto$ and let $\longmapsto^*$ denote its transitive closure.

The next two lemmas show that the property of being single-threaded is preserved under the transition semantics $\longmapsto$.

`val array:int->int->array`	the *generator*, with the first argument stating the size and the second the initial value.
`val update:int->int->array->array`	the *mutator*, with the first argument giving the subscript and the second argument the value.
`val sub:int->array->int`	the *selector*, with the first argument giving the subscript

Figure 1: The data-type *int array* in SML/NJ

Lemma 3.1 *Let N be an active sub-term of a single-threaded term M, i.e $M \equiv F[N]$. Let N' be a closed term satisfying the following conditions:*

- *It is single-threaded.*
- *It has an active sub-term of type τ only if N has an active sub-term of type τ.*
- *If $F[\ ]$ contains an active value A of type τ and N' contains an active value A' of type τ, then $A \equiv A'$.*

If N is not a value of type τ then $F[N']$ is a single-threaded term.

Proof: By case analysis on the types of M and N. □

Lemma 3.2 *Let M be a closed single-threaded term. If $M \longmapsto^* M'$, then M' is single-threaded.*

Proof: We need to prove the preservation of single-threadedness over a single step of the reduction process. The proof for an arbitrary number of steps follows by induction.

Let $M \equiv E[N] \longmapsto M' \equiv E[N']$. By Lemma 3.1, M' is single-threaded if N' satisfies the following three conditions:

1. It is single-threaded.
2. It has active sub-terms of type τ only if N does.
3. If $E[\ ]$ contains an active value A of type τ and N' contains a value A' of type τ, then $A \equiv A'$.

There are two non-trivial redexes to consider:

- $N \equiv (\mathcal{M}\ i\ j\ A) \longmapsto A'$, where $A' = \delta(\mathcal{M}, i, j, A)$. This result follows from Lemma 2.1
- $N \equiv (\lambda x.P)V \longmapsto P\{x/V\}$,

  If $V : \tau$ then it must be some abstract value A. P is single-threaded by assumption. By Lemma 2.2, abstractions do not contain free variables of type τ, thus $x : \tau$ is not free in any abstraction in P. Hence $V : \tau$ cannot be substituted into the body of any abstraction in P. Thus abstractions in $P\{x/V\}$ are single-threaded. $P\{x/A\}$ satisfies Condition A, because Condition A applies for both variables and values. Thus $P\{x/A\}$ is single-threaded. Statement (2) holds as N contains an active term of type τ. By single-threadedness, $E[\ ]$ can only contain A as an abstract value of type τ. By Lemma 2.2, P contains no abstract value of type τ, thus $P\{x/A\}$ can contain only the value A of type τ. Hence statement(3) is satisfied.

  If the single-threaded term V is not of type τ, then the syntactic structure of values in the language shows that it has no active sub-term of type τ. It is single-threaded, hence its substitution retains $P\{x/V\}$ as single-threaded. $(\lambda x.\ P)$ is single-threaded. Hence, by Lemma 2.2, P has no active sub-term of type τ. As observed earlier nor does V, hence nor does $P\{x/V\}$. Thus statement (2) and (3) do not apply.

  □

The preservation of single-threadedness over the reduction process is not valid in a call-by-name evaluation strategy because an argument to a function even though not of type τ may contain a value of type τ, e.g. $(\mathcal{S}\ i\ A)$ is a valid argument of type *int*.

The order of arguments in the type of the constructs $\mathcal{M}$ and $\mathcal{S}$ is critical in the proof of Lemma 3.2. If the order of the arguments in $S : int \rightarrow \tau \rightarrow int$ is switched to $S : \tau \rightarrow int \rightarrow int$ then $\mathcal{S}\ A$ would be a value of type $int \rightarrow int$ but would contain a value of type τ. Hence single-threadedness would not be preserved.

By Lemma 2.3, a closed single-threaded term M contains exactly one value of the type τ and all its occurrences are active. Hence to formally handle the operation of in-place updates in an implementation, we may store this single active value of type τ in a *ref-cell* and replace all its occurrences by its location in memory. This may be formalised by a transition semantics, in a manner similar to [11], with a single-element global store with address w_0. This new language, where all updates are performed in-place, no longer has any values of type τ: they are all replaced by values of type $\tau\ ref$. As there is a single address in the memory, there is only one value of type $\tau\ ref : w_0$. Thus, $\mathcal{M} : int \rightarrow int \rightarrow \tau\ ref \rightarrow \tau\ ref$ and $\mathcal{S} : int \rightarrow \tau\ ref \rightarrow int$. The transition semantics with a global representation for the value of type τ is given as follows,

$$\rho\langle w_0, A\rangle.\ E[(\lambda x.M)V] \Longrightarrow \rho\langle w_0, A\rangle.\ E[M\{x/V\}]$$

$$\rho\langle w_0, A\rangle.\ E[\mathcal{M}\ i\ j\ w_0] \Longrightarrow \rho\langle w_0, A'\rangle.\ E[w_0]$$
$$\text{where } A' = \delta(\mathcal{M}, i, j, A)$$

$$\rho\langle w_0, A\rangle.\ E[\mathcal{S}\ i\ w_0] \Longrightarrow \rho\langle w_0, A\rangle.\ E[j]$$
$$\text{where } j = \delta(\mathcal{S}, i, A)$$

Just as for $\longmapsto$, $\stackrel{n}{\Longrightarrow}$ denotes the n-fold composition of the transition relation $\Longrightarrow$. Obviously the $\Longrightarrow$ transition relation is a deterministic operational semantics.

The safety of in-place updates can now be conceived as follows: Consider a single-threaded term M. By Lemma 2.3 it contains exactly one value A of type τ. Consider a term M' obtained by replacing all occurrences of this value A by w_0. Then at every stage in the evaluation of $\rho\langle w_0, A\rangle.M'$ under the $\Longrightarrow$ semantics, replacing w_0 in the term by its value in the memory must exactly match the term obtained from the evaluation of M under the $\longmapsto$ semantics.

Definition: $M' \overset{\{w_0/A\}}{\simeq} M$ iff $M'\{w_0/A\} \equiv M$ and $M\{A/w_0\} \equiv M'$.

Theorem 3.1 (In-place Updates are Safe) *Consider a closed single-threaded term M_1 such that $M_1' \overset{\{w_0/A\}}{\simeq} M_1$. If $M_1 \overset{n}{\longmapsto} M_2$ and $\rho\langle w_0, A\rangle.M_1' \overset{n}{\Longrightarrow} \rho\langle w_0, A'\rangle.M_2'$, then $M_2' \overset{\{w_0/A'\}}{\simeq} M_2'$*

Proof: We need to prove the substitution property for a one-step transition, the rest follows by induction. This is because by Lemma 3.2 single-threadedness is preserved over transition.
There is only one important case to be considered:
$E[\mathcal{M}ijA] \longmapsto E[A']$
$\rho\langle w_0, A\rangle.E'[\mathcal{M}ijw_0] \Longrightarrow \rho\langle w_0, A'\rangle.E'[w_0]$
By assumption $E'[\,] \overset{\{w_0/A\}}{\simeq} E[\,]$, i.e $E'[\,]\{w_0/A\} \equiv E[\,]$ and $E[\,]\{A/w_0\} \equiv E'[\,]$. As $[\,]$ is a redex of type τ, by Lemma 2.1, $E[\,]$ does not contain the value A of type τ. Therefore $E[\,] \equiv E[\,]\{A/w_0\}$. As $E[\,]$ and $E'[\,]$ are related by a renaming substitution, $E'[\,]$ contains no occurrence of w_0. Hence,

$$E'[\,]\{w_0/A\} \equiv E'[\,] \equiv E[\,] \tag{1}$$

We need to prove that $E'[\,] \overset{\{w_0/A'\}}{\simeq} E[\,]$, i.e.

- $E'[\,]\{w_0/A'\} \equiv E[\,]$. As $E'[\,]$ contains no occurrence of w_0, $E'[\,]\{w_0/A'\} \equiv E'[\,]$. Hence by (1) $E'[\,]\{w_0/A'\} \equiv E[\,]$.
- $E[\,]\{A'/w_0\} \equiv E'[\,]$. As $[\,]$ is a redex of type τ , by Lemma 2.1, $E[\,]$ does not contain the value A' of type τ. Hence $E[\,]\{A'/w_0\} \equiv E[\,]$. Thus by (1) we have $E[\,]\{A'/w_0\} \equiv E'[\,]$.

□

4 Extending the Language with Control Operators

4.1 Introduction

We would like to extend the language with two control operators. One of them is the *if-then-else* statement and the other is the *call with current continuation*. The definition of single-threaded terms can be extended in a straight-forward way to accommodate the *if-then-else* statement. To prove that in-place updates are safe, when a term satisfies the extended definition, we present a translation of an *if-then-else* term M to a term M' in the original language such that M' is single-threaded iff, M is single-threaded by the extended definition.

In the presence of continuations in-place updates of single-threaded terms are unsafe. A simple extension of the definition of single-threadedness, though fixing the problem, turns out to be too restrictive. Hence we impose a run-time condition on the evaluation and prove that if this condition is satisfied, in-place updates, in the presence of continuations, are safe.

4.2 The if-then-else statement

The *if-then-else* term is a basic control operator present in most call-by-value languages. The conventional semantics for an *if-then-else* statement involves a boolean value. In our language it is assumed that booleans are generated from pre-defined functions on *int*, like '=', '$\neq$', '$<$' etc.

The extended language is defined by the grammar:

$$\begin{aligned} M \quad ::= \quad & \imath \mid x \mid A \mid \mathcal{M} \mid \mathcal{S} \mid \lambda x.M \mid MM \\ & \mid \textbf{if } (M_1\ P\ M_2) \textbf{ then } M_3 \textbf{ else } M_4 \\ P \quad ::= \quad & \neq \mid = \mid < \mid \ldots \end{aligned}$$

The standard operational semantics is as follows:

$$\textbf{if } \mathit{true} \textbf{ then } M_1 \textbf{ else } M_2 \longmapsto M_1$$

$$\textbf{if } \mathit{false} \textbf{ then } M_1 \textbf{ else } M_2 \longmapsto M_2$$

Let $M \equiv \textbf{if } (M_1\ P\ M_2) \textbf{ then } M_3 \textbf{ else } M_4$. From the operational semantics it is seen that both M_1 and M_2 are evaluated before either M_3 or M_4 is, and exactly one of M_3, M_4 is evaluated. Thus condition A(1) for single-threadedness should not consider M_3 and M_4 as disjoint terms. Also as M_1 and M_2 are of type *int* and are evaluated strictly before M_3 and M_4 condition A(1) should not consider M_1/M_2 disjoint from M_3/M_4.

$F'[\,]$, as defined below, is an extension of $F[\,]$. The context $F'[\,]$ defines active terms in the extended language.

$$\begin{aligned} F'[\,] \quad ::= \quad & [\,] \mid (F'[\,])M \mid M(F'[\,]) \\ & \mid \textbf{if } (F'[\,]\ P\ M_2) \textbf{ then } M_3 \textbf{ else } M_4 \\ & \mid \textbf{if } (M_1\ P\ F'[\,]) \textbf{ then } M_3 \textbf{ else } M_4 \\ & \mid \textbf{if } (M_1\ P\ M_2) \textbf{ then } F'[\,] \textbf{ else } M_4 \\ & \mid \textbf{if } (M_1\ P\ M_2) \textbf{ then } M_3 \textbf{ else } F'[\,] \end{aligned}$$

We define a new context $G'[\,]$ to conveniently refer to an active term in the *if* part of the conditional.

$$G'[\,] \ ::= \ (F'[\,]\,P\,M) \mid (M\,P\,F'[\,])$$

A revised version of condition A(1) is as follows:

A. Non-Interference

1. If N is of type τ then if N contains disjoint active sub-terms N_1 and N_2 of type τ, then:
   - $N_1 \equiv N_2 \equiv A$, or
   - $N_1 \equiv N_2 \equiv v$ for some variable v, or
   - (**if** ... **then** $F_1'[N_1]$ **else** $F_2'[N_2]$), or (**if** $G'[N_1]$ **then** $F_2'[N_2]$ **else** ...), or (**if** $G'[N_1]$ **then** ... **else** $F_2'[N_2]$), is an active sub-term of N. If $A_1 : \tau$ is an active value in N_1 and $A_2 : \tau$ is an active value in N_2 then $A_1 \equiv A_2$

But a revised condition involves revising the proofs given in the previous section. Hence we try to give the *if-then-else* statement a new syntax so that it can fit into the old framework. The pre-defined function '=' is removed from the language and instead we introduce a function $P_=$[2] which returns a projection function instead of a boolean value, i.e.

$$P_= \ i\ i \longmapsto \lambda x.\lambda y.\ x$$

$$P_= \ i\ j \longmapsto \lambda x.\lambda y.\ y \ , \text{ if } i \neq j$$

[2] As this is a simply-typed language, technically, we are introducing a set of functions $P_=^{\eta}$ for every type η, such that, $P_=^{\eta}\ i\ i \longmapsto \lambda x \ \ \eta\ \lambda y\ .\ \eta\ x$,

Similarly the set of functions $P_<$ and $P_{\neq}$ are defined: *true* corresponds to *first projection* and *false* to *second projection.*

Thus the term,

$$\textbf{if}(s = t)\ \textbf{then}\ (M_1 : \eta)\ \textbf{else}\ (M_2 : \eta)$$

can now translated to,

$$[P_= \ s\ t\ (\lambda w.M_1)\ (\lambda w.M_2)]\ ()$$

where $w \notin Fv(M_1) \cup Fv(M_2)$ and () is new value of type *unit*. A dummy abstraction and application on M_1 and M_2 is required because the language is call-by-value.

The above translation is faithful to the operational semantics but may fail to be single-threaded when the original term is. This is because by Condition B(2), if the term $(\lambda w : unit.\ M_1)$ is single-threaded, then M_1 can have no active sub-term of type τ. Hence a translation involving an abstraction variable of type *unit* may not work.

Let $M \equiv \textbf{if}\ (s = t)\ \textbf{then}\ M_1\ \textbf{else}\ M_2$, be a sub-term of a single-threaded closed term N. A translation of M is defined by cases:

- There is no enclosing λ for M in N, i.e. M is active in N. Since M is closed, if M_i does not contain a value of type τ then M_i is not of type τ. Thus if a value of type τ is not present in M the following representation is single-threaded iff M is,

  $$[P_= \ s\ t\ (\lambda x.M_1)\ (\lambda x.M_2)]\ ()$$

  If a value A of type τ is present in M then the following representation is single-threaded iff M is,

  $$[P_= \ s\ t\ (\lambda x.M_1)\ (\lambda x.M_2)]\ (A)$$

  This is because, by the modified Condition A, the closed single-threaded term M has exactly one value A of type τ.

- $\lambda x.$ is the nearest enclosing abstraction for M in N, i.e. $N \equiv \ldots(\lambda x.F'[M])\ldots$. Then the following representation is single-threaded iff M is,

  $$[P_= \ s\ t\ (\lambda x.M_1)\ (\lambda x.M_2)]\ (x)$$

  If $x : \eta$, where $\eta \not\equiv \tau$, then by Condition B(2) M_1 and M_2 do not contain active terms of type τ. Hence $(\lambda x.M_1)$, $(\lambda x.M_2)$ are single-threaded.

  If $x : \tau$ then by Condition B(1), both M_1, M_2 do not contain values of type τ nor a variable of type τ distinct from x. Thus $(\lambda x.M_1)$,$(\lambda x.M_2)$ are single-threaded.

4.3 First-Class Continuations

We next extend the language with the control operators, *letcc* and *throw*. Due to technical reasons the construct *letcc* is chosen over the more traditional *callcc*. The extended language, of terms M and values V, is defined below:

$$\begin{array}{lll} \mathbf{M} & ::= & x \mid i \mid A \mid \lambda x.M \mid MM \mid \mathcal{M} \mid \mathcal{S} \mid \\ & & \textbf{letcc}\ k\ \textbf{in}\ M \mid \textbf{throw} \\ \mathbf{V} & ::= & \imath \mid x \mid A \mid \mathcal{M} \mid \mathcal{S} \mid \mathcal{M}i \mid \mathcal{M}ij \mid \mathcal{S}\imath \mid \\ & & \lambda x.M \mid \textbf{throw} \mid \textbf{throw}\ (\lambda x.M) \end{array}$$

The operational semantics for the new constructs are as follows:

$$E[\textbf{letcc}\ k\ \textbf{in}\ M] \longmapsto E[M\{k/(\lambda x.\ E[x])\}]$$
$$E[\textbf{throw}\ (\lambda x.\ M)\ V] \longmapsto M\{x/V\}$$

The additional typing rules involved are:

$$\frac{\Gamma,\ \mathrm{k} : \eta\ \mathrm{cont} \vdash \mathrm{M} : \eta}{\Gamma \vdash \textbf{letcc k in M} : \eta}$$

$$\Gamma \vdash \textbf{throw} : \eta_1\ cont \rightarrow \eta_1 \rightarrow \eta_2$$

The standard definition of single-threadedness depends on the types of the terms and not on the terms and constructs themselves. Thus the earlier definition still applies to the terms of the language extended with the constructs *letcc* and *throw*. Under this operational semantics single-threadedness is not preserved. This is because of the *letcc* transition. Consider a single-threaded term N, where $N \overset{*}{\longmapsto} E[\textbf{letcc}\ k\ \textbf{in}\ M]$. If $E[\textbf{letcc}\ k\ \textbf{in}\ M]$ is single-threaded and [ ] is of type τ, then by Lemma 2.1, $E[\,]$ contains no term of type τ disjoint from [ ]. Hence $(\lambda x.\ E[x])$ is single-threaded. But if [ ] is not of type τ, $E[\,]$ may contain terms of type τ disjoint from [ ]. Thus the captured continuation $(\lambda x.\ E[x])$ may contain values of type τ. But this violates Condition B(2) as $(\lambda x.\ E[x])$ is of type $\eta_1 \rightarrow \eta_2$, where $\eta_1 \not\equiv \tau$.

Figure 2 presents a counter-example demonstrating that in-place updates are not safe in terms initially single-threaded. In term (1) in Figure 2 the captured continuation $k \equiv (\lambda f.\ fA)$ contains a value A of type τ. Thus the continuation to which k is bound is not single-threaded. Performing the updates in place returns the value A', instead of the correct answer A. The term is initially single-threaded, but as the captured continuation is not a single-threaded abstraction, the instant *letcc* is executed the term loses its single-threadedness. The source of the problem is the fact that a captured non-τ continuation may fail to be single-threaded, i.e. may possess a value of type τ.

The obvious solution is to disallow such continuations. This solution is inordinately restrictive on programs. For example, let $\tau \equiv int\ array$. Then consider the term $(update\ 4\ (\textbf{letcc}\ k\ \textbf{in}\ M)\ A)$. The term M is a simple arithmetic expression which takes in a continuation k to short circuit the evaluation when we have a multiplication by 0. This program, however, must be disallowed as k will be bound to $(\lambda x.\ update\ 4\ x\ A)$, which contains a value of the type *int array*. In many programs continuations are captured for such short-circuit operations and they must be likewise disallowed. Hence the obvious solution is not acceptable.

As it is our aim to allow continuations to contain values of type τ, denoting a captured continuation by $(\lambda x.\ E[x])$ will always violate Condition B(2). So we introduce a new binding construct for continuations *Cnt*. The continuation $(\lambda x.\ E[x])$ is now represented as $(Cnt\ x.\ E[x])$. Just like λ, terms enclosed by *Cnt* are not considered *active*. The revised clauses for the operational semantics are:

$$E[\textbf{letcc}\ k\ \textbf{in}\ M] \longmapsto E[M\{k/(Cnt\ x.\ E[x])\}]$$
$$E[\textbf{throw}\ (Cnt\ x.\ M)\ V] \longmapsto M\{x/V\}$$

An additional typing rule is required:

$$\frac{\Gamma,\ \mathrm{x} : \eta_1 \vdash \mathrm{M} : \eta_0}{\Gamma \vdash (Cnt\ \mathrm{x.\ M}) : \eta_1\ \mathrm{cont}}$$

$[\textbf{letcc } k \textbf{ in } (\lambda s_1.(\lambda s_2.\mathcal{M}\ (\textbf{throw}\, k\, (\lambda x.x))\ 4\ s_2))(\mathcal{M}\ 1\ 4\ s_1)]A$
$\longmapsto\ [(\lambda s_1.(\lambda s_2.\mathcal{M}\ (\textbf{throw}\, (\lambda f.\ fA)\, (\lambda x.x))\ 4\ s_2))(\mathcal{M}\ 1\ 4\ s_1)]A$ (1)
$\longmapsto\ [(\lambda s_2.\mathcal{M}\ (\textbf{throw}\, (\lambda f.\ fA)\, (\lambda x.x))\ 4\ s_2)\ (\mathcal{M}\ 1\ 4\ A)]$
$\longmapsto\ [(\lambda s_2.\mathcal{M}\ (\textbf{throw}\, (\lambda f.\ fA)\, (\lambda x.x))\ 4\ s_2)\ A']$, where $A' = \delta(\mathcal{M}, 1, 4, A)$
$\longmapsto\ [\mathcal{M}\ (\textbf{throw}\, (\lambda f.\ fA)\, (\lambda x.x))\ 4\ A']$
$\longmapsto\ (\lambda x.x)A$
$\longmapsto\ A$

$\rho\langle w_0, A\rangle.\ [\textbf{letcc } k \textbf{ in } (\lambda s_1.(\lambda s_2.\mathcal{M}\ (\textbf{throw}\, k\, (\lambda x.x))\ 4\ s_2))(\mathcal{M}\ 1\ 4\ s_1)]w_0$
$\Longrightarrow\ \rho\langle w_0, A\rangle.\ [(\lambda s_1.(\lambda s2.\mathcal{M}\ (\textbf{throw}\, (\lambda f.\ fw_0)\, (\lambda x.x))\ 4\ s_2))(\mathcal{M}\ 1\ 4\ s_1)]w_0$
$\Longrightarrow\ \rho\langle w_0, A\rangle.\ [(\lambda s_2.\mathcal{M}\ (\textbf{throw}\, (\lambda f.\ fw_0)\, (\lambda x.x))\ 4\ s_2)\ (\mathcal{M}\ 1\ 4\ w_0)]$
$\Longrightarrow\ \rho\langle w_0, A'\rangle.\ [(\lambda s_2.\mathcal{M}\ (\textbf{throw}\, (\lambda f.\ fw_0)\, (\lambda x.x))\ 4\ s_2)\ w_0]$, where $A' = \delta(\mathcal{M}, 1, 4, A)$
$\Longrightarrow\ \rho\langle w_0, A'\rangle.\ [\mathcal{M}\ (\textbf{throw}\, (\lambda f.\ fw_0)\, (\lambda x.x))\ 4\ w_0]$
$\Longrightarrow\ \rho\langle w_0, A'\rangle.\ (\lambda x.x)w_0$
$\Longrightarrow\ \rho\langle w_0, A'\rangle.\ w_0$

Figure 2: A violation of safety of in-place updates

The rest of the typing rules are exactly the same as before. Let a term M be single-threaded when it satisfies the conditions A and B defined in Section 2. Thus $(Cnt\ x.\ N)$ is single-threaded when N is. This is intuitively what it should be, because when the execution of N begins the surrounding context is thrown away completely. It is important to note that Lemma 2.3 does not hold anymore. All occurrences of abstract values of type τ need not be active and need not be the same value. A single-threaded term in this language may contain several different values of type τ (of course, exactly one of these values is active). In the counter-example given previously, it is seen that single-threadedness can no longer guarantee that if we start with a term with exactly one value of type τ then the term will continue to have exactly one value of type τ. Notice that if the captured continuation in Figure 2 is represented as $(Cnt\ f.fA)$, the term retains its single-threadedness.

Definition: In a term (**letcc** k **in** M), the sub-term M is considered as the **syntactic scope** of the variable k.

A variable k 'escapes' a syntactic scope M if, during the evaluation of M, k or a closure containing k, is returned as a value or is passed as a parameter to a free variable in M. Consider the single-threaded expression (**letcc** k **in** M). If k is a non-τ continuation then M is not of type τ. By single-threadedness, M does not contain an active mutating operation. If k does not escape its syntactic scope, then, even if k captures a continuation containing a value of type τ, in-place updates should be safe. This is because all points of application of k are contained in its syntactic scope, the body of M, which contains no active mutating operations. In fact, if no continuation escapes its own syntactic scope then it is possible to define an operational semantics in which it is not necessary to bind a continuation to a variable. This is described in [6]. In this case, as there are no captured continuations, single-threadedness is retained all along.

Actually we can do better: suppose

$$M \xrightarrow{*} E[\textbf{letcc } k \textbf{ in } N]$$

Let us assume that $E[\textbf{letcc } k \textbf{ in } N]$ does not contain any captured continuation. If k is a non-τ continuation then N does not contain an active mutating operation. If no continuation captured within N escapes the syntactic scope of k, the body of N, then in-place updates should be safe. Within the syntactic scope of k, it is not necessary to restrict captured continuations from escaping their own scopes. A formal presentation of the correctness of this idea is given in the next section.

5 Non-Escaping Continuations

5.1 Operational Semantics

The operational semantics given in the previous section does not keep track of the scope of the captured continuation. As discussed earlier, at any instant it is necessary to keep track of the scope of a single non-τ continuation: the outermost one. Let us denote it by $(Cnt\ x.\ c_1)$, where c_1 is a special symbol not used anywhere else in the program. The actual continuation denoted by c_1 is carried around with the term as an explicit context $E[\,]$. Thus the operational semantics is defined on a pair (M, L), where M is a term and L is a list of terms at most one element long. L, if non-null, contains the continuation $E[\,]$ denoted by c_1. As we are concerned exclusively with non-τ continuations, the terms of type $\tau\ cont$ are labelled: (**letcc** $k : \tau$ **in** ...), $(Cnt\, k : \tau.\ \ldots)$.

To represent the concept that no non-τ continuation in a syntactic scope escapes, a predicate *Non-Tau-Free* is defined on terms. This predicate is true if there are no non-τ continuations present in a term, i.e. all the continuations in the term are of the form $(Cnt\ k : \tau.\ \ldots)$. When a value is thrown to the continuation denoted by c_1 or the syntactic scope of the continuation denoted by c_1 evaluates to a value, the predicate *Non-Tau-Free* ensures that there are no non-τ continuations in the value.

A non-standard operational semantics is presented in Figure 3. The intent of this non-standard operational semantics, containing clauses with guards, is to present a run-time condition for the safety of in-place updates. Hence programs that can be statically proven to satisfy these conditions, at run-time, must allow safe in-place updates.

5.2 In-place Updates are Safe

Definition: An evaluation context $E[\,]$ is defined single-threaded, if $(Cnt\ x.\ E[x])$ is single-threaded. $E[\,]$ is of type $\eta\ cont$ if $[\,]$ is of type η.

The following two lemmas are proved with the fact that values, not of type τ, do not contain active values of type τ,

$$
\begin{array}{rcl}
(E[(\lambda x.\ M)V]\ ,\ L) & \longmapsto_s & (E[M\{x/V\}]\ ,\ L) \\
(E[\mathcal{M}\ i\ j\ A]\ ,\ L) & \longmapsto_s & (E[A']\ ,\ L) \text{ where } A' = \delta(\mathcal{M}, i, j, A) \\
(E[\mathcal{S}\ i\ A]\ ,\ L) & \longmapsto_s & (E[\jmath]\ ,\ L) \text{ where } \jmath = \delta(\mathcal{S}, i, A) \\
(E[\textbf{letcc } k : \tau \textbf{ in } M]\ ,\ L) & \longmapsto_s & (E[M\{k/(Cnt\ x : \tau.\ E[x])\}]\ ,\ L) \\
(E[\textbf{letcc } k \textbf{ in } M]\ ,\ nil) & \longmapsto_s & (M\{k/(Cnt\ x.\ c_1)\}\ ,\ [E[\,]]) \\
(E[\textbf{letcc } k \textbf{ in } M]\ ,\ [E'[\,]]) & \longmapsto_s & (E[M\{k/(Cnt\ x.\ E[x])\}]\ ,\ [E'[\,]]) \\
(E[\textbf{throw } (Cnt\ x : \tau.\ E'[x])\ A]\ ,\ L) & \longmapsto_s & (E'[A]\ ,\ L) \\
(E[\textbf{throw } (Cnt\ x.\ E'[x])\ V]\ ,\ L) & \longmapsto_s & (E'[V]\ ,\ L) \\
(E[\textbf{throw } (Cnt\ x.\ c_1)\ V]\ ,\ [E'[\,]]) & \longmapsto_s & (E'[V]\ ,\ nil) \quad \underline{\text{if } Non\text{-}Tau\text{-}Free(V)} \\
(V\ ,\ E[\,]) & \longmapsto_s & (E[V]\ ,\ nil) \quad \underline{\text{if } Non\text{-}Tau\text{-}Free(V)} \\
(V\ ,\ nil) & \longmapsto_s & V
\end{array}
$$

Figure 3: Operational Semantics

and the use of Lemma 3.1, which is still valid in the extended language.

Lemma 5.1 *If $E[\,]$ is a closed single-threaded context of type η cont, and $V : \eta$ is a single-threaded value, then $E[V]$ is single-threaded.*

Lemma 5.2 *If E[**letcc** k **in** M] is a closed single-threaded term, then $E[M\{k/(\text{Cnt } x.\ E[x])\}]$ is also single-threaded.*

The following lemma is crucial for the safety of in-place updates:

Lemma 5.3 *Consider a closed single-threaded term N_0 in* **Muser**.
If $(N_0, []) \overset{*}{\longmapsto}_s (N_1\ ,\ L_1) \longmapsto_s (N_2\ ,\ L_2)$
The following are invariant over transition:

1. *N_i is single-threaded*
2. *If $N_i : \tau$, then $Non\text{-}Tau\text{-}Free(N_i)$*
3. *If $L_i \equiv [E[\,]]$, then $Non\text{-}Tau\text{-}Free(E[])$ and $E[\,]$ is single-threaded.*
4. *If $L_i \equiv [E[\,]]$, then N_i is not of type τ.*
5. *If $L_i \equiv nil$, then $Non\text{-}Tau\text{-}Free(N_i)$*

Proof: By case analysis, using Lemmas 5.1-2 and Subject Reduction Theorem for Types. □

Lemma 5.4 *Consider a closed single-threaded term N_0 in* **Muser**.
If $(N_0, []) \overset{n}{\longmapsto}_s (\ldots(Cnt\ x : \tau.\ E[x])\ldots\ ,\ L_1)$ *then $E[x]$ contains no value A of type τ, active or otherwise.*

Proof: By induction on the step m in which a continuation of type τ is generated. It is to be noticed that terms of the form $(Cnt\ x : \tau.\ \ldots)$ are not generated anywhere except at the **letcc** $k : \tau$ redex. Let,

$$(N_0, []) \overset{m}{\longmapsto}_s (M \equiv E[\textbf{letcc } k : \tau \textbf{ in } \ldots], L).$$

Let $(Cnt\ x : \tau.\ E[x])$ be generated during the next transition. By single-threadedness M must be of type τ. Hence by Lemma 5.3, $Non\text{-}Tau\text{-}Free(M)$ and $L \equiv nil$.

The continuation $(Cnt\ x : \tau.\ E[x])$ is a closed value. By Lemma 2.1, $E[\,]$ contains no active terms of type τ, thus no active values A. As M is single-threaded, values of type τ can only be found in captured continuations within M. But $Non\text{-}Tau\text{-}Free(M)$, therefore $E[\,]$ has no non-τ continuations. Any continuation of type τ present in $E[\,]$ must have been generated earlier. By the induction hypothesis they do not contain values of type τ, active or otherwise.

□

A theorem analogous to Theorem 3.1 can be stated regarding the safety of in-place updates in the presence of non-escaping continuations.

Theorem 5.1 (In-place Updates are Safe) *If*
$(F_0\ ,\ nil) \overset{*}{\longmapsto}_s (F_1\ ,\ L_1) \longmapsto_s (F_2\ ,\ L_2)$ *and*
$\rho\langle w_0, A''\rangle.(F_0'\ ,\ nil) \overset{*}{\Longrightarrow}_s \rho\langle w_0, A\rangle.(F_1'\ ,\ L_1')$
$\Longrightarrow_s \rho\langle w_0, A'\rangle.(F_2'\ ,\ L_2')$.

If F_0 is single-threaded and $F_0' \overset{\{w_0/A''\}}{\simeq} F_0$, then this substitution property is invariant over transition, i.e.
$F_2' \overset{\{w_0/A'\}}{\simeq} F_2$ *and* $L_2' \overset{\{w_0/A'\}}{\simeq} L_2$

Proof: The proof is by induction on the length of the transition. By Lemma 5.3, single-threadedness is preserved. Hence we only need to prove that if the substitution property holds for $(F_1\ ,\ L_1)$ then it holds for $(F_2\ ,\ L_2)$. Let $F_1 \equiv E_1[M_1]$. As w_0 is a special variable different from

bound variables and no redex except *mutation* changes the contents of w_0, the substitution property is trivial for all redexes except the mutation redex. If F_i is single-threaded and contains an active redex of type τ then $F_i : \tau$. By Lemma 5.3, If $F_i : \tau$, then $Non\text{-}Tau\text{-}Free(F_i)$ and $L_i \equiv nil$. In the case of the mutation redex,

- $(E[\mathcal{M}\ i\ j\ A]\ ,\ [\,]) \longmapsto_s (E[A']\ ,\ [\,])$
  $\rho\langle w_0, A\rangle.\ (E'[\mathcal{M}\ \ i\ \ j\ \ w_0]\ \ ,\ \ [\quad]) \Longrightarrow_s$
  $\rho\langle w_0, A'\rangle.\ (E'[w_0]\ ,\ [\,])$

  By assumption $E'[\,] \overset{\{w_0/A\}}{\cong} E[\,]$. To prove the substitution property, we must show that $E'[\,]\{w_0/A'\} \equiv E[\,]$. By transitivity this is the same as proving $E'[\,]\{w_0/A'\} \equiv E'[\,]\{w_0/A\}$. Hence if $E'[\,]$ does not contain any occurrence of w_0, the property holds. But if $E'[\,]$ contains an occurrence of w_0, then $E[\,]$ contains an occurrence of A. There are two possible cases:

  - A is an active value of type τ: this immediately violates Lemma 2.1, for single-threaded terms.
  - A is not an active value, hence by Lemma 5.4 must be enclosed in a non-τ continuation. As $F_1 \equiv (E[\mathcal{M}ijA])$ is of type τ, by Lemma 5.3, $Non\text{-}Tau\text{-}Free(F_1)$. Thus F_1 has no non-τ continuations. Thus $E'[]$ cannot have an occurrence of w_0 enclosed in a non-τ continuation.

□

6 Static Analysis of Programs

6.1 First-Order Continuations

Continuations of type $\eta\ cont$, where η is not of a function type or continuation type, are termed as first-order continuations.

Theorem 6.1 *Let M be single-threaded term where all* letcc *terms capture first-order continuations. Evaluation of the term M with in-place updates is safe.*

Proof: By Theorem 5.1, if transitions are made under the non-standard operational semantics defined in Section 5.1, then in-place updates are always safe. As certain transitions are not allowed in the non-standard operational semantics, it is a restricted version of the standard operational semantics. But a term with first-order continuations will not be restricted by the non-standard semantics: it never gets 'stuck'. This is because there are two clauses with guards:

$$(E[\textbf{throw}\ (Cnt\ x.\ c_1)\ V]\ ,\ [E'[\,]]) \longmapsto_s (E'[V]\ ,\ nil) \quad \underline{\text{if } Non\text{-}Tau\text{-}Free(V)}$$

$$(V\ ,\ E[\,]) \longmapsto_s (E[V]\ ,\ nil) \quad \underline{\text{if } Non\text{-}Tau\text{-}Free(V)}$$

As all continuations are first-order, the value V does not have a closure nor is it a continuation, hence $Non\text{-}Tau\text{-}Free(V)$ always holds. Thus if M evaluates to a value under the standard semantics, it also evaluates to the same value under the non-standard semantics. An application of Theorem 5.1 now gives the required result.

□

```
signature CPS = sig
eqtype var
datatype value = VAR of var
               |INT of int
datatype primop =
          * | - | + | div | ~ |
          < | <= | > | >= | =
datatype cexp =  APP of value * value list
   | FIX of
       (var * var list * cexp) list * cexp
   | PRIMOP of
       primop * value list * var list *
       cexp list
end
```

Figure 4: The CPS data-type

In the presence of higher-order continuations a term M evaluating under the non-standard semantics may get 'stuck' because of the guards to the two transitions. Theorem 5.1 guarantees that if a term does not get 'stuck' when evaluating under the non-standard semantics then in-place updates are always safe. A term under evaluation fails to satisfy the guard only if a non-τ continuation attempts to escape the scope of the outermost non-τ continuation enclosing it. Hence the strategy of the static analysis is to ensure that a non-τ continuation does not escape the syntactic scope of the outermost non-τ continuation enclosing it.

6.2 The CPS Language

In this section we describe a technique to statically analyse programs to determine whether they contain *escaping-continuations*. The language for this data-flow analysis is the *CPS data-type* that is the language used by the Standard ML of New Jersey Compiler [1]. The use of a different language is not an overhead as the analysis is to be used by the compiler itself. The advantage of using this language for analysis is the fact that all intermediate terms are values which have names, and *letcc* and *throw* are changed into function applications. A description of the language is given in Figure 4 as a SML/NJ *signature*.

The conditional is expressed as two continuations associated with the $PRIMOP\ '='$

$$PRIMOP(\ =\ ,\ [\ a\ ,\ b\ ]\ ,\ [\,]\ ,\ [\ C_1\ ,\ C_2\ ])$$

The convention being that if the equality indeed holds between a and b, then C_1 is executed otherwise C_2 is. The language allows functions defined within a *FIX* to be recursive.

This language does not contain *callcc/letcc* operations, as continuations are passed around as arguments. Figure 5 gives a translation of λ-terms into the *CPS data-type*, by a recursive function $\mathcal{F}$. $\mathcal{F}$ is defined on an object language where an abstraction is represented by FN, applications are represented explicitly by a constructor APP, recursive functions use an explicit FIX and variables use the constructor VAR. The translation of a term M, in the object language, is performed by calling $\mathcal{F}(M\ ,\ (fn\ x => x))$.

6.3 A Simple Analysis

The fundamental problem with higher-order analysis of functional programs is termination [5]. Theorem 5.1 presents a very liberal condition for safety of in-place updates. From the translation $\mathcal{F}$ it is seen that for every term (**letcc** k_1 **in** M), we are going to have the term:

$$FIX(\ [\ (k,[x],c(VAR\ x)),\ (f,[k_1,k_2],\ N)\ ],\ APP(f,[VAR\ k,\ VAR\ k]))$$

where $N \equiv \mathcal{F}(M, \lambda z.APP(VAR\ k_2\ ,\ [z]))$

The strategy for static analysis is to first locate every syntactically outermost declaration of a *non-*τ continuation. Let (**letcc** k_1 **in** M) be one such declaration. We must ensure that k_1 does not escape its syntactic scope. As seen in the definition of the term N, above, the $\mathcal{F}$-translation of M is made with the initial continuation, $\lambda z.APP(VAR\ k_2\ ,\ [z])$. Thus any captured continuation in the body of M must contain k_2 in its closure. As explained at the end of Section 4 it is also necessary to ensure that no captured continuation in the body of M escapes the scope demarcated by M. In the $\mathcal{F}$-translation this is equivalent to stating that k_2 does not escape the scope of $N \equiv \mathcal{F}(M, \lambda z.APP(VAR\ k_2\ ,\ [z]))$.

Our static analysis returns a set of functions that may escape the syntactic scope demarcated by M. If these escaping functions contain k_1 or k_2 in their closures then either k_1 or any continuation captured in the body of M may escape: hence, in-place updates are conservatively declared unsafe. If neither k_1 nor k_2 escapes, then static analysis will have to be repeated for every escaping function. This is because a declaration of a *non-*τ continuation, $P \equiv$ (**letcc** k **in** ...), in an escaping function, may become the outermost **letcc** term at run-time.

The question whether captured continuations, in the scope demarcated by M, escape becomes the question as to whether k_1 or k_2 escapes N. The key step in the solution is control-flow analysis, i.e. computing the set of defined-functions/free-variables to which a formal parameter of a function may be bound at run-time. The technique employed is similar to the one used in [9]. If a function escapes then we would like to know whether its closure contains either k_1 or k_2. But a statically computed closure contains free variables. Hence we have to use the control-flow information to find out whether a closure contains the variable k_1 or k_2. This will be an iterative computation as a statically-computed closure may contain the variable k_1 or k_2, or it may contain a parameter that may be bound to k_1 or k_2, or it may contain a closure which may contain k_1 or k_2, and so on recursively. This information is computed simultaneously with the control-flow data.

The flat closure associated with a function is the set of free variables present in the function body. In the absence of mutual recursion, flat closures can be computed in a single pass. All the $PRIMOP$s in the language take non-function arguments and return non-function results. Since we are interested exclusively in functions that escape, these non-functional terms are ignored. The function CLOSURE, as defined in Figure 6, returns a tuple: set of free variables of the term, and a function CL which maps every defined function to its flat closure.

Let (_, CL) = CLOSURE N

Notation:

- $FV(N)$ denotes the set of free variables of N.
- $\{\alpha \mapsto S_1\} \sqcup \{\alpha \mapsto S_2\} = \{\alpha \mapsto (S_1 \cup S_2)\}$, if S_1 and S_2 are sets.
- $\{\alpha \mapsto \beta_1\} \sqcup \{\alpha \mapsto \beta_2\} = \{\alpha \mapsto (\beta_1 \vee \beta_2)\}$, if β_1 and β_2 are boolean values.
- $\bigvee(S)$ is the disjunction on a set S, of boolean values.
- $(A - B - C) \equiv ((A - B) - C)$, i.e. the set difference operator is assumed to be left associative.

Control-flow analysis is done by a function CONTROL which takes in four parameters: a term T, and functions CL, $\mathcal{C}$ and $\mathcal{T}$. The function CL is the mapping of function names to their flat closures. The function $\mathcal{C}$ maps a parameter of a function to a set of functions to which it may be bound at run-time. The function $\mathcal{T}$ maps a function name to *true*, if its closure at run-time contains k_1 or k_2. $\mathcal{T}$ maps a parameter to *true*, if it may be bound to a closure containing k_1 or k_2. By default, it is assumed that $\mathcal{C}$ maps defined functions to themselves and $\mathcal{T}$ maps k_1 and k_2 to *true*. The function CONTROL, as defined in Figure 7, returns a tuple containing updated versions of $\mathcal{C}$ and $\mathcal{T}$. Control-flow analysis begins with (CONTROL N CL ($fn\_ \mapsto \phi$) ($fn\_ \mapsto$ *false*)) and terminates when CONTROL N CL $\mathcal{C}$ $\mathcal{T}$ = ($\mathcal{C}$, $\mathcal{T}$) . Iteration to fixed point of this routine must terminate, as each parameter of a defined function can be bound to only finitely many defined functions.

Once the functions $\mathcal{C}$ and $\mathcal{T}$ have been computed, we can compute the actual escape points of the term N. Escape points within N are points where there is an application to a free variable within N. To ensure that neither k_1 nor k_2 escape, all we need do, is verify that the arguments supplied at the escape points have their $\mathcal{T}$ value false. A function ANALYSE is defined, which takes in a term and the function $\mathcal{C}$, and returns the set of escaping functions of the term. As the control flow information is already available we can get more sophisticated than ANALYSE and throw out the escape points associated with functions that are not used.

- ANALYSE $PRIMOP(P\ ,\ \vec{l_0}\ ,\ \vec{l_1}\ ,\ [t_1\ ,\ \ldots, t_n])\ \mathcal{C}$
  = Let $\forall i,\ A_i = (\text{ANALYSE}\ t_i\ \mathcal{C})$
  in $(\bigcup_{i=1}^{n} A_i - \{\vec{l_0}\} - \{\vec{l_1}\})$
- ANALYSE $APP(VAR\ f\ ,\ [l_1\ ,\ \ldots\ ,\ l_n])\ \mathcal{C}$
  = if $(\text{FV}(N) \cap \mathcal{C}(f)) \neq \phi$
  then $\bigcup_{i=1}^{n} \mathcal{C}(l_i)$
  else ϕ
- ANALYSE $FIX([(g_1, \vec{v_1}, t_1), \ldots, (g_n, \vec{v_n}, t_n)]\ ,\ t_0)\ \mathcal{C}$
  = Let $\forall i,\ A_i = (\text{ANALYSE}\ t_i\ \mathcal{C})$
  in $(\bigcup_{i=1}^{n} A_i) \cup (\text{ANALYSE}\ t_0\ \mathcal{C})$

Let A = (ANALYSE N $\mathcal{C}$). If $[\bigvee(\mathcal{T}\ A) = true]$ then either k_1 or k_2 escapes. If neither k_1 nor k_2 escapes then the analysis has to be repeated for the body of every function in A.

Notation: $k : \eta\ cont$ denotes a non-τ continuation variable, i.e $\eta \not\equiv \tau$

$$\mathcal{F}(VAR\,v\,,\,c) \;=\; c(VAR\,v)$$

$$\mathcal{F}(FN(v,E),c) = FIX([(f,[v,k],\mathcal{F}(E,\lambda z.APP(VAR\ k,[z])))],c(VAR\ f))$$

$$\mathcal{F}(APP(PRIM\ throw,E),c) = \mathcal{F}(E,\lambda k.FIX([(f,[x,j],APP(k,[VAR\ x]))],c(VAR\ f)))$$

$$\mathcal{F}(APP(F,E),c) = FIX([(r,[x],c(VAR\ x))],\mathcal{F}(F,\lambda f.\mathcal{F}(E,\lambda e.APP(f,[e,\ VAR\ r]))))$$

$$\mathcal{F}(FIX(f,FN(v,B),E),c) = FIX([(f,[v,k],\mathcal{F}(B,\lambda z.APP(VAR\ k,[z])))],\mathcal{F}(E,c))$$

$$\begin{aligned}\mathcal{F}(LETCC(k_1,F),c) = FIX(\ &[\ (k,[x],c(VAR\ x)),\\ &\quad (f,[k_1,k_2],\mathcal{F}(F,\lambda z.APP(VAR\ k_2,[z])))\\ &],\\ &APP(f,[VAR\ k,\ VAR\ k]))\end{aligned}$$

Figure 5: Translation to CPS

- CLOSURE $PRIMOP(P\ ,\ \vec{l_0}\ ,\ \vec{l_1}\ ,\ [t_1\ ,\ \ldots,t_n])$
  $=$ Let $\forall i \le n\ \ (S_i\ ,\ \mathrm{CL}_i) = (\mathrm{CLOSURE}\ \ t_i)$
  in $(\bigcup_{i=1}^{n} S_i\ -\{\vec{l_0}\}\ -\{\vec{l_1}\}\ ,\ \bigsqcup_{i=1}^{n} \mathrm{CL}_i)$
- CLOSURE $APP(VAR\ f\ ,\ [l_1\ ,\ \ldots\ ,\ l_n])$
  $=\ (\{f\} \cup \{l_1\ ,\ \ldots\ ,\ l_n\},\ \phi)$
- CLOSURE $FIX([(g_1,\vec{v_1},t_1),\ldots,(g_n,\vec{v_n},t_n)]\ ,\ t)$
  Let $\forall i \le n\ \ (S_i\ ,\ \mathrm{CL}_i) = (\mathrm{CLOSURE}\ \ t_i)$
  $=$ $(S\ ,\ \mathrm{CL}) = (\mathrm{CLOSURE}\ \ t)$
  in $(S \bigcup_{i=1}^{n}(S_i - \{\vec{v_i}\ ,\ g_i\})\ ,\ \mathrm{CL} \bigsqcup_{i=1}^{n}(\mathrm{CL}_i \sqcup (g_i \mapsto (S_i - \{\vec{v_i}\ ,\ g_i\}))))$

Figure 6: Computing the Closure

- CONTROL $PRIMOP(P\ ,\ \vec{l_0}\ ,\ \vec{l_1}\ ,\ [t_1\ ,\ \ldots,t_n])\ \ \mathrm{CL}\ \ \mathcal{C}_0\ \ \mathcal{T}_0$
  $=$ Let $\forall i \le n\quad (\mathcal{C}_i\ ,\ \mathcal{T}_i)\ =\ (\mathrm{CONTROL}\ \ t_i\ \ \mathrm{CL}\ \ \mathcal{C}_{i-1}\ \ \mathcal{T}_{i-1})$
  in $(\mathcal{C}_n\ ,\ \mathcal{T}_n)$
- CONTROL $APP(VAR\ f\ ,\ [l_1\ ,\ \ldots\ ,\ l_n])\ \ \mathrm{CL}\ \ \mathcal{C}\ \ \mathcal{T}$
  Let $\_\ =\ \forall g \in \mathcal{C}(f)\quad$ if $g\ \equiv (VAR\ h\ ,\ [\alpha_1,\ldots,\alpha_n])$
  $=$ then $\mathcal{C} \longleftarrow \mathcal{C} \bigsqcup_{i=1}^{n}\{\alpha_i \longmapsto \mathcal{C}(l_i)\}$
  $\mathcal{T} \longleftarrow \mathcal{T} \bigsqcup_{i=1}^{n}\{\alpha_i \longmapsto \bigvee \mathcal{T}(\mathcal{C}\ l_i)\}$
  in $(\mathcal{C}\ ,\ \mathcal{T})$
- CONTROL $FIX([(g_1,\vec{v_1},t_1),\ldots,(g_n,\vec{v_n},t_n)]\ ,\ t_0)\ \ \mathrm{CL}\ \ \mathcal{C}_0\ \ \mathcal{T}_0$
  Let $(\mathcal{C}_1\ ,\ \mathcal{T}_1)\ =\ \mathrm{CONTROL}\ \ t_1\ \ \mathrm{CL}\ \ \mathcal{C}_0\ \ \mathcal{T}_0$
  $=$ $\forall i > 1\quad (\mathcal{C}_i\ ,\ \mathcal{T}_i)\ =\ \mathrm{CONTROL}\ \ t_i\ \ \mathrm{CL}\ \ \mathcal{C}_{i-1}\ \ (\mathcal{T}_{i-1} \sqcup (g_{i-1} \longmapsto \bigvee\ \mathcal{T}(\mathrm{CL}\ g_{i-1}))$
  in $\mathrm{CONTROL}\ \ t_0\ \ \mathrm{CL}\ \ \mathcal{C}_n\ \ (\mathcal{T}_n \sqcup (g_n \longmapsto \bigvee\ \mathcal{T}(\mathrm{CL}\ g_n)))$

Figure 7: Control Flow Analysis

The algorithm for static analysis can be summarised as follows,

```
Static_Analyse(M);
{

stack = nil;
For every outermost term of the form
   (f, [k1 : η cont, k2 : η cont], N) in M
  stack = (f, [k1, k2], N) :: stack;
While (stack ≠ nil) do
{
  (f, [k1, k2], N) = hd stack;
  stack = tail stack;
  ( _, CL ) = CLOSURE( N );
  ( C, T ) = CONTROL N CL C T ;
  /* Fix Point Computation */
  S = ANALYSE N C ;
  If ⋁(T(S))
  then return ( In-place updates are unsafe )
  else For every function g ∈ S, defined within N
         For every outermost term of the form
            (f, [k1 : η cont, k2 : η cont], N) in g
           stack = (f, [k1, k2], N) :: stack
}
return(In-place updates are safe);

}
```

7 Relation to Existing Work

A different approach to safe in-place updates is taken in [3]. In that paper a non-standard semantics is defined for a first-order language, where the store keeps an explicit count of the number of references to a value of the *abstract data-type*. Any term which is single-threaded by the criterion presented here can have safe in-place updates when analysed by Hudak's criteria. In a language with multi-argument first-order functions with a strict left-to-right evaluation the criterion presented here can be improved. But even with an improved criterion the reverse containment does not hold. This is because while the criterion presented here is strictly context-free, Hudak's criterion uses information about the call-sites of the function. Let $P \equiv (\lambda x.\ M) : \tau \rightarrow \eta$, where $\eta \not\equiv \tau$, be a single-threaded term. M cannot contain an active mutating operation. But if in all positions where P is used there is no subsequent use of a value of type τ then in-place mutation in M should be safe. This can be detected by Hudak's analysis.

The analysis considered in [3] does not extend in a simple-way to handle *callcc* and *throw*. The source of the problem is *throw*. This is because the current context is thrown away during the *throw* operation and thus reference counts of memory locations drop by an arbitrary number.

8 Conclusion

A simple condition is presented for the safety of in-place updates in purely functional programs with higher-order control operators. The verification of this condition takes no more time than required to perform any other control-flow analysis by the compiler. Thus there is no significant overhead on the compiler to implement this optimisation. By Theorem 6.1, if all continuations in a single-threaded program are first-order, then in-place updates are always safe. As this is an extremely common case, it is seen that control-flow analysis need not be performed for a large number of cases.

An important extension of the above work, is to present it as an abstract interpretation of the λ-calculus and then present a finite or computable abstraction of the interpretation. In many situations it is desirable to have a type system which can statically reject programs which do not allow safe in-place updates. The analysis presented here is a static approximation of the run-time situation. In the presence of control operators it is doubtful that a type system that is not overtly constraining can be obtained.

Acknowledgements

The author is grateful for technical and editorial help from Carl Gunter.

References

[1] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. The MIT Press, 1992.

[3] P. Hudak. A semantic model of reference counting and its abstraction. In *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987. (Preliminary version appeared in Proceedings 1986 ACM Conference on LISP and Functional Programming, August 1986, pp. 351-363).

[4] P. Hudak. Mutable abstract datatypes. Technical Report YALEU/DCS/RR-914, Yale University, 1993.

[5] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Principles of Programming Languages.* ACM, 1986.

[6] P. Curien R. Cartwright and M.Felleisen. Fully abstract models of observably sequential languages. Technical Report TR93-219, Rice University, 1993.

[7] J. Raoult and R. Sethi. The global storage needs of a subcomputation. In *Principles of Programming Languages.* ACM, 1984.

[8] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Language and Systems*, 1985.

[9] O. Shivers. Control flow analysis in scheme. In *Conference on Programming Language Design and Implementation.* ACM, 1988.

[10] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* The MIT Press, 1977.

[11] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Department of Computer, Rice University, 1991.