The Higher-order Aggregate Update Problem

Christos Dimoulas and Mitchell Wand

College of Computer and Information Science, Northeastern University,
360 Huntington Avenue, Boston, Massachusetts 02115
{chrdimo,wand}@ccs.neu.edu

Abstract. We present a multi-pass interprocedural analysis and transformation for the functional aggregate update problem. Our solution handles untyped programs, including unrestricted closures and nested arrays. Also, it can handle programs that contain a mix of functional and destructive updates. Correctness of all the analyses and of the transformation itself is proved.

1 Introduction

The update of aggregate data structures like arrays is expensive in a functional language because it involves copying the whole data structure. Naively adding destructive update to a functional language does not solve the problem, because the combination loses the compositional properties of functional languages.

A series of papers [1, 4, 5, 7–9, 12] have pursued the idea of destructive update transformation: 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 this paper we present and prove the correctness of an algorithm for destructive update transformation that allows arrays to be nested in arrays and stored in closures. It allows destructive updates and functional updates to coexist in the source program if desired. Furthermore, it does not rely on any type information from the underlying program. To our knowledge, this is the first algorithm with all these features.

The transformation is based on a multi-pass inter-procedural program analysis. We first perform a control-flow analysis, which is used to construct a reachability analysis. We then perform a liveness analysis. The results of these analyses are combined to obtain a live variable analysis.

In section 2 we present a simple example to demonstrate the problem and show the use of our method. In section 3, we give the syntax and the semantics of our language. In section 4, we describe the architecture of our framework, the intermediate layers of analyses and their properties. In section 5 we define the transformation and sketch its correctness proof. Section 6 reviews previous research results in the field.

2 Examples

Consider the following program where each expression is marked with a unique label, the $\texttt{NEW}(n, v)$ operator creates a new array with n cells of value v and the $\texttt{UPD}(l, i, v)$ operator functionally updates the i-th cell of the array l with the new value v.

$^{0}(\,^{1}\lambda\texttt{x}.^{2}\texttt{UPD}(^{3}\texttt{x}, ^{4}\texttt{1}, ^{5}\texttt{x})$
$\quad{}^{6}(\,^{7}\lambda\texttt{y}.^{8}\texttt{UPD}(^{9}\texttt{y}, ^{10}\texttt{1}, ^{11}\texttt{444})$
$\quad\quad{}^{12}(\,^{13}\lambda\texttt{f}.^{14}(^{15}\texttt{f}\ ^{16}\texttt{333})$
$\quad\quad\quad{}^{17}\lambda\texttt{z}.^{18}\texttt{NEW}(^{19}\texttt{4}, ^{20}\texttt{z}))))$

By constructing the control-flow graph of the program, we observe that the UPD-expression with label 8 is applied on l_1 which is created by the NEW-expression. At the time of the evaluation of the UPD-expression, l_1 is is not reachable from any closures or arrays in the continuation of the UPD-expression. Also, l_1 is not reachable from the new array l_2 that the UPD-operation will return to the program. Thus l_1 is not live after the execution of the UPD-operator. So, replacing the functional update with a destructive one does not change the semantics of the program.

The UPD-expression with label 2 is applied on variable y which is bound to l_2. The result of the UPD-operation l_3 contains l_2. Thus l_2 is live after the evaluation of the UPD-expression and the functional update cannot be replaced by a destructive one. If the UPD-expression was replaced by a destructive update then the original and the transformed program would not agree on the contents of l_2.

From the above, we can conclude that the original program can be transformed to the following one without changing its semantics:

$^{0}(\,^{1}\lambda\texttt{x}.^{2}\texttt{UPD}(^{3}\texttt{x}, ^{4}\texttt{1}, ^{5}\texttt{x})$
$\quad{}^{6}(\,^{7}\lambda\texttt{y}.^{8}\texttt{UPD!}(^{9}\texttt{y}, ^{10}\texttt{1}, ^{11}\texttt{444})$
$\quad\quad{}^{12}(\,^{13}\lambda\texttt{f}.^{14}(^{15}\texttt{f}\ ^{16}\texttt{333})$
$\quad\quad\quad{}^{17}\lambda\texttt{z}.^{18}\texttt{NEW}(^{19}\texttt{4}, ^{20}\texttt{z}))))$

Each time we replace a functional update with a destructive update we avoid copying the array, making our programs more efficient in terms of time. Our framework aims to provide a method that can be used to detect such plausible replacement points in programs by statically predicting if a program variable is live or not.

3 The Language

Our language is a variant of the call-by-value untyped lambda calculus with operators for array manipulation.

Every expression and value comes with a unique label θ. Values v are either basic values $\texttt{c}$, function closures $(\lambda\texttt{x}.E, \rho)$, or memory locations l of arrays of values. Expressions include conditionals, primitive operators p, and array operators g. See figure 1 for details. Our language is untyped, so it can express recursive procedures. Our analysis would of course work if the language were restricted to a typed subset.

$$
\begin{array}{rcl}
E & ::= & {}^{\theta}T \\
\\
T & ::= & \mathtt{x} \mid \mathtt{c} \mid l \mid \lambda\mathtt{x}.E \mid (E_1\ E_2) \mid g(E_1,\ldots) \mid p(E_1,\ldots) \\
& \mid & \mathtt{if}\ E_0\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2
\end{array}
$$

where $\mathtt{x} \in \mathit{Var}$, $\theta \in \mathit{Lab}$, $\mathtt{c} \in \mathit{Scalar}$, $p \in \mathit{Prim}$, $g \in \{\mathtt{UPD}, \mathtt{UPD!}, \mathtt{NEW}, \mathtt{REF}\}$.

Fig. 1. Language Syntax.

$$
\begin{array}{rcl}
S & ::= & \langle \mathit{halted}, v, \Sigma\rangle \mid \langle \alpha, \rho, G, K, \Sigma\rangle \\
\\
G & ::= & E \mid v \\
\\
R & ::= & {}^{\theta}(E\ {}^{\theta}[\,]) \mid {}^{\theta}({}^{\theta}[\,]\ v) \\
& \mid & {}^{\theta}g(v_1,\ldots,v_{i-1},{}^{\theta}[\,],E_{i+1},\ldots,E_n) \mid {}^{\theta}p(v_1,\ldots,v_{i-1},{}^{\theta}[\,],E_{i+1},\ldots,E_n) \\
& \mid & {}^{\theta}\mathtt{if}\ {}^{\theta}[\,]\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 \\
\\
K & ::= & \mathit{halt} \mid \langle \alpha, \rho, R, K\rangle \\
\\
v \in V & ::= & {}^{\theta}\mathtt{c} \mid {}^{\theta}l \mid ({}^{\theta}\lambda\mathtt{x}.E, \rho) \\
\\
A & ::= & {}^{\alpha.\theta}\langle v_1,\ldots,v_n\rangle \\
\\
\mathit{Loc} & ::= & \alpha.\theta
\end{array}
$$

where $\alpha, \theta \in N^*$, $l \in \mathit{Loc}$, $\rho \in \mathit{Var} \rightarrow_{\mathit{fin}} V$, $\Sigma \in \mathit{Loc} \rightarrow_{\mathit{fin}} A$.

Fig. 2. Machine Configurations.

We use small-step operational semantics with environments ρ, continuations K, and stores Σ [2]. The configurations of our machine and the continuation frames also include a structured computational address that serves as a time stamp [12]. Time stamp $\alpha.i$ marks the beginning of the evaluation of the i-th subexpression of the expression being evaluated at time stamp α. The body of an n-ary procedure is evaluated at time $\alpha.(n+1)$. When a `NEW` operator is executed, a new location is created using the time stamp. The new location is added in the store domain and points to an array containing the specified values. The new array has a label that is equal to the computational address of the new location. This label has two parts: a dynamic one originating from the time-stamp and the label of the `NEW` expression. The last one is also the label of the newly created location, which is returned as the result of the operator. The `REF` operator is also straightforward. The two update operators `UPD` and `UPD!` are the heart of

$$\langle \alpha, \rho, {}^{\theta} g({}^{\theta_1} T_1, {}^{\theta_2} T_2, \ldots, {}^{\theta_n} T_n), K, \Sigma \rangle$$
$$\rightarrow \langle \alpha.1, \rho, {}^{\theta_1} T_1, \langle \alpha, \rho, {}^{\theta} g({}^{\theta_1}[\,], {}^{\theta_2} T_2, \ldots, {}^{\theta_n} T_n), K \rangle, \Sigma \rangle$$

$$\langle \alpha.i, \rho_i, v_i, \langle \alpha, \rho, {}^{\theta} g(v_1, \ldots, v_{i-1}, {}^{\theta_i}[\,], {}^{\theta_{i+1}} T_{i+1}, {}^{\theta_{i+2}} T_{i+2}, \ldots, {}^{\theta_n} T_n), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha.(i+1), \rho, {}^{\theta_{i+1}} T_{i+1}, \langle \alpha, \rho, {}^{\theta} g(v_1, \ldots, v_{i-1}, v_i, {}^{\theta_{i+1}}[\,], {}^{\theta_{i+2}} T_{i+2}, \ldots, {}^{\theta_n} T_n), K \rangle, \Sigma \rangle$$

$$\langle \alpha.2, \rho_2, v, \langle \alpha, \rho, {}^{\theta}\texttt{NEW}({}^{\theta_n} n, {}^{\theta_v}[\,]), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha, \rho, {}^{\theta}\alpha.\theta, K, \Sigma[\alpha.\theta \rightarrow {}^{\alpha.\theta}\langle v, \ldots, v \rangle] \rangle$$

$$\langle \alpha.2, \rho_2, {}^{\theta_j} j, \langle \alpha, \rho, {}^{\theta}\texttt{REF}({}^{\theta_l} l, {}^{\theta_j}[\,]), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha, \rho, v_j, K, \Sigma \rangle$$
$$\text{where } \Sigma(l) = {}^{\alpha'.\theta_l}\langle v_1, \ldots, v_n \rangle$$

$$\langle \alpha.3, \rho_3, v, \langle \alpha, \rho, {}^{\theta}\texttt{UPD}({}^{\theta_l} l, {}^{\theta_j} j, {}^{\theta_v}[\,]), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha, \rho, {}^{\theta}\alpha.\theta, K, \Sigma[\alpha.\theta \rightarrow {}^{\alpha.\theta}\langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n \rangle] \rangle$$
$$\text{where } \Sigma(l) = {}^{\alpha'.\theta_l}\langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n \rangle$$

$$\langle \alpha.3, \rho_3, v, \langle \alpha, \rho, {}^{\theta}\texttt{UPD!}({}^{\theta_l} l, {}^{\theta_j} j, {}^{\theta_v}[\,]), K \rangle, \Sigma \rangle$$
$$\rightarrow \langle \alpha, \rho, {}^{\theta_l} l, K, \Sigma[l \rightarrow {}^{\alpha.\theta_l}\langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n \rangle] \rangle$$
$$\text{where } \Sigma(l) = {}^{\alpha'.\theta_l}\langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n \rangle$$

Fig. 3. Machine Reduction Rules.

our problem. The `UPD` primitive is similar to the `NEW` primitive and creates a new location and a new array. The only difference is that the content of the new array is the updated content of the old array. The `UPD!` primitive does not create a new location. It updates the contents of the array to which the location points and returns the location and its label unchanged. But the label of the array that the location points to changes: the dynamic part of the label is modified to match the time-stamp of the machine state. This way, we can discriminate between two destructive updates to the same array. Figures 2 and 3 present the details of the behavior of the most interesting operators of our language. The rest follow the standard semantics of left-to-right call-by-value evaluation.

Throughout the rest of this paper, we work within a finite universe of expressions $\mathcal{U}$ closed under subexpressions and containing the initial program E_0. The initial configuration $\langle \epsilon, \emptyset, E_0, \mathit{halt}, \emptyset \rangle$ consists of the initial program in the empty environment and store. *Reachable* configurations are those reachable from this initial configuration. Note that if $E_0 \in \mathcal{U}$, then in any reachable configuration, every expression that appears in the configuration will be drawn from $\mathcal{U}$.

4 The Analysis

Our goal is to identify expressions of the form $\texttt{UPD}(\texttt{x}, E_1, E_2)$ and replace them with $\texttt{UPD!}(\texttt{x}, E_1, E_2)$, if we can prove that this does not affect the semantics of the program. A sufficient correctness condition is that there is no alias to `x` that

is subsequently accessible from other parts of the program. The existence of higher-order functions and nested arrays in our language implies that closures and arrays can hide aliases of locations from the context. For this purpose we use a reachability analysis, which tracks how variables, closures and locations are connected with each other. To build the reachability analysis, we begin with a flow analysis [10].

4.1 The Control Flow Analysis

0-CFA is an analysis for constructing the flow graph of a program with higher order functions based on standard abstract interpretation techniques. Each expression is assigned a unique label. These labels are used as abstract values $\hat{V}$.

The result of the 0-CFA analysis is a function ϕ from labels of expressions and variables to sets of abstract values, i.e. labels of the possible results and bindings. However, the existence of stores demands the extension of the analysis to predict the possible contents of each location in the store. We accomplish that by developing another prediction function σ that describes the shape of the store.

Definition 1 (Store Shape Analysis). *A store shape analysis is a map* σ*:* $\mathrm{Lab} \rightarrow_{fin} \mathcal{P}(\hat{V})$*, mapping a label for an array to a set of abstract values.*
We say σ *describes a store* Σ*,* $\sigma \models \Sigma$*, iff* $\forall l \in dom(\Sigma).\ (\Sigma(l) = {}^{\alpha.\theta}\langle v_1, \ldots, v_n\rangle \implies \forall 0 \leq i \leq n.\ lab(v_i) \in \sigma(\theta))$.

Definition 2 (Control Flow Analysis). *A control flow analysis is a map* ϕ*:* $(\mathrm{Lab} \cup \mathrm{Var}) \rightarrow_{fin} \mathcal{P}(\hat{V})$*, from labels and variables to a set of abstract values.*
We say ϕ *describes an environment* ρ*,* $\phi \models \rho$*, iff* $\forall \mathtt{x} \in dom(\rho).\ lab(\rho(\mathtt{x})) \in \phi(\mathtt{x}) \wedge (\rho(\mathtt{x}) = ({}^{\theta}\lambda \mathtt{y}.E, \rho') \implies \phi \models \rho')$
We say $\langle \phi, \sigma\rangle$ *describes an expression* ${}^{\theta}T \in \mathcal{U}$ *,* $\langle \phi, \sigma\rangle \models {}^{\theta}T$*, iff for all* ρ*,* Σ*, if* $\phi \models \rho$*,* $\sigma \models \Sigma$*,* $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle$ *is a reachable configuration and* $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle$ *, then*

1. $lab(v) \in \phi(\theta)$
2. *if* $v = ({}^{\theta'}\lambda \mathtt{x}.E, \rho'')$*, then* $\phi \models \rho''$
3. $\phi \models \rho'$
4. $\sigma \models \Sigma'$

We say $\langle \phi, \sigma\rangle$ *is sound for* $\mathcal{U}$ *iff* $\forall\, E \in \mathcal{U},\ \langle \phi, \sigma\rangle \models E$.

Note that in this definition, in the reduction $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle$, K is the same in both sides. This reduction represents the evaluation of ${}^{\theta}T$ to the value v.

We can find a sound analysis for a universe of expression $\mathcal{U}$ by solving the set constraints $\mathcal{C}[\![\mathcal{U}]\!]$ presented in figure 4. These constraints are all Horn clauses, and so are solvable by standard techniques. The most interesting are the constraints that apply to the `UPD`, `UPD!` operators. Given a `UPD` expression ${}^{\theta}\mathtt{UPD}({}^{\theta_a}T_a, {}^{\theta_j}T_j, {}^{\theta_v}T_v)$, if $\theta' \in \phi(\theta_a)$, then the new array contains all the values

$$\frac{{}^{\theta}\lambda \mathtt{x}.E \in \mathcal{U}}{(\theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(a) \quad \frac{{}^{\theta}l \in \mathcal{U}}{(\theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(b) \quad \frac{{}^{\theta}\mathtt{c} \in \mathcal{U}}{(\theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(c)$$

$$\frac{{}^{\theta}\mathtt{x} \in \mathcal{U}}{(\phi(\mathtt{x}) \subseteq \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(d)$$

$$\frac{{}^{\theta}({}^{\theta_1}T_1 \ {}^{\theta_2}T_2) \in \mathcal{U} \qquad {}^{\theta'}\lambda \mathtt{x}.{}^{\theta_\lambda}T_\lambda \in \mathcal{U}}{\begin{array}{l}(\theta' \in \phi(\theta_1) \implies (\phi(\theta_2) \subseteq \phi(\mathtt{x}))) \in \mathcal{C}[\mathcal{U}] \\ (\theta' \in \phi(\theta_1) \implies (\phi(\theta_\lambda) \subseteq \phi(\theta))) \in \mathcal{C}[\mathcal{U}]\end{array}}(e)$$

$$\frac{{}^{\theta}\mathtt{if}\ {}^{\theta_0}T_0\ \mathtt{then}\ {}^{\theta_1}T_1\ \mathtt{else}\ {}^{\theta_2}T_2 \in \mathcal{U}}{((\phi(\theta_1) \cup \phi(\theta_2)) \subseteq \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(f) \quad \frac{{}^{\theta}p({}^{\theta_1}T_1, \ldots, {}^{\theta_n}T_n) \in \mathcal{U}}{(\theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(g)$$

$$\frac{{}^{\theta}\mathtt{NEW}({}^{\theta_n}T_n, {}^{\theta_v}T_v) \in \mathcal{U}}{((\phi(\theta_v) \subseteq \sigma(\theta) \wedge \theta \in \phi(\theta)) \in \mathcal{C}[\mathcal{U}]}(h)$$

$$\frac{{}^{\theta}\mathtt{REF}({}^{\theta_a}T_a, {}^{\theta_a}T_a) \in \mathcal{U}}{(\theta' \in \phi(\theta_a) \Rightarrow (\sigma(\theta') \subseteq \phi(\theta))) \in \mathcal{C}[\mathcal{U}]}(i)$$

$$\frac{{}^{\theta}\mathtt{UPD}({}^{\theta_a}T_a, {}^{\theta_j}T_j, {}^{\theta_v}T_v) \in \mathcal{U}}{(\theta' \in \phi(\theta_a) \Rightarrow (\sigma(\theta') \subseteq \sigma(\theta) \wedge \phi(\theta_v) \subseteq \sigma(\theta) \wedge \theta \in \phi(\theta))) \in \mathcal{C}[\mathcal{U}]}(j)$$

$$\frac{{}^{\theta}\mathtt{UPD!}({}^{\theta_a}T_a, {}^{\theta_j}T_j, {}^{\theta_v}T_v) \in \mathcal{U}}{(\theta' \in \phi(\theta_a) \Rightarrow (\phi(\theta_v) \subseteq \sigma(\theta') \wedge \theta' \in \phi(\theta))) \in \mathcal{C}[\mathcal{U}]}(k)$$

Fig. 4. Set Constraints for 0-CFA Analysis $\langle \phi, \sigma \rangle$.

that are possibly contained in the old array, $(\sigma(\theta') \subseteq \sigma(\theta))$, plus possible results of the last operand $(\phi(\theta_v) \subseteq \sigma(\theta))$. Since the result of the operation is a new location with static label θ, the label is added to the possible results of the operation, $(\theta \in \phi(\theta))$. On the other hand, given a UPD! expression, ${}^{\theta}\mathtt{UPD!}({}^{\theta_a}T_a, {}^{\theta_j}T_j, {}^{\theta_v}T_v)$, no new location is created so the constraints just have to add the possible values of the third operand to the possible values of the existing array , $(\phi(\theta_v) \subseteq \sigma(\theta'))$ and add the static label of the updated location to the possible results of the operation $(\theta' \in \phi(\theta))$.

Theorem 1 (Soundness of $\langle \phi, \sigma \rangle$). *If $\langle \phi, \sigma \rangle$ satisfies the constraints $\mathcal{C}[\mathcal{U}]$ in figure 4 then $\langle \phi, \sigma \rangle$ is sound for $\mathcal{U}$.*

Proof: Following [13], we extend the constraints from constraints on expressions to constraints on configurations S. The most interesting new constraints are presented in figure 5. We write $X \in S$ iff X occurs in S. $lab_{[\,]}(R)$ denotes the label of the hole of R, $lab(R)$ denotes the label of the expression of the frame R, and $lab_{[\,]}(K)$ denotes the label of the hole of the top frame of

K. We show that if $S \rightarrow S'$, the constraints for S imply the constraints for S'. Then we obtain the desired result by induction on the length of reduction $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle$. ■

$$\frac{\rho \in \Sigma \quad \mathtt{x} \in dom(\rho)}{(\phi(lab(\rho(\mathtt{x}))) \subseteq \phi(\mathtt{x})) \in \mathcal{C}[S]}(l) \qquad \frac{\Sigma \in S \quad l \in dom(\Sigma) \quad \Sigma(l) = {}^{\alpha.\theta}\langle v_1, \ldots, v_n\rangle \in \Sigma}{(\forall i \leq n.\phi(lab(v_i)) \subseteq \sigma(\theta)) \in \mathcal{C}[S]}(m)$$

$$\frac{\langle \alpha, \rho, R, K\rangle \in S}{(\phi(lab(R)) \subseteq \phi(lab_{[\,]}(K))) \in \mathcal{C}[S]}(n) \qquad \frac{\langle \alpha, \rho, E, \langle \alpha', \rho', R, K\rangle, \Sigma\rangle \in S}{(\phi(lab(E)) \subseteq \phi(lab_{[\,]}(R))) \in \mathcal{C}[S]}(o)$$

Fig. 5. Extended Set Constraints for Constraints 0-CFA Analysis $\langle \phi, \sigma\rangle$.

Consider the example from section 2. The only constraints relevant to $\phi(\mathtt{y})$ are: $\mathtt{18} \in \phi(\mathtt{18}) \subseteq \phi(\mathtt{14}) \subseteq \phi(\mathtt{12}) \subseteq \phi(\mathtt{y})$. So the smallest solution for ϕ gives $\phi(\mathtt{y}) = \{\mathtt{18}\}$. Similarly, in the smallest solution $\sigma(\mathtt{18}) = \{\mathtt{20}\}$, $\phi(\mathtt{8}) = \{\mathtt{8}\}$, $\phi(\mathtt{x}) = \{\mathtt{8}\}$ and $\phi(\mathtt{2}) = \{\mathtt{2}\}$.

4.2 The Reachability Analysis

The control flow analysis does not describe how values and expressions are associated through the store. In order to describe this relation we use the notion of reachability.

Definition 3 (Reachable Value). *A value w is reachable from a value v in a store Σ,* $\mathrm{reach}(w, v, \Sigma)$*, iff either:*

1. $v = w$*, or*
2. $v = {}^{\theta}l'$ *and* $\Sigma(l') = {}^{\alpha.\theta}\langle v_1, \ldots, v_n\rangle$ *and* $\exists i$ *s.t.* $\mathrm{reach}(w, v_i, \Sigma)$*, or*
3. $v = ({}^{\theta}\lambda\mathtt{x}.{}^{\theta'}T, \rho)$ *and* $\exists \mathtt{y} \in fv(\lambda\mathtt{x}.{}^{\theta'}T)$ *s.t.* $\mathrm{reach}(w, \rho(\mathtt{y}), \Sigma)$.

Following the same path as before, we build an analysis that returns a function $\mathcal{R}$ that associates an expression with the labels of all values reachable from its value.

Definition 4 (Reachability Analysis). *A reachability analysis is a map* $\mathcal{R}$*:* $(\mathrm{Lab} \cup \mathrm{Var}) \rightarrow_{fin} \mathcal{P}(\hat{V})$*, mapping each label or variable to a set of abstract values.*
We say $\mathcal{R}$ *describes a store* Σ*,* $\mathcal{R} \models \Sigma$*, iff* $\forall l \in dom(\Sigma).(\Sigma(l) = {}^{\alpha.\theta}\langle v_1, \ldots, v_n\rangle \implies \forall 0 \leq i \leq n.(\mathrm{reach}(w, v_i, \Sigma) \implies lab(w) \in \mathcal{R}(\theta)))$.
We say $\mathcal{R}$ *describes an environment* ρ *under a store* Σ*,* $\mathcal{R} \models^{\Sigma} \rho$*, iff* $\forall \mathtt{x} \in dom(\rho).\ (\mathrm{reach}(w, \rho(\mathtt{x}), \Sigma) \implies lab(w) \in \mathcal{R}(\mathtt{x}))$.
We say $\mathcal{R}$ *describes an expression* ${}^{\theta}T \in \mathcal{U}$*,* $\mathcal{R} \models {}^{\theta}T$*, iff for all* ρ*,* Σ*, if* $\mathcal{R} \models^{\Sigma}$

ρ, $\mathcal{R} \models \Sigma, \langle \alpha, \rho, {}^{\theta}T, K, \Sigma \rangle$ is a reachable configuration and $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma \rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma' \rangle$, then

1. $\mathrm{reach}(l, v, \Sigma')$ *implies* $lab(l) \in \mathcal{R}(\theta)$
2. *if* $v = ({}^{\theta'}\lambda \mathtt{x}.E, \rho'')$, *then* $\mathcal{R} \models^{\Sigma'} \rho''$
3. $\mathcal{R} \models^{\Sigma'} \rho'$
4. $\mathcal{R} \models \Sigma'$

We say $\mathcal{R}$ is sound for a universe of expressions $\mathcal{U}$, iff $\forall\, E \in \mathcal{U}$, $\mathcal{R} \models E$.

In order to build a sound $\mathcal{R}$, we use a sound control flow analysis to compute the possible values that an expression may be evaluated to or a variable may be bound to. Then, we perform a sort of transitive closure operation inside the environments and the store.

Theorem 2 (Soundness of $\mathcal{R}$). *Given a sound control flow analysis $\langle \phi, \sigma \rangle$ for $\mathcal{U}$, define $\mathcal{R}$ to be the smallest function (in the partial function ordering) that satisfies the equations*

- $\mathcal{R}(t) = \phi(t) \cup \mathcal{M}(t) \cup \mathcal{N}(t)$
- $\mathcal{M}(t) = \{\theta' | \theta'' \in \sigma(t) \wedge \theta' \in \mathcal{R}(\theta'')\}$
- $\mathcal{N}(t) = \{\theta' | \theta'' \in \phi(t) \wedge {}^{\theta''}\lambda \mathtt{x}.E \in \mathcal{U} \wedge \mathtt{y} \in fv({}^{\theta''}\lambda \mathtt{x}.E) \wedge \theta' \in \mathcal{R}(y)\}$

where $t \in Lab \cup Var$. Then $\mathcal{R}$ is sound for $\mathcal{U}$.

Proof: By induction on the definition of $reach(w, v, \Sigma)$ using the soundness of $\langle \phi, \sigma \rangle$. ■

The most interesting part of the definition of $\mathcal{R}$ is the auxiliary set $\mathcal{N}$ that talks about λ-terms, ${}^{\theta}\lambda \mathtt{x}.E$. Then the possibly reachable locations from this term are the locations hidden in all the possible environments that are used for the evaluation of the term. Thus, the reachable locations from the term are the reachable locations from all its free variables, $\forall \mathtt{y} \in fv({}^{\theta}\lambda \mathtt{x}.E).\mathcal{R}(\mathtt{y}) \subseteq \mathcal{R}(\theta)$.

Consider the example from section 2. The only constraints relevent to $\mathcal{R}(\mathtt{y})$ are: $\mathtt{18} \in \mathcal{R}(\mathtt{18}) \subseteq \mathcal{R}(\mathtt{12}) \subseteq \mathcal{R}(\mathtt{y})$. So the smallest solution for $\mathcal{R}$ gives $\mathcal{R}(\mathtt{y}) = \{\mathtt{18}\}$. Also, the only constraints relevant to $\mathcal{R}(\mathtt{8})$ are: $\{\mathtt{8}\} \subseteq \{\mathtt{8}\} \cup \mathcal{R}(\mathtt{16}) \subseteq \mathcal{R}(\mathtt{8})$. So in the smallest solution $\mathcal{R}(\mathtt{8}) = \{\mathtt{8}\}$. Similarly in the smallest solution $\mathcal{R}(\mathtt{x}) = \{\mathtt{8}\}$ and $\mathcal{R}(\mathtt{2}) = \{\mathtt{2}, \mathtt{8}\}$.

4.3 The Liveness Analysis

In order to define the liveness of a location in a machine configuration, we need first to extend the notion of reachability to continuations K.

A value w is reachable from a continuation K in a store Σ if it is reachable in Σ from a value v or variable $\mathtt{x}$ of the top frame of the continuation stack or if it is reachable in Σ by a lower frame of the continuation stack.

Definition 5 (Reachable Value from a Continuation). *Given a store Σ, reachability from a continuation K,* $\mathrm{reach}(w, K, \Sigma)$, *is defined as follows.*

- *No value is reachable from halt in any store* Σ.
- w *is reachable from* $\langle \alpha, \rho, R, K\rangle$ *iff either:*
  1. $\exists v$ *that occurs in* R *s.t.* $\mathrm{reach}(w, v, \Sigma)$, *or*
  2. $\exists \mathtt{x} \in fv(R)$ *s.t.* $\mathrm{reach}(w, \rho(\mathtt{x}), \Sigma)$, *or*
  3. $\mathrm{reach}(w, K, \Sigma)$.

A location is live in a configuration iff it is reachable *after* the evaluation of the current expression. If the expression is a value, this means simply that it is reachable from the value itself or from the current continuation.

Definition 6 (Live Location in a Configuration). *Liveness in a configuration is defined as follows:*

- $\langle halted, v, \Sigma\rangle$. $l \in dom(\Sigma)$ *is live iff* $\mathrm{reach}({}^{\theta}l, v, \Sigma)$
- $\langle \alpha, \rho, v, K, \Sigma\rangle$. $l \in dom(\Sigma)$ *is live iff* $\mathrm{reach}({}^{\theta}l, v, \Sigma)$, *or* $\mathrm{reach}(l, K, \Sigma)$
- $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle$. $l \in dom(\Sigma)$ *is live iff* $(\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle \xrightarrow{*} \langle \alpha', \rho', v, K, \Sigma'\rangle \Rightarrow$ l *live in* $\langle \alpha', \rho', v, K, \Sigma'\rangle$.

Definition 7 (Liveness Analysis).
A liveness analysis $\mathcal{Z}$ *is a map from expression labels* θ *to sets of labels.* $\mathcal{Z}$ *is sound iff for each label* θ *and for all reachable machine configurations* $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma\rangle$, l *live in the configuration implies* $lab(l) \in \mathcal{Z}(\theta)$ *where* $l = \alpha_l.\theta_l$ *and* $lab(l) = \theta_l$.

Given an expression ${}^{\theta}T$, we wish to enumerate the labels of all the locations that could be live following an evaluation of ${}^{\theta}T$. Assume that ${}^{\theta}T$ occurs in a context $E = {}^{\theta'}g({}^{\theta_1}T_1, \ldots, {}^{\theta_{i-1}}T_{i-1}, {}^{\theta}T, {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n)$.

Every evaluation of ${}^{\theta}T$ occurs as part of a sequence of reduction steps

$$\begin{array}{l}\langle \alpha, \rho, E, K, \Sigma\rangle \\ \xrightarrow{*} \langle \alpha.i, \rho, {}^{\theta}T, \langle \alpha, \rho, g(v_1, \ldots, v_{i-1}, [\,], {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n, K\rangle, \Sigma'\rangle \\ \xrightarrow{*} \langle \alpha.i, \rho, v, \langle \alpha, \rho', g(v_1, \ldots, v_{i-1}, [\,], {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n, K\rangle, \Sigma''\rangle\end{array}$$

We need to enumerate the labels of all locations reachable from v or from $K' = \langle \alpha, \rho, g(v_1, \ldots, v_{i-1}, [\,], {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n, K\rangle$. By the definition of reachability, there are exactly four ways in which a location l could be reachable from v or K':

1. l could be reachable from v. This leads to the constraint $\mathcal{R}(\theta) \subseteq \mathcal{Z}(\theta)$.
2. l could be reachable from one of v_1,...,v_{i-1}. Since each of these v_j is the value of ${}^{\theta_j}T_j$, this leads to the constraint $\mathcal{R}(\theta_j) \subseteq \mathcal{Z}(\theta)$ $(1 \le j \le i-1)$.
3. l could be reachable from the value that a free variable is bound to in ${}^{\theta_{i+1}}T_{i+1}, \cdots, {}^{\theta_n}T_n$. This leads to the constraint $\mathtt{x} \in fv({}^{\theta_j}T_j) \implies \mathcal{R}(\mathtt{x}) \subseteq \mathcal{Z}(\theta)$ $(i+1 \le j \le n)$.
4. l could be reachable from K. This leads to the constraint $\mathcal{Z}(\theta') \subseteq \mathcal{Z}(\theta)$.

A similar analysis applies if ${}^{\theta}T$ appears as an argument of a primitive operator p or as the operator or operand of an application or as the test of an `if`-expression. If ${}^{\theta}T$ appears as the body of a λ-expression ${}^{\theta_\lambda}\lambda\mathtt{x}.{}^{\theta}T$ or as a branch of an `if`-expression only cases 1 and 4 apply. If ${}^{\theta}T$ is the expression in the initial configuration, then only case 1 applies. This is summarized in figure 6.

$$\frac{{}^{\theta}T \text{ is the initial expression}}{(\mathcal{R}(\theta) \subseteq \mathcal{Z}(\theta)) \in \mathcal{C}[\mathcal{U}]}(a)$$

$$\frac{{}^{\theta_\lambda}\lambda \mathtt{x}.{}^{\theta}T \in \mathcal{U} \quad {}^{\theta'}({}^{\theta_1}T_1 \ {}^{\theta_2}T_2) \in \mathcal{U} \quad \theta_\lambda \in \phi(\theta_1)}{\begin{array}{l}(\mathcal{R}(\theta) \subseteq \mathcal{Z}(\theta)) \ \in \mathcal{C}[\mathcal{U}] \\ (\mathcal{Z}(\theta') \subseteq \mathcal{Z}(\theta)) \in \mathcal{C}[\mathcal{U}]\end{array}}(b)$$

$$\frac{\begin{array}{c}{}^{\theta}({}^{\theta_1}T_1 \ {}^{\theta_2}T_2) \text{ or} \\ {}^{\theta}p({}^{\theta_1}T_1, \ldots, {}^{\theta_{i-1}}T_{i-1}, {}^{\theta_i}T_i, {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n) \text{ or} \\ {}^{\theta}g({}^{\theta_1}T_1, \ldots, {}^{\theta_{i-1}}T_{i-1}, {}^{\theta_i}T_i, {}^{\theta_{i+1}}T_{i+1}, \ldots, {}^{\theta_n}T_n) \in \mathcal{U}\end{array}}{\begin{array}{ll}\forall i, 1 \leq i \leq n.(\mathcal{R}(\theta_i) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}] \\ \forall i, 1 \leq i \leq n, 1 \leq j \leq i-1.(\mathcal{R}(\theta_j) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}] \\ \forall i, 1 \leq i \leq n, i+1 \leq j \leq n.\mathtt{x} \in fv({}^{\theta_j}T_j) \implies (\mathcal{R}(\mathtt{x}) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}] \\ \forall i, 1 \leq i \leq n.(\mathcal{Z}(\theta) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}]\end{array}}(c)$$

$$\frac{{}^{\theta}\mathtt{if} \ {}^{\theta_1}T_1 \ \mathtt{then} \ {}^{\theta_2}T_2 \ \mathtt{else} \ {}^{\theta_3}T_3 \in \mathcal{U}}{\begin{array}{ll}\forall i, 1 \leq i \leq 3.(\mathcal{R}(\theta_i) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}] \\ \forall j, 2 \leq j \leq 3.\mathtt{x} \in fv({}^{\theta_j}T_j) \implies (\mathcal{R}(\mathtt{x}) \subseteq \mathcal{Z}(\theta_1)) & \in \mathcal{C}[\mathcal{U}] \\ \forall i, 1 \leq i \leq 3.(\mathcal{Z}(\theta) \subseteq \mathcal{Z}(\theta_i)) & \in \mathcal{C}[\mathcal{U}]\end{array}}(d)$$

Fig. 6. Set Constraints for Liveness Analysis $\mathcal{Z}$.

Theorem 3 (Soundness of $\mathcal{Z}$). *Given a control flow analysis $\langle \phi, \sigma \rangle$ and a reachability analysis $\mathcal{R}$, both sound for $\mathcal{U}$, if $\mathcal{Z}$ satisfies the constraints in figure 6 then $\mathcal{Z}$ is sound for $\mathcal{U}$.*

Proof: We apply the same technique used for the proof of the soundness of the control flow analysis. The most interesting extended constraints are presented in figure 7.

■

$$\frac{\langle \alpha, \rho, R, K \rangle \in S}{(\mathcal{Z}(lab_{[\,]}(K)) \subseteq \mathcal{Z}(lab(R))) \in \mathcal{C}[S]}(e) \qquad \frac{\langle \alpha, \rho, E, \langle \alpha', \rho', R, K \rangle, \Sigma \rangle \in S}{(\mathcal{Z}(lab_{[\,]}(R)) \subseteq \mathcal{Z}(lab(E))) \in \mathcal{C}[S]}(f)$$

Fig. 7. Extended Set Constraints for Liveness Analysis $\mathcal{Z}$.

The goal of the live variable analysis is to determine which variables of an expression will be bound to live locations.

Definition 8 (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 all reachable machine configurations $\langle \alpha, \rho, {}^{\theta}T, K, \Sigma \rangle$, $\rho(\mathtt{x})$ live in the configuration implies* $\mathtt{x} \in \mathcal{L}(\theta)$.

Theorem 4 (Soundness of $\mathcal{L}$). *Given a flow analysis $\langle \phi, \sigma \rangle$ for $\mathcal{U}$ and a liveness analysis $\mathcal{Z}$, both sound for $\mathcal{U}$, $\mathcal{L}(\theta) = \{\mathtt{x} \in fv(\theta) | (\phi(\mathtt{x}) \cap \mathcal{Z}(\theta)) \neq \emptyset\}$ is sound for $\mathcal{U}$.*

Proof: Straightforward by the definitions of live variable analysis and soundness of $\mathcal{Z}$. ■

Consider the example from section 2. The constraints relevant to $\mathcal{R}(8)$ are: $\mathcal{Z}(6) \cup \mathcal{R}(8) \subseteq \mathcal{Z}(8)$, $\{8\} \subseteq \mathcal{R}(8)$. But in the smallest solution $\mathcal{Z}(6) = \emptyset$, $\mathcal{Z}(8) = \{8\}$ and $\phi(\mathtt{y}) = \{18\}$. So $\mathcal{L}(8) = \emptyset$. Similarly, $\mathcal{Z}(0) \cup \mathcal{R}(2) \subseteq \mathcal{Z}(2)$. Again in the smallest solution, $\mathcal{Z}(0) = \emptyset$, $\mathcal{R}(8) = \{2, 8\}$, $\phi(\mathtt{x}) = \{8\}$. So $\mathcal{L}(2) = \{\mathtt{x}\}$.

5 Replacing Functional with Destructive Updates

5.1 The Transformation

The soundness of the live variable analysis guarantees that if a free variable $\mathtt{x}$ occurs in an expression ${}^{\theta}T$ and $\rho(\mathtt{x})$ is live after the evaluation of ${}^{\theta}T$ in some configuration, then $\mathtt{x} \in \mathcal{L}(\theta)$. In the case where ${}^{\theta}T = {}^{\theta}\mathtt{UPD}({}^{\theta_x}\mathtt{x}, {}^{\theta_1}T_1, {}^{\theta_2}T_2)$, we infer that if $\mathtt{x} \notin \mathcal{L}(\theta)$, then $\mathtt{x}$ is bound to a location l that is not live after the evaluation of ${}^{\theta}T$. So we can replace the functional update, UPD, with a destructive one, UPD!, without affecting the meaning of the program that ${}^{\theta}T$ appears in, because l is not accessible by any other part of the program.

Definition 9 (The Transformation $(-)^*$). *Let $E \in \mathcal{U}$ and $\mathcal{L}$ a sound live variable analysis for $\mathcal{U}$. Also let Θ be a set of labels s.t. every $\theta \in \Theta$ labels an update of the form ${}^{\theta}\mathtt{UPD}({}^{\theta_x}\mathtt{x}, E_1, E_2)$, where $\mathtt{x} \notin \mathcal{L}(\theta)$. Then E^* is the result of replacing ${}^{\theta}\mathtt{UPD}({}^{\theta_x}\mathtt{x}, {}^{\theta_1}T_1, {}^{\theta_2}T_2)$ by ${}^{\theta}\mathtt{UPD!}({}^{\theta_x}\mathtt{x}, {}^{\theta_1}{T_1}^*, {}^{\theta_2}{T_2}^*)$ for each $\theta \in \Theta$.*

In the example from section 2, from the results of the live variable analysis on the program, we concluded that $\mathtt{y} \notin \mathcal{L}(8)$. So $\Theta = \{8\}$ and the transformation gives us the expected result from section 2.

5.2 Correctness Proof

We claim that the initial and the transformed program have the same observable behavior.

In order to prove our claim, we define a similarity relation $\sim$ between two configurations. The similarity relation is parameterized by a one-to-one function f that records the correspondence between locations on the left and locations on the right. The relation $\sim^f$ is defined by induction on the various structures

- $\langle \alpha, \rho, E, K, \Sigma \rangle \sim \langle \alpha^*, \rho^*, E^*, K^*, \Sigma^* \rangle$ iff $\alpha = \alpha^*$ and there exists a one-to-one function $f : (D \subseteq dom(\Sigma)) \to dom(\Sigma^*)$ such that $\rho \sim^f_{fv(E)} \rho^*$, $\Sigma \sim^f \Sigma^*$, $E \sim^f E^*$, and $K \sim^f K^*$
- $\rho \sim^f_Y \rho^*$ iff $\forall \mathtt{x} \in Y.\ \rho(\mathtt{x}) \sim^f \rho^*(\mathtt{x})$
- $\Sigma \sim^f \Sigma^*$ iff $(l, l^*) \in f$ implies $\Sigma(l) = \langle v_1, \ldots, v_n \rangle$, $\Sigma^*(l^*) = \langle v_1^*, \ldots, v_n^* \rangle$ and $\forall i \leq n.\ v_i \sim^f v_i^*$.
- ${}^{\theta} l \sim^f {}^{\theta^*} l^*$ iff $f(l) = l^*$.
- ${}^{\theta}\mathtt{UPD}(E_l, E_j, E_v) \sim^f {}^{\theta}\mathtt{UPD!}(E_l^*, E_j, E_v^*)$ iff $E_l \sim^f E_l^*$, $E_j \sim^f E_j^*$, $E_v \sim^f E_v^*$ and $\theta \in \Theta$

Fig. 8. The Similarity Relation $\sim$ (selected cases)

involved. The function f avoids the need for a coinductive definition. The key portions of the definition of $\sim$ are shown in figure 8.

Two configurations are similar iff there is a one-to-one function f that makes each of their components similar mod f. Similarity of environments is always done relative to a set of variables Y; ρ and ρ^* are similar mod f iff for each $x \in Y$, their values are similar mod f. Two stores Σ and Σ^* are similar mod f iff for each $(l, l^*) \in f$, $\Sigma(l)$ and $\Sigma^*(l^*)$ are arrays of the same length whose components are similar mod f. Two locations are similar mod f iff they are related by the function f. All the other cases are defined by the obvious structural recursion, except that an `UPD`-term is similar to an `UPD!`-term if they have the same label $\theta \in \Theta$ and their subterms are similar. Similar expressions always have the same label, unless they are $\sim^f$-related locations.

Clearly, the initial states of the original and the transformed program are similar, using the empty function for f. We then prove, by induction on the length of the computation, that as the original and transformed program compute, they stay in similar configurations. Therefore, the machines halt with similar values: if the values are constants, then they must be the same constant.

There are two non-trivial cases: when the machines are at an (`UPD!`, `UPD!`) pair (lemma 2), and when they are at a (`UPD`, `UPD!`) pair (lemma 6). The former illustrates why f must be injective, and the latter is a point at which the transformation has been applied.

In the first case, we use the one-to-one property of f to show that the destructive updates do not disturb the similarity relation of the produced configurations.

Consider the two following configurations:
$S = \langle \alpha.3, \rho_3, v, \langle \alpha, \rho, {}^{\theta}\mathtt{UPD!}({}^{\theta_l} l, j, {}^{\theta_v}[\,]), K \rangle, \Sigma \rangle$,
$S^* = \langle \alpha.3, \rho_3^*, v^*, \langle \alpha, \rho^*, {}^{\theta}\mathtt{UPD!}({}^{\theta_{l^*}} l^*, j^*, {}^{\theta_v}[\,]), K^* \rangle, \Sigma^* \rangle$. Assume that $S \sim^f S^*$. By the semantics of the language we know that $S \to S'$ and $S^* \to S^{*\prime}$ where
$S' = \langle \alpha, \rho, {}^{\theta_l} l, K, \Sigma' \rangle$,
$S^{*\prime} = \langle \alpha, \rho^*, {}^{\theta_{l^*}} l^*, K^*, \Sigma^{*\prime} \rangle$
where $\Sigma' = \Sigma[l \to {}^{\alpha.\theta_l} \langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n \rangle]$,
$\Sigma^{*\prime} = \Sigma^*[l^* \to {}^{\alpha.\theta_{l^*}} \langle v_1^*, \ldots, v_{j-1}^*, v^*, v_{j+1}^*, \ldots, v_n^* \rangle]$

Lemma 1. $\Sigma' \sim^f \Sigma^{*\prime}$

Proof: From $S \sim^f S^*$, we know that ${}^{\theta_l} l \sim^f {}^{\theta_{l^*}} l^*$, $\forall 0 \leq i \leq n.\ v_i \sim^f v_i^*$ and $v \sim^f v^*$. Furthermore, f is an one-to-one function. So there is no other $l' \in dom(\Sigma)$ or $l^{*\prime} \in dom(\Sigma^*)$ such that $l' \sim^f l^*$ or $l \sim^f l^{*\prime}$. Since $\Sigma \sim^f \Sigma^*$ we can conclude that $\Sigma[l \rightarrow {}^{\alpha.\theta_l}\langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n\rangle] \sim^f \Sigma^*[l^* \rightarrow {}^{\alpha.\theta_{l^*}}\langle v_1^*, \ldots, v_{j-1}^*, v^*, v_{j+1}^*, \ldots, v_n^*\rangle]$. ■

Lemma 2. $S' \sim^f S^{*\prime}$

Proof: Straightforward by $S \sim^f S^*$ and lemma 1. ■

In the second case, we take advantage of the definition of the transformation and the fact that this case arises only when the updated location is not live. We define a new one-to-one function f' which makes the two resulting configurations similar.

Consider the two following configurations:
$S = \langle \alpha, \rho, {}^{\theta}\texttt{UPD}(\texttt{x}, E_j, E_v), K, \Sigma\rangle$,
$S^* = \langle \alpha, \rho^*, {}^{\theta}\texttt{UPD!}(\texttt{x}, E_j^*, E_v^*), K^*, \Sigma^*\rangle$
where $\texttt{x} \notin \mathcal{L}(\theta)$ and $S \sim^f S^*$. Assume that $S \xrightarrow{*} S''$ and $S^* \xrightarrow{*} S^{*\prime\prime}$, where
$S'' = \langle \alpha, \rho, {}^{\theta}\alpha.\theta, K, \Sigma''\rangle$,
$S^{*\prime\prime} = \langle \alpha, \rho, {}^{\theta_{l^*}} l^*, K^*, \Sigma^{*\prime\prime}\rangle$
and where $\forall 1 \leq i \leq n.\ v_i \in \Sigma'(l)$, $\forall 1 \leq i \leq n.\ v_i^* \in \Sigma^{*\prime}(l^*)$,
$\Sigma'' = \Sigma'\ [\alpha.\theta \rightarrow {}^{\alpha.\theta}\langle v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n\rangle]$,
$\Sigma^{*\prime\prime} = \Sigma^{*\prime}[l^* \rightarrow {}^{\alpha.\theta_{l^*}}\langle v_1^*, \ldots, v_{j-i}^*, v^*, v_{j+1}^*, \ldots, v_n^*\rangle]$.

Then there must be configurations
$S' = \langle \alpha.3, \rho_3, v, \langle \alpha, \rho, {}^{\theta}\texttt{UPD}({}^{\theta_l} l, {}^{\theta_j} j, {}^{\theta_v}[\,]), K\rangle, \Sigma'\rangle$,
$S^{*\prime} = \langle \alpha.3, \rho_3^*, v^*, \langle \alpha, \rho^*, {}^{\theta}\texttt{UPD!}({}^{\theta_{l^*}} l^*, {}^{\theta_j} j, {}^{\theta_v}[\,]), K^*\rangle, \Sigma^{*\prime}\rangle$
such that $S \xrightarrow{*} S' \rightarrow S''$ and $S^* \xrightarrow{*} S^{*\prime} \rightarrow S^{*\prime\prime}$. Assume that $S' \sim^f S^{*\prime}$.

Consider now the following f':
– $f'(\alpha.\theta) = l^*$
– if l' live in S'' and $l' \neq \alpha.\theta$ then $f'(l') = f(l')$.

Lemma 3. *f' is a one-to-one function.*

Proof: By $S' \sim^f S^{*\prime}$, $f(l) = l^*$. f is a one-to-one function. Let g be the function defined by the second branch of the definition of f'. g is a subset of f. Since l is not live in in S'', g is a one-to-one function that does not include the pair (l, l^*). Also, there is no $l' \in dom(\Sigma')$ such that $l' \neq l$ and $f(l') = l^*$. So there is no $l' \in dom(\Sigma')$ such that $g(l') = l^*$. The extension of g with the pair $(\alpha.\theta, l^*)$ defines f'. From the above we can conclude that f' is a one-to-one function. ■

Lemma 4. $\forall w, w^*$ *if* $\text{reach}(w, K, \Sigma'')$ *and* $w \sim^f w^*$ *then* $w \sim^{f'} w^*$.

Lemma 5. *Let* $\Sigma''(\alpha.\theta) = {}^{\alpha.\theta}\langle w_1, \ldots, w_n\rangle$. $\forall 1 \leq i \leq n, w, w^*$ *if* $\text{reach}(w, w_i, \Sigma'')$ *and* $w \sim^f w^*$ *then* $w \sim^{f'} w^*$.

Proof: We prove these two lemmas by induction on the depth of w. The interesting part is in the base case of the inductive proof, when w is a location, where we proceed by case analysis on whether w is the updated location l. ■

Lemma 6. $S'' \sim^{f'} S^{*\prime\prime}$

Proof: By lemmas 4 and 5, we can conclude that all the values that are reachable from S'' are related through f' with the corresponding values that occur in $S^{*\prime\prime}$. Thus by lemma 3 and the definition of the similarity relation all the elements of the two resulting configurations are similar. ■

Observe that these lemmas imply that similar configuration either both halt or both take a step to similar configurations. Combining these lemmas gets us:

Theorem 5 (Correctness of $(-)^*$ - The main theorem). *Let E_0 be the initial program, and let E_0^* be the result of applying the transformation on E_0. Then $\langle \epsilon, \emptyset, E_0, halt, \emptyset \rangle \xrightarrow{n} \langle halted, v, \Sigma \rangle$ iff $\langle \epsilon, \emptyset, E_0^*, halt, \emptyset \rangle \xrightarrow{n} \langle halted, v^*, \Sigma^* \rangle$ where for some f $v \sim^f v^*$ and $\Sigma \sim^f \Sigma^*$*

6 Related Work

Our effort is strongly related to previous work on inter-procedural aggregate update analysis. Hudak and Bloss [4, 5] propose an aggregate update analysis for strict first-order languages with flat arrays. Their approach combines abstract interpretation with conventional flow analysis. In their turn, Draghicescu and Purushothaman [1] presented an aggregate optimization for non-strict first-order languages with flat arrays. The transformation is based on a liveness analysis.

The analysis of Wand and Clinger [12], which extends the analysis of [7, 8], presents a modular framework. They build a propagation analysis, and on top of that an alias analysis and finally a live variables analysis. The transformation replaces functional updates of dead variables with destructive updates. Their analysis deals only with first-order languages with arrays of scalar values and they prove the soundness of their analysis using environmental semantics and store erasure.

Shankar [9] proposed a method for safe destructive update in strongly-typed higher-order functional languages with eager order of evaluation. The structure of his framework is very similar to that of [12]: it uses an alias analysis to create a live variable analysis and then uses the results of the analysis to perform the transformation. The analysis handles higher-order programs by means of a fixed point calculation of the live variables. However the analysis cannot handle nested arrays.

From a different perspective, efforts like [3, 6] are influenced by the generalization of linear type systems [11] and try to solve the problem using type annotations. None of these can handle untyped programs as our does.

Our optimization is based on the analysis of [12]. We share the spirit of a modular framework which consists of layers with different analysis in each layer

computed symbolically. But our analysis differs from that of [12] in a number of ways. Instead of a propagation and an alias analysis we use control-flow analysis [10]. This makes our framework capable of handling higher-order languages and arrays of any kind of values. Also, we can handle source code with both functional and destructive updates. Finally we use a different proof technique for the correctness of the transformation by constructing a bisimulation of the initial and the transformed program based on store shape properties.

References

1. M. Draghicescu and S. Purushothaman. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118(2):231–262, 1993.
2. M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, page 314, 1987.
3. J. C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.
4. P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 300–314, 1985.
5. P. Hudak and A. Bloss. Avoiding copying in functional and logic programming languages. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 300–314, 1985.
6. M. Odersky. How to make destructive updates less destructive. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 25–26, 1991.
7. A.V.S. Sastry. *Efficient Array Update Analysis of Strict Functional Languages*. PhD thesis, Computer and Information Science, University of Oregon, 1994.
8. A.V.S. Sastry, W. D. Clinger, and Z. 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.
9. N. Shankar. Static analysis for safe destructive updates in a functional language. In *International Workshop on Logic-based Program Synthesis and Transformation*, pages 1–24, 2002.
10. O. Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1991. Technical Report CMU-CS-91-145.
11. P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359, 1990.
12. M. Wand and W. D. Clinger. Set constraints for destructive array update optimization. *Journal of Functional Programming*, 11(3):319–346, May 2001.
13. G. B. Williamson. *Flow analysis for higher-order multithreaded computations*. PhD thesis, College of Computer and Information Science, Northeastern University, Boston, Massachusetts, 2004.