

How to Make Destructive Updates Less Destructive

Martin Odersky *

18th ACM Symposium on Principles of Programming Languages, January 1991

Abstract

We present a safe embedding of mutable data structures in functional languages. With safety we mean that confluence and (in some sense) referential transparency are maintained. We develop a static criterion based on abstract interpretation which checks that any side-effect which a function may exert via a destructive update remains invisible. The technique opens up the possibility of designing safe and efficient wide-spectrum languages which combine functional and imperative language constructs.

1 Introduction

We are interested in the design of programming notations which augment a functional core with imperative statements and mutable data structures such as arrays. These *wide-spectrum languages* [14] promise to bridge the gap between abstract and declarative specifications/prototypes and efficient implementations. There are advantages to having one notation instead of two for both programming activities: We can derive the program from the specification by local transformations (which can be proved correct in advance). Since we do not switch notations in the process, every intermediate step is a well defined program. Moreover, the transformation process can stop as soon as the result is efficient enough. If the strong points of both programming concepts are to be retained, however, their combination must be *safe*. Safety means that the usual semantics

of functional and imperative programs should be preserved. This is the semantics of predicate transformers in the imperative part and lambda-calculus reduction in the functional part. As the semantics carry over to the combinations, so should the main theorems. In particular, even in the presence of assignment, the functional part should remain confluent and referentially transparent.

Combining functional expressions with statements means that, first, statements such as assignments can contain function applications and, second, that function bodies can contain statements. To obtain safety, we must control all side-effects caused by statements in a function body. An obvious first step is to allow only read access to variables which are declared outside the scope of a function. This is not enough, however. A more complex problem, which we attack in this paper, is presented by arguments to a function which are modified in the function's statement part. Consider the following implementation of function `swap`:

```
swap a i j =  x := a.i ; a.i := a.j ; a.j := x
              ; a.
```

To prevent a side-effect in `swap`, we either would have to implement array updates non-destructively, or we would have to pass array parameters by value. Both choices have a disastrous effect on efficiency. One could optimize array updates by using reference counts [7] or trailers [1]. However, these optimizations impose a sizable run-time overhead themselves, and it is difficult for a programmer to convince themselves of their effectiveness.

We propose instead to choose the efficient implementation for both parameter passing and array updating. Function side-effects are then possible but we will insist that all such effects remain invisible. This means that we have to impose some restrictions on the contexts in which a side-effecting function may be used. For example,

*IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

$f(\text{swap } a \ i \ j) \ (a.i)$

is illegal, since the value of $a.i$ depends on the evaluation order of f 's arguments. A legal application, which avoids the “race condition” would be:

$x := a.i ; f(\text{swap } a \ i \ j) \ x.$

A context is clearly legal if the old value of an updated array a is never again accessed after the update took place. We say in this case: a is *single-threaded* [15]. To check single-threadedness, we develop a static well-formedness criterion for expressions. The criterion guarantees that:

- the evaluation of a well-formed expression is confluent, and
- if a well-formed expression is transformed in a sequence of substitutions of equals into another well-formed expression, then both expressions denote the same value. This is the case even if the result of some intermediate step is not well-formed.

The criterion is decidable using an abstract interpretation of the source program. Indeed, it can be efficiently computed by a compiler. In general, our effect analysis does not need any user-declared effect specifications, as effects can be reconstructed from the program text.

Related Work

Existing designs of wide-spectrum languages have not yet addressed the problem of maintaining confluence in the presence of destructive array updates. Either loss of confluence is accepted [8, 18], or array updates are presumed to be implemented non-destructively [11, 13, 14].

An alternative way to handle the problem of mutations in functional languages relies on static optimization techniques in the compiler. Data structures are then always conceptually immutable, but the optimizer might transform copying updates into in-place updates whenever it is recognized that the transformation is semantics-preserving. Some published techniques for update optimization are [1, 4, 5, 6].

Optimization techniques have the advantage that they do not change the context-rules of the compiled language. Even if optimization fails, the program is still legal. The downside is that a programmer must have a very good understanding of the optimizer's capabilities in order to know which program can be optimized and which cannot. Such knowledge is sometimes crucial since unrecognized opportunities for optimization can add a linear factor to resource consumption. An example is the quicksort function whose space requirements can be $O(\log n)$ or $O(n \log n)$, depending on whether update optimization is successful or not [10].

All published optimization techniques are global optimizations and therefore do not coexist well with separate compilation. Moreover, due to their costs they are applicable only for small programs. Adaptation of any one of these techniques for program-checking purposes would encounter difficulties due to the large (notational and computational) overhead.

Besides optimizations, several program checking techniques have been developed for the update problem. Gifford and Lucassen present in [12] a notational framework which captures information about function side-effects. Their treatment is based on heaps instead of variables and they do not give an effect reconstruction algorithm. Wadler [16] developed linear data types, which admit destructive updates but require every variable to be used exactly once. Again, type information has to be given explicitly, no reconstruction algorithm is presently known. A more recent approach [17], also presented by Wadler, represents an array as an abstract data type on which only one operation, *init*, is defined. *init* creates a fresh array, applies a state transformer (an argument function) to it, and subsequently discards the array while returning a computed result (which is never an array). This interesting approach has the advantage that the single-reference property is established by design. However, arrays can be manipulated only in a restricted manner. They cannot be passed to functions and state transformers can operate only on a single array at a time.

The problem we tried to solve in this paper is very similar to the one addressed by work of Hudak and Guzmán [9]. They present a type-system to capture single-threadedness, whereas our method is based on abstract interpretation. We simplify their approach in that we are able to drop without loss of precision their distinction between single- and multiple-threaded accesses. We extend their results with better approximations to the sharing behavior of a program. We model sharing configurations by an inclusion hierarchy over the storage which is reachable from bound variables. Sharing can be quite complex, as in the case where an array contains a list whose elements are again arrays. Our analysis is able to detect in that case that modifications to the root array do not change the values of the list elements, whereas modifications to a list element do change the values of all variables referring to it. Such accuracy is definitely needed for the analysis of languages with a fully orthogonal type system in which mutable and immutable types can be freely combined. The main restriction with respect to the work of Hudak and Guzmán is that we consider here only first-order languages. The reason for this restriction is that the issues of automatic effect reconstruction in the presence of higher-order functions are not yet fully resolved (this holds for our

approach as well as theirs). Automatic effect reconstruction faces the problem that, in contrast to types in the Hindley-Milner system [3], a function does not always have a most general effect.

The rest of this paper is organized as follows: Section 2 presents a small language with both imperative and functional components on which the subsequent treatment is based. Section 3 introduces abstract domains used in the analysis. Section 4 presents the abstract interpretation function. Section 5 discusses human-readable representations. Section 6 concludes.

2 An Example Language

To base the following discussion on a concrete foundation, we introduce a small single assignment language with the following syntax:

$$\begin{array}{l}
 e ::= e \text{ where } \{f \{x\} = e\} \\
 \quad | \quad x \\
 \quad | \quad f\{e\} \\
 \quad | \quad \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
 \quad | \quad x := e_1 ; e_2.
 \end{array}$$

The first four clauses define a purely functional first-order language with local definitions, bound variables, function applications, and a conditional expression. The only statement in the language is the assignment, whose syntax is given in the fifth clause. Variables may not be assigned to more than once. Our method can be extended to more general functional/imperative combinations by mapping them into the above language. The transformation is similar in spirit to a translation into single assignment form [2].

We assume lazy evaluation in general, except for the evaluation of sequential composition. The expression $x := e_1 ; e_2$ is evaluated by first reducing e_1 to normal form and then reducing e_2 . If reduction terminates, the value of this construct is equal to e_2 where $x = e_1$. We say that “;” is hyperstrict in its left argument.

Structured data types in our language are lists or arrays. Lists are representatives of more general algebraic data types found in most functional languages. They are constructed with the operator *cons* and taken apart with the selectors *hd* and *tl*. Arrays have dynamic size; their indices are integers. Arrays are created and manipulated with the following operators:

<code>vec l</code>	array of (references to) list <i>l</i> ’s elements,
<code>upd a i x</code>	array with (a reference to) element <i>x</i> at index position <i>i</i> which is otherwise equal to <i>a</i> ,
<code>a.i</code>	the <i>i</i> ’th component of <i>a</i> .

Our type system is fully orthogonal. The component type of an array may be any type. The elements of a list may also assume any type, including an array type.

3 The Abstract Domains

Since the implementation of updates is intended to be destructive, we have to ascertain that the “old” value of an array is never accessed after an update. A static technique for doing so is presented in this section and the next. We will present an abstract interpretation which yields for every expression a description of its effect on all variables of a program. The effect on a single variable shall be captured in an abstract use. Some requirements for this *Use* domain can be derived from the following examples, (1) – (7). Assume that *f* is a function which accesses both of its elements without modifying them. Nothing is known about the order of accesses in *f*, and hence, because of lazy evaluation, about the order of argument reduction.

(1)	<code>f a a</code>	<i>safe</i>
(2)	<code>f a (upd a i x)</code>	<i>unsafe</i>
(3)	<code>g c c where</code> <code> g a b = f a (upd b i x)</code>	<i>unsafe</i>
(4)	<code>b := a ; f b (upd a i x)</code>	<i>unsafe</i>
(5)	<code>b := c.(a.i) ; f b (upd a j x)</code>	<i>safe</i>
(6)	<code>b := a.i ; f b (upd a j x)</code>	<i>safe</i>
(7)	<code>b := a.i ; f b (upd a.i j x)</code>	<i>unsafe</i>

(1) is clearly safe since nothing is updated. (2) is unsafe, since the second argument might be evaluated before the first one, thereby producing a visible side-effect. Cases (1) and (2) can be distinguished by having different abstract uses *rd* for read and *wr* for write accesses. Line (3) shows that effect information has to be propagated beyond function boundaries. Our abstract interpretation will map every function to an effect signature, which maps argument liabilities to a result liability.

Since (*:=*) means reference assignment, (4) is essentially the same situation as (2), i.e. unsafe. The similar looking case (5) however, is safe. Because of the hyperstrict evaluation of “;” the read access of value *a* occurs before *a* is updated. To distinguish cases (4) and (5) we need to keep information about variable aliases. This is done by having a value *id* for aliasing reads in the *Use* domain as well as *rd* (*rd* guarantees absence of aliasing).

Line (6) presents some difficulties. In the assignment $b := a.i$, should *a* be abstracted to *rd* or to *id*? *rd* seems to be too optimistic, since *b* does contain a reference to a part of *a*. Indeed, abstracting *a* to *rd* would lead to

acceptance of the (unsafe) case (7). If we abstract a to id , however, we get the same analysis as in case (4), leading to a rejection of an expression which is perfectly safe. (6) is safe, because an update of an array does leave (the storage occupied by) its elements unchanged. Hence, the modification of a by evaluation of f 's second argument does not interfere with the first argument, b , which is bound to $a.i$. This is true in our standard semantics which assumes uniform access by reference. It would also be true if array elements were uniformly updated and retrieved by value. It would not be true for an implementation in which array elements are assigned by value but retrieved by reference, but then, this is not the semantics we consider here.

The example shows that we have to distinguish between variables and their elements in the abstract domain. This is done by mapping every variable to a pair of uses $u_1.u_2$, one for the variable itself and another one for its elements. The value of every expression is also split into the value itself and its elements. Hence, there are three combinations of sharing: An entity (which stands for either a variable or the elements of one) might be shared by the expression's value, or it might be shared by the value's elements, or it might be shared by both. The cases correspond to the abstract uses id , el , and sh .

To summarize: The effect on an entity x (either a bound variable or the elements of one) of evaluating an expression e is abstracted to:

- $-$, if x is not accessed,
- rd , if x is read, but does not form part of e 's normal form,
- id , if x is a potential alias of e 's value,
- el , if x is a potential element of e 's value,
- sh , if x is potentially an alias or an element of e 's value,
- wr , if x is potentially updated (overwritten) during evaluation of e (in this case, the old value of x may of course be neither an alias nor an element of the result),
- \top , in case of conflicting accesses, where confluence might be lost.

The partial ordering of Use is depicted in Figure 1. The total effect of an expression is abstracted to a *liability*, a mapping from the program's bound variables to pairs of uses.

$$Lia = BVar \rightarrow Use \times Use$$

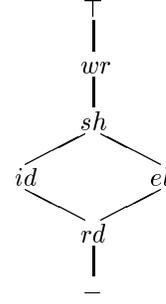


Figure 1: The Use domain

A pair of uses u_1 and u_2 will be written $u_1.u_2$. Given that the liability of expression e in variable x is $u_1.u_2$, u_1 represents the effect of e on x itself while u_2 represents the effect on x 's elements. In the product domain, all pairs $x.\top$ and $\top.y$ are merged into the single error element \top .

Remark: The term “liability” (instead of “effect”) emphasizes the point that every non-bottom use in the liability of an expression restricts the contexts in which this expression may occur.

Example: The liabilities of some simple expressions are:

$$\begin{aligned}
 x & : x \mapsto id.el \\
 x + y & : x \mapsto rd.rd, y \mapsto rd.rd \\
 cons\ x\ y & : x \mapsto el.el, y \mapsto el.el \\
 a.i & : a \mapsto rd.sh, i \mapsto rd.rd \\
 upd\ a\ i\ x & : a \mapsto wr.el, i \mapsto rd.rd, x \mapsto el.el.
 \end{aligned}$$

4 Computation of Liabilities

This section presents our abstract interpretation for effect analysis. First, we need some operators on uses and liabilities. The liability of a compound expression is computed from the liabilities of its constituents using the combinators \sqcup (least upper bound), \parallel (parallel composition) and $;$ (sequential composition). The latter two are defined by:

$$\begin{aligned}
 \top \parallel y & = \top \\
 x \parallel \top & = \top \\
 - \parallel y & = y \\
 x \parallel - & = x \\
 x \parallel y & = \top, & \text{if } x = wr \vee y = wr \\
 & = x \sqcup y, & \text{otherwise}
 \end{aligned}$$

$$\begin{aligned}
\top ;; y &= \top \\
x ;; \top &= \top \\
- ;; y &= y \\
x ;; - &= wr && \text{if } x = wr \\
&= rd, && \text{if } x \neq wr \\
x ;; y &= \top, && \text{if } x = wr \\
&= y, && \text{if } x \neq wr
\end{aligned}$$

All operators defined in this section are assumed to be extended pointwise to the liability domain where appropriate.

A function $f \ x_1 \dots x_n$ is abstracted to an effect signature of the form:

$$\lambda l_1, \dots, l_n. l_{global} \ || \ (lb \ x_1) : l_1 \ || \ \dots \ || \ (lb \ x_n) : l_n.$$

That is, the effect of a function application is the parallel composition of the function's effect on every parameter (lb stands for the liability of its body), and its effect l_{global} on non-local variables. This amounts to regarding a function as a “black box”, in which nothing is known about execution paths. The expression $x : l$ represents the result of applying a function with an effect x on its formal parameter to an argument expression whose liability is l . “:” is defined by:

$$\begin{aligned}
\top : x.y &= \top \\
u_1.u_2 : \top &= \top \\
u_1.u_2 : x.y &= (u_1.u_2 : x).(u_1.u_2 : y) \\
-, - : x &= - \\
u_1.u_2 : x &= x, && \text{if } x \in \{-, rd, wr\} \\
u_1.u_2 : id &= u_1 \\
u_1.u_2 : el &= u_2 \\
u_1.u_2 : sh &= u_1 \sqcup u_2.
\end{aligned}$$

If a function accesses local variables only, we can combine its type and effect signature. We write argument uses before argument types. For example, the signatures of the predefined functions are:

$$\begin{aligned}
(+ &:: rd.rd \ \text{Int} \rightarrow rd.rd \ \text{Int} \rightarrow \text{Int} \\
&\quad \text{(and likewise for } -, *, \dots) \\
\text{cons} &:: el.el \ \alpha \rightarrow el.el \ [\alpha] \rightarrow [\alpha] \\
\text{hd} &:: rd.sh \ [\alpha] \rightarrow \alpha \\
&\quad \text{(and likewise for tl)}
\end{aligned}$$

$$\begin{aligned}
\text{vec} &:: rd.el \ [\alpha] \rightarrow \text{Array } \alpha \\
(.) &:: rd.sh \ \text{Array } \alpha \rightarrow rd.rd \ \text{Int} \rightarrow \alpha \\
\text{upd} &:: wr.el \ \text{Array } \alpha \rightarrow rd.rd \ \text{Int} \\
&\quad \rightarrow el.el \ \alpha \rightarrow \text{Array } \alpha.
\end{aligned}$$

The sequential composition $x := e_1; e_2$ is computed by evaluating first e_1 and then e_2 . Accordingly, the liability of the whole expression is the sequential composition of the liability l_1 of e_1 and the liability l_2 of e_2 . Inside e_2 , variable x is associated with a liability which reflects the normal form of e_1 , and which is computed by a composition of the function $norm$ with l_1 .

$$\begin{aligned}
norm \ u &= -, && \text{if } u \in \{rd, wr\} \\
&= u, && \text{otherwise}
\end{aligned}$$

Note that $norm$ is not monotonic. We will therefore have to give proof that the abstract interpretation as a whole is well-defined. Figure 2 details our effect analysis. There are two abstract interpretation functions: L_D , which abstracts sets of function definitions to function environments and L , which abstracts expressions to liabilities. The functions take two environments as parameters, one for bound variables and one for functions. We assume that all bound variable names and all function names in a program are distinct from each other.

Notation: The list abstraction $[f \ x \mid x \leftarrow xs]$ stands for f applied to all elements x of list xs . Reduction of a list with a binary operator such as \parallel or \sqcup is expressed by prefixing the list with the operator. All operators which we use in this context are associative and have an identity element. Reduction of an empty list yields the identity of the reducing operator. The function zip takes two lists and combines corresponding elements in a list of pairs. Mappings from variable names V to some domain D are expressed as follows: The expression $x \mapsto d$ denotes the function which maps variable $x \in V$ to $d \in D$ and which maps all other variables in V to $-_D$. $-_{V \rightarrow D}$ stands for the function which maps all variables in V to $-_D$. Other mappings can be constructed from these with the operators \sqcup (pointwise upper bound) and \circ (function composition). $f|_V$ denotes the restriction of function f to sub-domain V .

A program is well-formed if its liability maps no bound variable to \top :

$$wf \llbracket e \rrbracket = \forall x \in BVar : L \llbracket e \rrbracket \ \text{initfe} \ -_{BEnv} \ x \neq \top.$$

The initial bound variable environment maps all variables to $-$. The initial function environment has been presented above.

Syntactic Domains

$x \in BVar$	(bound variables)
$f \in FVar$	(functions)
$e \in Exp$	(expressions)

Semantic Domains

$Use = \{-, rd, id, el, sh, wr, \top\}$	
$Lia = BVar \rightarrow Use \times Use$	(liabilities)
$BEnv = BVar \rightarrow Lia$	(bound variable environments)
$FEnv = FVar \rightarrow Lia^* \rightarrow Lia$	(function environments)

Interpretation Functions

$L_D :: Dcl^* \rightarrow FEnv \rightarrow BEnv \rightarrow FEnv$
$L :: Exp \rightarrow FEnv \rightarrow BEnv \rightarrow Lia$

$L_D \llbracket dcls \rrbracket fe\ be$	$= fe'$
whererec	
fe'	$= \sqcup [L_F d \mid d \leftarrow dcls] \sqcup fe$
$L_F \llbracket f\ xs = b \rrbracket$	$= f \mapsto \lambda ys. \parallel [L_B x : y \mid (x, y) \leftarrow zip\ xs\ ys] \parallel L_B _{BVar \setminus xs}$
where	
L_B	$= L \llbracket b \rrbracket fe' be'$
be'	$= \sqcup [x \mapsto (x \mapsto id.el) \mid x \leftarrow xs] \sqcup be$
ys	is a list of fresh variables, s.t. $\text{length } ys = \text{length } xs$
$L \llbracket e\ \text{where } dcls \rrbracket fe\ be$	$= L \llbracket e \rrbracket (L_D \llbracket dcls \rrbracket fe\ be)\ be$
$L \llbracket x \rrbracket fe\ be$	$= be \llbracket x \rrbracket$
$L \llbracket f\ es \rrbracket fe\ be$	$= (fe \llbracket f \rrbracket) [L \llbracket e \rrbracket fe\ be \mid e \leftarrow es]$
$L \llbracket x := e1 ; e2 \rrbracket fe\ be$	$= L \llbracket e1 \rrbracket fe\ be \ ; \ ; \ L \llbracket e2 \rrbracket fe\ be'$
where	
be'	$= (x \mapsto ((x \mapsto id.el) \sqcup norm \circ (L \llbracket e1 \rrbracket fe\ be))) \sqcup be$
$L \llbracket \text{if } e1\ \text{then } e2\ \text{else } e3 \rrbracket fe\ be$	
	$= L \llbracket e1 \rrbracket fe\ be \ ; \ ; \ (L \llbracket e2 \rrbracket fe\ be \sqcup L \llbracket e3 \rrbracket fe\ be)$

Figure 2: Effect Analysis

The non-monotonic behavior of *norm* might raise some doubts whether our abstract interpretation function is well-defined. Indeed, L_F fails to be monotonic. This can be seen by considering the expression

$$E \equiv x := f y ; \\ \text{cons (upd } x \ 0 \ 0) \ (\text{upd } x \ 0 \ 0).$$

If the function environment fe associates f with $\lambda l.(id.el : l)$ (that is, f is typed $id.el \ T_1 \rightarrow T_2$), we have:

$$L \llbracket E \rrbracket fe be = x \mapsto \top, y \mapsto \top$$

On the other hand, assuming $L_F \llbracket f \rrbracket = \lambda l.(wr.el : l)$, we get:

$$L \llbracket E \rrbracket fe be = x \mapsto \top, y \mapsto wr$$

Hence, here is an example where $fe < fe'$ but $L \llbracket E \rrbracket fe be > L \llbracket E \rrbracket fe' be$. Fortunately, the non-monotonic behavior of L_F is restricted to error cases, i.e. situations where both result values contain \top . L_F can be shown to be monotonic in a setting where all error states of domain Lia are merged into a single element. This suggests that we should work in a lifted liability domain \overline{Lia} whose elements are equivalence classes of liabilities. The equivalence classes are given by:

$$\bar{l} = \top_{\overline{Lia}}, \quad \text{if } \exists x : l x = \top \\ = \{l\}, \quad \text{otherwise.}$$

All functions on liabilities are lifted to \overline{Lia} in the obvious way. The lifting is well-defined, as shown by the following reasoning: $\top_{\overline{Lia}}$ is the only equivalence class in \overline{Lia} which corresponds to more than one element of Lia . Since all operators on abstract uses preserve \top_{Use} , we have that all operators on liabilities map error liabilities to error liabilities. Hence, every lifted function on \overline{Lia} maps $\top_{\overline{Lia}}$ to a unique value, namely itself.

Theorem 1 (Monotonicity) In the lifted domain \overline{Lia} , for all $e \in Exp$, $be \in BEnv$, $fe, fe' \in FEnv$:

$$fe \leq fe' \Rightarrow L_F \llbracket e \rrbracket fe be \leq L_F \llbracket e \rrbracket fe' be$$

The proof of Theorem 1 is given in appendix A. From Theorem 1, and the fact that all domains are finite, Theorem 2 follows.

Theorem 2 (Computability) wf is well-defined and computable.

Example: We compute the liability of function *scatter*, which updates array a with elements of array b . The update indices come from list xs and are translated by table c .

$$\text{scatter } xs \ a \ b \ c = \\ \text{if } xs = [] \ \text{then } a \\ \text{else } \text{scatter } (\text{tl } xs) \\ \quad (\text{upd } a \ (c.(hd \ xs)) \ (b.(hd \ xs))) \\ \quad b \ c$$

According to our rule for function definition, the liability $l_{scatter}$ of *scatter* is:

$$l_{scatter} = \\ \lambda l_{xs} \ l_a \ l_b \ l_c. \\ (lb \ xs) : l_{xs} \ || \ (lb \ a) : l_a \ || \ (lb \ b) : l_b \ || \ (lb \ c) : l_c.$$

The liability lb of the body of *scatter* is:

$$lb = xs \mapsto rd.rd ; ; \\ ((a \mapsto id.el) \sqcup \\ \quad l_{scatter} \ (xs \mapsto rd.sh) \\ \quad (a \mapsto wr.el, b \mapsto rd.el, \\ \quad \quad c \mapsto rd.rd, xs \mapsto rd.rd) \\ \quad (b \mapsto id.el) \\ \quad (c \mapsto id.el) \\).$$

Substituting the definition of $l_{scatter}$ and rearranging gives:

$$lb = (xs \mapsto rd.rd ; ; \\ \quad ((lb \ xs) : rd.sh \ || \ (lb \ a) : rd.rd) \\ \quad a \mapsto id.el \sqcup (lb \ a) : wr.el \\ \quad b \mapsto (lb \ a) : rd.el \ || \ (lb \ b) : id.el \\ \quad c \mapsto (lb \ a) : rd.rd \ || \ (lb \ c) : id.el \\)$$

The least fixpoint of this equation is:

$$xs \mapsto rd.rd, a \mapsto wr.el, b \mapsto rd.el, c \mapsto rd.rd.$$

Translated into prose, we have that array a is written, that the elements of array b are elements of the result, and that xs and c are read. This describes precisely our intuitive understanding of the function's effect.

5 Representation

In this section, we discuss the human-readable representation of liabilities and effect signatures. We would expect that liabilities are in general computed automatically, but there are also situations where their representation in human-readable form is required. First, if single-threadedness is violated, the precise cause of the error has to be pinpointed by the compiler. Communication in the other direction is needed if we adopt a module system like Haskell’s [11]. The compilation rules of Haskell are such that the clients of a module may be submitted to compilation before the module itself is completed; the only requirement is that the module’s interface be defined. In the absence of an implementation, the programmer has to define the types and liabilities of all exported entities. Finally, because our technique is a checking algorithm (as opposed to an optimization), programmers have to anticipate the results of the checker, that is they have to reason about liabilities.

We should therefore look for a notation which is as concise and natural as possible. The *UsePair* domain which we have employed in the last two sections leaves something to be desired in this respect. It is rather large, comprising $6^2 + 1$ states. Some of these states can never be reached. For example, the element-part of a use-pair can never be *id*, since the element-part stands for elements of differing nesting depths. Generally, the use-pair notation is better suited for automatic effect analysis than for human reasoning.

We have found it useful to employ another representation, in which all use-pairs of interest are given names of their own. We thus have to write just one identifier instead of two but pay the price of a larger *Use* domain. Table 1 lists the interesting use-pairs together with their descriptive names and example expressions. Every row describes a use value and contains an example in which the variable x is abstracted to this use.

copy is used if an expression’s result contains the elements of a variable, but not the variable itself. *select* is used if a selection yields an element of a variable. *alias* and *include* are used for variables which are identical to the result or an element of it. *share* stands for any combination of *select*, *alias*, and *include*. Of the various *write*- values, only *write* itself is likely to occur often in liabilities. It describes the effect of an update on a variable. *writeread* and *writeshare* stand for cases where an updated variable is subsequently discarded; only its elements are read or shared by the result. *deepwrite* is used for variables whose elements are modified. Figure 3 presents some examples of effect signatures of predefined and user-defined functions.

Name	Pair	Example
–	–,–	
<i>read</i>	<i>rd.rd</i>	length x
<i>copy</i>	<i>rd.el</i>	vec x
<i>select</i>	<i>rd.sh</i>	$x.i$
<i>include</i>	<i>el.el</i>	upd $a\ i\ x$
<i>alias</i>	<i>id.el</i>	x
<i>share</i>	<i>sh.sh</i>	if p then x else tail x
<i>writeread</i>	<i>wr.rd</i>	length (upd $x\ i\ y$)
<i>write</i>	<i>wr.el</i>	upd $x\ i\ y$
<i>writeshare</i>	<i>wr.sh</i>	(upd $x\ i\ y$) j
<i>deepwrite</i>	<i>wr.wr</i>	upd ($x.j$) $i\ y$
\top	$\top.\top$	

Table 1: Interesting use-pairs

upd	::	<i>write</i> Array $\alpha \rightarrow$ <i>read</i> Int \rightarrow <i>include</i> $\alpha \rightarrow$ Array α
vec	::	<i>copy</i> [α] \rightarrow Array α
cons	::	<i>include</i> $\alpha \rightarrow$ <i>include</i> [α] \rightarrow [α]
hd	::	<i>select</i> [α] \rightarrow α
swap	::	<i>write</i> Array $\alpha \rightarrow$ <i>read</i> Int \rightarrow <i>read</i> Int \rightarrow Array α
scatter	::	<i>read</i> [α] \rightarrow <i>write</i> Array $\alpha \rightarrow$ <i>copy</i> Array $\alpha \rightarrow$ <i>read</i> Array $\alpha \rightarrow$ Array α
histogram	::	<i>read</i> Int \rightarrow <i>copy</i> [(Int, α)] \rightarrow Array α
histogram n xs	=	accum (vec[0 $i \leftarrow [0..n - 1]$]) xs
accum	::	<i>write</i> Array Int \rightarrow <i>copy</i> [(Int, α)] \rightarrow Array Int
accum a []	=	a
accum a (Cons (i, x) xs)	=	$y := a.i ;$ upd $a\ i\ (x + y)$

Figure 3: Example effect signatures

6 Discussion

We have presented a framework which allows us to reason about side-effects in a language which combines functional expressions with destructive array updates. The framework gives us a criterion for programs without observable side-effects.

The two main simplifications of the abstract semantics relative to the standard semantics are:

- functions are black boxes, outside of which nothing is known about execution paths, and
- all elements of a structured value are treated the same.

One might think of relaxing the second simplification by maintaining more information about structure elements, for example by distinguishing elements according to inclusion level. This might be useful for analyzing updates of structure elements. Our current analysis just marks as written the containing variable and all its elements. However, considerably more knowledge is needed for analyzing these “deep updates”. If we update a structure element we usually want to re-link the updated datum into the enclosing structure. In our example language, this would be achieved by something like

$$\text{upd } a \text{ i } (\text{upd } (a.i) \text{ j } x)$$

The expression is safe only if all elements of a are distinct (which could be detected by an extension of our analysis along the lines of Wadler’s linear data types [16]), and the two ‘i’ indices agree. Since the indices might be arbitrary expressions, a useful analysis must be able to deduce equality of expressions, at least in some restricted cases.

Also desirable would be an extension of effect analysis to higher order functions. One problem in this area is that functional arguments do not always have a most general effect signature. Consider the function declaration

$$F \text{ f } g \text{ a} = \text{cons } (\text{upd } a \text{ i } x) (\text{f } (g \text{ a}))$$

This function is confluent only if a is not “passed through” $f \circ g$. That is, either f or g can have an effect signature $id.el \tau_1 \rightarrow \tau_2$, but not both of them. This shows that effect signatures do not possess the principal type property. Therefore, the best information an effect reconstruction algorithm could produce is a set of admissible signatures.

A more pragmatic immediate fix to the problem is also possible: One could require the effect signatures of function valued parameters to be declared. If no declaration is present, unrestricted sharing but no writing could be assumed as a default, in order to stay compatible with the purely functional case. Our existing analy-

sis could then check conformance with the programmer-supplied declarations.

Acknowledgements

This work was inspired by a talk of Paul Hudak in which he presented a preliminary version of [9]. It benefitted greatly from subsequent discussions with Paul Hudak and Juan-Carlos Guzmán.

A Monotonicity of the Abstract Interpretation Functions

The appendix contains a proof of Theorem 1, which states that function L_F of Figure 2 is monotonic in its function environment argument. From this theorem the computability of our abstract interpretation function L follows.

The main complication in the proof arises from the non-monotonic behavior of function $norm$. This prevents a simple structural induction over the form of expressions and mandates an analysis of the context in which $norm$ appears.

Theorem 1 (Monotonicity) In the lifted domain \overline{Lia} , for all $e \in Exp$, $be \in BEnv$, $fe, fe' \in FEnv$:

$$fe \leq fe' \Rightarrow L_F \llbracket e \rrbracket fe be \leq L_F \llbracket e \rrbracket fe' be$$

Proof: We first introduce a simplification. Rather than regarding a liability as a mapping from variables to use-pairs, we split every variable into two parts, each of which is mapped to a single use. A variable x is split into x_{id} and x_{el} , and, given that the liability l of x is $u_1.u_2$, we define $l \ x_{id} = u_1$ and $l \ x_{el} = u_2$. This is clearly only a notational modification.

The proof proceeds by structural induction on the form of expressions. With the exception of $norm$, all operators which construct liabilities are monotonic. The only nontrivial case is hence the sequential composition, which looks in our simplified model as follows:

$$L \llbracket x = e_1 ; e_2 \rrbracket fe be = L \llbracket e_1 \rrbracket fe be ;; L \llbracket e_2 \rrbracket fe be'$$

where

$$be' = (x \mapsto ((x_{id} \mapsto id) \sqcup (x_{el} \mapsto el) \sqcup norm (L \llbracket e_1 \rrbracket fe be))) \sqcup be.$$

To give names to the parts of these equations, we define:

$$\begin{aligned} h &= \lambda l. (x_{id} \mapsto id) \sqcup (x_{el} \mapsto el) \sqcup norm \ l, \\ g &= \lambda l. L \llbracket e_2 \rrbracket fe ((x \mapsto l) \sqcup be), \\ f &= \lambda l. l ;; (g \circ h) \ l. \end{aligned}$$

We have then:

$$L \llbracket x = e_1 ; e_2 \rrbracket fe be = f(L \llbracket e_1 \rrbracket fe be).$$

In the following, we will mostly reason in the original liability domain Lia but will need sometimes also the lifted domain \overline{Lia} . To distinguish both cases, we will overline operators in \overline{Lia} . Our task is to show that \overline{f} is monotonic in its liability argument. i.e:

$$l_1 \leq l_2 \Rightarrow \overline{f} l_1 \leq \overline{f} l_2. \quad (1)$$

To show (1), we take a closer look at the possible forms of f , g , and h . Given fixed environments fe and be , all these functions are L-Mappings, i.e. they satisfy Definition 1.

Definition 1

An *L-Mapping* is a function $\Phi : Lia \rightarrow Lia$ which is of one of the forms (i) – (v).

- (i) $\Phi = id$
- (ii) $\Phi = K l'$
- (iii) $\Phi = (u :) \circ \Phi_1$
- (iv) $\Phi = \Phi_1 \odot \Phi_2$
- (v) $\Phi = id ;; (\Phi_1 \circ (\Phi_2 \sqcup norm \circ \Phi_3))$

Here, $u \in Use$, $l' \in Lia$, Φ_1 and Φ_2 are L-Mappings, and \odot is one of $\sqcup, ||, ;;$.

Two useful properties of L-Mappings are:

Lemma 1

For all L-Mappings Φ , $x_1, x_2 \in BVar$, $u_1, u_2 \in Use$:

- (a) $\Phi((x_1 \mapsto u_1) \sqcup (x_2 \mapsto u_2)) = \Phi(x_1 \mapsto u_1) \sqcup \Phi(x_2 \mapsto u_2)$
- (b) $x_1 \neq x_2 \Rightarrow \Phi(x_1 \mapsto u_1) x_2 = \Phi(x_1 \mapsto u_2) x_2$

The proofs of properties (a) and (b) are simple inductions over the form of L-Mappings given in Definition 1. They are left out here since they present no technical difficulties.

Proof of Theorem 1 continued: Because of property (a), we can rewrite f and \overline{f} as follows:

$$\begin{aligned} f l &= \bigsqcup_{z \in BVar} f(z \mapsto l z) \\ \overline{f} l &= \bigsqcup_{z \in BVar} \overline{f}(z \mapsto l z) \end{aligned}$$

Hence, \overline{f} is monotonic if we can show that:

$$\begin{aligned} \forall z \in BVar, u_1, u_2 \in Use : & \quad (2) \\ u_1 \leq u_2 \Rightarrow \overline{f}(z \mapsto u_1) & \leq \overline{f}(z \mapsto u_2) \end{aligned}$$

The only non-trivial case in equation (2) is given by $u_1 \in \{id, el\}$, $u_2 = wr$. This is the only case where function *norm* shows non-monotonic behavior, namely *norm* $u_1 = u_1$ and *norm* $u_2 = -$. We have to show that the decrease in *norm* is “cancelled out” in the whole of \overline{f} , i.e. that the following holds:

$$\overline{f}(z \mapsto id) \leq \overline{f}(z \mapsto wr) \quad (3)$$

$$\overline{f}(z \mapsto el) \leq \overline{f}(z \mapsto wr) \quad (4)$$

A case analysis is used to show (3).

Case 1: $\overline{f}(z \mapsto id) \neq \top_{\overline{Lia}}$.

In this case, $\overline{f}(z \mapsto id) = f(z \mapsto id)$. Let $y \in BVar$. We have to show that

$$f(z \mapsto id) y \leq f(z \mapsto wr) y.$$

If $y \neq z$, this follows from Lemma 1(b), with “ \leq ” strengthened to “ $=$ ”. On the other hand, if $y = z$, we get:

$$\begin{aligned} & f(y \mapsto id) y \\ & \leq \text{(case assumption)} \\ & \quad wr \\ & \leq \text{(properties of ;;)} \\ & \quad ((y \mapsto wr) ;; (g \circ h)(y \mapsto wr)) y \\ & = \text{(definition of } f) \\ & \quad f(y \mapsto wr) y \end{aligned}$$

Case 2: $\overline{f}(z \mapsto id) = \top_{\overline{Lia}}$. To show in this case:

$$\overline{f}(z \mapsto wr) = \top_{\overline{Lia}} \quad (5)$$

From the case assumption follows that there exists $y \in BVar$ such that $f(z \mapsto id) y = \top_{Use}$. If $y \neq z$, Lemma 1(b) yields $f(z \mapsto wr) y = \top_{Use}$, which implies (5). To prove (5) in case $y = z$, we need inequality (6):

$$\begin{aligned} (g \circ h)(y \mapsto wr) y = - & \Rightarrow \quad (6) \\ (g \circ h)(y \mapsto id) y & \leq (g \circ h)(y \mapsto id) x_{id}. \end{aligned}$$

The proof of (6) proceeds by structural induction on the definition of the L-Mapping g .

Case 2.2 (i): $g = id$

$$\begin{aligned}
& (id \circ h) (y \mapsto id) y \\
= & \text{(definition of } h) \\
& ((x_{id} \mapsto id) \sqcup (x_{el} \mapsto el) \sqcup (y \mapsto id)) y \\
= & \text{(\beta-reduction)} \\
& id \\
= & \text{(\beta-abstraction)} \\
& ((x_{id} \mapsto id) \sqcup (x_{el} \mapsto el) \sqcup (y \mapsto id)) x_{id} \\
= & \text{(definition of } h) \\
& (id \circ h) (y \mapsto id) x_{id}
\end{aligned}$$

Case 2.2 (ii): $g = K l'$

$$\begin{aligned}
& (K l' \circ h) (y \mapsto id) y \\
= & \text{(\beta-reduction)} \\
& l' y \\
= & \text{(\beta-abstraction)} \\
& (K l' \circ h) (y \mapsto wr) y \\
= & \text{(premise of (6))} \\
& - \\
\leq & (K l' \circ h) (y \mapsto id) x_{id}
\end{aligned}$$

Case 2.2 (iii): $g = (u :) \circ \Phi$

$$\begin{aligned}
& ((u :) \circ \Phi) (y \mapsto id) y \\
= & \text{(\beta-reduction)} \\
& u : (\Phi (y \mapsto id) y) \\
\leq & \text{(induction hypothesis, monotonicity of ':')} \\
& u : (\Phi (y \mapsto id) x_{id}) \\
= & \text{(\beta-abstraction)} \\
& ((u :) \circ \Phi) (y \mapsto id) x_{id}
\end{aligned}$$

The proofs of cases (iv) and (v) are like the proof of case (iii) simple induction steps; we leave them out for brevity. This concludes our proof of proposition (6). But (6) implies (5) as shown by the following reasoning:

$$\begin{aligned}
& f (y \mapsto id) y = \top \\
\Rightarrow & \text{(definition of } f) \\
& ((y \mapsto id) ;; (g \circ h) (y \mapsto id)) y = \top \\
\Rightarrow & \text{(properties of } ;;) \\
& (g \circ h) (y \mapsto id) y = \top \\
\Rightarrow & \text{(6)} \\
& (g \circ h) (y \mapsto wr) y \neq - \vee \\
& (g \circ h) (y \mapsto id) x_{id} = \top
\end{aligned}$$

$$\begin{aligned}
\Rightarrow & \text{(properties of } ;;, \text{ Lemma 1(b))} \\
& ((y \mapsto wr) ;; (g \circ h) (y \mapsto wr)) y = \top \vee \\
& (g \circ h) (y \mapsto wr) x_{id} = \top \\
\Rightarrow & \text{(definition of } f) \\
& f (y \mapsto wr) y = \top \vee f (y \mapsto wr) x_{id} = \top \\
\Rightarrow & \text{(definition of } \bar{f}) \\
& \bar{f} (y \mapsto wr) = \top_{\overline{Lia}}
\end{aligned}$$

This concludes our proof of (3). The proof of (4) differs only in that occurrences of x_{id} are replaced by x_{el} . (3) and (4) together imply (2) and with it the main theorem.

References

- [1] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proc. 4th Conf. on Functional Programming and Computer Architecture*, August 1989.
- [2] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. An efficient method for computing static single assignment form. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, January 1989.
- [3] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212, January 1982.
- [4] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, January 1990.
- [5] M. Draghicescu and S. Purushothaman. A compositional analysis of evaluation order and its application. In *Proc. ACM Conference on LISP and Functional Programming*, June 1990.
- [6] W.L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation* 2(3/4), October 1989.
- [7] K. Gharachorloo, V. Sarkar and J.L. Hennessy. A simple and efficient implementation approach for single assignment languages. In *Proc. ACM Conference on LISP and Functional Programming*, 1988.
- [8] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. 5th ACM Symposium on Principles of Programming Languages*, 1978.

- [9] J.C. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proc. 5th IEEE Symposium on Logic in Computer Science*, June 1990.
- [10] 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 Ltd., 1987.
- [11] P. Hudak and P. Wadler (editors). Report on the functional programming language Haskell. Technical Report YALEU/DCS/RR666, Yale University, Department of Computer Science, November 1988.
- [12] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, January 1988.
- [13] J. McGraw et al. SISAL: streams and iterators in a single assignment language, language reference manual. Technical Report M-146, LLNL, March 1985.
- [14] The Munich project CIP, volume 1: The wide spectrum language CIP/L. Springer Verlag LNCS 183, 1985.
- [15] D.A. Schmidt. Denotational semantics as a programming language. Internal report CSR-100, Computer Science Department, University of Edinburgh, 1982.
- [16] P. Wadler. Linear types can change the world! *Proc. IFIP TC2 Working Conference on Programming Concepts and Methods*, April 1990.
- [17] P. Wadler. Comprehending monads. In *Proc. ACM Conference on LISP and Functional Programming*, June 1990.
- [18] J.H. Williams and E.L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line?. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, January 1988.