Theoretical Computer Science 118 (1993) 231–262
Elsevier

A uniform treatment of order of evaluation and aggregate update*

M. Draghicescu

EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, USA

S. Purushothaman

Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA

Communicated by A.R. Meyer
Received January 1991
Revised February 1992

Abstract

Draghicescu, M. and S. Purushothaman, A uniform treatment of order of evaluation and aggregate update, Theoretical Computer Science 118 (1993) 231–262.

The article presents an algorithm for the destructive update optimization in first-order lazy functional languages. The main component of the method is a new static analysis of the *order of evaluation of expressions* which, compared to other published work, has a much lower complexity and is not restricted to pure lazy evaluation. The other component, which we call *reduction to variables*, is a method of detecting the variables which denote locations where the result of an expression might be stored.

Starting with the operational semantics of the language, we introduce some *markers* for the values in the basic domain. By choosing appropriately the set of markers M and the method of propagating them during evaluation, we can extract some property of the evaluation in which an expression can participate by looking at the marker of its value. We then define an equivalent denotational semantics and derive the above analyses, in a uniform way, by abstract interpretation over a subdomain of $P(M_\perp)$.

1. Introduction

A characteristic feature of functional languages is their *referential transparency*, which makes them suitable for parallel execution. On sequential machines, however,

Correspondence to: M. Draghicescu, EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, USA.

* Funded in part by NSF CDA-89-14587. A preliminary version of this paper appeared in [7].

this quality becomes a serious obstacle to an efficient implementation. The impossibility to compute through side-effects greatly reduces the efficiency of functional languages which manipulate large data structures, such as arrays, records, or lists. In a functional language, an object, once created, is never changed; so, modifying such a structure implies making a new copy. This is inefficient not only because large structures must be copied but also because of the additional load on the garbage collector. Traditionally, designers of functional languages either do not provide these data structures or introduce "impure" operations which destroy the referential transparency.

To use such structures efficiently in a pure functional language we must detect the structure modifications (updates) which can be done *destructively* or *in place* without affecting the semantics of the language. This can be done either by some run-time checks (e.g., by keeping track of reference counts) or through compile-time analysis. The latter approach is the topic of the present work.

The destructive update optimization has been considered in the literature before, one of the early works being [13]. In [10] the problem is discussed in an operational model based on graph reduction. An applicative-order language is treated in [9] using an abstraction of reference counting (reference counting offers a *run-time* solution to this optimization problem). A related analysis (detection of *single-threaded* definitions) is presented in [17, 18], also in an applicative-order setting. The problem is also discussed in [1, 2] as an application of the path analysis (see below); the method thus obtained is very expensive computationally. A variation of path analysis is also used in [8] for a language with call-by-value semantics.

We present here another solution to this problem. The general idea used in this article, and in most of the works cited above, is the following: an object can be updated destructively only if it is not accessed after the update. To detect this at compile time, we need some information about (a) the possible *sharing* of this object and (b) the run-time *order of evaluation* of expressions.

The article presents new solutions to these two static-analysis problems for lazy functional languages. They are needed for the destructive–update procedure and they are also of independent interest. Our method is based on *abstract interpretation*, a semantically based general technique for compile-time analysis.

Sharing information can be presented under different forms; we called our analysis *reduction to variables*. It detects the variables which may denote the location where the result of an expression evaluation will be stored at run-time and is related to *targeting* [8]. The analysis is also related to *aliasing*, a much-studied problem, especially for imperative languages (a solution based on abstract interpretation is presented in [14]).

The evaluation-order analysis is simple in an applicative-order model. The first solution for normal-order languages that use pure lazy evaluation is presented in [3]. The most general solution to date is *path analysis* presented in [1, 4]. Unlike these works, our analysis is not restricted to lazy evaluation, but applies to all evaluation strategies compatible with the semantics of the language (for example, strict arguments can be evaluated in any order or even in parallel). The method can also be adapted, yielding a sharper analysis, to any predefined order of evaluation of arguments to primitive functions. Its complexity is exponential in the number of variables, which is a significant improvement over the $O(2^{N!+(N-1)!+\cdots+1})$ complexity of path analysis. The most important application of evaluation-order analysis is to the destructive-update problem; other optimizations based on this information are mentioned in [1, 5].

The article is organized as follows: Section 2 describes the syntax and semantics of the language used for illustration. We define two equivalent semantics: an operational and a denotational one. A general nonstandard semantic scheme (both operational and denotational), which constitutes the starting point of the analyses developed in the following sections, is also defined. The nonstandard semantic scheme is intended to capture information that can be gleaned from the standard operational semantics, but in a more accessible form. The idea is to *mark* the values in the basic domain and define the method of propagating the markers during evaluation so that we can extract some property of the evaluation in which an expression can participate by looking at the marker of its value. The section also contains some examples which give a motivation to the present work.

Section 3 contains a short presentation of abstract interpretation and its classical application to *strictness analysis*. We also introduce some definitions and notations used in the rest of the article and compute, by abstract interpretation of the nonstandard semantics, a general relation between the variables of an expression.

The reduction-to-variables and evaluation-order analyses are presented in Sections 4 and 5, respectively. They are first defined as predicates over the reduction sequences engendered by the standard operational semantics. It is then shown how this information can be obtained as particularizations of the general relation mentioned above.

The procedure for the destructive-update problem is discussed in Section 6. The use of the procedure is shown with several examples; for the functional version of the quicksort algorithm considered in [9], the procedure yields a linear-space complexity.

The conclusions and plans for future work are presented in Section 7. To summarize, the contributions of the paper are: (a) evaluation-order analysis, (b) reduction to variables and destructive update, and, importantly, (c) a methodology for static analysis starting from the operational semantics.

2. A first-order language

We will consider a language L of first-order recursion equations with normal-order semantics. The data types include integers, booleans, and one-dimensional arrays of integers with fixed lower and upper bounds; the lower bound is always 1.

This section contains formal definitions of the syntax and semantics of L. We also define a general nonstandard semantics on which the analyses developed in the following sections are based.

2.1. Abstract syntax

$c, [c_1, \ldots, c_n], p \in Con$	(constants, primitive functions),
$x \in Var$	(variables),
$f \in Fn$	(function names),
$e, body \in Exp$	(expressions),
$pr \in Prog$	(programs),

where

$$e ::= c \mid [c_1, \ldots, c_n] \mid x \mid p(e_1, \ldots, e_n) \mid f(e_1, \ldots, e_n),$$

$$\begin{aligned} pr ::= \ & f_1(x_{11}, \ldots, x_{1k_1}) = body_1, \\ & f_2(x_{21}, \ldots, x_{2k_2}) = body_2, \\ & \quad \vdots \\ & f_n(x_{n1}, \ldots, x_{nk_n}) = body_n. \end{aligned}$$

$[c_1, \ldots, c_n]$ denotes the constant array of size n, with elements $c_1, \ldots, c_n$. For simplicity, we did not include an expression in the definition of a program but, instead, we will require that f_1, the first function, takes no arguments and a program is "run" by calling f_1. We assume that the formal parameters of all user-defined functions are distinct variables. Let P be a given program.

Notations.

$body_f$ is the body of the function f in P.
Exp is the set of expressions in P.
$M = cardinality(Exp)$.
Exp_f is the set of subexpressions of $body_f$.
Var is the set of variables in P.
$N = cardinality(Var)$.
Var_e is the set of variables which occur in the expression e.
Var_f is the set of variables which are formals of the function f ($Var_{body_f} \subseteq Var_f$).

We will use lowercase letters from the end of the alphabet to denote variables and capital letters for sets of variables. We will denote arbitrary expressions by e (possibly with subscripts or superscripts), nonfunctional constants by c, general primitive functions by p, and user-defined functions by f, g, or h.

2.2. Standard semantics

For a set S denote by $S_\perp$ the flat domain $S \cup \{\perp\}$ ordered by $\perp \sqsubset s$ for all $s \in S$.

Semantic domains

$Z=\{\ldots, -1, 0, 1, \ldots\}$	(integers),
$B=\{true, false\}$	(booleans),
$A=Z+Z^2+\cdots$	(arrays),
$D=(Z+B+A)_\perp$	(basic domain),
$Env=Var\to D$	(variables environment),

where + is the separated sum operation.

Semantic functions

$\mathscr{F}: Fn\to D^*\to D$	(gives meaning to function names),
$\mathscr{E}: \to Exp\to Env\to D$	(gives meaning to expressions),
$\mathscr{C}: Con\to D^*\to D$	(gives meaning to constants).

We will use the informal method of presenting the semantics from [12], which consists in defining $\mathscr{E}$ and $\mathscr{F}$ through a set of mutually recursive equations. $\mathscr{F}$ corresponds to the "function variable environment" which is expressed as the least fixed point of an operator in a more traditional presentation.

Semantic equations

$$\mathscr{F}[\![f_i]\!]=\lambda d_1\ldots d_{k_i}\cdot\mathscr{E}[\![body_i]\!]\,[d_j/x_{ij}],$$

$$\mathscr{E}[\![c]\!]\,\rho=\mathscr{C}[\![c]\!],$$

$$\mathscr{E}[\![x]\!]\,\rho=\rho[\![x]\!],$$

$$\mathscr{E}[\![p(e_1,\ldots,e_n)]\!]\,\rho=\mathscr{C}[\![p]\!]\,\mathscr{E}[\![e_1]\!]\,\rho\ldots\mathscr{E}[\![e_n]\!]\,\rho,$$

$$\mathscr{E}[\![f(e_1,\ldots,e_n)]\!]\,\rho=\mathscr{F}[\![f]\!]\,\mathscr{E}[\![e_1]\!]\,\rho\ldots\mathscr{E}[\![e_n]\!]\,\rho.$$

The following typical primitive functions will be used throughout the article:

(1) *if*: the polymorphic conditional.

(2) $+, <, \ldots$: arithmetic and relational operators.

(3) *select*(a, i): returns the ith element of the array a.

(4) *update*(a, i, q): returns an array identical to a except for the ith element which is q.

(5) $\oplus$: array addition.

(6) *length*: the length (size) of an array.

$\mathscr{C}[\![c]\!]=c$ ($c\in Z+B$ is the semantic value of c),

$\mathscr{C}[\![[c_1,\ldots,c_n]]\!]=\langle c_1,\ldots,c_n\rangle$ (the constant array of size n; $c_i\in Z$)

$\mathscr{C}[\![+]\!]=\lambda d_1 d_2\cdot d_1+d_2$, where the right-hand side + denotes the strict addition in $Z_\perp$,

$\mathscr{C}[\![if]\!]=\lambda d_1 d_2 d_3\cdot if\ d_1\ then\ d_2\ else\ d_3$,

$\mathscr{C}[\![select]\!]=\lambda\langle k_1,\ldots,k_n\rangle i\cdot if\ i>n\ then\ \perp\ else\ k_i$,

$$\mathscr{C}[\![update]\!]=\lambda\langle k_1,\dots,k_i,\dots,k_n\rangle iq\cdot if\ i>n\ then\ \perp\ else\ \langle k_1,\dots,q,\dots,k_n\rangle,$$

$$\mathscr{C}[\![\oplus]\!]=\lambda\langle a_1,\dots,a_m\rangle\langle b_1,\dots,b_n\rangle\cdot if\ m\neq n\ then\ \perp\ else$$

$$\langle a_1+b_1,\dots,a_m+b_m\rangle,$$

$$\mathscr{C}[\![length]\!]=\lambda\langle k_1,\dots,k_n\rangle\cdot n.$$

We will usually write *if x then y else z*, $x+y$, and $a[i]$ instead of $if(x, y, z)$, $+(x, y)$, and *select*(a, i), respectively.

Note that we assumed all programs to be well-typed. The size of an array is not part of its type. Type checking can be done statically using a Hindley–Milner type algorithm.

Throughout this paper, we will assume a *lazy* evaluation strategy, i.e., call-by-name plus the fact that function arguments are evaluated at most once, subsequent references using the already computed values. We will also assume that, operationally, the value of an expression is a *reference* (location, pointer). This reference might be to a newly created object (integer, boolean, or array) or it might be to an already existing one. The same object might be created many times as the result of evaluating different expressions, but an existing object is never duplicated explicitly. For example, evaluating $1+4$ and $2+3$ will create two copies of the object 5; however, if

$$\max(x, y)=if\ x\geqslant y\ then\ x\ else\ y,$$

then the evaluation of $\max(1, 2+3)$ will return a reference to the unique 5 created when its second argument is evaluated. These assumptions are valid, for example, in an execution model based on *graph reduction* [15].

The purpose of the destructive-update analysis is to determine at compile time whether a given expression *update*$(e, \dots)$ in a given program P can be evaluated, without affecting the meaning of P, *in place* (i.e., destructively, by rewriting the array e instead of creating a new array).

Example 2.1.

$$minus(a)=minus1(a, 1)),$$

$$minus1(a, i)=\text{if}\ i>length(a)\ then\ a\ else\ minus1(update(a, i, -a[i]), i+1).$$

If called on an array of length 100, *minus* will generate 100 new arrays. However, it is clear that, if the original value of a is not needed after any of the calls to *minus* in a given program, all the evaluations of *update* can be done in place.

The following examples will illustrate some of the problems that we must solve when trying to detect (at compile time) the *update*s which can be done in place. Solutions to these problems will be discussed in the rest of the article.

Example 2.2.

$$f(u, v) = u \oplus v,$$

$$g(x) = f(x, \mathit{update}(x, \ldots)).$$

The *update* can or cannot always be done in place depending on the order of evaluation of the arguments of *f*. In this example the *update* cannot be done in place if $\oplus$ might evaluate its arguments right to left. In general, the run-time order of evaluation cannot be computed at compile time; the challenge is to find a good approximation of this order which is statically computable.

$$g(x) = f(x, \mathit{update}(x, \ldots)) \oplus x, \quad f \text{ as above.}$$

The *update* cannot be done in place no matter what the (fixed) order of evaluation of $\oplus$ is.

Example 2.3.

$$\ldots \mathit{update}(x \oplus y, \ldots)\ldots$$

This *update* can always be done in place; $x \oplus y$ is a new, nameless, array which cannot be referenced anywhere else in the program, so it can be destroyed safely.

$$\ldots \mathit{update}(\mathit{update}(x, \ldots), \ldots)\ldots$$

The first (outside) *update* can always be done in place. Even if the inside *update* is done in place, we can consider its value to be a new object (after all we know that x will never be needed again, otherwise the inside *update* could have not been done in place).

A key observation is that an object can be referenced in more than one place only if it is denoted by a variable. The following example will further illustrate this idea. We will assume from now on that $\oplus$ is always evaluated left to right.

Example 2.4.

$$f(x) = \mathit{update}(g(x), \ldots) \oplus x,$$

$$g(y) = y.$$

We can immediately determine that the *update* cannot be done in place; see below.

$$g(y) = y \oplus y, \quad f \text{ as above.}$$

The *update* can be done in place now. The difference between these two examples is that, in the former case $g(x)$ and x refer to the same object (operationally, x and the result returned by $g(x)$ are the same reference), while in the latter case they denote different objects. In the former case, we will say that $g(x)$ *reduces* to x.

$$g(y) = \mathit{if} \ldots \mathit{then}\ y\ \mathit{else}\ y \oplus y, \quad f \text{ as above.}$$

We cannot know, at compile time, whether $g(x)$ will reduce to x or not; therefore, the safe decision must be that the *update* cannot be done in place. We will say, in this case, that $g(x)$ *might* reduce to x.

$$f(x, y) = \mathit{update}(x, \ldots) \oplus y,$$

$$g(u, v) = \text{if} \ldots \mathit{then}\ u\ \mathit{else}\ v,$$

$$h(p, q, r) = f(g(p, q), g(q, r)).$$

The *update* cannot be done in place: both $g(p, q)$ and $g(q, r)$ might reduce to q, so x and y might denote the same object; therefore, x cannot be destroyed.

Example 2.5.

$$f(x, y) = x \oplus y \oplus x,$$

$$h(u) = f(u, \mathit{update}(u, \ldots)).$$

The *update* cannot be done in place. f will evaluate x before y, but it will also *access* x again, after y is evaluated. This example shows that we must also consider the relative order in which variables are accessed and not only the order in which they are evaluated (under lazy evaluation, they are evaluated when first accessed).

$$h(u) = f(g(u), \mathit{update}(u, \ldots)), \quad f \text{ as above, } g \text{ as in Example 2.4.}$$

If $g(u)$ might reduce to u (e.g., $g(y) = y$) then the *update* cannot be done in place. On the other hand, if $g(u)$ never reduces to u (e.g., $g(y) = y \oplus y$) then the *update* could be done in place: $g(u)$ is evaluated when x is first accessed; its (new) value is stored and the second access to x refers to this stored value, so u is not needed after the *update*.

The following examples will show the limits of the approach presented in this paper:

Example 2.6.

$$f(x) = \mathit{minus}(x) \oplus x,$$

where *minus* is defined in Example 2.1. The *update* in *minus*1 cannot be done in place because x is needed later; this means that *minus*1 will generate $\mathit{length}(x)$ arrays all of which, except the last one, are useless, intermediate results, which could be destroyed even if the value of x is needed later. The optimization which consists in evaluating the *update* normally once and then destructively $\mathit{length}(x) - 1$ times is beyond the scope of the present work: for a given (statical) *update*, we decide only whether it can *always* be evaluated in place or not.

However, our analysis will determine that x is the variable which prevents the *update* from being done destructively and an optimizing compiler could easily transform f into:

$$f(x) \ = \mathit{minus}(\mathit{new\_copy}(x)) \oplus x,$$

where *new_copy* is a special built-in function which returns a new copy of its argument.[1] Now the *update* in *minus*1 can be done in place, so the optimized program will do only one array copy (by *new_copy*) instead of $length(x)$.

Example 2.7.

$$f(x) = length(update(x, \dots) + length(x)).$$

Assuming that + is evaluated left to right, we will decide that the *update* cannot be done in place because x is accessed after the *update*. We do not treat separately functions like *length* which are not affected by any *update*s of their argument. It is not too difficult to modify our procedure to take into account such situations; the following example, however, illustrates a much more interesting and difficult problem:

$$f(a, i, x) = update(a, i, x)[i] + a[i+1].$$

The first operand of + is equivalent to x, but the point here is that we will again conclude that the *update* cannot be done in place because a is accessed after the *update*. In reality the *update* could be safely made in place: only the $(i+1)$th element of a is needed after the ith one is lost. We make no attempt to analyze statically the possible values of array *indices*.

2.3. Operational semantics

The notions of order of evaluation and sharing can be defined only in an operational manner. The operational semantics presented in this section is a simplified version (adapted to our first-order language) of the operational semantics of PCF presented in [16]. The only difference is the presence of an environment and the rule (1), which allows the reduction of expressions containing free variables. Note, however, that the variables are used only at the first level; function calls do not introduce new variables nor do they change the environment (rule (6)).

For each boolean, integer, or array $d \in D$, denote by $\hat{d}$ its syntactic representation. We have $\hat{d} \in Con$ and $\mathscr{C}[\![\hat{d}]\!] = d$. Let, by definition, $\hat{\bot} = \omega$, where ω is some expression whose standard value is $\bot$, for example,

$$\omega = f(\ ),\ \text{where } f \text{ is a function with no arguments defined as } f(\ ) = f(\ ).$$

For each $\rho \in Env$ the reduction relation $\rightarrow_\rho$ between expressions is defined by the following rules:

$$x \rightarrow_\rho \widehat{\rho(x)} \quad (x \in Var), \tag{1}$$

[1] If we would need to define it ourselves, then $new\_copy(u) = update(u, 1, u[1])$ will do the trick; $new\_copy(u) = u$ is not good because it does not copy its argument.

$$\frac{e_i \rightarrow_\rho e'_1}{p(e_1 \ldots e_i \ldots e_n) \rightarrow_\rho p(e_1 \ldots e'_i \ldots e_n)} \quad (p \neq \mathit{if}), \tag{2}$$

$$\frac{e_1 \rightarrow_\rho e'_1}{\mathit{if}(e_1, e_2, e_3) \rightarrow_\rho \mathit{if}(e'_1, e_2, e_3)}, \tag{3}$$

$$\frac{e_2 \rightarrow_\rho e'_2}{\mathit{if}(\mathit{true}, e_2, e_3) \rightarrow_\rho \mathit{if}(\mathit{true}, e'_2, e_3)}, \quad \frac{e_3 \rightarrow_\rho e'_3}{\mathit{if}(\mathit{false}, e_2, e_3) \rightarrow_\rho \mathit{if}(\mathit{false}, e_2, e'_3)}, \tag{4}$$

$$\mathit{if}(\mathit{true}, c, e) \rightarrow_\rho c, \quad \mathit{if}(\mathit{false}, e, c) \rightarrow_\rho c \quad (c \in \mathit{Con}), \tag{5}$$

$$f(e_1 \ldots e_n) \rightarrow_\rho \mathit{body}_f[e_i/x_i]. \tag{6}$$

To these rules we will also add the following rule scheme specifying the action of the primitive functions other than *if* on all possible combinations of constant arguments:

$$p(c_1 \ldots c_n) \rightarrow_\rho \hat{d}, \quad d = \mathscr{C}[\![p]\!]\, c_1 \ldots c_n, \quad p \neq \mathit{if}, \quad c_i \in \mathit{Con}. \tag{7}$$

Note that rule (6) specifies call-by-name as the evaluation strategy. Note also that the condition of *if* must be completely reduced before any reduction can take place in one of the branches (rules (3)–(5)); therefore, the evaluation proceeds in a pure lazy manner (as opposed, for example, to an evaluation which uses strictness information to change the order of evaluation; a strategy which allows such changes will be discussed in Section 5.3). A reduction sequence might not be unique because we do not impose any order on the reduction of arguments of the primitive functions other than *if*. An expression will either reduce to a constant or its reduction will not terminate. We can easily prove that if c is a constant and if $e \overset{*}{\rightarrow}_\rho c$ then any reduction of e will terminate in c ($\overset{*}{\rightarrow}_\rho$ is the transitive–reflexive closure of $\rightarrow_\rho$). We can, therefore, define the evaluation function $\mathit{Eval}: \mathit{Exp} \rightarrow \mathit{Env} \rightarrow D$ by

$$\mathit{Eval}(e, \rho) = \begin{cases} d & \text{if } e \overset{*}{\rightarrow}_\rho \hat{d}, \\ \bot & \text{otherwise.} \end{cases}$$

The following theorem states the equivalence between the denotational and operational semantics (for a proof, see [19]).

Theorem 2.8. *For all* $e \in \mathit{Exp}$, $\rho \in \mathit{Env}$,

$$\mathit{Eval}(e, \rho) = \mathscr{E}[\![e]\!]\, \rho.$$

2.4. Nonstandard semantics

The standard semantics of L does not contain all the information needed for the analyses presented in this article. We now define a general nonstandard semantics by adding some extra information to the standard one. The idea is to "mark" the elements of D. The marker of an expression is computed from the markers of its components following some rules. By choosing these rules appropriately, we will obtain different particularizations of this general semantics.

Let $M=\{m_1, \ldots, m_n\}$ be a finite set of markers. The nonstandard basic domain is

$$D_n=((Z+B+A)\times M)_\perp$$

and the nonstandard domain of environments is

$$Env_n = Var \to D_n.$$

By identifying $\perp \in D_n$ with $\langle \perp, \perp \rangle \in D \times M_\perp$, we will consider D_n to be a subdomain of $D \times M_\perp$. Define the two projections

$$\begin{aligned} &content: D_n \to D, && marker: D_n \to M_\perp,\\ &content(\langle d, m\rangle)=d, && marker(\langle d, m\rangle)=m. \end{aligned}$$

Note that $marker(x)=\perp$ iff $content(x)=\perp$ iff $x=\perp$. For $t=\langle \hat{d}, m\rangle \in D_n, t\neq\perp$, let $t=\langle \hat{d}, m\rangle \in Con \times M$ (its "syntactic representation") and let $\hat{\perp}=\omega$. The markers associated with constants and primitive functions are given by the strict functions:

$$\tilde{p}: M_\perp^n \to M_\perp \quad (\text{for all } p \neq if \text{ of arity } n \geqslant 0)$$

$$\widetilde{if}: M_\perp^2 \to M_\perp.$$

In particular, the marker of a constant c is $\tilde{c} \in M$.

It is more convenient to define the new reduction relations $\to_{\rho_n}$ for $\rho_n \in Env_n$ between expressions in a new language L_M. The set of constants of L_M is $Con \times M$; the rest of the syntax is identical to that of L. For an expression e in L we will denote by e_M the expression in L_M obtained from e by replacing each $c \in Con$ by $\langle c, \tilde{c}\rangle$. To define $\to_{\rho_n}$, we will introduce the computations on markers into the rules (1)–(7). The new rules are:

$$x \to_{\rho_n} \widehat{\rho_n(x)} \quad (x \in Var), \tag{8}$$

$$\frac{e_i \to_{\rho_n} e_i'}{p(e_1 \ldots e_i \ldots e_n) \to_{\rho_n} p(e_1 \ldots e_i' \ldots e_n)} \quad (p \neq if), \tag{9}$$

$$\frac{e_1 \to_{\rho_n} e_1'}{if(e_1, e_2, e_3) \to_{\rho_n} if(e_1', e_2, e_3)}, \tag{10}$$

$$\begin{gathered} \frac{e_2 \to_{\rho_n} e_2'}{if(\langle true, m\rangle, e_2, e_3) \to_{\rho_n} if(\langle true, m\rangle, e_2', e_3)},\\ \frac{e_3 \to_{\rho_n} e_3'}{if(\langle false, m\rangle, e_2, e_3) \to_{\rho_n} if(\langle false, m\rangle, e_2, e_3')}, \end{gathered} \tag{11}$$

$$\left.\begin{aligned} &if(\langle true, m_1\rangle, \langle c, m_2\rangle, e) \to_{\rho_n} \langle c, \widetilde{if}(m_1, m_2)\rangle,\\ &if(\langle false, m_1\rangle, e, \langle c, m_2\rangle) \to_{\rho_n} \langle c, \widetilde{if}(m_1, m_2)\rangle \end{aligned}\right\} \quad (c \in Con), \tag{12}$$

$$f(e_1 \dots e_n) \to_{\rho_n} body_f[e_i/x_i], \tag{13}$$

$$p(\langle c_1, m_1\rangle \dots \langle c_n, m_n\rangle) \to_{\rho_n} \langle \hat{d}, \tilde{p}(m_1 \dots m_n)\rangle,$$

$$d = \mathscr{C}[\![p]\!]\, c_1 \dots c_n,\ p \neq if,\ c_i \in Con. \tag{14}$$

The nonstandard reductions mirror exactly the standard ones. The markers are computed in parallel with the standard values but they do not influence the reduction sequence. The nonstandard reduction is, therefore, confluent and we can define the evaluation function $Eval_n: Exp \to Env_n \to D_n$ by

$$Eval_n(e, \rho_n) = \begin{cases} \langle d, m\rangle & \text{if } e_M \overset{*}{\to}_{\rho_n} \langle \hat{d}, m\rangle, \\ \perp & \text{otherwise.} \end{cases}$$

It is easy to prove that the standard semantics can be obtained from the nonstandard one by ignoring the markers.

Theorem 2.9. *For all $e \in Exp$, $\rho_n \in Env_n$,*

$$content(Eval_n(e, \rho_n)) = Eval(e, content \circ \rho_n),$$

where $\circ$ denotes the left-to-right function composition.

We will define now an equivalent nonstandard denotational semantics. The semantic functions $\mathscr{E}_n$ and $\mathscr{F}_n$ are defined similarly to $\mathscr{E}$ and $\mathscr{F}$ from the standard semantics (Section 2.2), while $\mathscr{C}_n$ will include now the action on markers given by $\tilde{p}$:

$$\mathscr{C}_n[\![p]\!] = \langle \mathscr{C}[\![p]\!] \circ content^n, \tilde{p} \circ marker^n\rangle \text{ for any } n\text{-argument } p \neq if,\ n \geq 0 \tag{15}$$

$$\begin{aligned} \mathscr{C}_n[\![if]\!] = \lambda xyz\,.\,&\text{case } content(x): \\ & true :: \langle content(y), \widetilde{if}(marker(x), marker(y))\rangle, \\ & false :: \langle content(z), \widetilde{if}(marker(x), marker(z))\rangle, \\ & \perp :: \perp. \end{aligned} \tag{16}$$

The following analogue of Theorem 2.8 also holds for the two nonstandard semantics.

Theorem 2.10. *For all $e \in Exp$, $\rho_n \in Env_n$,*

$$Eval_n(e, \rho_n) = \mathscr{E}_n[\![e]\!]\, \rho_n.$$

The nonstandard semantics defined above depends on the set of markers M and the marker propagation functions $\tilde{p}$. By specifying M and $\tilde{p}$ for each primitive function p, we can obtain different semantics. Two such particularizations will be used for the evaluation-order and reduction-to-variables analyses.

3. Abstract interpretation

This section presents some classical results from the theory of abstract interpretation of first-order functional languages first developed in [13].

The idea of the abstract interpretation method is to obtain some information about a function f by projecting the semantic domain D on some abstract domain $D^{\#}$ and then computing the abstract semantic value of f in $D^{\#}$. Under the conditions described below, there is a relation between the normal semantic value and the abstract one. $D^{\#}$ is chosen such that (a) the abstract semantic value of f gives us the required information, and (b) computing the abstract semantic values can be done at compile time. (b) is satisfied, for example, if $D^{\#}$ is finite, which is usually the case.

The classic example is the *rule of signs* in arithmetic, which enables us to find the sign of a multiplication knowing the signs of the operands, without having to actually perform the multiplication. Here $D=Z$ and $D^{\#}=\{0, +, -\}$.

The following are some simple facts from domain theory: for a flat domain X, the *Hoare powerdomain* $P(X)$ is defined as

$$P(X)=\{A\subseteq X \mid \bot\in A\},$$

ordered by subset inclusion. For $A\subseteq X$, denote by $\bar{A}=A\cup\{\bot\}\in P(X)$ (the closure of A). If X and Y are flat domains, a function $f\colon X^n\rightarrow Y$ can be extended to a function $f\colon P(X)^n\rightarrow P(Y)$ by defining

$$f(A_1, \ldots, A_n)=\overline{\{f(a_1, \ldots, a_n) \mid a_i\in A_i\}}.$$

In Mycroft's abstract interpretation method, the powerdomain $P(D)$ is projected on the abstract domain $D^{\#}$. More exactly, we define the continuous abstraction and concretization functions,

$$Abs\colon P(D)\rightarrow D^{\#}, \qquad Conc\colon D^{\#}\rightarrow P(D),$$

which must satisfy

$$Abs\circ Conc=id_{D^{\#}}, \qquad Conc\circ Abs\supseteq id_{P(D)}. \tag{17}$$

The abstract valuation functions $\mathscr{E}^{\#}$ and $\mathscr{F}^{\#}$ are defined in the same way as $\mathscr{E}$ and $\mathscr{F}$ (see Section 2.2). For each n-argument primitive p, we define:

$$\mathscr{C}^{\#}[\![p]\!]=Abs\circ\mathscr{C}[\![p]\!]\circ Conc^n. \tag{18}$$

Under these conditions, the correctness theorem of Mycroft is stated as follows.

Theorem 3.1 (*Mycroft* [13]). *For each n-argument user-defined function f,*

$$\mathscr{F}[\![f]\!]\subseteq Conc\circ\mathscr{F}^{\#}[\![f]\!]\circ Abs^n,$$

where $\mathscr{F}[\![f]\!]$ is lifted to $P(D)$.

$\mathscr{F}^{\#}[\![f]\!]$ can be computed at compile time by finite fixpoint iteration, yielding some information about f. The following section illustrates the application of this method for computing strictness information.

3.1. Strictness analysis

We will say that a function $f: D^n \rightarrow D$ is strict in its ith argument if

$$\forall d_j \in D \quad f(d_1, \ldots, d_{i-1}, \perp, d_{i+1}, \ldots, d_n) = \perp.$$

Strictness analysis allows us to detect such information. The importance of the analysis is that the parameters in which a function is strict can be passed by value, avoiding the need for building a closure. Not all cases will be discovered because strictness is, in general, undecidable.

The abstract domain is $2 = \{0, 1\}$, with $0 \sqsubset 1$. Intuitively, 0 represents the undefined element (nontermination) and 1 represents possible termination. The abstraction and concretization functions are:

$$Abs: P(D) \rightarrow 2, \qquad Conc: 2 \rightarrow P(D),$$

$$Abs(S) = 0 \text{ iff } S = \{\perp\},$$

$$Conc(0) = \{\perp\}, \qquad Conc(1) = D.$$

Equation (18) translates to

$$c^{\#} = 1 \quad (c \in Con),$$

$$x +^{\#} y = x \wedge y, \text{ etc.},$$

$$if^{\#}(x, y, z) = x \wedge (y \vee z),$$

where we denoted $\mathscr{C}^{\#}[\![p]\!]$ by $p^{\#}$.

Example 3.2.

$$fac(x) = if\ x = 0\ then\ 1\ else\ x * fac(x-1),$$

$$fac^{\#}(x) = (x \wedge 1) \wedge (1 \vee x \wedge fac^{\#}(x \wedge 1)) = x.$$

The equation defining $fac^{\#}$ is not recursive, so there is no need for fixpoint iteration. We can conclude that fac is strict because $fac^{\#}(0) = 0$ which implies, by the correctness theorem of abstract interpretation, $fac(\perp) = \perp$ (more exactly, $\mathscr{F}[\![fac]\!] \perp = \perp$).

We can consider an arbitrary expression to be a function of its free variables. The relation $\downarrow$ (read "is strict in") between expressions and variables is defined as follows.

Definition 3.3. For $e \in Exp_f$ and $x \in Var_f$,

$$e \downarrow x \text{ iff } \mathscr{E}^{\#}[\![e]\!][0/x, 1/y(y \neq x)] = 0.$$

The correctness of strictness analysis implies that

$$e \downarrow x \Rightarrow \forall \rho \in Env . \mathscr{E}[\![e]\!]\rho[\perp/x] = \perp.$$

3.2. Abstractions of the nonstandard semantics

The analyses developed in the rest of the paper are expressed as particularizations of the following general problem. For each expression e, we want to approximate some property of the variables of e which cannot be computed at compile time. The properties that we are interested in can be formulated using the nonstandard semantics defined in Section 2: the variables of e have the desired property iff, whenever we mark them in a certain way, we obtain a certain marker of the (nonstandard) value of e. More exactly, we are interested in the k-ary relations r between variables of the following general form.

Definition 3.4. Let $k \geqslant 1$ and $M_0, M_1, \ldots, M_k, M_{k+1} \in P(M_\perp)$ be fixed.

For $e \in Exp_f$, $x_1, \ldots, x_n \in Var_f$, and $\rho \in Env$,

$$\langle x_1, \ldots, x_k \rangle \in r(e, \rho) \text{ iff } marker(\mathscr{E}_n[\![e]\!]\rho_n) \in M_{k+1},$$

for all $\rho_n \in Env_n$ satisfying:

$$content \circ \rho_n = \rho, \quad marker(\rho_n(x_i)) \in M_i \ (i = 1, \ldots, k),$$
$$marker(\rho_n(x)) \in M_0 \ (x \neq x_i, \ i = 1, \ldots, k).$$

In other words, $\langle x_1, \ldots, x_k \rangle \in r(e, \rho)$ iff the marker of the value of e in a nonstandard environment obtained from ρ by marking x_i with something in M_i (and everything else with something in M_0) is in M_{k+1}. Note that the M_i's are not just sets of markers, but elements of $P(M_\perp)$, i.e., $\perp \in M_i$ (this is necessary because $\perp$ can only be marked with $\perp$). We are interested only in the behavior of terminating computations; $\perp \in M_{k+1}$, therefore, if the evaluation of e in ρ does not terminate, $r(e, \rho)$ is the total relation.

By abstracting the nonstandard semantics, we will obtain a statically computable approximation (a subset) of r which does not depend on an environment. The idea is to ignore the standard values and consider only the markers. The abstract values are sets of possible markers; more exactly, the abstract domain A is an arbitrary subset of $P(M_\perp)$ which contains $M_\perp$ and is closed under set intersection ($A = P(M_\perp)$ is such a domain). Different abstract domains generate, in general, different approximations; the relationship between them is discussed later in this section. For $S \subseteq M_\perp$, let $a(S)$ be the least element of A such that $S \subseteq a(S)$ (it always exists because $M_\perp \in A$ and A is closed under intersection). The abstractization and concretization functions are:

$$Abs = a \circ marker : P(D_n) \to A, \qquad Conc = marker^{-1} : A \to P(D_n). \tag{19}$$

We will use the superscript a to denote the abstractions of the valuation functions. The abstractions of the predefined functions are given by the following lemma.

Lemma 3.5.

$$\mathscr{C}_n^a[\![p]\!] = a \circ \tilde{p} \quad (p \neq \mathit{if}),$$

$$\mathscr{C}_n^a[\![\mathit{if}]\!] = \lambda xyz \,.\, a(\widetilde{\mathit{if}}(x, y) \cup \widetilde{\mathit{if}}(x, z)).$$

Proof. Immediate from (18), (15), (16), and the definitions of *Abs* and *Conc*. □

The definition of the relation r^a is as follows.

Definition 3.6. For $e \in Exp_f$ and $x_1, \ldots, x_n \in Var_f$,

$$\langle x_1, \ldots, x_k \rangle \in r^a(e) \textit{ iff}$$

$$\mathscr{E}_n^a[\![e]\!]\,[a(M_i)/x_i\,(i=1, \ldots, k),\, a(M_0)/x\,(x \neq x_i)] \subseteq M_{k+1}.$$

The correctness of the approximation is given by the following theorem.

Theorem 3.7. *For all $e \in Exp_f$ and $\rho \in Env$,*

$$r^a(e) \subseteq r(e, \rho).$$

Proof. Let $e \in Exp_f$ and $x_1, \ldots, x_n \in Var_f$ such that $\langle x_1, \ldots, x_k \rangle \in r^a(e)$. From Definition 3.6 we have

$$\mathscr{E}_n^a[\![e]\!]\,[a(M_i)/x_i\,(i=1, \ldots, k),\, a(M_0)/x\,(x \neq x_i)] \subseteq M_{k+1}.$$

Conc is monotonic; therefore,

$$Conc(\mathscr{E}_n^a[\![e]\!]\,[a(M_i)/x_i\,(i=1, \ldots, k),\, a(M_0)/x\,(x \neq x_i)]) \subseteq Conc(M_{k+1}),$$

or, using the first equality in (17),

$$Conc(\mathscr{E}_n^a[\![e]\!]\,[Abs(Conc(a(M_i)))/x_i\,(i=1, \ldots, k),$$
$$Abs(Conc(a(M_0)))/x\,(x \neq x_i)]) \subseteq Conc(M_{k+1}).$$

We can now apply Theorem 3.1 to obtain

$$\mathscr{E}_n[\![e]\!][Conc(a(M_i))/x_i\,(i=1, \ldots, k),\, Conc(a(M_0))/x\,(x \neq x_i)] \subseteq Conc(M_{k+1}).$$

Using the definition of *Conc* in (19), this is equivalent to

$$marker(\mathscr{E}_n[\![e]\!]\,\rho_n) \in M_{k+1},$$

for all $\rho_n \in Env_n$ such that $marker(\rho_n(x_i)) \in a(M_i)$, $marker(\rho_n(x)) \in a(M_0)$ $(x \neq x_i$, $i=1, \ldots, k)$. But $a(M_i) \supseteq M_i$, therefore, from Definition 3.4, $\langle x_1, \ldots, x_k \rangle \in r(e, \rho)$ for all $\rho \in Env$. □

The theoretical complexity of computing the abstractions of all user-defined functions by fixpoint iteration is $O(|A|^N)$, with the constant depending on the structure of A (the maximum number of fixpoint iterations is the height of the domain of monotonic functions from A^N to A, which is $O(|A|^N)$). In some instances, due to some special properties of A, the exact complexity can be much lower (such a case will be discussed in the next section).

While decreasing the complexity of the computation, the use of a smaller abstract domain will generate, in general, a weaker approximation (more information is lost by abstraction). More precisely, the approximation over the smaller domain can be obtained by abstract interpretation from the approximation over the larger domain. We have, thus, a hierarchy of approximations corresponding to the hierarchy of subdomains of $P(M_\perp)$. This result is presented in the following lemma.

Lemma 3.8. *If A and A' are two subsets of $P(M_\perp)$ closed under intersection such that $M_\perp \in A \subseteq A'$ then $r^a \subseteq r^{a'}$.*

Proof. The functions

$$Abs: A' \to A, \quad Abs(S') = a(S'),$$

$$Conc: A \to A', \quad Conc(S) = S$$

satisfy the conditions (17); therefore, from Theorem 3.1,

$$\mathscr{E}_n^{a'} [\![e]\!] \rho' \subseteq \mathscr{E}_n^a [\![e]\!] a \circ \rho',$$

for all expressions e and abstract environments ρ' over A'. The lemma is proven by taking $\rho' = [a'(M_i)/x_i \ (i = 1, \ldots, k), a'(M_0) \ /x \ (x \neq x_i)]$. □

Under the conditions specified in the following lemma, the approximation over a smaller domain is the same as the approximation over a larger one. This fact can be used to simplify the abstraction without losing any information.

Lemma 3.9. *If A and A' are two subsets of $P(M_\perp)$ closed under intersection such that $\{M_\perp, M_{k+1}\} \subseteq A \subseteq A'$ and, for all predefined p of n arguments, $a \circ \mathscr{C}_n^{a'} [\![p]\!] = \mathscr{C}_n^a [\![p]\!] \circ a^n$ (as functions from A'^n to A) then $r^a = r^{a'}$.*

Proof. $\mathscr{E}_n^a$ and $\mathscr{E}_n^{a'}$ are the (finite) limits of their fixpoint approximations and the following equality can be proven easily by induction on these approximations:

$$a(\mathscr{E}_n^{a'} [\![e]\!] \rho') = \mathscr{E}_n^a [\![e]\!] a \circ \rho',$$

for all expressions e and abstract environments ρ' over A'. The lemma then follows from the fact that a is monotonic and $a(M_{k+1}) = M_{k+1}$ (because $M_{k+1} \in A$). □

4. Reduction to variables

Under our assumption that expressions evaluate to references (locations, pointers), it is easy to see that the value of an expression e is either (a) a reference to a newly created object, or (b) the reference denoted by some variable x in e. In the second case, we say that e *reduces* to x.

As mentioned before, we assume that no object is copied during evaluation; more precisely, we assume that

(1) *if* never creates a new object but just returns the reference of the selected branch,

(2) all primitive functions except *if* always create a new object as their result, i.e., a call to such a function can never reduce to a variable, and

(3) user-defined functions return the references obtained by evaluating their bodies.

The purpose of the analysis defined in this section is to define a statically computable approximation (superset) of the reduction-to-variables relation. To consider that every expression might reduce to any of its variables is an approximation which is safe, but too coarse to be useful. The analysis is an essential component of the destructive-update algorithm presented in Section 6 (see Examples 2.3 and 2.4).

The standard semantics does not offer all the necessary information – in particular, we cannot determine when new locations are accessed. Consider, for example, the expressions *if true then* x *else* 0 and $x+0$. The standard values of these two expressions are equal, but the first one reduces to x, while the second one generates a new reference.[2] In order to differentiate between such expressions, we will use a particularization of the nonstandard semantics defined in Section 2.4. We will then derive the desired approximation by abstract interpretation.

4.1. Exact reduction to variables

We will denote by $e \Downarrow x(\rho)$ the fact that e reduces to x when evaluated in environment ρ. Using the operational semantics defined in Section 2.3, we can define $\Downarrow$ as follows.

Definition 4.1. For $e \in Exp_f$, $x \in Var_f$, and $\rho \in Env$, $e \Downarrow x(\rho)$ iff all reduction sequences of e in ρ terminate and the last step in any such sequence is a reduction of x based on rule (1).

In order to obtain an equivalent definition without mentioning explicitly the reduction sequences, we will mark the value of x with a special marker which will be propagated to the final result iff rule (1) is used for the last reduction. We will take

$$M = \{old, new\},$$

[2] We will assume that any constant folding is carried out *before* the update analysis.

where *old* is used to mark the variable x and *new* is used for everything else and also for all "newly generated" markers.[3] All primitive functions generate *new* and all constants are marked with *new*; therefore, we define:

$$\tilde{p}(m_1, \ldots, m_n) = \mathit{if}\ \exists i\ m_i = \bot\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{new}\ (p \neq \mathit{if}, n \geqslant 0). \tag{20}$$

The marker generated by *if* is the marker of the respective alternative, i.e.,

$$\widetilde{\mathit{if}}(m_1, m_2) = \mathit{if}\ m_1 = \bot \mathit{then}\ \bot\ \mathit{else}\ m_2. \tag{21}$$

Theorem 4.2. *For all* $e \in Exp_f$, $x \in Var_f$, $\rho \in Env$,

$$e \Downarrow x(\rho)\ \mathit{iff}\ \mathit{marker}(\mathit{Eval}_n(e, \rho_n)) = \mathit{old},$$

where $\rho_n \in Env_n$ *is defined by*

$$\mathit{content}(\rho_n) = \rho, \qquad \mathit{marker}(\rho_n(x)) \sqsubseteq \mathit{old}, \qquad \mathit{marker}(\rho_n(y)) \sqsubseteq \mathit{new} \quad (y \neq x).$$

Proof. The left-to-right implication follows immediately from the definition of $\Downarrow$. We will prove the other implication by induction on the number of reduction steps of e:

$e = c$ (0 reduction steps): $\mathit{marker}(\mathit{Eval}_n(e, \rho_n)) = \tilde{c} = \mathit{new}$ (definition (20)).

e is not a constant: the last step in any finite reduction of e is obtained by one of the rules (1), (12), or (14). In the first case, if the reduced variable is not x, and also in the last case, $\mathit{marker}(\mathit{Eval}_n(e, \rho_n)) \sqsubseteq \mathit{new}$ by the definition of ρ_n and, by definition (20). In the second case, use definition (21) and the induction hypothesis applied to the selected branch of the *if*. □

Corollary 4.3. *For all* $e \in Exp_f$, $x \in Var_f$, $\rho \in Env$, and ρ_n as above,

$$e \Downarrow x(\rho)\ \mathit{iff}\ \mathit{marker}(\mathscr{E}_n [\![e]\!] \rho_n) = \mathit{old}.$$

Example 4.4.

$$\begin{aligned} e_1 &::= \text{if } \mathit{true}\ \mathit{then}\ x\ \mathit{else}\ 0, \\ e_2 &::= x + 0. \end{aligned}$$

For any $\rho \in Env$ such that $\rho(x) \neq \bot$, $e_1 \Downarrow x(\rho)$, $e_2 \not\Downarrow x(\rho)$.

4.2. Approximative reduction to variables

We will obtain now a statically computable approximation of the reduction-to-variables relation defined in the previous section.

Let r be the complement of $\Downarrow$, i.e., the relation "does not reduce to a variable". We are interested in r because we will need an approximation to $\Downarrow$ from above (i.e., with

[3] These names are justified by the fact that we can interpret these markers as special "references". *new* corresponds to the "newly generated" references, while *old* corresponds to all other references. An equivalent analysis can, indeed, be obtained by abstracting a store semantics along this idea.

a weaker relation), which is the same thing as the complement of an approximation of r from below (r^a defined in Section 3.2 is such an approximation). We can put the relation r in the form presented in Section 3.2 by choosing $k=1$, $M_0=M_2=\{\bot, new\}$, $M_1=\{\bot, old\}$. We obtain the following definition:

$$x \in r(e, \rho) \text{ (or } e \not\Downarrow x(\rho)) \text{ iff } marker(\mathscr{E}_n[\![e]\!]\rho_n)) \in \{\bot, new\}$$

for all $\rho_n \in Env_n$ such that $content(\rho_n)=\rho$, $marker(\rho_n(x)) \sqsubseteq old$, $marker(\rho_n(y)) \sqsubseteq new$, $y \neq x$. For the approximation r^a, we will choose the abstract domain

$$A=\{\{\bot, new\}, \{\bot, old, new\}\}.$$

We can check easily that the conditions in Lemma 3.9 (for $A'=P(M_\bot)$) are satisfied, so we do not lose any information by abstracting over A instead of $P(M_\bot)$. Denoting $\{\bot, new\}$ by 0 and $\{\bot, old, new\}$ by 1, we have $A=\{0, 1\}$, with $0 \sqsubset 1$. The abstractions of the primitive functions are obtained from the definitions (20) and (21) using Lemma 3.1:

$$\mathscr{C}_n^a[\![p]\!]=\lambda x_1 \ldots x_n . 0 \quad (p \neq if,\ n \geqslant 0)$$

$$\mathscr{C}_n^a[\![if]\!]=\lambda xyz \,.\, y \vee z.$$

The desired approximation to $\Downarrow$ is the complement of r^a. It will be denoted also by $\Downarrow$; no confusion is possible because the approximation does not depend on an environment.

Definition 4.5. For $e \in Exp_f$ and $x \in Var_f$,

$$e \Downarrow x \text{ iff } x \notin r^a(e) \text{ iff } \mathscr{E}_n^a[\![e]\!]\,[1/x, 0/y\,(y \neq x)]=1.$$

Example 4.6.

$$f(x, y, z)=if\ x=0\ then\ y\ else\ f(x-1, z, y).$$

Let $e::=f(7, v, w)$. If $\rho(w) \neq \bot$ then $e \Downarrow w(\rho)$. $\mathscr{E}_n^a[\![e]\!]=v \vee w$; therefore, $e \Downarrow v$ and $e \Downarrow w$.

The following correctness theorem for $\Downarrow$ is a direct consequence of the correctness of r^a with respect to r.

Theorem 4.7. *For any $e \in Exp$, $x \in Var$, and $\rho \in Env$,*

$$e \Downarrow x(\rho) \Rightarrow e \Downarrow x.$$

Both the strictness relation $\downarrow$ and the reduction to variables relation $\Downarrow$ are defined by abstract interpretation over a two-element domain. The height of the domain of n-argument monotonic functions over this domain is 2^n+1; therefore, 2^N+1 is an upper limit on the number of fixpoint iterations needed to compute the abstraction of an arbitrary function. While the complexity of strictness analysis was indeed proven in [11] to be $O(2^N)$, the complexity of the reduction-to-variables analysis is much lower because its defining abstraction has the following special property:

Lemma 4.8. *For any $e \in Exp_f$ and $\rho_1, \rho_2 \in Env_n^a$,*

$$\mathscr{E}_n^a[\![e]\!](\rho_1 \vee \rho_2) = \mathscr{E}_n^a[\![e]\!]\,\rho_1 \vee \mathscr{E}_n^a[\![e]\!]\,\rho_2.$$

Proof. It is easy to prove by induction on k that the equality holds for all fixpoint approximations $\mathscr{E}_n^{ak}$ of $\mathscr{E}_n^a$, etc. □

Corollary 4.9. *$\mathscr{E}_n^a$ can be computed in O(N) time.*

Proof. Follows from the fact that the height of the domain of n-argument monotonic functions on (0, 1) satisfying

$$f(x_1, \ldots, x_n) \vee f(y_1, \ldots, y_n) = f(x_1 \vee y_1, \ldots, x_n \vee y_n)$$

is n. □

5. Evaluation order

Information about the order in which different expressions will be evaluated when the program is run can be used for several compile-time optimizations. Unfortunately, this order cannot be determined completely at compile time. This is true for all run-time evaluation strategies (assuming, of course, that the strategy preserves the normal-order semantics of the language). This section will explore different ways of defining the evaluation order and methods of obtaining statically computable approximations.

5.1. Exact evaluation order of variables

In this section we will define formally an exact order of evaluation relation between variables and in Section 5.2 we will obtain a statically computable approximation of this relation.

We will assume a pure lazy evaluation strategy, as defined by the operational semantics in Section 2.3; other strategies will be considered in Section 5.3.

We will say that a terminating reduction sequence $e_1 \rightarrow_\rho \cdots \rightarrow_\rho e_n \rightarrow_\rho c$ *evaluates* a variable x at step i if the reduction $e_i \rightarrow_\rho e_{i+1}$ is specified either by rule (1) or by one of the rules (2) or (3) with (1) as precondition. For a given ρ, either all reductions of e evaluate x or none does.

The operational evaluation-order relation $\prec$ between variables is defined as follows.

Definition 5.1. For $e \in Exp_f$, $x, y \in Var_f$, $x \neq y$, $\rho \in Env$:
$x \prec y(e, \rho)$ iff all reductions of e in ρ terminate evaluating both x and y and at least one such reduction evaluates first x and then y.

Example 5.2.

$$e ::= \mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ y+z.$$

For all environments ρ in which e terminates, $x \prec y(e, \rho)$. If also $\rho(x) = \mathit{false}$ then $x \prec z(e, \rho)$, $y \prec z(e, \rho)$, and $z \prec y(e, \rho)$.

This definition of $\prec$ is not very useful since it depends on all steps of all reduction sequences of e. We will develop another definition which depends only on the final results of the reductions by using the nonstandard semantics defined in Section 2.4. Let

$$M = \{m_x, m_y, m_{xy}, m_z\}.$$

The nonstandard semantics is based on the following idea: reduce e in an environment in which x and y are marked with m_x and m_y, respectively, and all other variables are marked with m_z. Define $\tilde{p}$ such that a possible evaluation of x before y will generate the marker m_{xy} which is propagated to the final result. Then $x \prec y(e, \rho)$ iff e reduces to a constant marked with m_{xy}. The definitions of $\tilde{p}$ are:

$$\tilde{p}(m_1 \ldots m_n) = \begin{cases} \bot & \text{if } \exists i\, m_i = \bot, \\ m_x & \text{if } \forall i\, m_i \in \{m_x, m_z\} \wedge \exists i\, m_i = m_x, \\ m_y & \text{if } \forall i\, m_i \in \{m_y, m_z\} \wedge \exists i\, m_i = m_y \\ m_z & \text{if } \forall i\, m_i = m_z, \\ m_{xy} & \text{if } \exists i\, m_i = m_{xy} \vee \exists i, j\, m_i = m_x, m_j = m_y, \end{cases} \tag{22}$$

for $p \neq \mathit{if}$, and

$$\widetilde{\mathit{if}}(m_1, m_2) = \begin{cases} \bot, & \text{if } m_1 = \bot \vee m_2 = \bot, \\ m_x & \text{if } m_1, m_2 \in \{m_x, m_z\} \wedge \langle m_1, m_2 \rangle \neq \langle m_z, m_z \rangle, \\ m_y & \text{if } m_1 = m_y \vee (m_1 = m_z \wedge m_2 = m_y), \\ m_z & \text{if } m_1 = m_2 = m_z, \\ m_{xy} & \text{if } m_1 = m_{xy} \vee (m_1 \neq \bot, m_y \wedge m_2 = m_{xy}) \\ & \quad \vee (m_1 = m_x \wedge m_2 = m_y). \end{cases} \tag{23}$$

Note that $\tilde{c} = m_z$ for all constants c. We can now define $\prec$ in terms of Eval_n without explicitly mentioning the reduction sequences.

Theorem 5.3. *For any $e \in \mathit{Exp}_f$, $x, y \in \mathit{Var}_f$, $\rho \in \mathit{Env}$,*

$$x \prec y(e, \rho)\ \mathit{iff}\ \mathit{marker}(\mathit{Eval}_n(e, \rho_n)) = m_{xy},$$

where

$$\mathit{content} \circ \rho_n = \rho,\quad \mathit{marker}(\rho_n(x)) \sqsubseteq m_x,\quad \mathit{marker}(\rho_n(y)) \sqsubseteq m_y,$$

$$\mathit{marker}(\rho_n(z)) \sqsubseteq m_z\ (z \neq x, y).$$

Proof. Immediate from the following lemma. □

Lemma 5.4. *For any e, x, y, ρ, and ρ_n as in Theorem 5.8, all reduction sequences of e*

(1) *terminate without evaluating either x or y iff $marker(Eval_n(e, \rho_n)) = m_z$;*

(2) *terminate, evaluate x, and do not evaluate y iff $marker(Eval_n(e, \rho_n)) = m_x$;*

(3) *terminate, evaluate y, and either do not evaluate x or evaluate x after y iff $marker(Eval_n(e, \rho_n)) = m_y$.*

Proof. By induction on the number of reduction steps of e. □

Corollary 5.5. *For any $e \in Exp_f$, $x, y \in Var_f$, $\rho \in Env$, and ρ_n as in Theorem 5.3,*

$$x \prec y(e, \rho) \text{ iff } marker(\mathscr{E}_n[\![e]\!]\rho_n)) = m_{xy}.$$

5.2. Approximative order of evaluation

To obtain a statically computable approximation of $\prec$ from above, we will, again, (a) define the complement of $\prec$ as a particularization of the general relation r from Section 3.2, (b) define an approximation r^a, and (c) take the complement of r^a as the desired approximation of $\prec$.

The complement of $\prec$ is a relation r as in Section 3.2 if we take $k = 2$, $M_0 = \{\perp, m_z\}$, $M_1 = \{\perp, m_x\}$, $M_2 = \{\perp, m_y\}$, $M_3 = \{\perp, m_x, m_y, m_z\}$. To obtain the approximation r^a, we will choose the abstract domain

$$A = \{z, xz, yz, xyz, \top\},$$

where

$$z = \{\perp, m_z\}, \quad xz = \{\perp, m_x, m_z\}, \quad yz = \{\perp, m_y, m_z\}, \quad xyz = \{\perp, m_x, m_y, m_z\},$$

$$\top = \{\perp, m_x, m_y, m_z, m_{xy}\}.$$

We can again check that the conditions in Lemma 3.9 are satisfied, so we do not lose any information by choosing this abstract domain instead of $P(M_\perp)$. The abstractions of the primitive functions are obtained from the definitions (22) and (23) using Lemma 3.5:

$$\mathscr{C}_n^a[\![p]\!](x_1 \ldots x_n) = \begin{cases} z & \text{if } \forall i\, x_i = z, \\ xz & \text{if } \forall i\, x_i \subseteq xz, \\ yz & \text{if } \forall i\, x_i \subseteq yz \\ \text{xyz} & \text{if } \exists i\, x_i = xyz \wedge \forall j \neq i\, x_j = z, \\ \top & \text{otherwise,} \end{cases} \tag{24}$$

for $p \neq if$, and

$$\mathscr{C}^a_n[\![if]\!](a, b, c) = \begin{cases} z & \text{if } a=b=c=z, \\ xz & \text{if } a, b, c \sqsubseteq xz, \\ yz & \text{if } (a, b, c \sqsubseteq yz) \vee (a=yz, b, c \neq \top), \\ xyz & \text{if } (a=xyz, b=c=z) \vee (a=z, b, c \neq \top), \\ \top & \text{otherwise,} \end{cases} \tag{25}$$

where, on each line, we assume that the conditions on the previous lines are not satisfied. The maximum number of iterations needed for computing all abstractions is $3 \cdot 5^N + 1$ (the height of the domain of monotonic functions from A^N to A).

The approximation to $\prec$ is the complement of r^a and will be denoted also by $\prec$; no confusion is possible because the approximation does not depend on any environment. From Definition 3.6, we obtain the following definition.

Definition 5.6. For $e \in Exp_f$ and $x, y \in Var_f$,

$$x \prec y(e) \;\; \textit{iff} \;\; \mathscr{E}^a_n[\![e]\!][xz/x, yz/y, z/z\,(z \neq x, y)] = \top.$$

Intuitively, $x \prec y(e)$ if x might be evaluated before y. For $x, y \in Var_f$, we will usually write $x \prec y$ instead of $x \prec y(body_f)$.

Other order relations between variables can be defined in a similar manner. In particular, the following relation will be needed for the destructive-update algorithm.

Definition 5.7. For $e \in Exp_f$, $x, y \in Var_f$, $x \neq y$, $\rho \in Env$:
$x \mathrel{\prec\!\cdot} y(e, \rho)$ iff all reductions of e in ρ terminate, evaluate x, and either (a) no reduction evaluates y, or (b) there is a reduction which evaluates x before y.

Using Lemma 5.4 and the definition of $\prec$, we can characterize $\mathrel{\prec\!\cdot}$ as follows:

$$x \mathrel{\prec\!\cdot} y(e, \rho) \text{ iff } marker(\mathscr{E}_n[\![e]\!]\rho_n)) \in \{m_x, m_{xy}\} \text{ iff } x \prec y(f(e, y), \rho),$$

where f is any function which evaluates its arguments from left to right, e.g.,

$$f(u, v) \;=\; \textit{if } u=u \textit{ then } v \textit{ else } v.$$

This relation can be used to find an approximation for $\mathrel{\prec\!\cdot}$ in terms of the approximation of $\prec$. We can also approximate $\mathrel{\prec\!\cdot}$ directly by abstract interpretation. Using the same markers and the same abstract domain as for $\prec$, we obtain the following approximation.

Definition 5.8. For $e \in Exp_f$ and $x, y \in Var_f$,

$$x \mathrel{\prec\!\cdot} y(e) \text{ iff } \mathscr{E}^a_n[\![e]\!][xz/x, yz/y, z/z\,(z \neq x, y)] \sqsupseteq xz.$$

Intuitively, $x \mathrel{\prec\!\cdot} y(e)$ if there might be a reduction sequence which either evaluates x before y or evaluates x but not y.

5.3. Other evaluation strategies

Assume now that we have some additional information about the evaluation strategies to which the evaluation-order analysis must be applied. A relation $\prec' \subseteq \prec$, which would be valid only for the strategies under consideration, would contain more order information and would yield a sharper analysis.

In particular, we can adapt $\prec$ to evaluation strategies which impose some restrictions on the order in which primitive functions evaluate their arguments. Suppose, for example, that $+$ evaluates its arguments from left to right. This information can be included in the operational semantics defined in Section 2.2 by replacing, for $+$, rule (2) by the rules

$$\frac{e_1 \to_p e_1'}{e_1 + e_2 \to_p e_1' + e_2}, \tag{26}$$

$$\frac{e \to_p e'}{c + e \to_p c + e'} \quad (c \in Con). \tag{27}$$

In the nonstandard semantics defined in Section 5.1, we must change definition (22) for $\tilde{+}$ and set $\tilde{+} = \tilde{if}$ (both specify that the first argument is always evaluated first).

If, on the contrary, we want our evaluation-order analysis to be applicable to a larger set of evaluation strategies than the one considered in the previous sections, we must define a weaker relation $\prec' \supseteq \prec$. For example, we must weaken $\prec$ to make it applicable to the evaluation strategies which might use information from strictness analysis to change the pure lazy order of evaluation. These strategies are used widely in the implementation of functional languages, so the problem of finding a suitable order relation is important.

Example 5.9.

$$e ::= \mathit{if}\ x > 0\ \mathit{then}\ y + x\ \mathit{else}\ y - x.$$

According to our previous definition, $y \not\prec x(e)$ (no reduction evaluates y before x). This is not correct under an evaluation strategy that uses the fact that $e \downarrow y$ to evaluate y before x.

To adapt our operational semantics to an evaluation strategy which uses strictness information to change the order of evaluation, we will replace rule (1) by

$$\frac{e \downarrow x}{e \to_p e[\widehat{\rho(x)}/x]}. \tag{28}$$

Note that (1) is a particular instance of (28); therefore, any reduction in the original semantics is also a reduction in the new semantics.

Unfortunately, we cannot obtain an exact semantics defining the new evaluation-order relation in the way we obtained one for pure lazy evaluation (Section 5.1). The problem can be traced back to rule (12) in the general operational semantics defined in Section 2.4. We would need some information about the unevaluated branch (expression e) which cannot be obtained no matter how we define $\widetilde{if}$. This information, however, can easily be included directly in the abstract semantics if we replace equation (25) by

$$\mathscr{C}_n^a[\![if]\!](a, b, c) = \begin{cases} z & \text{if } a = b = c = z, \\ xz & \text{if } a, b, c \sqsubseteq xz, \\ yz & \text{if } (a, b, c \sqsubseteq yz) \vee (a = yz, b, c \neq \top,\ (b \sqsubseteq yz \vee c \sqsubseteq yz)), \\ xyz & \text{if } (a = xyz, b = c = z) \vee (a = z, b, c \neq \top), \\ \top & \text{otherwise.} \end{cases} \tag{29}$$

5.4. *Access order of variables*

The relation $\prec$ allows us to approximate the order in which variables are evaluated, but not the order in which they are *accessed*. In a graph-reduction-based implementation, the evaluation of a variable takes place when it is first accessed; subsequent references to the variable use its already computed value. A variable is evaluated only once but can be accessed many times. Moreover, for the destructive-update problem, we need to have some information about the order in which *references* denoted by variables are accessed.

Here, and in the rest of the paper, by "expression" we will mean a particular *instance* of an expression; we will assume implicitly that all expressions in a given program are uniquely labeled. We will use integer superscripts to differentiate between occurrences of the same variable; thus, if x is a variable, x^k is an expression.

We will define an evaluation-order relation (also denoted by $\prec$) between variables and *expressions* as follows: if $e', e \in Exp_f$ such that e' is a subexpression of e and $x \in Var_f$,

$$x \prec e'(e) \text{ iff } x \prec w(e[w/e']),$$

where $w \notin Var_f$ is a new variable and $e[w/e']$ is the expression obtained from e by replacing e' by w. Intuitively, $x \prec e'(e)$ if x might be evaluated before e'. If the original $\prec$ is known (i.e., we know the abstractions of all user-defined functions), the new $\prec$ can be computed in one step (no recursion is involved). Evaluation-order relations between expressions and variables and between expressions can be defined similarly. From now on, we will denote by $\prec$ the union of all these relations; arguments of $\prec$ can be, independently, either variables or expressions. The relation $\prec\!\!\cdot$ will be also extended to expressions in a similar way.

We will define now the relation $\prec_a$ between variables such that for $x, y \in Var_f$, $x \prec_a y$ if, during the evaluation of $body_f$, the reference denoted by y might be accessed after x is evaluated. $\prec_a$ is the least fixed point of the following recursive definition.

Definition 5.10. For $x, y \in Var_f$, $x \prec_a y$ iff

(1) there exists an occurrence y^k of y such that $x \prec y^k(body_f)$, or

(2) there exists a function call $h(\dots e_u \dots e_v \dots)$ in $body_f$ such that $x \in Var_{e_v}$, $e_v \Downarrow y$, and $u \prec_a v$ (u, v are the formals of h corresponding to e_u, e_v, respectively).

If $\prec$ is known, $\prec_a$ can be computed in at most N^2 fixpoint iterations. Similarly to $\prec$, we can extend $\prec_a$ to a relation between expressions and variables, also denoted by $\prec_a$. Intuitively, $e \prec_a y$ if the reference denoted by y can be accessed after the evaluation of e.

Example 5.11.

$$f(x, y) = \mathit{if}\ x^1 \geqslant 0\ \mathit{then}\ y^1 + x^2\ \mathit{else}\ y^2 - x^3,$$

$$g(u, v) = f(\mathit{if}\ u^1 = 0\ \mathit{then}\ 0\ \mathit{else}\ u^2, v^1).$$

Assuming pure lazy evaluation, $x \prec y$, $u \prec v$:

$x \prec y^1 + x^2$, $x^2 \prec y^1$, $x^1 \prec y$, etc.;

$x \prec_a y$ (because $x \prec y^1$), $y \prec_a x$ (because $y \prec x^2$), $x \prec_a x$ (because $x \prec x^2$), $u \prec_a v$ (because $u \prec v^1$), $v \prec_a u$ (because $y \prec_a x$ and *if*... $\Downarrow u$), $u \prec_a u$ (because $u \prec u^2$);

$u^1 \prec_a u$ (because $u^1 \prec u^2$), $u^2 \prec_a u$ (because $x \prec_a x$ and *if*... $\Downarrow u$), etc.

If we replace u^2 by $u^2 + 1$ then $v \nprec_a u$, $u^2 \nprec_a u$.

6. Destructive update

The destructive-update problem can be defined informally as follows: given the expression *update*(e_1, e_2, e_3), determine at compile time, if possible, that the object denoted by e_1 will not be referenced after the *update* is performed; in such a case, a compiler can generate code to update in place. The relative order in which references to different objects are accessed depends on the evaluation strategy adopted.

The destructive-update procedure uses the analyses presented in the previous sections. The algorithm is based on the following observation: *update*(e_1, e_2, e_3) can always be done in place if the value of e_1 is not referenced by a variable, for then we are sure that it is not used elsewhere in the program. The other case is when e_1 reduces to a variable x; we must decide now, using evaluation-order information, whether the reference denoted by x is used in the rest of the program. We must also consider all actual arguments corresponding to x and see if they might reduce to a variable, etc.

6.1. The destructive-update algorithm

The following algorithm accepts as input a program P and an expression e' of the form *update*(e, ...) in P and decides whether the *update* can be done in place or not. It uses a set R of variables and two sets of pairs of variables, A and E, with $A \supseteq E$.

Intuitively, $x \in R$ if x might denote the value of e and $\langle x, y\rangle$ is in A (E) if x and y are formals of the same function and x might denote the value of e while y might be accessed (evaluated) after the *update*. The *update* can be done in place only if there is no variable z such that $\langle z, z\rangle \in A$.

Algorithm

(1) Set $R=\{x \mid e \Downarrow x\}$, $A=\{\langle x, y\rangle \mid e \Downarrow x,\ e' \prec_a y\}$ and $E=\{\langle x, y\rangle \mid e \Downarrow x, e' \prec y\}$.

(2) Repeat this step until all variables in R have been considered: choose $x \in R$ not considered so far; suppose $x \in Var_f$. For each expression $e''=f(\ldots, e_x, \ldots)$ (e_x is the actual corresponding to x) and for each variable u such that $e_x \Downarrow u$, set

$$R=R \cup \{u\},$$

$$A=A \cup \{\langle u, v\rangle \mid e'' \prec_a v\},$$

$$E=E \cup \{\langle u, v\rangle \mid e'' \prec v\}.$$

(3) If all pairs in A have been considered then stop; the *update* can be done in place; if $\exists z \in Var$ such that $\langle z, z\rangle \in A$ then stop; the *update* cannot be done in place.

(4) Choose $\langle x, y\rangle \in A$ not considered so far. Suppose $x, y \in Var_f$.
For each expression $e''=f(\ldots, e_x, \ldots, e_y, \ldots)$ (e_x, e_y are the actuals corresponding to x, y) and for each variable u such that $e_x \Downarrow u$, set

$$A=A \cup \{\langle u, v\rangle \mid e_y \Downarrow v\}.$$

If $\langle x, y\rangle \in E$ then set

$$A=A \cup \{\langle u, v\rangle \mid v \in Var_{e_y}\};$$

$$E=E \cup \{\langle u, v\rangle \mid v \in Var_{e_y},\ e_y \prec v\}.$$

Go to step (3).

The execution time of the algorithm is dominated by the time needed to compute $\prec$. Its time complexity is, thus, $O(5^N)$.

Theorem 6.1 (Safety). *Suppose $e' = update(e, \ldots)$ appears in a program P. If the value of e is accessed after e' is evaluated during the execution of P, then the above algorithm will conclude that the update cannot be done in place.*

Proof. In a graph-reduction evaluation model, only the primitive functions other than *if* "destroy" the reference to an actual argument, i.e., neither transmit it to other functions nor propagate it as their result.

A particular use of a particular reference r is characterized by a dynamic sequence of function invocations

$$f_n(\ldots, e_n, \ldots),\ \ldots\ , f_1(\ldots, e_1, \ldots),$$

where the call to f_i takes place in the body of f_{i+1} $(i<n)$, $f_2, \ldots, f_n$ are user-defined functions (not necessarily distinct), and f_1 is a primitive function other than *if*. r is created as the (store) value of e_n, is destroyed by f_1, and is transmitted along this chain as the value of the e_i's. The e_i's collect together all function calls that propagate r. For $i \geqslant 2$, let x_i be the formal parameter of f_i corresponding to e_i. Then, during this sequence of function calls, all x_i's denote r and each e_{i-1} reduces to x_i.

Now let r be the reference to the value of e which is accessed after the *update* and let a sequence as above, with $f_1 = update$ and $e_1 = e$, represent the use of r in *update*.

If r is used after the *update* then there must exist a k_0 such that x_{k_0} is accessed after the *update*. We will prove that, for all $k \geqslant 2$, x_k is added to R, for all variables y_k of f_k which can be accessed after the *update* $\langle x_k, y_k \rangle$ is added to A and, if y_k can be evaluated (i.e., first accessed) after the *update*, it is also added to E. It follows that $\langle x_{k_0}, x_{k_0} \rangle$ will be in A which will cause the algorithm to stop and conclude that the *update* cannot be done in place.

The proof is by induction on k.

(1) $k=2$. f_2 is the function where *update*$(e_1, \ldots)$ appears and $e_1 \Downarrow x_2$. In step 1, x_2 is put into R and, for all variables y_2 which can be accessed (evaluated) after the *update* $\langle x_2, y_2 \rangle$, is added to A (E).

(2) $k>2$. $x_{k-1} \in R$, so x_k is also added to R in step 2. If y_k is accessed after the *update* then either (a) it is accessed after the call to f_{k-1} in which case $\langle x_k, y_k \rangle$ is added to A in step 2 or (b) there exists a variable y_{k-1} of f_{k-1}, accessed or evaluated after the *update*, such that y_{k-1} and y_k play the roles of y and v in step 4 of the algorithm (f, x, and e_x in the algorithm are f_{k-1}, x_{k-1}, and e_{k-1}, respectively). By induction hypothesis, $\langle x_{k-1}, y_{k-1} \rangle$ is in A (E), so $\langle x_k, y_k \rangle$ is added to A in step 4. The proof for E is similar.

6.2. *Examples*

The following example is from [9]:

```
result( ) = quicksort([c₁, ..., cₙ]),

quicksort(vect₁) = qsort(vect₁, 1, length(vect₁)),

qsort(vect₂, first, last)

   = if first ≥ last then vect₂ else scanright(vect₂, first, last,

     vect₂[first], first, last),

scanright(v₁, l₁, r₁, pivot₁, left₁, right₁)

   = if l₁ = r₁ then finish(update(v₁, l₁, pivot₁), l₁, left₁, right₁) else

     if v₁[r₁] ≥ pivot₁ then scanright(v₁, l₁, r₁ − 1, pivot₁, left₁, right₁)

     else scanleft(update(v₁, l₁, v₁[r₁]), l₁ + 1, r₁, pivot₁, left₁, right₁),
```

$$\begin{aligned}
&\mathit{scanleft}(v_2, l_2, r_2, \mathit{pivot}_2, \mathit{left}_2, \mathit{right}_2)\\
&\quad = \mathit{if}\ l_2 = r_2\ \mathit{then}\ \mathit{finish}(\mathit{update}(v_2, l_2, \mathit{pivot}_2), l_2, \mathit{left}_2, \mathit{right}_2)\ \mathit{else}\\
&\qquad \mathit{if}\ v_2[l_2] \leqslant \mathit{pivot}_2\ \mathit{then}\ \mathit{scanleft}(v_2, l_2 + 1, r_2, \mathit{pivot}_2, \mathit{left}_2, \mathit{right}_2)\\
&\qquad \mathit{else}\ \mathit{scanright}(\mathit{update}(v_2, r_2, v_2[l_2]),\ l_2, r_2 - 1, \mathit{pivot}_2, \mathit{left}_2, \mathit{right}_2),\\
&\mathit{finish}(\mathit{vect}_3, \mathit{mid}, \mathit{left}_3, \mathit{right}_3) = \mathit{qsort}(\mathit{qsort}(\mathit{vect}_3, \mathit{left}_3, \mathit{mid} - 1),\\
&\quad \mathit{mid} + 1,\ \mathit{right}_3).
\end{aligned}$$

This program sorts the array $[c_1, \ldots, c_n]$ using the quicksort algorithm. The only information that we assume about the order of evaluation of arguments of predefined functions other than *if* is that the first argument of *update* is evaluated last. The relation $\prec$ on variables is

$$\begin{aligned}
&\mathit{first} \prec \mathit{vect}_2;\ \mathit{last},\ \mathit{last} \prec \mathit{vect}_2, \mathit{first};\\
&\mathit{mid} \prec \mathit{vect}_3, \mathit{left}_3, \mathit{right}_3;\ \mathit{left}_3 \prec \mathit{vect}_3;\ \mathit{right}_3 \prec \mathit{vect}_3, \mathit{mid}, \mathit{left}_3;\\
&v_i \prec \mathit{pivot}_i, \mathit{left}_i, \mathit{right}_i;\ l_i \prec v_i, r_i, \mathit{pivot}_i, \mathit{left}_i, \mathit{right}_i;\ r_i \prec v_i, l_i, \mathit{pivot}_i, \mathit{left}_i, \mathit{right}_i;\\
&\mathit{pivot}_i \prec v_i, \mathit{left}_i, \mathit{right}_i;\ \mathit{left}_i \prec v_i, \mathit{pivot}_i;\ \mathit{right}_i \prec v_i, \mathit{pivot}, \mathit{left}_i, \quad i = 1, 2.
\end{aligned}$$

The relation $\prec_a$ on variables contains all pairs except the following:

$$\begin{aligned}
&\mathit{vect}_3 \not\prec_a \mathit{mid},\ \ \mathit{vect}_3 \not\prec_a \mathit{vect}_3,\ \ \mathit{mid} \not\prec_a \mathit{mid},\ \ \mathit{left}_i \not\prec_a \mathit{left}_i,\ \ \mathit{right}_i \not\prec_a \mathit{right}_i,\\
&\quad i = 1, 2, 3.
\end{aligned}$$

For the first *update* in *scanright*, the algorithm will end with $E = \emptyset$,

$$R = \{v_1, \mathit{vect}_2, \mathit{vect}_3, \mathit{vect}_1\}$$

and

$$\begin{aligned}
A = \{&\langle v_1, l_1\rangle, \langle v_1, \mathit{left}_1\rangle, \langle v_1, \mathit{right}_1\rangle, \langle \mathit{vect}_2, \mathit{first}\rangle, \langle \mathit{vect}_2, \mathit{last}\rangle,\\
&\langle \mathit{vect}_3, \mathit{left}_3\rangle, \langle \mathit{vect}_3, \mathit{right}_3\rangle\}.
\end{aligned}$$

The algorithm will terminate without detecting any conflict, so the *update* can be done in place. For the second *update* in *scanright*, we get the same R,

$$\begin{aligned}
A = \{&\langle v_1, r_1\rangle, \langle v_1, \mathit{pivot}_1\rangle, \langle v_1, \mathit{left}_1\rangle, \langle v_1, \mathit{right}_1\rangle,\\
&\langle \mathit{vect}_2, \mathit{first}\rangle, \langle \mathit{vect}_2, \mathit{last}\rangle, \langle \mathit{vect}_3, \mathit{left}_3\rangle, \langle \mathit{vect}_3, \mathit{right}_3\rangle\}
\end{aligned}$$

and

$$E = \{\langle v_1, \mathit{left}_1\rangle, \langle v_1, \mathit{right}_1\rangle\}.$$

The algorithm will conclude again that the *update* can be done in place. We can prove similarly that the other *update*s can also be done in place, so the optimized program matches the linear space complexity of Hoare's original algorithm.

7. Conclusions and future work

Using a unified framework, we have presented two static analyses for a lazy first-order functional language: reduction to variables and evaluation order. Using these analyses, we developed a practical procedure for the important destructive-update optimization. Both problems are formulated in a general operational semantics and the analyses are obtained by abstract interpretation from a nonstandard denotational semantics equivalent to the operational one. The primary contributions of the paper are the evaluation-order analysis and the methodology of basing the analysis on operational semantics.

The analyses can be extended to higher-order languages using the methods developed in [6, 11]. These methods, originally, were developed for strictness analysis, which is obtained by abstracting the standard semantics, but they can easily be adapted to our nonstandard semantics.

The destructive-update algorithm uses, in an essential way, the fact that the language is first-order; its formulation for higher-order languages is the main topic of our future work. We are also studying the possibility of extending our work to languages with a nonflat basic domain, e.g., to languages which take into account the internal structure of an array.

References

[1] A. Bloss, Path analysis and the optimization of non-strict functional languages, Ph.D. Thesis, Yale University, 1989.

[2] A. Bloss, Update analysis and the efficient implementation of functional aggregates, in: *Proc. 4th Internat. Conf. on Functional Programming and Computer Architecture* (ACM, New York, 1989) 26–38.

[3] A. Bloss and P. Hudak, Variations on strictness analysis, in: *Proc. 1986 ACM Conf. on LISP and Functional Programming* (ACM, New York, 1986) 132–142.

[4] A. Bloss and P. Hudak, Path semantics, *Mathematical Foundations of Programming Language Semantics*, Lecture Notes in Computer Science, Vol. 298 (Springer, Berlin, 1987) 476–489.

[5] A. Bloss, P. Hudak and J. Young, Code optimizations for lazy evaluation, *Lisp and Symbolic Computation* **1** (1968) 147–164.

[6] G.L. Burn, C. Hankin and S. Abramsky, Strictness analysis for higher-order functions, *Sci. Comput. Programming* **7** (1986) 250–278.

[7] M. Draghicescu, and S. Purushothaman, A compositional analysis of evaluation-order and its application, in: *Proc. 1990 ACM Conf. on LISP and Functional Programming* (ACM, New York, 1990) 242–250.

[8] K. Gopinath and J.L. Hennessy, Copy elimination in functional languages, in: *16th ACM Symp. on Principles of Programming Languages* (ACM, New York, 1989) 303–314.

[9] P. Hudak, A semantic model of reference counting and its abstraction, in: S. Abramsky and C. Hankin, eds., *Abstract Interpretation of Declarative Languages* (Ellis Horwood, Chichester, 1987).

[10] P. Hudak and A. Bloss, The aggregate update problem in functional programming systems, in: *12th ACM Symp. on Principles of Programming Languages* (ACM, New York, 1985) 300–314.

[11] P. Hudak and J. Young, Higher-order strictness analysis in untyped lambda calculus, in: *13th ACM Symp. on Principles of Programming Languages* (ACM, New York, 1986) 97–109.

[12] J. Hughes, Analysing strictness by abstract interpretation of continuations, in: S. Abramsky and C. Hankin, eds., *Abstract Interpretation of Declarative Languages* (Ellis Horwood, Chichester, 1987).

[13] A. Mycroft, Abstract interpretation and optimising transformations for applicative programs, Ph.D. Thesis, University of Edinburgh, 1981.

[14] A. Neirynck, P. Panangaden and A. Demers, Computation of aliases and support sets, in: *Proc. 14th ACM Symp. on Principles of Programming Languages* (ACM, New York, 1987) 274–283.

[15] S.L. Peyton Jones, *The Implementation of Functional Programming Languages* (Prentice-Hall, Englewood Cliffs, NJ, 1987).

[16] G.D. Plotkin, LCF considered as a programming language, *Theoret. Comput. Sci.* **5** (1977) 223–255.

[17] D.A. Schmidt, Detecting global variables in denotational specifications, *ACM Trans. on Programming Languages and Systems*, **7** (1985) 299–310.

[18] D.A. Schmidt, Detecting stack-based environments in denotational definitions, *Sci. Comput. Programming* **11** (1988) 107–131.

[19] J.E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory* (MIT Press, Cambridge, MA, 1977).