J. Functional Programming **11** (3): 319–346, May 2001. Printed in the United Kingdom

Set constraints for destructive array update optimization

MITCHELL WAND and WILLIAM D. CLINGER*

College of Computer Science, Northeastern University, 161 Cullinane Hall, Boston MA 02115 USA
(*e-mail:* {wand,will}@ccs.neu.edu)

Abstract

Destructive array update optimization is critical for writing scientific codes in functional languages. We present set constraints for an interprocedural update optimization that runs in polynomial time. This is a multi-pass optimization, involving interprocedural flow analyses for aliasing and liveness. We characterize and prove the soundness of these analyses using small-step operational semantics. We also prove that any sound liveness analysis induces a correct program transformation.

Capsule Review

The authors present two small-step operational semantics – one based on an environment, and the other on a store – for a first-order call-by-value functional language with flat arrays. The store semantics permits destructive array updates, whereas the environment semantics does not. A transformation is defined from the pure language to the one with destructive updates, and it is shown that for any sound live variable analysis, the transformation is correct; i.e. program meaning is preserved. The proof architecture adopted in showing correctness is unique, as is the notion of "computation addresses", which both record order of evaluation and serve as addresses in the store semantics. The authors then introduce a propagation analysis based on set constraints to determine whether an expression that evaluates to an array could be propagating an array that was passed to it. Together with an alias analysis also based on set constraints, they then derive a sound live variable analysis, thus leading to a correct method for array update optimization.

1 Introduction

In this paper we use set constraints to reformulate an algorithm for interprocedural array update optimization in a call-by-value functional language. We then prove the correctness of the program transformation that introduces imperative assignments, as well as the correctness of the analyses on which this transformation is based.

* Work supported by the National Science Foundation under grants numbered CCR-9404646, CCR-9629801, and CCR-9804115.

1.1 The destructive update problem

Since there are no assignments or other side effects in functional languages, updating an array or other data structure generally involves creating a copy that is like the original except at the index being updated. Such copying can drastically degrade the performance of the program, and can easily increase its asymptotic complexity.

Destructive update transformation is an optimization that transforms functional updates into assignments whenever a flow analysis reveals that the array value being updated is dead following the update. In an imperative language this is always the case, because updating by assignment kills the previous value of the array.

Our optimization is based on the flow analysis of Sastry *et al.* (1993; 1994). As reported previously, this is a very effective optimization. It was also the first interprocedural update analysis to run in polynomial time. As originally presented, this algorithm's efficiency depended upon a special technique for symbolic computation of a least fixed point (Chuang and Goldberg, 1992). In the reformulated algorithm, the generated set constraints are easily seen to be data flow inequalities, which can be solved in polynomial time using standard techniques (Aiken *et al.*, 1993).

1.2 Proof architecture

We use a small-step operational semantics. (A small-step operational semantics represents a computation by a linear sequence of *configurations*, whereas a big-step operational semantics represents a computation by a tree of subcomputations.) We assume that for each program E, there are an infinite number of possible starting configurations C_0, representing the different computations possible using E (say, by running it on different data). We say a configuration C is *reachable* from E iff it appears as a configuration in the computation starting from some initial configuration C_0.

We imagine that we have some transformation we would like to apply to programs. We can apply the transformation to our program E to get a transformed program E^*, and to an initial configuration C_0 to get a transformed initial configuration C_0^*. The goal is to show that C_0 reaches a final configuration iff C_0^* reaches a final configuration representing the same answer.[1]

We do this by introducing a relation $\mathscr{R}$ between configurations of the original program and configurations of the transformed program. Our transformation will be correct if $\mathscr{R}$ has suitable properties on initial and terminal configurations, and is preserved under computation, that is,

$$(C \mathrel{\mathscr{R}} D) \wedge (C \to C') \wedge (D \Rightarrow D') \Longrightarrow (C' \mathrel{\mathscr{R}} D') \tag{1}$$

where → is the transition relation of the original program and ⇒ is the transition relation of the transformed program.

Unfortunately, this is usually false. It fails just when the two configurations are

[1] For simplicity, we consider only transformations that do not change the number of steps in the computation; it is conceptually easy but notationally messy to remove this restriction.

executing a phrase in the program that has been transformed. For example, if we are replacing a copying array update in C by an in-place update in D, the side-effects of the in-place update might be visible, destroying any correspondence $\mathscr{R}$ we might have.

Hence we fall back to the weaker condition

$$(C \text{ reachable}) \wedge (C \mathrel{\mathscr{R}} D) \wedge (C \rightarrow C') \wedge (D \Rightarrow D') \Longrightarrow (C' \mathrel{\mathscr{R}} D') \tag{2}$$

which is enough to establish the desired result. We use the reachability information by finding a property P such that

$$(C \text{ reachable}) \Longrightarrow (C \models P) \tag{3}$$

and

$$(C \models P) \wedge (C \mathrel{\mathscr{R}} D) \wedge (C \rightarrow C') \wedge (D \Rightarrow D') \Longrightarrow (C' \mathrel{\mathscr{R}} D') \tag{4}$$

In this proof architecture, the primary purpose of program analysis is to find such a P. Property (3) expresses the *soundness* of the analysis, and (4) expresses the *correctness* of the transformation as justified by the analysis.

We use the paradigm of *set-based analysis* to find such properties P. In this paradigm, we first characterize a class of propositions P to be found; this gives us a notion of soundness such as (3). For any program E, we then show how to generate a set of constraints on the form of P. Typically these constraints can be solved using well-known algorithms (e.g. Heintze and Jaffar (1990), and many others). To show the constraint generation system is correct, we prove that if P is any solution to the constraints generated from E, then property (3) holds.

In general, the proof of (3) may require course-of-values induction on the length of the execution sequence leading to C, that is, it may involve not only the immediate predecessor of C in the sequence, but also the global structure of the sequence (see, for example, case 1(c) of Theorem 24). We may also need to deal with propositions that are not just about a single configuration, but about larger computations. Nielson calls such propositions 'second-order analyses' (Nielson, 1985). Our propagation analysis $\mathscr{P}[\![-]\!]$ (section 6) is an example.

In this paper, Definition 7 corresponds to property (3); Theorem 9 corresponds to property (4), and Theorem 24 shows the correctness of the constraint-generation system.

1.3 Outline

We begin in section 2 by describing our functional source language and its environment semantics, which is a small-step operational semantics using an environment and a continuation. In section 3 we obtain our imperative target language by adding destructive updates to the source language. The semantics of destructive updates are not expressible in the environment semantics, so we must introduce a store semantics.

Section 4 presents our main result: We state the soundness property for live-variable analysis, define a transformation from source to target languages, and assert that any sound live-variable analysis yields a correct transformation.

http://dx.doi.org/10.1017/S0956796801003938

E, F	$::=$	$\theta : T$	expressions (labelled terms)
θ	$\in$	Lab	expression labels
T	$::=$	$x \mid \phi(E, \ldots)$	terms
	$\mid$	$\textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } E_2$	
ϕ	$::=$	$f_1 \mid \ldots \mid f_N \mid g$	function symbols
g	$::=$	$\mathrm{NEW} \mid \mathrm{REF} \mid \mathrm{UPD} \mid p$	primitives
p	$\in$	$Prim$	scalar primitives

Fig. 1. Syntax of expressions.

EC	$::=$	$\langle \textbf{halted}, v \rangle \mid \langle \alpha, \rho, G, K \rangle$	configurations
α	$\in$	N^*	computation addresses (ordered lexicographically)
ρ	$\in$	$Var \rightarrow_{\mathrm{fin}} Val$	environments
$v \in Val$	$::=$	$bv \mid lv$	denoted values
bv	$::=$	$true \mid false \mid 0 \mid 1 \mid \ldots$	scalar (basic) values
$lv \in LabVal$	$::=$	$\alpha{:}\langle bv, \ldots \rangle$	labelled arrays of scalar values
G	$::=$	$E \mid v$	partially reduced expressions
	$\mid$	$\theta : \phi(v_1, \ldots, v_{i-1}, E_i, \ldots, E_n)$	
	$\mid$	$\theta : \textbf{if } v \textbf{ then } E_1 \textbf{ else } E_2$	
R	$::=$	$\theta : \phi(v_1, \ldots, v_{i-1}, [\,], E_{i+1}, \ldots, E_n)$	return contexts
	$\mid$	$\theta : \textbf{if } [\,] \textbf{ then } E_1 \textbf{ else } E_2$	
K	$::=$	$\textbf{halt} \mid \langle \alpha, \rho, R, K \rangle$	continuations

Fig. 2. Configurations of the environment semantics.

Sections 5 through 8 develop a live-variable analysis and prove its correctness. To do this, we first present a propagation analysis, which determines when the output of a procedure can be the same array as one of its inputs. We then use the propagation analysis to develop an alias analysis that keeps track of when two variables in an environment may denote the same array. The use of instrumented values allows us to do both of these within the environment semantics. Finally, we use both the propagation and alias analyses to develop the live-variable analysis. In each case, the presentation follows the pattern suggested above: First we characterize a kind of flow analysis. Next we define a notion of soundness for this kind of analysis. Then we write down a set of constraints generated from the source program. Lastly, we prove a theorem showing that any solution to these constraints is sound.

Finally, section 9 proves the correctness of the transformation that introduces destructive updates.

We conclude with a discussion of related work and possible extensions.

$$\langle \alpha,\ \rho,\ \theta\!:\!x,\ K\rangle \rightarrow \langle \alpha,\ \rho,\ \rho(x),\ K\rangle \qquad \text{[FETCH]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!\phi(v_1,\ldots,v_{i-1},E_i,E_{i+1},\ldots,E_n),\ K\rangle\\ &\quad\rightarrow\ \langle \alpha.i,\ \rho,\ E_i,\ \langle \alpha,\ \rho,\ \theta\!:\!\phi(v_1,\ldots,v_{i-1},[\,],E_{i+1},\ldots,E_n),\ K\rangle\rangle\end{aligned} \qquad \text{[PUSH]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ v,\ \langle \alpha',\ \rho',\ R,\ K\rangle\rangle\\ &\quad\rightarrow\ \langle \alpha',\ \rho',\ R[v],\ K\rangle\end{aligned} \qquad \text{[RETURN]}$$

$$\langle \alpha,\ \rho,\ v,\ \textbf{halt}\rangle \rightarrow \langle \textbf{halted},\ v\rangle \qquad \text{[HALT]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!f_k(v_1,\ldots,v_n),\ K\rangle\\ &\quad\rightarrow\ \langle \alpha.(n+1),\ \{x_{k:1}\mapsto v_1,\ldots,x_{k:n}\mapsto v_n\},\ F_k,\ K\rangle\end{aligned} \qquad \text{[CALL]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!p(bv_1,\ldots,bv_n),\ K\rangle\\ &\quad\rightarrow\ \langle \alpha.(n+1),\ \rho,\ bv',\ K\rangle\\ &\quad\ \text{if } p^*(bv_1,\ldots,bv_n)=bv'\end{aligned} \qquad \text{[PRIMOP]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!\mathrm{NEW}(n,bv),\ K\rangle\\ &\quad\rightarrow\ \langle \alpha,\ \rho,\ \alpha\!:\!\langle bv,\ldots,bv\rangle,\ K\rangle\\ &\quad\ \text{NEW creates a new array filled with } n \text{ copies of } bv.\end{aligned} \qquad \text{[NEW-E]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!\mathrm{REF}(\beta\!:\!\langle bv_1,\ldots\rangle,j),\ K\rangle\\ &\quad\rightarrow\ \langle \alpha,\ \rho,\ bv_j,\ K\rangle\end{aligned} \qquad \text{[REF-E]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!\mathrm{UPD}(\beta\!:\!\langle bv_1,\ldots,bv_n\rangle,j,bv'),\ K\rangle\\ &\quad\rightarrow\ \langle \alpha,\ \rho,\ \alpha\!:\!\langle bv_1,\ldots,bv',\ldots,bv_n\rangle,\ K\rangle\\ &\quad\ \text{UPD produces a new copy of the array}\\ &\quad\ \text{with the } j\text{-th element changed to } bv'.\end{aligned} \qquad \text{[UPD-E]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!\textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } E_2,\ K\rangle\\ &\quad\rightarrow\ \langle \alpha.1,\ \rho,\ E_0,\ \langle \alpha,\ \rho,\ \theta\!:\!\textbf{if } [\,] \textbf{ then } E_1 \textbf{ else } E_2,\ K\rangle\rangle\end{aligned} \qquad \text{[PUSH-TEST]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!\textbf{if } \mathit{true} \textbf{ then } E_1 \textbf{ else } E_2,\ K\rangle\\ &\quad\rightarrow\ \langle \alpha.2,\ \rho,\ E_1,\ K\rangle\end{aligned} \qquad \text{[BRANCH-TRUE]}$$

$$\begin{aligned}&\langle \alpha,\ \rho,\ \theta\!:\!\textbf{if } \mathit{false} \textbf{ then } E_1 \textbf{ else } E_2,\ K\rangle\\ &\quad\rightarrow\ \langle \alpha.2,\ \rho,\ E_2,\ K\rangle\end{aligned} \qquad \text{[BRANCH-FALSE]}$$

Fig. 3. Reduction rules for the environment semantics.

2 Source language

Our source language is a first-order, call-by-value functional language, specified by recursion equations. Values are either scalar (basic) values bv or arrays of scalars. Each array is time-stamped with a *computation address*. These time-stamps are highly structured; their structure is crucial to the proofs.

The detailed syntax of expressions is given in Figure 1. Expressions are labelled first-order terms, including conditionals. Function symbols may be either procedure symbols f_k or primitive function symbols; the latter are divided into the array primitives NEW, REF, and UPD, and the scalar primitives. Each scalar primitive p takes scalar arguments and returns a scalar value given by a function $p^* : bv^n \to bv$. Every function symbol has a fixed arity, and the number of arguments must match this arity; for notational convenience, we do not include this information in the grammar, but we will use it implicitly throughout.

The semantics and the analysis assume that we are looking at a fixed program

$$\begin{array}{lcl} f_1(x_{1:1}, \ldots, x_{1:n_1}) & = & F_1 \\ f_2(x_{2:1}, \ldots, x_{2:n_2}) & = & F_2 \\ & \vdots & \\ f_N(x_{N:1}, \ldots, x_{N:n_N}) & = & F_N \\ \textbf{in } F_0 & & \end{array}$$

where the $x_{k:j}$ are distinct variables, and for each $i \in \{0, \ldots, N\}$ the free variables of F_i excluding function symbols, which we abbreviate as $\mathrm{fv}(F_i)$, are a subset of $\{x_{i:1}, \ldots, x_{i:n_i}\}$.

2.1 Environment semantics

We give this language a small-step operational semantics using environments and continuations. We refer to this as the *environment semantics*.

For the purpose of formulating and proving our theorems, the semantics is instrumented by adding a computation address to each configuration and to each continuation frame. Each array is time-stamped with the computation address at which it was created. These addresses are clearly extraneous to the computation; we omit the tedious formulation of an uninstrumented semantics and the forgetful transformation linking the instrumented and uninstrumented semantics.

As described in figure 2, a configuration either is of the form $\langle \textbf{halted}, v \rangle$ or else it consists of a computation address, an environment, a partially reduced expression, and a continuation. A configuration of the form $\langle \textbf{halted}, v \rangle$ is said to be *successful*. A continuation is either the initial continuation **halt**, or else it consists of a saved computation address, a saved environment, a return context (representing a return address and saved temporaries), and a nested continuation.

Definition 1 (*Initial, Reachable*)
An *initial* configuration is a configuration

$$\langle \alpha_0, \rho_0, F_0, \textbf{halt} \rangle$$

where $\alpha_0 = 1$ and ρ_0 contains only scalar values. A *reachable* configuration is any configuration that is reachable from an initial configuration by the rules of figure 3.

The initial configurations differ only in the initial environment ρ_0. The restriction of ρ_0 to scalar values can be removed; see section 12.

The operational semantics is given by the reduction rules in figure 3. The rule [PUSH] begins the evaluation of the i-th argument in a function call; when evaluation of the argument is completed, its value will be left in the E-register, and argument evaluation will be resumed by [RETURN]. When a value is returned to the initial continuation **halt**, the machine halts successfully.

If there are no more subexpressions to evaluate, the function is called. This may be either a procedure call ([CALL]) or a primitive. In the case of a procedure call, the procedure body is executed tail-recursively in an appropriate environment. In the case of a scalar primitive p, the interpretation p^* of p is invoked and the value returned in the E-register ([PRIMOP]). In the case of an array primitive, the appropriate transformation is performed. Note that each array is time-stamped with the computation address at which it was created. If any of the arguments is of the wrong type (not an integer, not a scalar, or not an array), the computation is *stuck* (halted unsuccessfully).

A similar set of rules ([PUSH-TEST], [BRANCH-TRUE] and [BRANCH-FALSE]) manages conditionals.

This notation suppresses the program, which is used in the [CALL] rule.

The environment semantics carries around more information than it needs because the environment component of a continuation often contains values for variables that aren't needed by the component that indicates the return context. We therefore define a congruence $\cong$ that relates configurations that differ only by such dead variables.

Definition 2 (*Congruence*, $\cong$)
We define congruence first for configurations and then for continuations:

- $\langle \textbf{halted}, v\rangle \cong \langle \textbf{halted}, v'\rangle$ iff $v = v'$.
- $\langle \alpha, \rho, G, K\rangle \cong \langle \alpha', \rho', G', K'\rangle$ iff
  1. $\alpha = \alpha'$,
  2. $G = G'$,
  3. for all $x \in \mathrm{fv}(G)$, $\rho(x) = \rho'(x)$, and
  4. $K \cong K'$
- **halt** $\cong$ **halt** always.
- $\langle \alpha, \rho, \theta{:}\phi(v_1,\ldots,v_{i-1},[\,],E_{i+1},\ldots,E_n), K\rangle$
  $\cong \langle \alpha', \rho', \theta{:}\phi(v'_1,\ldots,v'_{i-1},[\,],E'_{i+1},\ldots,E'_n), K'\rangle$ iff
  1. $\alpha = \alpha'$,
  2. for all $j \in [0, i-1]$ $v_j = v'_j$,
  3. for all $j \in [i+1, n]$ $E_j = E'_j$,
  4. for all $j \in [i+1, n]$, for all $x \in \mathrm{fv}(E_j)$, $\rho(x) = \rho'(x)$, and
  5. $K \cong K'$
- $\langle \alpha, \rho, \theta{:}\textbf{if } [\,] \textbf{ then } E_1 \textbf{ else } E_2, K\rangle$
  $\cong \langle \alpha', \rho', \theta{:}\textbf{if } [\,] \textbf{ then } E'_1 \textbf{ else } E'_2, K'\rangle$ iff
  1. $\alpha = \alpha'$,
  2. $E_1 = E'_1$ and $E_2 = E'_2$,
  3. for all $j \in \{1, 2\}$, for all $x \in \mathrm{fv}(E_j)$, $\rho(x) = \rho'(x)$, and
  4. $K \cong K'$
- These are the only cases for which the congruence holds.

Congruence is preserved by reduction; furthermore congruent configurations either both have reductions or both are terminal:

Lemma 3 (*Preservation of Congruence*)

If $EC_1 \cong EC_2$, and $EC_1 \rightarrow EC_1'$, then there exists a unique EC_2' such that $EC_2 \rightarrow EC_2'$. Furthermore $EC_1 \rightarrow EC_1'$ and $EC_2 \rightarrow EC_2'$ by the same rule, and $EC_1' \cong EC_2'$.

Proof

By examination of the reduction rules. □

3 Target language

Our imperative target language has exactly the same syntax as the purely functional source language, except that we add a destructive update operation UPD! to the list of primitive operations:

$$g ::= \text{NEW} \mid \text{REF} \mid \text{UPD} \mid p \mid \text{UPD!}$$

Unlike the UPD operation, which creates and returns a new array, the UPD! operation writes a new value into the storage occupied by the original array, and then returns a pointer to that storage. This side effect is not expressible in the environment semantics.

3.1 Store semantics

To express the semantics of this new operation, we must use a store semantics. A store is a finite function from locations to labelled values. Locations are computation addresses. We denote locations by α or β when we rely on this fact, or by l in contexts where this fact can be ignored. This trick allows locations to carry their own time stamps. As with the environment semantics, it would be easy enough to construct a bisimulation between this instrumented semantics and an uninstrumented one in which locations are modelled in some more primitive way.

Storable values are labelled values (arrays). Denoted values are now scalar values or locations. A halted configuration may contain a scalar value or an array. A non-halted configuration contains the same information as an environment configuration, plus a store.

The configurations of our store semantics are the same as those of the environment semantics, except they contain a store as a fifth component. Apart from the rules in figure 4, which show how arrays are managed in the store, the reduction rules for the store machines are the same as for the environment machines (adding an unchanging store component to each rule). Notice that a location is dereferenced before halting; this prevents the location at which an array is allocated from being observable outside the program, which would wreck the equivalence between unoptimized and optimized programs.

$$\begin{array}{lr}
\langle \alpha,\ \rho,\ \theta\!:\!\mathrm{NEW}(n, bv),\ K,\ \Sigma\rangle & \\
\quad \rightarrow\ \langle \alpha,\ \rho,\ \alpha,\ K,\ \Sigma[\alpha \mapsto \alpha\!:\!\langle bv, \ldots, bv\rangle]\rangle & [\textsc{new-s}] \\
\qquad \text{new array allocated at } \alpha \text{, filled with } n \text{ copies of } bv &
\end{array}$$

$$\begin{array}{lr}
\langle \alpha,\ \rho,\ \theta\!:\!\mathrm{REF}(\beta, j),\ K,\ \Sigma\rangle & \\
\quad \rightarrow\ \langle \alpha,\ \rho,\ bv_j,\ K,\ \Sigma\rangle & [\textsc{ref-s}] \\
\qquad \text{if } \Sigma(\beta) = \gamma\!:\!\langle bv_1, \ldots, bv_n\rangle &
\end{array}$$

$$\begin{array}{lr}
\langle \alpha,\ \rho,\ \theta\!:\!\mathrm{UPD}(\beta, j, bv'),\ K,\ \Sigma\rangle & \\
\quad \rightarrow\ \langle \alpha,\ \rho,\ \alpha,\ K,\ \Sigma[\alpha \mapsto \alpha\!:\!\langle bv_1, \ldots, bv', \ldots, bv_n\rangle]\rangle & [\textsc{upd-s}] \\
\qquad \text{where } \Sigma(\beta) = \beta'\!:\!\langle bv_1, \ldots, bv_j, \ldots, bv_n\rangle & \\
\qquad \text{UPD produces a new copy of the modified array,} & \\
\qquad \text{allocated at the fresh location } \alpha. &
\end{array}$$

$$\begin{array}{lr}
\langle \alpha,\ \rho,\ \theta\!:\!\mathrm{UPD!}(\beta, j, bv'),\ K,\ \Sigma\rangle & \\
\quad \rightarrow\ \langle \alpha,\ \rho,\ \beta,\ K,\ \Sigma[\beta \mapsto \alpha\!:\!\langle bv_1, \ldots, bv', \ldots, bv_n\rangle]\rangle & [\textsc{upd!}] \\
\qquad \text{where } \Sigma(\beta) = \beta'\!:\!\langle bv_1, \ldots, bv_j, \ldots, bv_n\rangle & \\
\qquad \text{UPD! destructively modifies the array.} &
\end{array}$$

$$\langle \alpha,\ \rho,\ bv,\ \mathbf{halt},\ \Sigma\rangle \rightarrow \langle \mathbf{halted},\ bv\rangle \qquad [\textsc{halt-b}]$$

$$\langle \alpha,\ \rho,\ l,\ \mathbf{halt},\ \Sigma\rangle \rightarrow \langle \mathbf{halted},\ \Sigma(l)\rangle \qquad [\textsc{halt-a}]$$

Fig. 4. Reduction rules for the store semantics.

3.2 Store elimination

We can map a store configuration to an environment configuration by *Store Elimination*: we replace any locations by their contents and replace any UPD! by UPD. We show representative portions of this inductive definition in figure 5.

We use store elimination to define the reachable configurations of the store semantics.

Definition 4 (*Initial, Reachable* (*store*))

A store configuration is initial iff its store component is empty and its *Elim* is an initial environment configuration. A store configuration is reachable iff it is reachable from some initial store configuration by the rules of the store semantics.

Storage elimination loses any sharing relationships that may hold in the store configuration. In the absence of destructive update this doesn't matter, so the successive configurations of the environment semantics can be obtained by store elimination from the successive configurations of the store semantics; this is Corollary 16 below. We will prove the correctness of our optimization by showing that this happens even in the presence of destructive updates, provided those updates are justified by a sound flow analysis.

$$
\begin{aligned}
&\mathit{Elim}\ \langle \alpha,\ \rho,\ G,\ K,\ \Sigma\rangle \\
&\quad = \langle \alpha,\ \mathit{Elim}\ \rho\ \Sigma,\ \mathit{Elim}\ G\ \Sigma,\ \mathit{Elim}\ K\ \Sigma\rangle \\
&\\
&(\mathit{Elim}\ \rho\ \Sigma)(x) = \mathit{Elim}(\rho(x))\ \Sigma \\
&\\
&\mathit{Elim}\ \alpha\ \Sigma = \Sigma(\alpha) \\
&\mathit{Elim}\ \mathit{bv}\ \Sigma = \mathit{bv} \\
&\mathit{Elim}\ \beta\ \Sigma = \Sigma(\beta) \\
&\mathit{Elim}\,[\,]\ \Sigma = [\,] \\
&\mathit{Elim}\ \mathrm{UPD!}\ \Sigma = \mathrm{UPD} \\
&\mathit{Elim}\ \phi\ \Sigma = \phi \quad (\phi \neq \mathrm{UPD!}) \\
&\mathit{Elim}(\theta : \phi(E_1, \ldots))\ \Sigma \\
&\quad = \theta : (\mathit{Elim}\ \phi\ \Sigma)((\mathit{Elim}\ E_1\ \Sigma), \ldots) \\
&\quad \vdots
\end{aligned}
$$

Fig. 5. Store elimination (partial definition).

4 The transformation

The goal of our analysis is to identify variables that denote locations that are certainly not live in a store continuation. If the program contains $\mathrm{UPD}(x, E_1, E_2)$, and we know that x can't be bound to a live location after E_1 and E_2 have been evaluated, then it will be safe to replace the UPD by a UPD!.

We will need two distinct but closely related notions of liveness, one for the environment semantics and another for the store semantics. These definitions are formally similar, because the structure of a continuation is the same in both the environment and the store semantics: a continuation is either the initial continuation **halt** or a tuple consisting of a computation address, an environment, a return context, and a nested continuation.

Definition 5 (*Live Location* (*environment*))
For the environment semantics, liveness is defined as follows.

- No location is live in **halt**.
- l is live in $\langle \alpha,\ \rho,\ R,\ K\rangle$ iff either:
  1. l is the label of an array that occurs in R, or
  2. there exists $x \in \mathrm{fv}(R)$ such that $\rho(x) = l{:}\langle \ldots \rangle$, or
  3. l is live in K.

Definition 6 (*Live Location* (*store*))
For the store semantics, liveness is defined as follows.

- No location is live in **halt**.
- l is live in $\langle \alpha,\ \rho,\ R,\ K\rangle$ iff either:
  1. l occurs in R, or
  2. there exists $x \in \mathrm{fv}(R)$ such that $\rho(x) = l$, or
  3. l is live in K.

We can now state the soundness condition for a live variable analysis $\mathscr{L}[\![-]\!]$.

Definition 7 (*Live Variable Analysis*)

A *live variable analysis* $\mathscr{L}[\![-]\!]$ is a map from expression labels θ to sets of variables. $\mathscr{L}[\![-]\!]$ is *sound* iff for each label θ, $\mathscr{L}[\![\theta]\!]$ is a set of variables such that for all reachable store configurations of the form $\langle \alpha, \rho, \theta{:}T, K, \Sigma\rangle$, $\rho(x)$ live in K implies $x \in \mathscr{L}[\![\theta]\!]$.

The idea behind the transformation is that if $\theta{:}\mathrm{UPD}(x, E_1, E_2)$ is an update in the program, and $x \notin \mathscr{L}[\![\theta]\!]$, then we can replace the UPD by UPD!, because the stores after the UPD and UPD! will agree on the locations that are live in the continuation K. We formulate the transformation as follows:

Definition 8 (*The Transformation* $(-)^*$)

Given a program or expression E, let Θ be a set of labels such that every $\theta \in \Theta$ labels an update of the form $\theta{:}\mathrm{UPD}(x, E_1, E_2)$, where $x \notin \mathscr{L}[\![\theta]\!]$. Then E^* is the result of replacing $\theta{:}\mathrm{UPD}(x, E_1, E_2)$ by $\theta{:}\mathrm{UPD}!(x, E_1^*, E_2^*)$ for each $\theta \in \Theta$.

Note that $\mathit{Elim}\, E^*\, \Sigma = \mathit{Elim}\, E\, \Sigma$, because the transformation $(-)^*$ merely replaces some occurrences of UPD by UPD!, and these differences are erased by *Elim*.

Our reduction rules depend upon the program because the rule for a call to f_k mentions its body F_k. We will use → to indicate reduction using the original program and ⇒ to indicate reduction using the transformed program.

We now state our main theorem, which establishes the correctness of the transformation. Its proof appears in section 9 below.

Theorem 9 (*Main Theorem*)

Let $\mathscr{L}[\![-]\!]$ be a sound live variable analysis, and let $\langle \alpha_0, \rho_0, F_0, \mathbf{halt}, \Sigma_0\rangle$ be an initial configuration. If

$$\langle \alpha_0, \rho_0, F_0, \mathbf{halt}, \Sigma_0\rangle \rightarrow^n \langle \alpha, \rho, G, K, \Sigma\rangle$$

and

$$\langle \alpha_0, \rho_0, F_0^*, \mathbf{halt}, \Sigma_0\rangle \Rightarrow^n \langle \alpha', \rho', G', K', \Sigma'\rangle,$$

then

$$\mathit{Elim}\langle \alpha, \rho, G, K, \Sigma\rangle \cong \mathit{Elim}\langle \alpha', \rho', G', K', \Sigma'\rangle$$

Corollary 10 (*Correctness of Transformation*)

If $\mathscr{L}[\![-]\!]$ is a sound live variable analysis, and $\langle \alpha_0, \rho_0, F_0, \mathbf{halt}, \Sigma_0\rangle$ is an initial configuration, then

$$\langle \alpha_0, \rho_0, F_0, \mathbf{halt}, \Sigma_0\rangle \rightarrow^n \langle \mathbf{halted}, v\rangle$$

if and only if

$$\langle \alpha_0, \rho_0, F_0^*, \mathbf{halt}, \Sigma_0\rangle \Rightarrow^n \langle \mathbf{halted}, v\rangle$$

Proof

From Theorem 9 and Lemma 3. □

5 Consistency

We will not be done, of course, until we show how to find a sound live-variable analysis $\mathscr{L}[\![-]\!]$. To get to the live-variable analysis, we perform two preliminary analyses – a propagation analysis and an alias analysis. We carry out these analyses on the environment semantics; in proving our theorems, we lift them to give useful results for the store semantics.

For each analysis we generate a set of constraints from the program, and show that any solution to those constraints yields a safety property of the environment semantics (and hence of the store semantics).

The soundness of all these analyses involves a notion that we call consistency. A configuration of the environment semantics is *consistent* iff each of its components was created after its subcomponents (which we formalize using the time stamps), every expression is a subexpression of the original program, and there are no unbound variables.

For any continuation K, the time at which K was created corresponds to the computation address of the configuration that created it. The reduction rules imply that this computation address crtime(K) can be computed as follows:

$$\begin{aligned}&\mathrm{crtime}(\textbf{halt}) = 1\\&\mathrm{crtime}(\langle\alpha,\ \rho,\ \theta\!:\!\phi(v_1,\ldots,v_{i-1},[\,],E_{i+1},\ldots,E_n),\ K\rangle) = \alpha.i\\&\mathrm{crtime}(\langle\alpha,\ \rho,\ \theta\!:\!\textbf{if}\ [\,]\ \textbf{then}\ E_1\ \textbf{else}\ E_2,\ K\rangle) = \alpha.1\end{aligned}$$

The formal definition of consistency is somewhat involved; an explanation follows the definition.

Definition 11 (*Environment Consistency*)

- $\langle\textbf{halted},\ v\rangle$ is consistent.
- $\langle\alpha,\ \rho,\ E,\ K\rangle$ is consistent iff
  1. E occurs in the program;
  2. $\mathrm{fv}(E) \subseteq \mathrm{dom}(\rho)$;
  3. $\forall\beta$ if $\beta\!:\!\langle\ldots\rangle \in \mathrm{ran}(\rho)$ then $\beta < \alpha$ (in the lexicographic order);
  4. crtime(K) is a prefix of α; and
  5. K is consistent.
- $\langle\alpha,\ \rho,\ v,\ K\rangle$ is consistent iff
  1. if $v = \beta\!:\!\langle bv,\ldots\rangle$ then $\beta < \alpha$ or α is a prefix of β;
  2. $\forall\beta$ if $\beta\!:\!\langle\ldots\rangle \in \mathrm{ran}(\rho)$ then $\beta < \alpha$;
  3. crtime(K) is a prefix of α; and
  4. K is consistent.
- $\langle\alpha,\ \rho,\ \theta\!:\!\phi(v_1\ldots v_{i-1},E_i,\ldots,E_n),\ K\rangle$ is consistent iff
  1. $\exists E_1,\ldots,E_{i-1}$ such that $\theta\!:\!\phi(E_1,\ldots,E_{i-1},E_i,\ldots,E_n)$ occurs in the program;
  2. $\forall j \in \{i,\ldots,n\}\ \mathrm{fv}(E_j) \subseteq \mathrm{dom}(\rho)$;
  3. $\forall j \in \{1,\ldots,i-1\}$ if $v_j = \beta\!:\!\langle\ldots\rangle$ then $\beta < \alpha$ or $(\alpha.j)$ is a prefix of β;
  4. $\forall\beta$ if $\beta\!:\!\langle\ldots\rangle \in \mathrm{ran}(\rho)$ then $\beta < \alpha$;
  5. crtime(K) is a prefix of α; and
  6. K is consistent.

- $\langle \alpha,\ \rho,\ \theta\!:\!\textbf{if}\ v\ \textbf{then}\ E_1\ \textbf{else}\ E_2,\ K\rangle$ is consistent iff
  1. $\exists E_0$ such that $\theta\!:\!\textbf{if}\ E_0\ \textbf{then}\ E_1\ \textbf{else}\ E_2$ occurs in the program;
  2. $\mathrm{fv}(E_1) \cup \mathrm{fv}(E_2) \subseteq \mathrm{dom}(\rho)$;
  3. if $v = \beta\!:\!\langle\ldots\rangle$ (a type error, since v should be a boolean), then $\beta < \alpha$ or $(\alpha.1)$ is a prefix of β;
  4. $\forall\beta$ if $\beta\!:\!\langle\ldots\rangle \in \mathrm{ran}(\rho)$ then $\beta < \alpha$;
  5. $\mathrm{crtime}(K)$ is a prefix of α; and
  6. K is consistent.
- **halt** is consistent.
- $\langle \alpha,\ \rho,\ \theta\!:\!\phi(v_1 \ldots v_{i-1}, [\,], E_{i+1}, \ldots, E_n),\ K\rangle$ is consistent iff there exists an expression E_i such that the configuration

$$\langle \alpha,\ \rho,\ \theta\!:\!\phi(v_1 \ldots v_{i-1}, E_i, E_{i+1}, \ldots, E_n),\ K\rangle$$

  is consistent.
- $\langle \alpha,\ \rho,\ \theta\!:\!\textbf{if}\ [\,]\ \textbf{then}\ E_1\ \textbf{else}\ E_2,\ K\rangle$ is consistent iff there exists an expression E_0 such that the configuration $\langle \alpha,\ \rho,\ \theta\!:\!\textbf{if}\ E_0\ \textbf{then}\ E_1\ \textbf{else}\ E_2,\ K\rangle$ is consistent.

To understand this definition, consider the clause for a configuration of the form $\langle \alpha,\ \rho,\ \theta\!:\!\phi(v_1 \ldots v_{i-1}, E_i, \ldots, E_n),\ K\rangle$. This configuration describes the situation at computation address α, during evaluation of the arguments to a function ϕ. The clause says that:

1. The partially reduced expression is obtained by taking a labelled expression from the original program and replacing the first $i-1$ subexpressions by their values.
2. There are no unbound variables in the unevaluated expressions.
3. For each evaluated argument, the value was either created prior to starting to evaluate these arguments, or else it was created during the evaluation of the corresponding actual parameter.
4. All the array values in ρ were created before starting to evaluate these arguments.
5. The embedded continuation K was created prior to α.
6. Finally, the embedded continuation K is similarly consistent.

Lemma 12 (*Preservation of Consistency*)
Consistency is preserved by the environment semantics.

Proof
By tedious examination of the reduction rules. The most interesting cases involve the [RETURN] and [CALL] rules.

Suppose the left hand side of the [RETURN] rule is consistent:

$$\langle \alpha,\ \rho,\ v,\ \langle \alpha',\ \rho',\ R,\ K\rangle\rangle \rightarrow \langle \alpha',\ \rho',\ R[v],\ K\rangle$$

Since the continuation $\langle \alpha',\ \rho',\ R,\ K\rangle$ is consistent, there exists an expression E for which $\langle \alpha',\ \rho',\ R[E],\ K\rangle$ is a consistent configuration. To show $\langle \alpha',\ \rho',\ R[v],\ K\rangle$ consistent, there remains only to show that if v is a labelled value $\beta\!:\!\langle bv, \ldots\rangle$, then $\beta < \alpha'$ or α' is a prefix of β. By consistency of the left-hand side, we know that

$\beta < \alpha$ or α is a prefix of β. We also know that crtime($\langle \alpha', \rho', R, K \rangle$) is a prefix of α, which implies that α' is a proper prefix of α. By lexicographic ordering it follows that if $\beta < \alpha$, then $\beta < \alpha'$ or α' is a prefix of β. If α is a prefix of β, then so is α'. Hence $\beta < \alpha'$ or α' is a prefix of β.

Suppose the left hand side of the [CALL] rule is consistent:

$$\langle \alpha,\ \rho,\ \theta\!:\! f_k(v_1,\ldots,v_n),\ K \rangle \rightarrow \langle \alpha.(n+1),\ \rho',\ F_k,\ K \rangle$$

where $\rho' = \{x_{k:1} \mapsto v_1, \ldots, x_{k:n} \mapsto v_n\}$. From the consistency of the left hand side it follows that

$$\forall \beta \text{ if } \beta\!:\!\langle\ldots\rangle \in \operatorname{ran}\rho',\ \text{then } \beta < (\alpha.(n+1)),$$

that crtime(K) is a prefix of α, hence of $(\alpha.(n+1))$, and that K is consistent. □

A configuration of the store semantics is consistent iff its store elimination is consistent and every allocated location was allocated at the same time or before its most recent update. Formally:

Definition 13 (*Store Consistency*)
A store configuration $SC = \langle \alpha, \rho, G, K, \Sigma \rangle$ is consistent iff

- *Elim SC* is defined; that is, none of its components refer to unallocated locations in the store;
- *Elim SC* is consistent; and
- for all $\alpha \in \operatorname{dom}\Sigma$ if $\Sigma(\alpha) = \beta\!:\!\langle\ldots\rangle$ then $\alpha \leqslant \beta$.

Although consistency is an invariant of reduction in the environment semantics, the store semantics does not necessarily preserve consistency. Part of the proof of our main theorem (Theorem 9) therefore involves showing that, for both the original and transformed program, every reachable configuration is consistent.

In Lemma 15 we show that the [UPD!] rule is the only rule that can destroy consistency. First we must show that the computation address of a consistent store configuration represents a fresh (unallocated) location.

Lemma 14
If $\langle \alpha, \rho, G, K, \Sigma \rangle$ is consistent, and β occurs in K, then $\beta < \alpha$.

Proof
By structural induction on K. No locations occur in **halt**, so the base case is trivial.

Let $K = \langle \alpha', \rho', R, K' \rangle$, where any location β that occurs in K' satisfies $\beta < \alpha$. Let l be a location that occurs in K. There are three cases.

If l occurs in R, then $\theta\!:\!\phi(v_1,\ldots,v_{i-1},[\,], E_{i+1},\ldots,E_n)$ and $l = v_j$ for some j with $1 \leqslant j < i$. Let $\Sigma(l) = \gamma\!:\!\langle\ldots\rangle$. By consistency, we know that $(\alpha'.i)$ is a prefix of α, $l \leqslant \gamma$, and either $\gamma < \alpha'$ or $(\alpha'.j)$ is a prefix of γ. Either way, $l \leqslant \gamma < \alpha$.

If $\rho'(x) = l$ then consistency tells us that $l < \alpha' < \alpha$.

If l occurs in K', then $l < \alpha$ by the induction hypothesis. □

Lemma 15 (*Preservation of Store Consistency*)
If SC is a consistent store configuration, and $SC \rightarrow SC'$ by any rule other than [UPD!], then SC' is consistent and $\mathit{Elim}\, SC \rightarrow \mathit{Elim}\, SC'$.

Proof
Let SC be consistent. Then $Elim\,SC$ is consistent. By examination of the rules, if $SC \to SC'$ by any rule other than [UPD!], and $Elim\,SC$ is consistent, then SC' is consistent. If $Elim\,SC \to Elim\,SC'$, then $Elim\,SC'$ is consistent by Lemma 12.

It therefore suffices to prove that $Elim\,SC \to Elim\,SC'$. This is trivial if $SC \to SC'$ by any rule that leaves the store unchanged, leaving [NEW-S] and [UPD-S] as the only rules for which we must prove $Elim\,SC \to Elim\,SC'$. Consider the rule [NEW-S]:

$$\begin{array}{rcl} SC & = & \langle \alpha,\ \rho,\ \theta\!:\mathrm{NEW}(n, bv),\ K,\ \Sigma\rangle \\ & \to & \langle \alpha,\ \rho,\ \alpha,\ K,\ \Sigma'\rangle \\ & = & SC' \end{array}$$

where $\Sigma' = \Sigma[\alpha \mapsto \alpha{:}\langle bv, \ldots, bv\rangle]$. By consistency of SC, α does not occur in ρ or in K, so $Elim\,\rho\Sigma' = Elim\,\rho\Sigma$ and $Elim\,K\Sigma' = Elim\,K\Sigma$. Therefore $Elim\,SC \to Elim\,SC'$.

The proof for [UPD-S] is similar. □

Recall that → indicates reduction using the original program, which contains no occurrences of UPD!.

Corollary 16 (*Simulation*)
Let SC_0 be an initial store configuration, and let $SC_0 \to^n SC$. Then $Elim\,SC_0 \to^n Elim\,SC$.

Most of our proofs are by induction on the length of a computation in a small-step semantics, as in Corollary 16, or by induction on the structure of a continuation, as in Lemma 14. Occasionally, however, we will need the more global view that a big-step semantics would provide. We conclude this section with an interpolation theorem that allows us to look at the global structure of a computation in the environment semantics.

Theorem 17 (*Interpolation Theorem*)

1. If $\langle \alpha,\ \rho,\ \theta\!:f_k(E_1, \ldots, E_n),\ K\rangle \to^* \langle \alpha',\ \rho',\ v,\ K\rangle$, then there exist values $v_1, \ldots, v_n$ such that

$$\begin{array}{rl} & \langle \alpha,\ \rho,\ \theta\!:f_k(E_1, \ldots, E_n),\ K\rangle \\ \to^* & \langle \alpha,\ \rho,\ \theta\!:f_k(v_1, E_2, \ldots, E_n),\ K\rangle \\ \to^* & \langle \alpha,\ \rho,\ f_k(v_1, v_2, E_3, \ldots, E_n),\ K\rangle \\ \ldots & \\ \to^* & \langle \alpha,\ \rho,\ f_k(v_1, \ldots, v_n),\ K\rangle \\ \to^* & \langle \alpha',\ \rho',\ v,\ K\rangle \end{array}$$

2. If $\langle \alpha,\ \rho,\ \theta\!:\mathbf{if}\ E_0\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2,\ K\rangle \to^* \langle \alpha',\ \rho',\ v,\ K\rangle$, then there exists a value v' such that

$$\begin{array}{rl} & \langle \alpha,\ \rho,\ \theta\!:\mathbf{if}\ E_0\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2,\ K\rangle \\ \to^* & \langle \alpha,\ \rho,\ \theta\!:\mathbf{if}\ v'\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2,\ K\rangle \\ \to^* & \langle \alpha,\ \rho',\ v,\ K\rangle. \end{array}$$

Proof
Observe that the rules treat the continuation as a stack, with [PUSH] and [PUSH-TEST] as the pushing operations and [RETURN] as the matching pop operation. So the computation sequence

$$\langle \alpha,\ \rho,\ \theta\!:\!f_k(E_1,\dots,E_n),\ K\rangle \rightarrow^* \langle \alpha',\ \rho',\ v,\ K\rangle$$

must contain some number of matching push/pop pairs. The computation must look like

$$\begin{array}{ll}
 & \langle \alpha,\ \rho,\ f_k(E_1,\dots,E_n),\ K\rangle \\
\rightarrow & \langle \alpha.1,\ \rho,\ E_1,\ \langle \alpha,\ \rho,\ f_k([\ ],E_2,\dots,E_n),\ K\rangle\rangle \\
\rightarrow^* & \langle \beta_1,\ \rho_1,\ v_1,\ \langle \alpha,\ \rho,\ f_k([\ ],E_2,\dots,E_n),\ K\rangle\rangle \\
\rightarrow & \langle \alpha,\ \rho,\ f_k(v_1,E_2,\dots,E_n),\ K\rangle \\
\rightarrow & \langle \alpha.2,\ \rho,\ E_2,\ \langle \alpha,\ \rho,\ f_k(v_1,[\ ],E_2,\dots,E_n),\ K\rangle\rangle \\
\rightarrow^* & \langle \beta_2,\ \rho_2,\ v_2,\ \langle \alpha,\ \rho,\ f_k(v_1,[\ ],E_2,\dots,E_n),\ K\rangle\rangle \\
\rightarrow & \langle \alpha,\ \rho,\ f_k(v_1,v_2,E_3,\dots,E_n),\ K\rangle \\
\dots & \\
\rightarrow^* & \langle \alpha,\ \rho,\ f_k(v_1,v_2,\dots,v_n),\ K\rangle \\
\rightarrow^* & \langle \alpha',\ \rho',\ v,\ K\rangle
\end{array}$$

The case for conditionals is similar. □

This interpolation theorem is a surrogate for a big-step (or ‘natural’) semantics in which the nodes of the computation tree might take the form

$$\alpha : \rho \vdash \theta\!:\!E \rightarrow v$$

signifying that $\theta\!:\!E$, started with environment ρ, terminates with value v; α then becomes the address of this node in the tree.

6 Propagation analysis

The propagation analysis determines whether the value of an expression is the same as that of one of its free variables. ‘Sameness’ is measured using the array labels (time stamps). This is yet another place where we take advantage of the instrumentation in our environment semantics.

Definition 18 (*Propagation Analysis*)
A *propagation analysis* $\mathscr{P}[\![-]\!]$ is a map from expressions to sets of variables. A propagation analysis $\mathscr{P}[\![-]\!]$ is *sound* iff whenever $\langle \alpha,\ \rho,\ E,\ K\rangle$ is consistent and $\langle \alpha,\ \rho,\ E,\ K\rangle \rightarrow^* \langle \alpha',\ \rho',\ \beta\!:\!\langle bv\dots\rangle,\ K\rangle$ then either

1. $\beta\!:\!\langle bv,\dots\rangle \in \{\rho(x) \mid x \in \mathscr{P}[\![E]\!]\}$ or
2. α is a prefix (not necessarily proper) of β.

These two conditions say that if E evaluates to an array, then either the array is the value of one of the variables in $\mathscr{P}[\![E]\!]$, or it was created during the execution of E.

To find the sets $\mathscr{P}[\![-]\!]$, we solve the constraints in Figure 6. Note that there is no constraint generated for any call to a primitive $g(E_1,\dots,E_n)$, so in the smallest

P1. $\mathscr{P}[\![x]\!] \supseteq \{x\}$.
P2. $\mathscr{P}[\![\textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } E_2]\!] \supseteq \mathscr{P}[\![E_1]\!] \cup \mathscr{P}[\![E_2]\!]$.
P3. If $x_{k:i} \in \mathscr{P}[\![F_k]\!]$, then $\mathscr{P}[\![f_k(E_1, \ldots, E_n)]\!] \supseteq \mathscr{P}[\![E_i]\!]$

Fig. 6. Set constraints for $\mathscr{P}[\![-]\!]$.

solution $\mathscr{P}[\![g(E_1, \ldots, E_n)]\!]$ will be empty. This is reasonable, since any such term always produces a fresh value; there are no destructive updates in the source program.

A straightforward fixed-point iteration suffices to find the smallest solution to the constraints. However, *any* solution to the constraints is a sound analysis:

Theorem 19 (*Correctness of* $\mathscr{P}[\![-]\!]$)
Any solution to the constraints P1–P3 is a sound propagation analysis.

Proof
We check the conditions for all E simultaneously, by induction on the length of the computation $\langle \alpha,\ \rho,\ E,\ K\rangle \rightarrow^* \langle \alpha',\ \rho',\ \beta{:}\langle\ldots\rangle,\ K\rangle$. The induction step will proceed by cases on E.

1. If E is a variable x, then we must have $\rho(x) = \beta{:}\langle\ldots\rangle$ and $x \in \mathscr{P}[\![x]\!]$. Therefore $\beta{:}\langle\ldots\rangle \in \{\rho(y) \mid y \in \mathscr{P}[\![x]\!]\}$.
2. If E is a primitive $g(E_1, \ldots, E_n)$, then it must be that g is either NEW or UPD, since all other primitives return basic values (by assumption). In either case $\beta = \alpha$, so α is a prefix of β.
3. If E is of the form $f_k(E_1, \ldots, E_n)$, then by Theorem 17 (Interpolation Theorem) the calculation must look like

$$\begin{array}{ll}
 & \langle \alpha,\ \rho,\ f_k(E_1, \ldots, E_n),\ K\rangle \\
\rightarrow^* & \langle \alpha,\ \rho,\ f_k(v_1, E_2, \ldots, E_n),\ K\rangle \\
\rightarrow^* & \langle \alpha,\ \rho,\ f_k(v_1, v_2, E_3, \ldots, E_n),\ K\rangle \\
\ldots & \\
\rightarrow^* & \langle \alpha,\ \rho,\ f_k(v_1, \ldots, v_n),\ K\rangle \\
\rightarrow & \langle \alpha.(n+1),\ \{x_{k:1} = v_1, \ldots, x_{k:n} = v_n\},\ F_k,\ K\rangle \\
\rightarrow^* & \langle \alpha',\ \rho',\ \beta{:}\langle\ldots\rangle,\ K\rangle
\end{array}$$

Let ρ^* denote the environment $\{x_{k:1} = v_1, \ldots, x_{k:n} = v_n\}$.
The last calculation is shorter than the original, and it begins in a consistent state, so the induction hypothesis applies. Hence either $\alpha.(n+1)$ is a prefix of β, so α is also a prefix of β, or else

$$\begin{array}{rcl}
\beta{:}\langle\ldots\rangle & \in & \{\rho^*(x) \mid x \in \mathscr{P}[\![F_k]\!]\} \\
 & = & \{v_i \mid x_{k:i} \in \mathscr{P}[\![F_k]\!]\}
\end{array}$$

If α is a prefix of β, we are done. So consider the second alternative. There exists an i such that $x_{k:i} \in \mathscr{P}[\![F_k]\!]$ and $\beta{:}\langle\ldots\rangle = v_i$. Since $x_{k:i} \in \mathscr{P}[\![F_k]\!]$, we know that $\mathscr{P}[\![E_i]\!] \subseteq \mathscr{P}[\![f_k(E_1 \dots E_n)]\!]$ by P3. So now consider the subcomputation

$$\begin{array}{ll} & \langle \alpha,\ \rho,\ f_k(v_1,\dots,v_{i-1},E_i,\dots,E_n),\ K\rangle \\ \rightarrow & \langle \alpha.i,\ \rho,\ E_i,\ \langle \alpha,\ \rho,\ f_k(v_1,\dots,v_{i-1},[\,],E_{i+1},\dots,E_n),\ K\rangle\rangle \\ \rightarrow^* & \langle \beta_i,\ \rho_i,\ \beta{:}\langle\dots\rangle,\ \langle \alpha,\ \rho,\ f_k(v_1,\dots,v_{i-1},[\,],E_{i+1},\dots,E_n),\ K\rangle\rangle \end{array}$$

Applying the induction hypothesis to the last part of this subcomputation, we deduce that either $\alpha.i$ is a prefix of β, so α is a prefix of β and we are done, or else

$$\begin{array}{rll} \beta{:}\langle\dots\rangle & \in \{\rho(x) \mid x \in \mathscr{P}[\![E_i]\!]\} & \\ & \subseteq \{\rho(x) \mid x \in \mathscr{P}[\![f_k(E_1,\dots,E_n)]\!]\} & \text{by P3} \end{array}$$

4. The calculation for conditionals is similar. □

7 Alias analysis

The propagation analysis determines when the output of a procedure can be the same array (or location) as one of its inputs. In our next analysis, we use the results of this analysis to determine when two of the inputs to a procedure can be the same.

Definition 20 (*Alias Analysis*)
An *alias set* $\mathscr{A}$ is a subset of $Var \times Var$.

To characterize the soundness of an alias analysis $\mathscr{A}$, we define a proposition $\langle \alpha,\ \rho,\ G,\ K\rangle \models \mathscr{A}$ that relates $\mathscr{A}$ to configurations of the environment semantics. Informally, this proposition says that if two variables x and y denote the same array value within some environment that appears anywhere within the configuration, then $(x, y) \in \mathscr{A}$; and similarly that if two actual parameters denote the same array value within some partially reduced expression or return context, and the corresponding formal parameters are x and y, then $(x, y) \in \mathscr{A}$.

Definition 21 ($(-) \models \mathscr{A}$)

- $\langle \alpha,\ \rho,\ G,\ K\rangle \models \mathscr{A}$ iff
  1. $\rho \models \mathscr{A}$,
  2. $G \models \mathscr{A}$, and
  3. $K \models \mathscr{A}$.
- $\rho \models \mathscr{A}$ iff $\rho(x) = \rho(y) \Longrightarrow (x, y) \in \mathscr{A}$.
- $G \models \mathscr{A}$ iff one of the following holds:
  1. G is an expression E.
  2. G is a value v.
  3. $G = \theta{:}f_k(v_1,\dots,v_{i-1},E_i,\dots,E_n)$ and for all $j, j' \in [1, i-1]$, $v_j = v_{j'} = \beta{:}\langle bv_1,\dots\rangle \Longrightarrow (x_{k:j}, x_{k:j'}) \in \mathscr{A}$.
  4. $G = \theta{:}$**if** v **then** E_1 **else** E_2.
- **halt** $\models \mathscr{A}$ always.
- $\langle \alpha,\ \rho,\ R,\ K\rangle \models \mathscr{A}$ iff
  1. $\rho \models \mathscr{A}$,
  2. $R \models \mathscr{A}$, and
  3. $K \models \mathscr{A}$.

A1. $\mathscr{A}$ is an equivalence relation.
A2. For each call $f_k(E_1, \ldots, E_n)$ that occurs in the program, if

$$A \star \mathscr{P}[\![E_j]\!] \cap \mathscr{A} \star \mathscr{P}[\![E_{j'}]\!] \neq \emptyset$$

then $(x_{k:j}, x_{k:j'}) \in \mathscr{A}$.

Fig. 7. Set constraints for $\mathscr{A}$.

- $f_k(v_1, \ldots, v_{i-1}, [\,], E_{i+1}, \ldots, E_n) \models \mathscr{A}$ iff for all $j, j' \in [1, i-1]$, $v_j = v_{j'} \Longrightarrow (x_{k:j}, x_{k:j'}) \in \mathscr{A}$.
- $g(v_1, \ldots, v_{i-1}, [\,], E_{i+1}, \ldots, E_n) \models \mathscr{A}$ always.
- **if** $[\,]$ **then** E_1 **else** $E_2 \models \mathscr{A}$ always.

An alias analysis is sound if it predicts all of the aliasing that can actually occur during the execution of the program:

Definition 22 (*Soundness of* $\mathscr{A}$)

$\mathscr{A}$ is a *sound alias analysis* iff $\langle \alpha, \rho, G, K \rangle \models \mathscr{A}$ for every reachable configuration $\langle \alpha, \rho, G, K \rangle$.

For $S \subseteq \mathit{Var}$, define

$$\mathscr{A} \star S = \{x \mid (x, y) \in \mathscr{A} \wedge y \in S\}$$

The constraints for $\mathscr{A}$ are given in figure 7. Once again, these constraints take the form of closure conditions, so the smallest solution can be found by iterating to find the least fixed point.

Theorem 23 (*Correctness of* $\mathscr{A}$)

If $\mathscr{P}[\![-]\!]$ is a sound propagation analysis and $\mathscr{A}$ satisfies the constraints A1–A2, then $\mathscr{A}$ is a sound alias analysis.

Proof

By induction on the number of steps taken to get from an initial configuration $\langle \alpha_0, E_0, \rho_0, K_0 \rangle$ to $EC = \langle \alpha, \rho, G, K \rangle$. By the definition of $(-) \models \mathscr{A}$, we need only be concerned with rules that change the environment component of a configuration ([CALL]) or create or change a partially reduced call or call-like return context that involves a function symbol f_k ([PUSH], [RETURN]). It is easy to see that the [CALL] and [PUSH] rules preserve $\mathscr{A}$. So consider a step that returns to a return context for a call to f_k. By Theorem 17 (Interpolation Theorem), the computation through this step must look like:

$$\begin{array}{ll}
 & \langle \alpha_0,\ E_0,\ \rho_0,\ K_0 \rangle \\
\rightarrow^* & \langle \alpha,\ \rho,\ \theta\!:\!f_k(E_1,\ldots,E_n),\ K \rangle \\
\rightarrow & \langle \alpha.1,\ \rho,\ E_1,\ \langle \alpha,\ \rho,\ \theta\!:\!f_k([\,],E_2,\ldots,E_n),\ K \rangle\rangle \\
\rightarrow^* & \langle \alpha_1,\ \rho_1,\ v_1,\ \langle \alpha,\ \rho,\ \theta\!:\!f_k([\,],E_2,\ldots,E_n),\ K \rangle\rangle \\
\rightarrow & \langle \alpha,\ \rho,\ \theta\!:\!f_k(v_1,E_2,\ldots,E_n),\ K \rangle \\
\rightarrow & \langle \alpha.2,\ \rho,\ E_2,\ \langle \alpha,\ \rho,\ \theta\!:\!f_k(v_1,[\,],E_3,\ldots,E_n),\ \rangle\rangle \\
\ldots & \\
\rightarrow^* & \langle \alpha_i,\ \rho_i,\ v_i,\ \langle \alpha,\ \rho,\ \theta\!:\!f_k(v_1\ldots v_{i-1},[\,],E_{i+1},\ldots,E_n),\ K \rangle\rangle \\
\rightarrow & \langle \alpha,\ \rho,\ \theta\!:\!f_k(v_1,\ldots,v_i,E_{i+1},\ldots,E_n),\ K \rangle
\end{array}$$

We assume by induction that all but the last of these configurations satisfy $\mathscr{A}$. We need to show that the last configuration also satisfies $\mathscr{A}$. All of the conditions for $EC \models \mathscr{A}$ are satisfied, except perhaps for $v_j = v_{j'} \implies (x_{k:j'}, x_{k:j'}) \in \mathscr{A}$. To check this, we observe that the conditions for the propagation analysis are satisfied at each $\alpha.j$, so we know that if $v_j = \beta\!:\!\langle\ldots\rangle$, then either $\alpha.j$ is a prefix of β, or $\beta\!:\!\langle\ldots\rangle \in \{\rho(x) \mid x \in \mathscr{P}[\![E_j]\!]\}$.

So assume that $v_j = v_{j'} = \beta\!:\!\langle\ldots\rangle$. We must establish that $(x_{k:j}, x_{k:j'}) \in \mathscr{A}$. There are three cases:

1. $\alpha.j$ and $\alpha.j'$ are both prefixes of β. Then $j = j'$. Therefore $(x_{k:j}, x_{k:j'}) \in \mathscr{A}$, since $\mathscr{A}$ is an equivalence relation.
2. $\alpha.j$ is a prefix of β and $\beta\!:\!\langle\ldots\rangle \in \{\rho(x) \mid x \in \mathscr{P}[\![E_{j'}]\!]\}$. Then $\beta < \alpha$ by consistency, yielding the contradiction $(\alpha.j) \leqslant \beta < \alpha$. So this case is impossible.
3. $\beta\!:\!\langle\ldots\rangle \in \{\rho(x) \mid x \in \mathscr{P}[\![E_j]\!]\} \cap \{\rho(x) \mid x \in \mathscr{P}[\![E_{j'}]\!]\}$.

In the third case, $\beta\!:\!\langle\ldots\rangle \in \{\rho(x) \mid x \in \mathscr{P}[\![E_j]\!]\}$. Therefore there exists a $y \in \mathscr{P}[\![E_j]\!]$ such that $\rho(y) = \beta\!:\!\langle\ldots\rangle$. Since $\rho \models \mathscr{A}$, if $\rho(x) = \beta\!:\!\langle\ldots\rangle$, then $(x, y) \in \mathscr{A}$. Therefore $\{x \mid \rho(x) = \beta\!:\!\langle\ldots\rangle\} \subseteq \mathscr{A} \star \mathscr{P}[\![E_j]\!]$. Similarly, $\{x \mid \rho(x) = \beta\!:\!\langle\ldots\rangle\} \subseteq \mathscr{A} \star \mathscr{P}[\![E_{j'}]\!]$. Therefore

$$\{x \mid \rho(x) = \beta\!:\!\langle\ldots\rangle\} \subseteq \mathscr{A} \star \mathscr{P}[\![E_j]\!] \cap \mathscr{A} \star \mathscr{P}[\![E_{j'}]\!]$$

Since the left-hand side of this inequality is nonempty, so is the right-hand side. Hence, by constraint A2, $(x_{k:j}, x_{k:j'}) \in \mathscr{A}$. □

8 Live variable analysis

With the propagation and alias analyses in hand, we can now proceed to the live variable analysis. By Definition 7, which characterizes the soundness of a live variable analysis, our goal is to find sets $\mathscr{L}[\![\theta]\!]$ such that for every reachable state of the form $\langle \alpha,\ \rho,\ \theta\!:\!T,\ K,\ \Sigma \rangle$,

$$\text{if } \rho(x) \text{ is live in } K, \text{ then } x \in \mathscr{L}[\![\theta]\!]$$

To motivate the constraints for $\mathscr{L}[\![-]\!]$, observe that if θ occurs in the program as

$$\theta'\!:\!\phi(E_1,\ldots,E_{i-1},\theta\!:\!T,E_{i+1},\ldots,E_n)$$

L1. If θ occurs in the context

$$\theta' : \phi(E_1, \ldots, E_{i-1}, \theta : T, E_{i+1}, \ldots, E_n)$$

then

L1.1 $\mathscr{A} \star \mathscr{P}[\![E_j]\!] \subseteq \mathscr{L}[\![\theta]\!]$ for each $j \in [1, i-1]$.
L1.2 $\mathscr{A} \star \mathrm{fv}(E_j) \subseteq \mathscr{L}[\![\theta]\!]$ for each $j \in [i+1, n]$.
L1.3 $\mathscr{L}[\![\theta']\!] \subseteq \mathscr{L}[\![\theta]\!]$.

L2a. If θ occurs in the context

$$\theta' : \textbf{if } \theta : T \textbf{ then } E_1 \textbf{ else } E_2$$

then

L2a.1 $\mathscr{A} \star \mathrm{fv}(E_j) \subseteq \mathscr{L}[\![\theta]\!]$ for each $j \in [1, 2]$.
L2a.2 $\mathscr{L}[\![\theta']\!] \subseteq \mathscr{L}[\![\theta]\!]$.

L2b. If θ occurs in a context of the form

$$\theta' : \textbf{if } E_0 \textbf{ then } \theta : T \textbf{ else } E_2$$

or

$$\theta' : \textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } \theta : T$$

then $\mathscr{L}[\![\theta']\!] \subseteq \mathscr{L}[\![\theta]\!]$.

L3. If θ is the label of a procedure body F_k, then for each call

$$\theta' : f_k(E_1, \ldots, E_n)$$

in the program,

$$\mathscr{L}[\![\theta']\!] \cap \mathscr{P}[\![E_i]\!] \neq \emptyset \Longrightarrow x_{k:i} \in \mathscr{L}[\![\theta]\!]$$

Fig. 8. Set constraints for $\mathscr{L}[\![-]\!]$.

and if $\langle \alpha,\ \rho,\ \theta : T,\ K,\ \Sigma \rangle$ is reachable, then K must be of the form

$$\langle \alpha,\ \rho,\ \theta' : \phi(v_1, \ldots, v_{i-1}, [\,], E_{i+1}, \ldots, E_n),\ K' \rangle$$

A location may be live in this continuation in one of three ways:

1. it might be one of the v_j's, for $1 \leqslant j < i$;
2. it might be the value of a variable free in one of the E_j's (for $i < j \leqslant n$);
3. it might be live in K'.

Hence, a variable in ρ might be bound to a live location in one of three ways:

1. it might be (or be aliased to) a variable whose value is returned by one of the E_j $(1 \leqslant j < i)$;
2. it might be (or be aliased to) a variable that is free in one of the E_j's $(i < j \leqslant n)$;
3. it might be bound to a location that is live in K'.

These considerations correspond to the first group of set constraints L1 in figure 8. The second group L2 is analogous. L3 says that for a procedure call $\theta' : f_k(E_1, \ldots, E_n)$, if the value of some variable that is live at θ' can appear as the result of some actual parameter E_i, then in the analysis of the procedure body $\theta : F_k$ we must treat

http://dx.doi.org/10.1017/S0956796801003938

the corresponding formal parameter $x_{k:i}$ as live. This reflects the fact that this is an interprocedural analysis; a non-interprocedural analysis would have to take the conservative approach of regarding all formal parameters as live at θ.

Theorem 24 (*Correctness of* $\mathscr{L}[\![-]\!]$)
If $\mathscr{P}[\![-]\!]$ is a sound propagation analysis, $\mathscr{A}$ is a sound alias analysis, and $\mathscr{L}[\![-]\!]$ satisfies constraints L1–L3, then $\mathscr{L}[\![-]\!]$ is sound.

Proof
We proceed by induction on the length of the computation to the reachable state. The induction step proceeds by cases on how θ appears in the program.

1. If θ occurs in the context

$$\theta' : \phi(E_1, \dots, E_{i-1}, \theta : T, E_{i+1}, \dots, E_n)$$

then the only way in which $\theta : T$ can appear as the expression component of a store configuration is by a computation of the form

$$\begin{array}{ll} & \langle \alpha_0,\ \rho_0,\ E_0,\ K_0,\ \Sigma_0 \rangle \\ \to^* & \langle \alpha,\ \rho,\ \theta' : \phi(E_1, \dots, E_{i-1}, \theta : T, E_{i+1}, \dots, E_n),\ K',\ \Sigma \rangle \\ \to & \langle \alpha.1,\ \rho,\ E_1,\ \langle \alpha,\ \rho,\ \theta' : \phi([\,], E_2, \dots, E_{i-1}, \theta : T, E_{i+1}, \dots, E_n),\ K' \rangle,\ \Sigma \rangle \\ \to^* & \langle \alpha_1,\ \rho_1,\ v_1,\ \langle \alpha,\ \rho,\ \theta' : \phi([\,], E_2, \dots, E_{i-1}, \theta : T, E_{i+1}, \dots, E_n),\ K' \rangle,\ \Sigma_1 \rangle \\ \to & \langle \alpha,\ \rho,\ \theta' : \phi(v_1, E_2, \dots, E_{i-1}, \theta : T, E_{i+1}, \dots, E_n),\ K',\ \Sigma_1 \rangle \\ \dots & \\ \to & \langle \alpha.i,\ \rho,\ \theta : T,\ \langle \alpha,\ \rho,\ \theta' : \phi(v_1, v_2, \dots, v_{i-1}, [\,], E_{i+1}, \dots, E_n),\ K' \rangle,\ \Sigma_{i-1} \rangle \end{array}$$

Let SC denote the last configuration in this computation. By Corollary 16 (Simulation), the sequence obtained by taking the *Elim* of each configuration in this store computation is an environment computation. Furthermore the *Elim* of each state is consistent and reachable. Also, by the soundness of $\mathscr{A}$, $\mathit{Elim}\ SC \models \mathscr{A}$. Let K denote the store continuation

$$\langle \alpha,\ \rho,\ \theta' : \phi(v_1, v_2, \dots, v_{i-1}, [\,], E_{i+1}, \dots, E_n),\ K' \rangle$$

We need to check that if $\rho(x)$ is live in K, then $x \in \mathscr{L}[\![\theta]\!]$. We proceed by cases on the definition of $\rho(x)$ live in K, assuming in each case that $\rho(x)$ is a location:

(a) There exists a $j \in [1, i-1]$ such that $\rho(x) = v_j$ and v_j is a location. Let $\Sigma(v_j) = \beta : \langle \dots \rangle$. By the soundness of $\mathscr{P}[\![-]\!]$ (in the underlying environment computation) either

(i) $\alpha.j$ is a prefix of β, and therefore $\alpha < \alpha.j \leqslant \beta$. But consistency requires that $\beta < \alpha$. So this case is impossible.

(ii) there exists a y such that $\rho(y) \in \mathscr{P}[\![E_j]\!]$ and $\Sigma(\rho(y)) = \beta : \langle \dots \rangle$. Since $\Sigma(\rho(x)) = \Sigma(\rho(y))$ and $\mathit{Elim}\ SC \models \mathscr{A}$, it follows that $(x, y) \in \mathscr{A}$. Therefore $x \in \mathscr{A} \star \mathscr{P}[\![E_j]\!] \subseteq \mathscr{L}[\![\theta]\!]$.

(b) There exists a $j \in [i+1, n]$ and $y \in \mathrm{dom}(\rho)$ such that $\rho(x) = \rho(y)$ and $y \in \mathrm{fv}(E_j)$. Therefore $x \in \mathscr{A} \star \mathrm{fv}(E_j) \subseteq \mathscr{L}[\![\theta]\!]$.

(c) $\rho(x)$ is live in K'. By the induction hypothesis (applied at the second line in the calculation above), $x \in \mathscr{L}[\![\theta']\!] \subseteq \mathscr{L}[\![\theta]\!]$.

2. The cases for conditionals are similar: The case in which θ appears in the test is just like the evaluation of the first actual parameter of a procedure call. The case in which θ appears in one of the consequents is simpler; the only way in which θ can appear as the expression component of a configuration is by a computation of the form

$$\begin{array}{ll} & \langle \alpha_0,\ \rho_0,\ E_0,\ K_0,\ \Sigma_0\rangle \\ \rightarrow^* & \langle \alpha,\ \rho,\ \theta' : \textbf{if } E \textbf{ then } \theta : T \textbf{ else } E_2,\ K',\ \Sigma\rangle \\ \rightarrow & \langle \alpha.1,\ \rho,\ E,\ \langle \alpha,\ \rho,\ \theta' : \textbf{if } [\ ] \textbf{ then } \theta : T \textbf{ else } E_2,\ K'\rangle,\ \Sigma\rangle \\ \rightarrow^* & \langle \alpha.1,\ \rho,\ \mathit{true},\ \langle \alpha,\ \rho,\ \theta' : \textbf{if } [\ ] \textbf{ then } \theta : T \textbf{ else } E_2,\ K'\rangle,\ \Sigma\rangle \\ \rightarrow & \langle \alpha,\ \rho,\ \theta' : \textbf{if } \mathit{true} \textbf{ then } \theta : T \textbf{ else } E_2,\ K',\ \Sigma\rangle \\ \rightarrow & \langle \alpha.2,\ \rho,\ \theta : T,\ K',\ \Sigma\rangle \end{array}$$

If $\rho(x)$ is live in K', it must be that $x \in \mathscr{L}[\![\theta']\!]$ (applying the induction hypothesis at the second line), hence $x \in \mathscr{L}[\![\theta]\!]$ by L2b.1.

3. If θ is the label of a procedure body F_k, then the only way in which θ can appear is by a computation of the form:

$$\begin{array}{ll} & \langle \alpha_0,\ \rho_0,\ E_0,\ K_0,\ \Sigma_0\rangle \\ \rightarrow^* & \langle \alpha,\ \rho,\ \theta' : f_k(E_1, \ldots, E_n),\ K,\ \Sigma\rangle \\ \rightarrow^* & \langle \alpha,\ \rho,\ \theta' : f_k(v_1, \ldots, v_n),\ K,\ \Sigma'\rangle \\ \rightarrow & \langle \alpha.(n+1),\ \{x_{k:1} = v_1, \ldots, x_{k:n} = v_n\},\ \theta : F_k,\ K,\ \Sigma'\rangle \end{array}$$

Furthermore, applying the induction hypothesis at the next-to-last line, if $\rho(x)$ is live in K, then $x \in \mathscr{L}[\![\theta']\!]$.
Now, by the soundness of $\mathscr{P}[\![-]\!]$, each v_i is either

(a) a basic value
(b) a location with $\alpha.i$ as a prefix, or
(c) an element of $\{\rho(x) \mid x \in \mathscr{P}[\![E_i]\!]\}$.

If v_i is a basic value, then it is not a location live in K. If it is a location with $\alpha.i$ as a prefix, then it cannot be live in K because that would violate consistency. In the last case, there must be some x such that $v_i = \rho(x)$ and $x \in \mathscr{P}[\![E_i]\!]$. But if $v_i = \rho(x)$, then $x \in \mathscr{L}[\![\theta']\!]$ by the induction hypothesis. Hence $\mathscr{P}[\![E_i]\!] \cap \mathscr{L}[\![\theta']\!]$ is non-empty, and therefore $x_{k:i} \in \mathscr{L}[\![\theta]\!]$, as required.

4. The final possibility is that θ is the label of F_0, the body of the program. In this case, θ appears only with the initial continuation K_0, in which no locations are free, so soundness is satisfied trivially. □

9 Proof of main theorem

We now present the proof of our main theorem (Theorem 9). Recall that → indicates reduction using the original program and ⇒ indicates reduction using the transformed program.

Proof

Let $\mathscr{L}[\![-]\!]$ be a sound live variable analysis, and let $\langle \alpha_0, \rho_0, F_0, \mathbf{halt}, \Sigma_0 \rangle$ be an initial configuration. The proof proceeds by induction on n with induction hypothesis: If

$$\langle \alpha_0, \rho_0, F_0, \mathbf{halt}, \Sigma_0 \rangle \rightarrow^n \langle \alpha, \rho, G, K, \Sigma \rangle$$

and

$$\langle \alpha_0, \rho_0, F_0^*, \mathbf{halt}, \Sigma_0 \rangle \Rightarrow^n \langle \alpha', \rho', G', K', \Sigma' \rangle,$$

then

$$\mathit{Elim}\langle \alpha, \rho, G, K, \Sigma \rangle \cong \mathit{Elim}\langle \alpha', \rho', G', K', \Sigma' \rangle$$

$\mathit{Elim}\ F_0^*\ \Sigma \equiv F_0$ so the base case is trivial.

Assume the induction hypothesis for n, and suppose

$$\langle \alpha_0, \rho_0, F_0, \mathbf{halt}, \Sigma_0 \rangle \rightarrow^n \langle \alpha, \rho, G, K, \Sigma \rangle \rightarrow \langle \alpha'', \rho'', G'', K'', \Sigma'' \rangle$$

and

$$\langle \alpha_0, \rho_0, F_0^*, \mathbf{halt}, \Sigma_0 \rangle \Rightarrow^n \langle \alpha', \rho', G', K', \Sigma' \rangle \Rightarrow \langle \alpha''', \rho''', G''', K''', \Sigma''' \rangle$$

Since $\mathit{Elim}\ G\ \Sigma = \mathit{Elim}\ G'\ \Sigma'$, either the same rule is applied in both transitions or else the [UPD-S] rule is applied in the → transition and the [UPD!] rule is applied in the ⇒ transition. By Corollary 16, $\mathit{Elim}\langle \alpha, \rho, G, K, \Sigma \rangle$ is consistent. Therefore $\mathit{Elim}\langle \alpha', \rho', G', K', \Sigma' \rangle$ would be consistent if not for the values of some dead variables, by the definition of congruence. Congruence also implies that $\alpha = \alpha'$, whence $\alpha'' = \alpha'''$.

Suppose the same rule is applied in both transitions. This rule cannot be the [UPD!] rule because there are no destructive updates in the original program. By Lemma 15 (Preservation of Store Consistency), $\mathit{Elim}\langle \alpha'', \rho'', G'', K'', \Sigma'' \rangle$ is consistent and

$$\mathit{Elim}\langle \alpha, \rho, G, K, \Sigma \rangle \rightarrow \mathit{Elim}\langle \alpha'', \rho'', G'', K'', \Sigma'' \rangle$$

By Lemma 3 (Preservation of Congruence),

$$\mathit{Elim}\langle \alpha', \rho', G', K', \Sigma' \rangle \Rightarrow \mathit{Elim}\langle \alpha''', \rho''', G''', K''', \Sigma''' \rangle$$

and

$$\mathit{Elim}\langle \alpha'', \rho'', G'', K'', \Sigma'' \rangle \cong \mathit{Elim}\langle \alpha''', \rho''', G''', K''', \Sigma''' \rangle$$

The above use of ⇒ instead of → is justified by the fact that for all E, Σ, and Σ', $\mathit{Elim}(\theta\!:\!E)\,\Sigma \equiv \mathit{Elim}(\theta\!:\!E^*)\,\Sigma'$.

Suppose the [UPD-S] rule is applied in the → transition and the [UPD!] rule is applied in the ⇒ transition. Then

$$\begin{array}{ll} & \langle \alpha, \rho, G, K, \Sigma \rangle \\ = & \langle \alpha, \rho, \theta\!:\!\mathrm{UPD}(\beta, j, bv'), K, \Sigma \rangle \\ \rightarrow & \langle \alpha, \rho, \alpha, K, \Sigma'' \rangle \end{array}$$

where $\Sigma(\beta) = \beta\!:\!\langle bv_1, \ldots, bv_j, \ldots, bv_n \rangle$ and $\Sigma'' = \Sigma[\alpha \mapsto \alpha\!:\!\langle bv_1, \ldots, bv', \ldots, bv_n \rangle]$. Also

$$\langle \alpha', \rho', G', K', \Sigma' \rangle$$

$$\begin{aligned} &= \langle \alpha,\, \rho',\, \theta\!:\!\mathrm{UPD}(\gamma, j, bv'),\, K',\, \Sigma' \rangle \\ &\Rightarrow \langle \alpha,\, \rho',\, \gamma,\, K',\, \Sigma''' \rangle \end{aligned}$$

where $\Sigma'(\gamma) = \beta\!:\!\langle bv_1, \dots, bv_j, \dots, bv_n \rangle$ and $\Sigma''' = \Sigma'[\gamma \mapsto \alpha\!:\!\langle bv_1, \dots, bv', \dots, bv_n \rangle]$. Clearly $\mathit{Elim}\ \alpha\ \Sigma'' = \mathit{Elim}\ \gamma\ \Sigma'''$. By Definition 8 there exists a variable x such that $\theta\!:\!\mathrm{UPD}(x, E_1, E_2)$ is in the original program and $x \notin \mathscr{L}[\![\theta]\!]$. By Theorem 24, $\rho(x) = \beta$ is not live in K, so there is no occurrence of $\Sigma(\beta) = \beta\!:\!\langle bv_1, \dots, bv_j, \dots, bv_n \rangle$ in $\mathit{Elim}\ K\ \Sigma$ except possibly as the value of some dead variable that is ignored by the congruence relation. If γ were live in K', then $\Sigma(\gamma) = \beta\!:\!\langle bv_1, \dots, bv_j, \dots, bv_n \rangle$ would be live in $\mathit{Elim}\ K'\ \Sigma'$, which would contradict $\mathit{Elim}\ K\ \Sigma \cong \mathit{Elim}\ K'\ \Sigma'$. Hence γ is not live in K'. Furthermore α is not live in K, by Lemma 14. Now we can deduce:

$$\begin{array}{ll} \mathit{Elim}\ K\ \Sigma'' & \\ \cong \mathit{Elim}\ K\ \Sigma & \text{since } \alpha \text{ is not live in } K \\ \cong \mathit{Elim}\ K'\ \Sigma' & \text{from IH} \\ \cong \mathit{Elim}\ K'\ \Sigma''' & \text{since } \gamma \text{ is not live in } K' \quad \square \end{array}$$

10 Computational complexity

For each analysis, we now calculate an upper bound for the worst-case time required to find the least fixed point of its set constraints. Let m be the size of the program's abstract syntax tree, let n_k be the arity of F_k, and let $n = n_1 + \cdots + n_N$ be the total number of local variables in the program. (In a typed language, we would only consider variables whose type is an array type.) We assume that each union or intersection requires time proportional to the size of its universe, as when sets are represented by bitvectors. This implies that each set operation requires $O(n_j)$ time for some j.

The least fixed point of the set constraints for Propagation Analysis will be found in at most n iterations, where each iteration computes $O(m)$ unions of size n_j for some j. The worst-case time for Propagation Analysis is therefore $O(mn^2)$.

The least fixed point of the set constraints for Alias Analysis will be found in at most n iterations, where each iteration computes $O(mn_k^2)$ intersections of size n_j for some j. The cost of merging equivalence classes is less than this, so it can be ignored. The worst-case time for Alias Analysis is therefore $O(mn^4)$.

The least fixed point of the set constraints for Liveness Analysis will be found in less than mn iterations, where each iteration computes at most $2m$ unions or intersections of size n_j for some j. The worst-case time for Liveness Analysis is therefore $O(m^2n^2)$.

Hence the set constraints can be solved in polynomial time using standard techniques. The worst-case asymptotic bounds that we have calculated are comparable to the worst-case bounds for standard flow analyses (Aho *et al.*, 1986). For typical programs, a prototype implementation of the earlier (and less obviously efficient) formulation of these analyses appears to run in near-linear time (Sastry and Clinger, 1994).

http://dx.doi.org/10.1017/S0956796801003938

11 Related work

Our optimization is based on the analysis of Sastry *et al.* (1993). We add to that paper by providing a proof of correctness for the transformation. The present analysis differs from that of Sastry *et al.* (1993) in a number of ways:

The analysis of Sastry *et al.* (1993) was presented as a sequence of abstract interpretations. This framework led to a formulation for which a naive implementation would lead to an exponential algorithm. The algorithm was reduced to polynomial complexity by computing the propagation analysis symbolically. The current formulation builds these symbolic representations into the analyses, so no sophisticated implementation techniques are necessary.

The analysis of Sastry *et al.* (1993) contains two phases not included in this paper: a selects-and-updates analysis and an order-of-evaluation analysis. The order-of-evaluation analysis chooses an order for the evaluation of arguments at each procedure call, seeking to minimize live variables. The selects-and-updates analysis computes quantities that are used in the order-of-evaluation analysis. Such analyses could easily be added, because reordering the evaluation of actual parameters would not change the soundness of either $\mathscr{P}[\![-]\!]$ or $\mathscr{A}$; we chose not to use A-normal form (Flanagan and Felleisen, 1995) in order to make this invariance clear.

Work on destructive update analysis goes back to Hudak and Bloss (1985) and Hudak (1986). These early analyses required exponential or doubly exponential time in the worst case, and were less effective than the combination of an order-of-evaluation analysis with the algorithm considered here (Sastry *et al.*, 1993; Sastry and Clinger, 1994).

Of more recent work, our analysis appears most similar to that of Draghicescu and Purushothaman (1993). Like us, they present an update optimization based on live variable analysis for first-order functional languages with flat arrays. Their analysis works for non-strict languages, however, and has exponential time complexity. As in most work of this kind, they prove the soundness of their analyses but do not prove the correctness of any program transformations.

The style of analysis as constraint-generation is drawn from Palsberg & Schwartzbach (1995), which in turn drew on several previous papers (Jones and Muchnick, 1982; Sestoft, 1988; Jones, 1981; Shivers, 1991). Steckler and Wand (1997) used similar ideas to prove the correctness of a closure-conversion algorithm. Flanagan and Felleisen (1995) used the same set of ideas to eliminate type-checks in Scheme programs. Their proof also used an architecture quite similar to ours.

A competing paradigm for finding safety properties is *abstract interpretation* (Cousot and Cousot, 1977). In this framework, the soundness of analyses like our $\mathscr{P}[\![-]\!]$ may be regarded as the correctness of a second-order collecting interpretation (Nielson, 1985; Schmidt, 1998). Set constraints avoid the complications of collecting interpretations. They are also a more direct generalization of the flow analysis that is familiar to compiler writers (Aho *et al.*, 1986). It does not appear possible to prove the correctness of the program transformation using abstract interpretation of the original program alone, because the store-free semantics of the original functional program cannot express the side effects of the imperative program that results from the transformation.

All the analyses here generate conditional constraints, which can be solved using standard closure algorithms (Aiken *et al.*, 1993; Heintze, 1992; Palsberg and Schwartzbach, 1994).

The idea of using a time stamp to model sharing in a store-free model goes back at least to section V.4 of Clinger's PhD thesis (1981), which constructed process identifiers from process identifiers and local time. This idea was suggested by Carl Hewitt.

12 Variations and extensions

A similar proof could be developed using big-step semantics. In such a proof, our use of non-local references to the structure of the computation, for which we have relied on Theorem 17 (Interpolation Theorem), would be replaced by references to the immediate predecessors of a node. However, the information we have localized in the continuation component would become distributed through the tree.

It is possible to allow arrays in initial configurations by parameterizing the analysis on the alias analysis $\mathscr{A}$, and by admitting any initial configuration that satisfies $\mathscr{A}$. All of our results extend directly to this case.

Sastry and Clinger (1994) extended the analysis of Sastry *et al.* (1993) to parallel computation regimes. It would clearly be desirable to extend our proofs of correctness as well.

Currently, our analysis works only for arrays containing scalar data. It would be desirable to extend this to allow arrays containing compound data, including other arrays. It would also be desirable to remove the restriction to first-order programs by allowing data to reside inside closures.

Acknowledgements

The authors thank Neil Jones for giving us the opportunity to present an earlier version of this work at the January 1997 Atlantique meeting. Michael Ashley and members of the Northeastern University Programming Languages seminar provided useful conversations and feedback.

References

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

Aiken, A., Kozen, D., Vardi, M. and Wimmers, E. (1993) The complexity of set constraints. *Computer Science Logic '93.*

Chuang, T. and Goldberg, B. (1992) A syntactic approach to fixed point computation on finite domains. *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pp. 109–118.

Clinger, W. D. (1981) *Foundations of actor semantics.* PhD thesis, MIT.

Cousot, P. and Cousot, R. (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252.

Draghicescu, M. and Purushothaman, S. (1993) A uniform treatment of order of evaluation and aggregate update. *Theor. Comput. Sci.*, **118**, 231–262.

Flanagan, C. and Felleisen, M. (1995) *Set-based analysis for full scheme and its use in soft-typing*. Technical report COMP TR95-253. Department of Computer Science, Rice University.

Heintze, N. (1992) *Set-based program analysis.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA.

Heintze, N. and Jaffar, J. (1990) A decision procedure for a class of set constraints. *Proc. 5th IEEE Symposium on Logic in Computer Science*, pp. 42–51. IEEE.

Hudak, Paul. (1986) A semantic model of reference counting and its abstraction. *Proc. 1986 ACM Symposium on Lisp and Functional Programming*, pp. 351–363. ACM.

Hudak, P. and Bloss, A. (1985) Avoiding copying in functional and logic programming languages. *Conference record of the 12th ACM Symposium on Principles of Programming Languages*, pp. 300–314. ACM.

Jones, N. D. (1981) Flow analysis of lambda expressions. *International Colloquium on Automata, Languages, and Programming: Lecture Notes in Computer Science 115*, pp. 114–128. Springer-Verlag.

Jones, N. D. and Muchnick, S. S. (1982) A flexible approach to interprocedural data flow analysis and progrms with recursive data structures. *Conference Record of the 9th ACM Symposium on Principles of Programming Languages*, pp. 66–74.

Nielson, F. (1985) Program transformations in a denotational setting. *ACM Trans. Programming Lang. & Syst.*, **7**(3), 359–379.

Palsberg, J. and Schwartzbach, M. I. (1994) *Object-oriented Type Systems.* Wiley.

Palsberg, J. and Schwartzbach, M. I. (1995) Safety analysis versus type inference. *Information & Computation*, **118**(1), 128–141.

Sastry, A. V. S. and Clinger, W. (1994) Parallel destructive updating in strict functional languages. *ACM Conference on Lisp and Functional Programming*, pp. 263–272. (*Lisp Pointers*, **7**(3).)

Sastry, A. V. S., Clinger, W. and Ariola, Z. (1993) Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. *Conference on Functional Programming Languages and Computer Architecture*, pp. 266–275.

Schmidt, D. A. (1998) Data flow analysis is model checking of abstract interpretation. *Proc. 1998 ACM Symposium on Principles of Programming Languages*, pp. 38–48.

Sestoft, P. (1988) *Replacing function parameters by global variables.* MPhil thesis, DIKU, University of Copenhagen.

Shivers, O. (1991) *Control-flow analysis of higher-order languages.* PhD thesis, Carnegie-Mellon University.

Steckler, P. A. and Wand, M. (1997) Lightweight closure conversion. *ACM Trans. Programming Lang. & Syst.*, January, 48–86. (Original version appeared in *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, 1994.)