Set Constraints for Destructive Array Update Optimization*

Mitchell Wand and William D. Clinger

College of Computer Science
Northeastern University
Boston, MA 02115

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 the soundness of these analyses using small-step operational semantics. We have also proved that any sound liveness analysis induces a correct program transformation.

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. As originally presented in [SCA93, SC94], this algorithm's efficiency depended upon a special technique for symbolic computation of a least fixed point [CG92]. The set constraints generated by the reformulated algorithm are easily seen to be data flow inequalities, which can be solved in polynomial time using standard techniques [AKVW93].

We also state theorems that assert the correctness of this optimization. In an advance over previous work, we have proved the correctness of the program transformation that introduces imperative assignments, as well as the correctness of the analyses on which this transformation is based. In this paper we merely state our main theorems, as their proofs are too long to be included here.

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 [SCA93, SC94]. As reported previously, this is a very effective optimization. It was also the first interprocedural update analysis to run in polynomial time.

1.2 Outline

We begin in Section 2 by describing our functional source language and sketching 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.

Sections 5 through 7 complete the story by develop-

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

Figure 1: Syntax of expressions.

ing a live-variable analysis. 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 pattern is the same: 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 state a theorem showing that any solution to these constraints is sound.

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

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 where the free variables $\mathrm{fv}(F_0)$ are a subset of $\{x_{0:i}, \ldots, x_{0:n_0}\}$.

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)

1. *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.

$$
\begin{array}{rcll}
EC & ::= & \langle \textbf{halted}, v \rangle \mid \langle \alpha, \rho, G, K \rangle & \text{configurations} \\[2ex]
\alpha & \in & N^* & \text{computation addresses} \\
 & & & \text{(ordered lexicographically)} \\[2ex]
\rho & \in & \mathit{Var} \to_{\text{fin}} \mathit{Val} & \text{environments} \\[2ex]
v \in \mathit{Val} & ::= & bv \mid lv & \text{denoted values} \\
bv & ::= & \mathit{true} \mid \mathit{false} \mid 0 \mid 1 \mid \ldots & \text{scalar (basic) values} \\
lv \in \mathit{LabVal} & ::= & \alpha{:}\langle bv, \ldots \rangle & \text{labelled arrays of scalar values} \\[4ex]
G & ::= & E \mid v & \text{partially reduced expressions} \\
 & \mid & \theta{:}\, \phi(v_1, \ldots, v_{i-1}, E_i, \ldots, E_n) & \\
 & \mid & \theta{:}\, \textbf{if } v \textbf{ then } E_1 \textbf{ else } E_2 & \\
R & ::= & \theta{:}\, \phi(v_1, \ldots, v_{i-1}, [\,], E_{i+1}, \ldots, E_n) & \text{return contexts} \\
 & \mid & \theta{:}\, \textbf{if } [\,] \textbf{ then } E_1 \textbf{ else } E_2 & \\
K & ::= & \textbf{halt} \mid \langle \alpha, \rho, R, K \rangle & \text{continuations}
\end{array}
$$

Figure 2: Configurations of the environment semantics.

2. *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 9.

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* (halts 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 continuations and then for configurations:*

- $\textbf{halt} \cong \textbf{halt}$ *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. *for all $j \in [i+1, n]$, for all $x \in \mathrm{fv}(E_j)$, $\rho(x) = \rho'(x)$, and*
    2. $K \cong K'$
- $\langle \alpha, \rho, \theta{:}\, \textbf{if } [\,] \textbf{ then } E_1 \textbf{ else } E_2, K \rangle$ $\cong \langle \alpha, \rho', \theta{:}\, \textbf{if } [\,] \textbf{ then } E_1 \textbf{ else } E_2, K' \rangle$ *iff*
    1. *for all $j \in \{1, 2\}$, for all $x \in \mathrm{fv}(E_j)$, $\rho(x) = \rho'(x)$, and*
    2. $K \cong K'$

$$\langle \alpha,\ \rho,\ \theta{:}\, x,\ K\rangle \to \langle \alpha,\ \rho,\ \rho(x),\ K\rangle \qquad [\textsc{fetch}]$$

$$\begin{array}{l}\langle \alpha,\ \rho,\ \theta{:}\, \phi(v_1, \ldots, v_{i-1}, E_i, E_{i+1}, \ldots, E_n),\ K\rangle \\ \quad \to \quad \langle \alpha.i,\ \rho,\ E_i,\ \langle \alpha,\ \rho,\ \theta{:}\, \phi(v_1, \ldots, v_{i-1}, [\,], E_{i+1|,\ldots,E_n)} K,\ \rangle\rangle\end{array} \qquad [\textsc{push}]$$

$$\begin{array}{l}\langle \alpha,\ \rho,\ v,\ \langle \alpha',\ \rho',\ R,\ K\rangle\rangle \\ \quad \to \quad \langle \alpha',\ \rho',\ R[v],\ K\rangle\end{array} \qquad [\textsc{return}]$$

$$\langle \alpha,\ \rho,\ v,\ \mathbf{halt}\rangle \to \langle \mathbf{halted},\ v\rangle \qquad [\textsc{halt}]$$

$$\begin{array}{l}\langle \alpha,\ \rho,\ \theta{:}\, f_k(v_1, \ldots, v_n),\ K\rangle \\ \quad \to \quad \langle \alpha.(n+1),\ \{x_{k:1} = v_1, \ldots, x_{k:n} = v_n\},\ F_k,\ K\rangle\end{array} \qquad [\textsc{call}]$$

$$\begin{array}{l}\langle \alpha,\ \rho,\ \theta{:}\, p(bv_1, \ldots, bv_n),\ K\rangle \\ \quad \to \quad \langle \alpha.(n+1),\ \rho,\ bv',\ K\rangle \\ \qquad\quad \text{if } p^*(bv_1, \ldots, bv_n) = bv'\end{array} \qquad [\textsc{primop}]$$

$$\begin{array}{l}\langle \alpha,\ \rho,\ \theta{:}\, \mathrm{NEW}(n, bv),\ K\rangle \\ \quad \to \quad \langle \alpha,\ \rho,\ \alpha{:}\langle bv, \ldots, bv\rangle,\ K\rangle \\ \qquad\quad \text{NEW creates a new array filled with } n \text{ copies of } bv\end{array} \qquad [\textsc{new-e}]$$

$$\begin{array}{l}\langle \alpha,\ \rho,\ \theta{:}\, \mathrm{REF}(\beta{:}\langle bv_1, \ldots\rangle, j),\ K\rangle \\ \quad \to \quad \langle \alpha,\ \rho,\ bv_j,\ K\rangle\end{array} \qquad [\textsc{ref-e}]$$

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

$$\begin{array}{l}\langle \alpha,\ \rho,\ \theta{:}\, \mathbf{if}\ E_0\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2,\ K\rangle \\ \quad \to \quad \langle \alpha.1,\ \rho,\ E_0,\ \langle \alpha,\ \rho,\ \theta{:}\, \mathbf{if}\ [\,]\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2,\ K\rangle\rangle\end{array} \qquad [\textsc{push-test}]$$

$$\begin{array}{l}\langle \alpha,\ \rho,\ \theta{:}\, \mathbf{if}\ \mathit{true}\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2,\ K\rangle \\ \quad \to \quad \langle \alpha.2,\ \rho,\ E_1,\ K\rangle\end{array} \qquad [\textsc{branch-true}]$$

$$\begin{array}{l}\langle \alpha,\ \rho,\ \theta{:}\, \mathbf{if}\ \mathit{false}\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2,\ K\rangle \\ \quad \to \quad \langle \alpha.2,\ \rho,\ E_2,\ K\rangle\end{array} \qquad [\textsc{branch-false}]$$

Figure 3: Reduction rules for the environment semantics.

- $\langle \mathbf{halted},\, v\rangle \cong \langle \mathbf{halted},\, v'\rangle$ *iff* $v = v'$.
- $\langle \alpha,\, \rho,\, G,\, K\rangle \cong \langle \alpha,\, \rho',\, G,\, K'\rangle$ *iff*

    1. *for all* $x \in \mathrm{fv}(G)$, $\rho(x) = \rho'(x)$, *and*
    2. $K \cong K'$

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:

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

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 use α or β when we rely on this fact, and 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.

$$\begin{aligned}
&\mathit{Elim}\,\langle \alpha,\, \rho,\, G,\, K,\, \Sigma\rangle \\
&\quad = \langle \alpha,\, \mathit{Elim}\ \rho\,\Sigma,\, \mathit{Elim}\ G\,\Sigma,\, \mathit{Elim}\ K\,\Sigma\rangle
\end{aligned}$$

$$(\mathit{Elim}\ \rho\,\Sigma)(x) = \mathit{Elim}\,(\rho(x))\Sigma$$

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

Figure 5: Store elimination (partial definition).

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 3 (Initial, Reachable (store)) *A store configuration is initial iff 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 Lemma 4 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.

Lemma 4 (Simulation) *Assume the original program has no occurrences of* UPD!. *Let* SC_0 *be an initial store configuration, and let* $SC_0 \to^n SC$. *Then* $\mathit{Elim}\,SC_0 \to^n \mathit{Elim}\,SC$.

4 The transformation

The goal of our analysis is to identify variables that denote locations that are certainly not live in a store con-

$$\langle \alpha,\, \rho,\, \mathrm{NEW}(n, bv),\, K,\, \Sigma \rangle \;\to\; \langle \alpha,\, \rho,\, \alpha,\, K,\, \Sigma[\alpha = \alpha{:}\langle bv, \ldots, bv\rangle] \rangle \qquad [\textsc{new-s}]$$

new array allocated at α, filled with n copies of bv

$$\langle \alpha,\, \rho,\, \mathrm{REF}(\beta, j),\, K,\, \Sigma \rangle \;\to\; \langle \alpha,\, \rho,\, bv_j,\, K,\, \Sigma \rangle \qquad [\textsc{ref-s}]$$

if $\Sigma(\beta) = \gamma{:}\langle bv_1, \ldots, bv_n\rangle$

$$\langle \alpha,\, \rho,\, \mathrm{UPD}(\beta, j, bv'),\, K,\, \Sigma \rangle \;\to\; \langle \alpha,\, \rho,\, \alpha,\, K,\, \Sigma[\alpha = \alpha{:}\langle bv_1, \ldots, bv', \ldots, bv_n\rangle] \rangle \qquad [\textsc{upd-s}]$$

where $\Sigma(\beta) = \beta'{:}\langle bv_1, \ldots, bv_j, \ldots, bv_n\rangle$
UPD produces a new copy of the modified array, allocated at the fresh location α.

$$\langle \alpha,\, \rho,\, \mathrm{UPD!}(\beta, j, bv'),\, K,\, \Sigma \rangle \;\to\; \langle \alpha,\, \rho,\, \beta,\, K,\, \Sigma[\beta = \alpha{:}\langle bv_1, \ldots, bv', \ldots, bv_n\rangle] \rangle \qquad [\textsc{upd!}]$$

where $\Sigma(\beta) = \beta'{:}\langle bv_1, \ldots, bv_j, \ldots, bv_n\rangle$
UPD! destructively modifies the array.

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

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

Figure 4: Reduction rules for the store semantics.

tinuation. 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!.

Recall that a store 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)

- *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 $\mathcal{L}[\![-]\!]$.

Definition 6 (Live Variable Analysis)
A live variable analysis *$\mathcal{L}[\![-]\!]$ is a map from expression labels θ to sets of variables. $\mathcal{L}[\![-]\!]$ is* sound *iff for each label θ, $\mathcal{L}[\![\theta]\!]$ is a set of variables such that for all reachable configurations of the form $\langle \alpha,\, \rho,\, \theta{:}T,\, K,\, \Sigma\rangle$, $\rho(x)$ live in K implies $x \in \mathcal{L}[\![\theta]\!]$.*

The idea behind the transformation is that if $\theta{:}\,\mathrm{UPD}(x, E_1, E_2)$ is an update in the program, and $x \notin \mathcal{L}[\![\theta]\!]$, 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 can formulate the transformation as follows:

Definition 7 (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 $x \notin \mathcal{L}[\![\theta]\!]$. 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 $\to$ to indicate reduction using the original program and $\Rightarrow$ to indicate reduction using the transformed program.

We now state our main theorem, which establishes the correctness of the transformation.

Theorem 8 (Main Theorem) *Let $\mathcal{L}[\![-]\!]$ 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

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

Theorem 9 (Correctness of Transformation) *If $\mathcal{L}[\![-]\!]$ 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

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

5 Propagation analysis

We will not be done, of course, until we show how to find a sound live-variable analysis $\mathcal{L}[\![-]\!]$. 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 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 one of the places where we take advantage of the instrumentation in our environment semantics.

The soundness of a propagation analysis 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 can be formalized using the time stamps), every expression is a subexpression of the original program, and there are no unbound variables; lack of space prevents us from giving the formal definition. A configuration of the store semantics is consistent iff its store elimination is consistent.

Consistency is an invariant of reduction in the environment semantics, but is not necessarily preserved by reduction in the store semantics. Part of the proof of Theorem 8 therefore involves showing that, for both the original and transformed program, every reachable configuration is consistent.

P1. $\mathcal{P}[\![x]\!] \supseteq \{x\}$.

P2. $\mathcal{P}[\![\mathbf{if}\ E_0\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2]\!] \supseteq \mathcal{P}[\![E_1]\!] \cup \mathcal{P}[\![E_2]\!]$.

P3. If $x_{k:i} \in \mathcal{P}[\![F_k]\!]$, then $\mathcal{P}[\![f_k(E_1,\ldots,E_n)]\!] \supseteq \mathcal{P}[\![E_i]\!]$

Figure 6: Set constraints for $\mathcal{P}[\![-]\!]$.

Definition 10 (Propagation Analysis) *A propagation analysis $\mathcal{P}[\![-]\!]$ is a map from expressions to sets of variables.*

A propagation analysis $\mathcal{P}[\![-]\!]$ is sound *iff whenever $\langle \alpha,\ \rho,\ E,\ K \rangle$ is consistent and $\langle \alpha,\ \rho,\ E,\ K \rangle \rightarrow^* \langle \alpha',\ \rho',\ \beta{:}\langle bv \ldots \rangle,\ K \rangle$ then either*

1. *$\beta{:}\langle bv \ldots \rangle \in \{\rho(x) \mid x \in \mathcal{P}[\![E]\!]\}$ 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 $\mathcal{P}[\![E]\!]$, or it was created during the execution of E.

To find the sets $\mathcal{P}[\![-]\!]$, we solve the constraints in Figure 6. Note that there is no constraint generated for any call to a primitive $g(E_1,\ldots,E_n)$, so in the smallest solution $\mathcal{P}[\![g(E_1,\ldots,E_n)]\!]$ 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 11 (Correctness of $\mathcal{P}[\![-]\!]$) *Any solution to the constraints P1–P3 satisfies the soundness conditions for $\mathcal{P}[\![-]\!]$.*

6 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 12 (Alias Analysis) *An* alias set *$\mathcal{A}$ is a subset of $\mathit{Var} \times \mathit{Var}$. For $S \subseteq \mathit{Var}$, define $\mathcal{A} \star S = \{x \mid (x,y) \in \mathcal{A} \wedge y \in S\}$.*

Each alias set $\mathcal{A}$ induces a proposition $\langle \alpha,\ \rho,\ G,\ K \rangle \models \mathcal{A}$ on configurations of the environment semantics. Again, lack of space prevents us from giving a formal definition. Informally, however,

A1. $\mathcal{A}$ is an equivalence relation.

A2. for each call $f_k(E_1, \ldots, E_n)$ that occurs in the program, if

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

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

Figure 7: Set constraints for $\mathcal{A}$.

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 $(x, y) \in \mathcal{A}$; 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 $(x, y) \in \mathcal{A}$. We use these propositions to characterize the soundness of $\mathcal{A}$:

Definition 13 (Soundness of $\mathcal{A}$)
$\mathcal{A}$ is a sound alias analysis iff $\langle \alpha,\ \rho,\ G,\ K\rangle \models \mathcal{A}$ for every reachable configuration $\langle \alpha,\ \rho,\ G,\ K\rangle$.

The constraints for $\mathcal{A}$ are given in Figure 7. Again the smallest solution can be found by iterating to find the least fixed point.

Theorem 14 (Correctness of $\mathcal{A}$) *If $\mathcal{P}[\![-]\!]$ is a sound propagation analysis and $\mathcal{A}$ satisfies the constraints A1–A2, then $\mathcal{A}$ is a sound alias analysis.*

7 Live variable analysis

With these two analyses in hand, we can now proceed to the live variable analysis. By Definition 6, which characterizes the soundness of a live variable analysis, our goal is to find sets $\mathcal{L}[\![\theta]\!]$ such that for every reachable state of the form $\langle \alpha,\ \rho,\ \theta{:}T,\ K,\ \Sigma\rangle$,

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

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

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

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:

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

L2a. If θ occurs in the context

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

then

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

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

L2b.1 $\mathcal{L}[\![\theta']\!] \subseteq \mathcal{L}[\![\theta]\!]$.

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,

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

Figure 8: Set constraints for $\mathcal{L}[\![-]\!]$.

1. it might be one of the v_j's, for $1 \leq j < i$;
2. it might be the value of a variable free in one of the E_j's (for $i < j \leq n$);
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 $(1 \leq j < i)$;
2. it might be (or be aliased to) a variable that is free in one of the E_j's $(i < j \leq n)$;
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 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 15 (Correctness of $\mathcal{L}[\![-]\!]$) *If $\mathcal{P}[\![-]\!]$ is a sound propagation analysis, $\mathcal{A}$ is a sound alias analysis, and $\mathcal{L}[\![-]\!]$ satisfies constraints L1–L3, then $\mathcal{L}[\![-]\!]$ is sound.*

8 Related work

The analysis of [SCA93] 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 $\mathcal{P}[\![-]\!]$ or $\mathcal{A}$; we chose not to use A-normal form [FF95] in order to make this invariance clear.

Work on destructive update analysis goes back to Hudak and Bloss [HB85] and Hudak [Hud86]. A detailed comparison with Hudak's work appears in [SCA93].

Of more recent work, our analysis appears most similar to that of Draghicescu and Purushothaman [DP93]. 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 [PS95], which in turn drew on [JM82, Ses88, Jon81, Shi91]. [SW97] used similar ideas to prove the correctness of a closure-conversion algorithm. Felleisen and Flanagan [FF95] 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* [CC77]. In this framework, the soundness of analyses like our $\mathcal{P}[\![-]\!]$ may be regarded as the correctness of a second-order collecting interpretation [Nie85, Sch98]. 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 [ASU86].

All the analyses here generate conditional constraints, which can be solved using standard closure algorithms like those in [AKVW93, Hei92, PS94].

9 Variations and extensions

We have also developed a separate proof of the same results using big-step semantics throughout. We hope to report on this elsewhere.

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

[SC94] extends the analysis of [SCA93] 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 one of us (Wand) the opportunity to present 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

[AKVW93] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In *Computer Science Logic '93*, Swansea, Wales, September 1993.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CG92] T. Chuang and B. Goldberg. A syntactic approach to fixed point computation on finite domains. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 109–118, 1992.

[DP93] M. Draghicescu and S. Purushothaman. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118:231–262, 1993.

[FF95] Cormac Flanagan and Matthias Felleisen. Set-based analysis for full scheme and its use in soft-typing. Technical Report COMP TR95-253, Department of Computer Science, Rice University, October 1995.

[HB85] Paul Hudak and Adrienne Bloss. Avoiding copying in functional and logic programming languages. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 300–314. ACM, 1985.

[Hei92] Nevin Heintze. *Set-Based Program Analysis*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1992.

[Hud86] Paul Hudak. A semantic model of reference counting and its abstraction. In *Proc. 1986 ACM Symposium on Lisp and Functional Programming*, pages 351–363. ACM, 1986.

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and progrms with recursive data structures. In *Conf. Rec. 9th ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Jon81] Neil D. Jones. Flow analysis of lambda expressions. In *International Colloquium on Automata, Languages, and Programming*, 1981.

[Nie85] Flemming Nielson. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, July 1985.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley Professional Computing, Chichester, 1994.

[PS95] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[SC94] A.V.S. Sastry and William Clinger. Parallel destructive updating in strict functional languages. In *ACM Conference on Lisp and Functional Programming*, Lisp Pointers 7(3), pages 263–272, 1994.

[SCA93] A.V.S. Sastry, William Clinger, and Zena Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 266–275, 1993.

[Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proc. 1998 ACM Symposium on Principles of Programming Languages*, pages 38–48, 1998.

[Ses88] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, October 1988.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

[SW97] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, pages 48–86, January 1997. Original version appeared in Proceedings 21st Annual ACM Symposium on Principles of Programming Languages, 1994.