Static Analyses for Guarded Optimizations of High Level Languages *

Pavol Černý[1] ** and Natarajan Shankar[2]

[1] ENS
Paris 75005
France
Pavol.Cerny@ens.fr
[2] Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com

Abstract. Functional programs are typically less efficient than imperative programs in terms of execution time and space. However, the situation can be improved for large classes of programs by doing static analysis at compile time. The most important bottleneck in performance of purely functional programs comes from non-destructive updates of aggregate structures. Static analysis can determine whether destructive updates can be safely used instead. Preliminary work to this end has already been done for PVS [?]. We have continued and improved on this work by increasing the range of destructive update optimization while decreasing the complexity of the analysis. In particular, no fixpoint computations on the structure of a program is needed in our analysis. We present also two other analyses that treat problems arising from overhead caused by multiple representations of arrays (closure, destructive, non-destructive) and from coupling/decoupling overhead of arguments of a function. The results obtained are quite encouraging. The typical performance of a simple optimized program is within a linear factor of two of the corresponding C program (compiled without using optimizations), with a similar space utilization. There is also a relatively small loss of performance when exploiting parametricity.

1 Introduction

Functional programming languages have many important advantages, including simple semantics, expressiveness (higher-order expressions) and referential transparency, while the performance is usually a drawback. However, the simple and rigorously defined semantics makes it significantly easier to reason about programs, and to optimize programs in a verifiably safe manner automatically. In this work we exploited the second possibility, optimizations.

* Funded by NSF Grant No. CCR-0082560 and SRI International
** Visiting SRI International

In the static analysis we present, the following general pattern can be identified. For a function definition given by: $f(x_1, \ldots, x_n) = e$, we analyze the expression e statically to construct:

- An optimized version $f^O(x_1, \ldots, x_n)$.
- Conditions for the use of the optimized version. These conditions must be verified by the actual parameters of the function, and/or the context in which it is used.

We call this pattern "guarded optimizations".

We present these static analyses methods for the executable fragment of the specification language PVS. [**?**]. [3] This fragment is essentially a strongly typed, higher-order language with an eager order of evaluation. The methods can be easily adapted to other functional languages, including those with a lazy evaluation order.

PVS is a widely used framework for specification and verification. By optimizing functions defined in the PVS specification language, specifications can be executed for the purposes of animation, validation, code generation, and fast simplification. The techniques is presented for a small functional language fragment of PVS.

The analysis methods are *interprocedural*. Each function definition is analyzed solely in terms of the *results* of the analysis of the previously defined functions and not their actual definitions. We also outline a proof of the correctness of the static analyses that are used.

We treat three problems that create important bottlenecks in runtime performance of functional programs. The first concerns the use of non-destructive updates. Purely functional programs do not allow notations for destructive updates. However, static analysis can determine whether destructive updates can be used safely instead. Secondly, not all updates can be made destructive, thus multiple array representations have to be used (destructive form, nondestructive form, closure). Thus a priori runtime checks are necessary before every access to an array. Static analysis (destructive variable analysis) can improve on this overhead. Third problem comes from overhead caused by coupling/decoupling of arguments of a function (arity analyses), as explained in Section **??**.

Our approach builds on the work of Shankar [**?**], and it inherits several advantages over previous approaches to update analysis. It is simple, efficient, interprocedural, and has been implemented for an expressive functional language. The implementation yields code that is competitive in performance with efficient imperative languages. The proofs of correctness are simple enough that the method can be adapted to other languages with only modest changes to the correctness argument. Here, we extend and improve on the work of N. Shankar by:

[3] The PVS system and related documentation can be obtained from the URL http://pvs.csl.sri.com. The presentation in this paper is for a generic functional language and requires no prior knowledge of PVS. The notation used is also somewhat different from that of PVS.

– Employing a notion of a flow analysis in presenting the analysis.
– Reducing the computational complexity of analysis algorithms, by using fixpoint computation on the table representing the results of flow analysis, not on the structure of the program.
– The analysis presented here easily allows construction of more aggressive safe expressions for recursive function definitions.
– Destructive variable analysis and arity analysis (cf. below).

These extensions are the novel contributions of this paper.

We have implemented our methods as part of a code generator for an executable functional fragment of the PVS specification language which generates Common Lisp programs. Functional programs, such as sorting procedures, written in this fragment of PVS execute at speeds that are roughly a factor of two slower than the corresponding programs written in C (compiled without optimizations), and with comparable space usage.

2 Update analysis

We present several simple examples in order to motivate and clarify the ideas and introduce the concepts. For this purpose, we use a simple language that we introduce informally. The full analysis given is for a higher-order language that includes lambda-abstractions. A function is defined as $f(x_1, \ldots, x_n) = e$ where e contains no free variables other than those in $\{x_1, \ldots, x_n\}$. Let `Arr` be an array from the subrange `[0..9]` to the integers. Let A, B, C be variables of type `Arr`. An array lookup is written as $A(i)$ for $0 \leq i \leq 9$. An array update has the form $A[(i) := a]$ and represents a new array A' such that $A'(i) = a$ and $A'(j) = A(j)$ for $j \neq i$. Pointwise addition on arrays $A + B$ is defined as the array C such that $C(i) = A(i) + B(i)$ for $0 \leq i \leq 9$.

In order to be safe, destructive updates have to verify certain conditions. For the first example we take the following function definition:

$$f_1(A) = A + A[(3) := 4].$$

Let I be an array such that $I(a) = a$.
If we evaluate the expression $f_1(I)(3)$, we obtain 7. But if we replace the update by a destructive one, the function f_1, we obtain $f_1(I)(3) = 8$. When executing $f_1(A)$, the update to A cannot be carried out destructively since the original array is an argument to the + operation. The evaluation of $A[(3) := 4]$ must return a reference to a new array that is a suitably modified copy of the array A.

Thus the optimization is safe, when array A is not referenced in the context where it is updated, as is the case in the following example:

$$f_2(A, B) = A + B[(3) := 4].$$

The optimization assumes that array B is not referenced in the context where $f_2(A, B)$ is evaluated. For example, in the definition:

$$f_3(A, B) = B + f_2(A, B).$$

it is unsafe to execute f_2 so that B is updated destructively since there is a reference to the original B in the context when $f_2(A, B)$ is evaluated.

In the case of $f_2(A, B)$, there is also a second condition on the use of the optimized version of a function: A and B must not be bound to the same array reference. This happens for instance in the definition:

$$f_4(A, B) = f_2(A, A).$$

However, the following definition of f_5, satisfies both of the conditions mentioned previously.

$$f_5(A, B) = A + f_2(A, B).$$

It is thus safe to replace the call to f_2 by a call to a destructive version f_2^D.

Let us consider now the case of recursive functions. On the following example we will see that there can be a certain tradeoff between making the recursive calls destructive and, on the other hand, making some of the updates in the function destructive. In the following definition

$$\begin{aligned} f_{rec}(n, A, B, C) = \texttt{IF } n = 0 \texttt{ THEN } & A + B[(3) := 4] \\ \texttt{ELSE } & C[(n) := 4](3) + f_{rec}(n, A, A, C)(3) \end{aligned}$$

we see that if we make the updates of the array (referenced by) B and C destructive, the recursive call cannot be executed destructively, because A and B are aliased to the same array. On the other hand, if we make the recursive call destructive, the update of B cannot be made destructive, for precisely the same reasons. (Of course the update of C can still be destructive.) We choose to make all the recursive calls destructive.

In order to compute the destructive version f^D of a function f, together with conditions for its use, we need the following notions. We introduce them informally.

1. A specific occurrence of a destructive update u of the form $e_1[(e_2) := e_3]$ in an expression e can be identified by decomposing e as $U\{u\}$, where U is an *analysis context* containing a single occurrence of the *hole* $\{\}$, and $U\{u\}$ is the result of filling the hole with the update expression u.
2. A *substitution* σ. When the expression $U\{u\}$ is evaluated, the free variables in it are bound to values through some substitution σ, and the free mutable variables are bound to values containing references.
3. For all variables x of a function f, the set $FA(f)(x)$ of variables. Let us denote by σ_0 the substitution in the moment when f is called. If it is possible that $\sigma_0(x) \neq \sigma_0(y)$ and there is a recursive call of f such that for the substitution σ valid in that moment $\sigma(x) = \sigma_0(y)$, then $y \in FA(f)(x)$.
4. The set $Lv(U)$ of *live* mutable variables in the update context U. The set $Lv(U)$ contains such variables x that $\sigma(x)$ is present in the partially evaluated context U' when the subexpression u is evaluated. This is a subset of the mutable variables in U.
5. The set $Ov(e)$ of the output array variables in e contains those array variables x such that the reference $\sigma(x)$ is a possible value of $\sigma(e)$.

6. The set $Av(e)$ of the output array variables in e contains those array variables x such that the reference $\sigma(x)$ is is present after e is evaluated. The set $Av(e)$ includes $Ov(e)$, but it contains also variables that are trapped in closures.
7. For all variables x of a function f, the set $LA(f)(x)$ has the following property: For every update expression u in a context U in the definition of f, $Lv(U) \subseteq LA(f)(x)$. Thus $LA(f)$ should be used as the guard in the guarded optimization f^D - it takes into account the possibility of aliasing, as it was the case of the example function f_4.
8. The set Lv_{rec} contains the variables that are live in the contexts where recursive calls are being evaluated.
9. A notion of *safe* destructive update. Informally, the update is safe, when the update does not influence the environment in which it is evaluated, i.e. if the array that is being updated is not live in the analysis context. More formally, we have:

   $$f(x_1, \ldots, x_n) = U\{e_1[(e_2) := e_3]\}.$$

   The update is safe, if

   $$FA(f)(Lv(U)) \cap (FA(f)(Ov(e_1)) \cup Lv_{rec}) = \emptyset.$$

   There is a similar condition for safe destructive function calls.

We demonstrate the use of the above notions on the example of the function f_rec.

$$f_{rec}(n, A, B, C) = \texttt{IF } n = 0 \texttt{ THEN } A + B[(3) := 4] \\ \texttt{ELSE } C[(n) := 4](3) + f_{rec}(n, A, A, C)(3)$$

We have:

$$FA(f_{rec})(A) = \{A\}$$
$$FA(f_{rec})(B) = \{B, A\}$$
$$FA(f_{rec})(C) = \{C\}$$

since A 'can become' B by the recursive call.
Also, $Lv_{rec}(f) = \emptyset$.
Then, for the context where B is updated, we have

$$Lv(U) = \{A\}$$
$$Ov(e_1) = \{B\}$$
$$FA(Lv(U)) = \{A\}$$
$$FA(Ov(e_1)) = \{B, A\}.$$

thus, we obtain

$$FA(Lv(U)) \cap (FA(Ov(e_1)) \cup Lv_{rec}) = \{B, A\} \neq \emptyset,$$

so B cannot be destructively updated. On the other hand, analysis shows that C can be updated destructively and, since we used FA in our analysis, the recursive call can also be destructive.

In the following subsections we define the above notions formally.

2.1 Language

We describe a small functional language and an update analysis procedure for this language that generates a destructive counterpart to each function definition. The language is strongly typed. Each variable or function has an associated type. Type rules identify the well-typed expressions. The type of a well-typed expression can be computed from the types of its constituent subexpressions. Types are exploited in the analysis, but the principles apply to untyped languages as well.

The base types consist of `bool`, `integer`, and *index types* of the form $[0 < \kappa]$, where κ is a numeral. The only type constructor is that for function types which are constructed as $[T_1, \ldots, T_n \rightarrow T]$ for types $T, T_1, \ldots, T_n$. The language admits subtyping so that $[0 < i]$ is a subtype of $[0 < j]$ when $i \leq j$, and these are both subtypes of the type `integer`. A function type $[S_1, \ldots, S_n \rightarrow S]$ is a subtype of $[T_1, \ldots, T_n \rightarrow T]$ iff $S_i \equiv T_i$ for $0 < i \leq n$, and S is a subtype of T. We do not explain more about the type system and the typechecking of expressions. Readers are referred to the formal semantics of PVS [**?**] for more details. An array type is a function type of the form $[[0 < i] \rightarrow W]$ for some numeral i and base type W, so that we are, for the present, restricting our attention to flat arrays.

The metavariable conventions are that W ranges over base types, S and T range over types, x, y, z range over variables, p ranges over primitive function symbols, f and g range over defined function symbols, a, b, c, d, and e range over expressions, L, M, N range over sets of array variables.

The expression forms in the language are

1. Constants: Numerals and the boolean constants `TRUE` and `FALSE`.
2. Variables: x
3. Abstraction: $(\lambda(x_1 : T_1, \ldots, x_n : T_n) : e)$, is of type $[T_1, \ldots, T_n \rightarrow T]$, where e is an expression of type T given that each x_i is of type T_i for $0 < i \leq n$. We often omit the types $T_1, \ldots, T_n$ for brevity.
4. Application: $e(e_1, \ldots, e_n)$ is of type T where e is of type $[T_1, \ldots, T_n \rightarrow T]$ and is either an expression, a defined operation f, or a primitive operation p (assumed to be nondestructive), and each e_i is of type T_i. Note that primitive and defined operations are not by themselves expressions.
5. Conditional: `IF` e_1 `THEN` e_2 `ELSE` e_3 is of type T, where e_1 is an expression of type `bool`, and e_2, and e_3 are expressions of type T.
6. Update: A nondestructive update expression $e_1[(e_2) := e_3]$ is of type $[[0 < i] \rightarrow W]$, where e_1 is of array type $[[0 < i] \rightarrow W]$, e_2 is an expression of type $[0 < i]$, and e_3 is an expression of type W. A destructive update expression $e_1[(e_2) \leftarrow e_3]$ has the same typing behavior as its nondestructive counterpart.

2.2 Analysis context

An *analysis context* U is an expression containing a single occurrence of a hole $\{\}$. An update context U has one of the forms

1. $\{\}$.
2. $\{\}(e_1, \ldots, e_n)$.
3. $e(e_1, \ldots, e_{j-1}, \{\}, e_{j+1}, \ldots, e_n)$.
4. $\texttt{IF}(\{\}, e_2, e_3)$, $\texttt{IF}(e_1, \{\}, e_3)$, or $\texttt{IF}(e_1, e_2, \{\})$.
5. $\{\}[(e_2) := e_3]$, $e_1[(\{\}) := e_3]$, or $e_1[(e_2) := \{\}]$.
6. $\{\}[(e_2) \leftarrow e_3]$, $e_1[(\{\}) \leftarrow e_3]$, or $e_1[(e_2) \leftarrow \{\}]$.
7. $U\{V\}$ for analysis contexts U and V.

The primary observation about analysis contexts is that the hole $\{\}$ can occur anywhere except within a lambda-abstraction. Note that the context of evaluation of an update expression within a lambda-abstraction is not easily calculated. For ease of explanation, the definition of analysis contexts above is conservative in not allowing holes $\{\}$ to occur within lambda-abstractions that occur in function positions of beta-redexes, e.g., let-expressions, even though the context of evaluation for the hole can be exactly determined.

2.3 Flow analysis

The idea of the computation of flow analysis is to collect for each variable x_i all the variables that reference an array, that could be later referenced by x_i. Thus, in a first-order setting, we would simply have:

$$FA^0(f)(x_i) = \{y \mid \exists\, U, a_1, \ldots, a_n :\ e^D \equiv U\{f^D(a_1, \ldots, a_n)\} \\ \wedge\, y \in Av(a_i)\}$$

We then compute $FA(f)$ as the transitive closure of the relation given by $FA^0(f)$, by any suitable standard algorithm:

$$FA(f) = Rtc(FA^0(f))$$

However, in a higher-order functional language, we need to consider "more hidden" recursive calls, such as

$$\ldots g((\lambda z_1, \ldots, z_n : f(z_1, B, \ldots, z_n)))(e_1, \ldots e_k) \ldots \tag{1}$$

Therefore, the definition of $FA^0(f)$ is more involved. In the definition of $FA^0(f)$, we also need the function $Av(e)$ that computes the active variables of the expression e and two auxiliary function.
The definitions below use the application form, defined as

$$(\lambda(x_1, \ldots, x_n) : S)(S_1, \ldots, S_n) = (S - \{x_1, \ldots, x_n\}) \cup \bigcup\{S_i | x_i \in S\}$$
$$S(S_1, \ldots, S_n) = S \cup S_1 \cup \ldots \cup S_n$$

Active variable analysis function. $Av(e)$ contains the set of free variables that persist in the value of e. It is an over-approximation of active variables.
$Avr(e)$, where e is a function definition, $f(x_1, \ldots, x_n)$ is a lambda-abstracted set

of abstract variables:

$$
\begin{aligned}
Avr(x) &= \{x\},\ \text{if } x \\
Avr(f_i) &= \emptyset Avr(a(a_1, \ldots, a_n)) \qquad = Avr(a)(Av(a_1), \ldots, Av(a_n)) \\
Avr(\lambda(x : [0 < i]) : e) &= \emptyset \\
Avr(\lambda(x_1, \ldots, x_n) : e) &= (\lambda(x_1, \ldots, x_n) : Av(e)) \\
Avr(\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3) &= Av(e_2) \cup Av(e_3) \\
Avr(a_1[(a_2) := a_3]) &= \emptyset \\
Avr(a_1[(a_2) \leftarrow a_3]) &= \emptyset \\
Av((\lambda(x_1, \ldots, x_n) : e)) &= Mv((\lambda(x_1, \ldots, x_n) : e)) \\
Av(a) &= Avr(a),\ \text{otherwise}
\end{aligned}
$$

Function Fclos. $Fclos(f, e)$ analyzes the expression e syntactically to determine whether it contains the symbol f trapped in a closure, that is not applied (i.e. is used as an argument to a function).

Flow analysis.

$$
\begin{aligned}
Flr(f)(x)(x_i) &= \emptyset \\
Flr(f)(f_i)(x_i) &= \emptyset \\
Flr(f)(a(a_1, \ldots a_n))(x_i) &= Flr(f)(a)(x_i)(Av(a_1), \ldots, Av(a_n)) \\
&\quad \cup Fl(a_1)(x_i) \cup \ldots \cup Fl(a_n)(x_i)) \\
&\quad \wedge \neg Fclos(f)(a(a_1, \ldots, a_n)) \\
Flr(f)(f(a_1, \ldots a_n))(x_i) &= Av(f)(a_i) \cup \\
&\quad Fl(a_1)(x_i) \cup \ldots \cup Fl(a_n)(x_i)) \\
Flr(f)(a(a_1, \ldots a_n))(x_i) &= Av(a_1) \cup \ldots \cup Av(a_n) \cup \\
&\quad Fl(f)(a)(x_i) \cup Fl(a_1)(x_i) \cup \ldots \cup Fl(a_n)(x_i)) \\
&\quad \text{otherwise} \\
Flr(f)(\lambda(x_1, \ldots, x_n) : e)(x_i) &= \lambda(x_1, \ldots, x_n) : Fl(f)(e)(x_i) \\
Flr(f)(\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3)(x_i) &= Fl(f)(e_1)(x_i) \cup Fl(f)(e_2)(x_i) \cup Fl(f)(e_3) \\
Flr(f)(e_1[e_2 \leftarrow e_3])(x_i) &= Fl(f)(e_1)(x_i) \cup Fl(f)(e_2)(x_i) \cup Fl(f)(e_3) \\
Flr(f)(e_1[e_2 := e_3])(x_i) &= Fl(f)(e_1)(x_i) \cup Fl(f)(e_2)(x_i) \cup Fl(f)(e_3) \\
Fl(f)(a)(x_i) &= S - \{x_1, \ldots, x_n\}, \\
&\quad \text{if } Fl(f)(e) = \lambda(x_1, \ldots, x_n) : S \\
Fl(f)(e) &= Flr(a),\ \text{otherwise}
\end{aligned}
$$

Given a sequence of definitions of functions $f_1, \ldots, f_m$, the table $FA(f_k)$ is, for each function f_k with variables $x_1, \ldots, x_n$, a map from the variable index i to the flow analysis for the definition of f_k and the variable x_i, i.e., $FA(f_k)(x_i) = Fl(f_k)(i)$.

We compute $FA(f)$ as the transitive closure of the relation given by $FA^0(f)$, by any suitable standard algorithm:

$$\begin{aligned} FA^0(f)(x_i) &= Fl(f)(e)(x_i) \\ FA(f) &= Rtc(FA^0(f)) \end{aligned}$$

2.4 Output & active variable analysis

Now we can compute output analysis table $OA(f_i)$ and active variable analysis table $AVA(f_i)$. Note that no fixpoint computation is needed, since we use the flow analysis table.
$Ov(e)$ is the set of free variables that possibly share structure with the value of e. $Ovr(e)$, where e is a function definition, $f(x_1, \ldots, x_n)$ is a lambda-abstracted set of output variables:

Output analysis function.

$$\begin{aligned} Ovr(x) &= \{x\} \\ Ovr(f_i) &= OA(f_i) \\ Ovr(a(a_1, \ldots, a_n)) &= Ovr(a)(Ov(a_1), \ldots, Ov(a_n)) \\ Ovr(\lambda(x : [0 < i]) : e) &= \emptyset \\ Ovr(\lambda(x_1, \ldots, x_n) : e) &= (\lambda(x_1, \ldots, x_n) : Ov(e)) \\ Ovr(\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3) &= Ov(e_2) \cup Ov(e_3) \\ Ovr(a_1[(a_2) := a_3]) &= \emptyset \\ Ovr(a_1[(a_2) \leftarrow a_3]) &= \emptyset \\ Ov(a) &= S - \{x_1, \ldots, x_n\}, \text{ if} \\ & \qquad Ovr(a) = (\lambda(x_1, \ldots, x_n) : S) \\ Ov(a) &= Ovr(a), \text{ otherwise} \end{aligned}$$

Output analysis table.

$$\begin{aligned} OA^0(f_i) &= \lambda(x_1, \ldots, x_n) : \emptyset \\ OA(f_i) &= \lambda(x_1, \ldots, x_n) : FA(f)(S) \\ & \qquad \text{if } Ovr(\lambda(x_1, \ldots, x_n) : e)) = \lambda(x_1, \ldots, x_n) : S \end{aligned}$$

Example:

$$\begin{aligned} f(n, A, B, C) = \texttt{IF } & n = 0 \texttt{ THEN } A \\ & \texttt{ELSE } f(n-1, B, C, A) \end{aligned}$$

The result of output analysis:
$Ovr(\lambda n, A, B, C : e) = \lambda n, A, B, C : \{A\}$

$OA(f) = \lambda n, A, B, C : \{A, B, C\}$.

Active variable analysis table.

$$\begin{aligned} AVA^0(f_i) &= \lambda(x_1, \ldots, x_n) : \emptyset \\ AVA(f_i) &= \lambda(x_1, \ldots, x_n) : FA(f)(S) \\ &\quad \text{if } Avr_t(f)(\lambda(x_1, \ldots, x_n) : e)) = \lambda(x_1, \ldots, x_n) : S \end{aligned}$$

Example:

$$\begin{aligned} f(n, A, B) = \texttt{IF } n = 0 \texttt{ THEN } (\lambda u : A(u + u)) \\ \texttt{ELSE } f(n - 1, B, A) \end{aligned}$$

The result of active variable analysis:
$Avr(e) = \lambda n, A, B : \{A\}$
$AVA(f) = \lambda n, A, B : \{A, B\}$

2.5 Liveness analysis

Live variables

$$\begin{aligned} Lv(\{\}) &= \emptyset \\ Lv(\{\}(e_1, \ldots, e_n)) &= \bigcup_i Mv(e_i) \\ Lv(e(e_1, \ldots, e_{j-1}, \{\}, \ldots, e_n)) &= Av(e) \cup \bigcup_{i<j} Av(e_i) \cup \bigcup_{i>j} Mv(e_i) \\ Lv(\texttt{IF}(\{\}, e_2, e_3)) &= Mv(e_2) \cup Mv(e_3) \\ Lv(\texttt{IF}(e_1, \{\}, e_3)) &= \emptyset \\ Lv(\texttt{IF}(e_1, e_2, \{\})) &= \emptyset \\ Lv(\{\}[(e_2) := e_3]) &= \emptyset \\ Lv(e_1[(\{\}) := e_3]) &= Mv(e_1) \cup Mv(e_3) \\ Lv(e_1[(e_2) := \{\}]) &= Mv(e_1) \\ Lv(U\{V\}) &= Lv(U) \cup Lv(V) \end{aligned}$$

Note: for destructive updates, we have the same definition as for non-destructive updates.

2.6 Liveness analyses

Informally, we want to store information that B is being updated destructively, in a context, where A is live. Example:

$$\begin{aligned} f_2^D(A, B) &= A + B[(3) \leftarrow 4] \\ LA(f_2^D) &= \langle B \rightarrow \{A\} \rangle \end{aligned}$$

$$
\begin{aligned}
LA^0(f^D)(x_i) = & \bot, \text{ if} \\
& x_i \text{ is not updateable, or} \\
& \forall\, U, e_1, e_2, e_3 : e \equiv U\{e_1[(e_2) \leftarrow e_3]\} \Rightarrow x_i \notin Ov(e_1), \text{ and} \\
& \forall\, U, g^D, a_1, \ldots, a_n : e^D \equiv U\{g^D(a_1, \ldots, a_m)\} \\
& \qquad\qquad \Rightarrow \forall\, j : x_j \in dom(LA(g^D)) : x_i \notin Ova_j \\
LA^0(f^D)(x_i) = & L_1 \cup L_2, \text{ otherwise, where} \\
& L_1 = \{y \mid \exists\, U, e_1, e_2, e_3 : \quad e^D \equiv U\{e_1[(e_2) \leftarrow e_3]\}, \text{ and} \\
& \qquad\qquad \wedge\, x_i \in FA(Ov(e_1)) \\
& \qquad\qquad \wedge\, y \in Lv(U)\} \\
& L_2 = \{y \mid \exists \quad U, g^D, a_1, \ldots, a_m : \\
& \qquad\qquad e^D \equiv U\{g^D(a_1, \ldots, a_m)\} \\
& \qquad\qquad \wedge\, (\exists\, j, l : \quad x_j \in dom(LA(g^D)) \\
& \qquad\qquad\qquad\qquad \wedge\, x_i \in FA(Ov(a_j)) \\
& \qquad\qquad\qquad\qquad \wedge\, x_l \in LA(g^D)(x_j) \\
& \qquad\qquad\qquad\qquad \wedge\, y \in Av(a_l) \cup Lv(U))\}
\end{aligned}
$$

We compute the variables, that are live in the context of recursive calls:

$$
\begin{aligned}
Lv_{rec}(f) = FA(\{y \mid \exists\, U, a_1, \ldots, a_n : \; & e \equiv U\{f(a_1, \ldots, a_n)\} \\
& \wedge\, y \in Lv(U)\})
\end{aligned}
$$

Then we compute the final liveness analysis table by applying the result of functional analysis *FA* to the first iteration as computed above, and by adding the variables that are live in the contexts of recursive calls Lv_{rec}.

$$
\begin{aligned}
Lv_{rec}(f) = FA(\{y \mid \exists\, U, a_1, \ldots, a_n : \; & e \equiv U\{f(a_1, \ldots, a_n)\} \\
& \wedge\, y \in Lv(U)\})
\end{aligned}
$$

$$
LA(f^D)(x_i) = FA(LA^0(f^D)(x_i)) \cup Lv_{rec}(f^D)
$$

Examples:

$$
\begin{aligned}
f_1(n, A, B) &= \texttt{IF } n = 0 \texttt{ THEN } A[(3) \leftarrow 4] \\
&\qquad \texttt{ELSE } B + f_1(n-1, A, B) \\
LA(f) &= \langle A \mapsto \{B\}\rangle \\
f_1(n, A, B, C) &= \texttt{IF } n = 0 \texttt{ THEN } B + A[(3) \leftarrow 4] \\
&\qquad \texttt{ELSE } f_1(n-1, A, C, B) \\
LA(f) &= \langle A \mapsto \{B, C\}\rangle
\end{aligned}
$$

2.7 Safe updates

Definition. An expression e^D is *safe* if

1. Every occurrence of $e_1[(e_2) \leftarrow e_3]$ in e^D within an update context U (i.e., $e^D \equiv U\{e_1[(e_2) \leftarrow e_3]\}$), satisfies $FA(Ov(e_1)) \cap (FA(Lv(U)) \cup Lv_{rec}) = \emptyset$ and $FA(Lv(U)) \subseteq LA(f^D)(x)$ for each variable x in $FA(Ov(e_1))$.
2. Every occurrence of a destructive function application of a function g, $g^D(a_1, \ldots, a_n)$ in e within an update context U (i.e., $e^D \equiv U\{g^D(a_1, \ldots, a_n)\}$) satisfies $FA(Ov(a_i)) \cap (FA((Lv(U) \cup Av(a_j))) \cup Lv_{rec}) = \emptyset$ for each x_i in the domain of $LA(g^D)$ and $x_j \in LA(g^D)(x_i)$. Furthermore, $FA(Lv(U) \cup Av(a_j)) \subseteq LA(f^D)(x)$ for each variable x in $FA(Ov(a_i))$ for x_i in the domain of $LA(g^D)$ and $x_j \in LA(g^D)(x_i)$.

Construction of a safe destructive function definition. We consider definitions of the form $f(x_1, \ldots, x_n) = e$. We construct the destructive version $f^D(x_1, \ldots, x_n) = e^D$ in the following way. We convert all occurrences of safe updates to destructive forms, all safe occurrences of function calls (of functions g other than f) $g(a_1, \ldots, a_n)$ to $g^D(a_1, \ldots, a_n)$ and all recursive function calls in update contexts $f(a_1, \ldots, a_n)$ to $f^D(a_1, \ldots, a_n)$.

Theorem 1 (Safe Definition). *The destructive definition $f^D(x_1, \ldots, x_n) = e^D$ obtained from the nondestructive definition $f(x_1, \ldots, x_n) = e$, is safe.*

Proof. All destructive updates and destructive function calls of function g other than f are safe by construction.
We must prove that all recursive function calls are safe. The main idea is that since we used the flow analysis and the set Lv_{rec}, we effectively ensured the safety of recursive calls. We will prove that the safety condition is verified:

1. $x_i \in dom(LA(f^D)) \wedge x_j \in LA(g^D)(x_i) \Rightarrow FA(Ov(a_i)) \cap FA(Av(a_j)) = \emptyset$. We will proof this statement by contradiction. Suppose a variable r is both in $FA(Ov(a_i))$ and $FA(Av(a_j))$, then by construction of FA, we have $r \in FA(x_i)$, $r \in FA(x_j)$. On the other hand, since $x_i \in dom(LA(f^D))$, x_i was safely updated in e^D. Suppose it was done by a safe destructive update $e_1[(e_2) \leftarrow e_3]\}$ in the context U (so we have $x_i \in FA(Ov(e_1))$) Reasoning is similar for the case of a destructive function call g other than f. Then $x_j \in FA(Lv(U))$, but since we have $r \in FA(x_j)$, we have also $r \in FA(Lv(U))$. By transitivity of FA, the condition $FA(Ov(e_1)) \cap (FA(Lv(U)) = \emptyset$ does not hold. Contradiction.
2. $x_i \in dom(LA(f^D)) \Rightarrow FA(Ov(a_i)) \cap FA((Lv(U) = \emptyset$
   Suppose a variable r is in $FA(Ov(a_i))$, then by construction of FA, we have $r \in FA(x_i)$. On the other hand, since $x_i \in dom(LA(f^D))$, x_i was safely updated (or was updated in another function g called from f) in e^D, thus the safety test for that update gives us that $FA(x_i) \cap Lv_{rec} = \emptyset$. Since $Lv(U) \subset Lv_{rec}$ by definition of Lv_{rec}, we can conclude.
3. $x_i \in dom(LA(f^D)) \Rightarrow FA(Ov(a_i)) \cap Lv_{rec} = \emptyset$ this follows from the fact that point 2. holds for all recursive calls.

Thus the safety condition for recursive calls is verified.

In this way, for each definition $f(x_1, \ldots, x_n) = e$, we construct its destructive counterpart $f^D(x_1, \ldots, x_n) = e^D$. The condition for its use of f^D is given by

liveness analysis.

Example:

$$f(A,B) = A + B[(3) := 4]$$
$$f^D(A,B) = A + B[(3) \leftarrow 4]$$
$$LA(f^D) = \langle A \rightarrow \{B\} \rangle$$

Complexity We consider definitions of the form $f(x_1, \ldots, x_n) = e$. The complexity of algorithm for $FA^0(x)$ is $O(n* \mid e \mid)$, we calculate it for at most n variables, which gives $O(n^2* \mid e \mid)$ as the complexity of the algorithm for FA^0. The complexity of simple standard algorithm for calculating transitive closure is n^3. Thus the complexity of the algorithm for FA is $O(n^3 + n^2* \mid e \mid)$.

3 Destructive variable analysis

Problem: Overhead for multiple representation of an array

- Closures (or closed lambda-expressions).
- Non-destructive array.
- Destructive array.

Thus, every time there is an access to an array, there has to be a check on the representation array, possibly followed by a conversion.For a definition $f^D(x_1, \ldots, x_n) = e^D$, static analysis can determine, which variables would contain only references updated destructively. For these variables, no runtime checks are necessary, if they are initially in destructive form. This can contribute significantly to resulting performance of the optimized function.

Function Dexpr. $Dexpr(S, a)$ detects if the expression a is in the set S (is a destructive variable) or it is a destructive update of a variable from the set S.

We compute the destructive variables:

$$\begin{aligned} Dv^0(f^D) = & L_1 \cup L_2 \\ & L_1 = \{y \mid \exists\, U, e_1, e_2, e_3 : \quad e^D \equiv U\{e_1[(e_2) \leftarrow e_3]\}\,, \text{ and} \\ & \qquad\qquad\qquad\qquad \wedge\, y \in FA(Ov(e_1))\} \\ & L_2 = \{y \mid \exists \quad U, g, a_1, \ldots, a_m : \\ & \qquad\qquad e^D \equiv U\{g^D(a_1, \ldots, a_m)\}, g \neq f \\ & \qquad\qquad \wedge\, (\exists\, j : \quad x_j \in Dv(g) \\ & \qquad\qquad\qquad\qquad \wedge\, y \in FA(Ov(a_j))\}) \end{aligned}$$

We compute the non-destructive variables:

$$
\begin{aligned}
NDv(f)(x) &= \emptyset \\
NDv(f)(f_i) &= \emptyset \\
NDv(f)(g^D(a_1,\ldots,a_n)) &= \{y \mid (\exists\ j : x_j \in NDv(g) \wedge y \in FA(Ov(a_j))\} \\
&\quad \cup \\
&\quad \{y \mid (\exists\ j : x_j \in Dv(g) \wedge \neg Dexpr(Dv(f), a_j) \\
&\quad \wedge y \in FA(Ov(a_j))\}), g \neq f \\
NDv(f)(f^D(a_1,\ldots,a_n)) &= \{y \mid (\exists\ j : x_j \in Dv(f) \wedge \neg Dexpr(Dv(f), a_j) \\
&\quad \wedge y \in FA(Ov(a_j))\}) \\
NDv(f)(a(a_1,\ldots,a_n)) &= Mv(a(a_1,\ldots,a_n)),\ \text{if } a \neq f \wedge a \neq g \\
NDv(f)(\lambda(x_1,\ldots,x_n) : e) &= Mv(\lambda(x_1,\ldots,x_n) : e) \\
NDv(f)(\texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3) &= NDv(f)(e_1) \cup NDv(f)(e_2) \cup NDv(f)(e_3) \\
NDv(f)(a_1[(a_2) := a_3]) &= Ov(e1) \cup NDv(f)(e_1) \cup NDv(f)(e_2) \cup NDv(f)(e_3) \\
NDv(f)(a_1[(a_2) \leftarrow a_3]) &= NDv(f)(e_1) \cup NDv(f)(e_2) \cup NDv(f)(e_3)
\end{aligned}
$$

$$
\begin{aligned}
Dv(f) &= Dv^0(f) - FA(f)(Dv^0(f) \cap NDv^0(f)) \\
NDv(f) &= NDv^0(f) \cup FA(f)(Dv^0(f) \cap NDv^0(f))
\end{aligned}
$$

Definition. An expression e^D is *DV-safe* with respect to the set of variables S, if:

1. For all x in S, for every occurrence in e^D of $e_1[(e_2) := e_3]$, x is not in $Ov(e_1)$.
2. For all x in S, for every occurrence in e^D of $g(e_1,\ldots,e_n)$ (or $g^D(e_1,\ldots,e_n)$) for all y_i in $NDv(g)$, x is not in $Ov(e_i)$.
3. For all x in S, for every occurrence in e^D of $e_l(e_1,\ldots,e_n), e_l \neq g$, x is not in $Mv(e_l(e_1,\ldots,e_n))$.

Note: the following construction should just give an intuitive idea on the purpose of destructive variable analysis. In the core language we present in this paper, there are no runtime checks; but of course we still can prove that DV-safety is preserved during evaluation, which gives us correctness argument for this optimization.

Construction an optimized version f^{Dv} of the destructive version of f^D, by removing all the runtime checks on the representation of variables from $Dv(f^D)$ (and accessing the corresponding arrays directly, since we know that they are in destructive form).

Condition for the use of optimized version: Arrays passed as one of the variables in $Dv(f^D)$ have to be in destructive form.

Theorem 2. *Destructive definition* $f^D(x_1,\ldots,x_n) = e^D$ *is DV-safe with respect to* $Dv(f)$.

Proof. The only case: $f(e_1, \ldots, e_n), x_i \in NDv(f), x_j \in Ov(e_i)$ implies $x_j \in NDv(f)$.

Example:

$$f^D_{rec}(n, A, B, C) = \texttt{IF } n = 0 \texttt{ THEN } A + B[(3) := 4] \\ \texttt{ELSE } C[(n) \leftarrow 4](3) + f^D_{rec}(n, A, A, C)(3)$$

C is updated only destructively, so if the actual parameter corresponding to C is in destructive form, we need not check on the representation on C in the update $C[(n) \leftarrow 4](3)$.

4 Arity analysis

In higher-order languages with type parametricity, it is quite common to have a situation where the arguments to a multiary operation are tupled. For example, in $map(f, l)$, if the operation f is multiary, then the argument l must be a list of tuples. Here the problem arises because the map operation is parametric in the element type of the list (and the domain type of f). The code generated for map is the same even when the element type a tuple. On the other hand, there are situations where the arity is known at compilation time so that a sort operation that is parametric in a comparison operator can be given with the signature $sort(ord, array)$. In the body of sort, ord is used as a binary operator. This means that when it is invoked with say the integer ordering $<$ as $sort(<, A)$, the $<$ here can be multiary, whereas in $map(<, list - of - pairs)$, the $<$ operation must be unary operation whose argument is a pair. Thus the generated code for $<$ must include both a unary version as well as a multiary version of the code.

Given a function definition of the form $f(x_1, .., x_n) = e$, arity analysis can be used to detect if it is safe to use the multiary form of an argument x_i. An argument x_i is unary in e if it is used in a unary application $x_i(a)$ or in a unary argument position $g(.., x_i, ...)$ to a function g. Otherwise, the argument x_i is treated as multiary. Thus when $f(a_1, .., a_n)$ is used in a definition, and x_i is a multiary argument to f, then it is safe to use the multiary version of a_i. Otherwise, we must use the unary version of a_i.

In popular functional languages, it is customary to Curry function application so that $< (a, b)$ is used as $< (a)(b)$. With such Curried application, multiary operations might seem unnecessary since any unary use of these functions over all the arguments involves the explicit construction of the appropriate unary operation. Thus $map(<, list - of - pairs)$ would be type-incorrect, and this would have to be represented as $map((lambda(x, y) < (x)(y)), list - of - pairs)$. This objection is true to some extent but since these languages do allow functions to be defined on tuples of arguments, arity analysis is relevant for these languages as well.

Problem:

```
p:VAR [nat,nat]

x,y:VAR nat

f:VAR [nat,nat->nat]

h(f,p,x):nat=f(p)+f(x,x)
```

is legal in PVS syntax.
How to pass the parameter f (as a unary or multiary function)? Note, that it is not possible to de-tuple the arguments always:

```
arity  [ U:TYPE,f: [U->nat] ]: THEORY
  BEGIN
    p:var U
    h(p):nat=f(p)
  END arity

test_arity : THEORY
  BEGIN
    x1,x2: VAR nat
    f(x1,x2):nat=x1+x2
    IMPORTING arity[[nat,nat],f]
  END test_arity
```

The only safe alternative is to tuple the arguments by default and pass unary functions.

```
arity [U:TYPE]: THEORY
BEGIN
  x:VAR U
  b:VAR bool
  f:VAR [U->U]
  g(b,f):[U->U]=if b then lambda x:x  else f endif
END arity

test_arity:THEORY
BEGIN
  IMPORTING arity8[[nat,nat]]
  f(x1:nat,x2:nat):[nat,nat]=(x1,x2)
  r:[nat,nat]=g(true,f)(2,3)
END arity
```

Static analysis can determine the cases, when it is possible to pass multiary function and thus avoid unnecessary coupling of arguments.
Example:

$$h(f,x) = f(x,x)$$

f can be passed as a multiary function.

We extract information from the types of formal parameters as well as from the expression.

– $Un^0(f)$ - variables of a functional type, whose domain is of type U, that can be instantiated as a tuple and variables of functional type, that appear "standalone" in the expression
– $Mult^0(f)$ - variables of a functional type, whose domain type is determined (in the sense that it cannot be instantiated)

$$Mult(f) = Mult^0(f) - FA(f)(Mult^0(f) \cap Un^0(f))$$
$$Un(f) = Un^0(f) \cup FA(f)(Mult^0(f) \cap Un^0(f))$$

In order to define the notion of safety corresponding to arity analysis problem, we introduce new type constructor into our type system, and a new expression form into the language. We denote the resulting language L_1 and type system T_1. Type constructor for tuple types: $[T_1, \ldots, T_n]$ for types $T_1, \ldots, T_n$.
Expression form for tuple expression is: $(e_1, \ldots e_n)$ is of type $[T_1, \ldots, T_n]$, where each e_i is of type T_i.

Definition. An expression e of a language L_1 is *AR-safe*, if it is well-typed according to type system T_1.

Construction of an AR-safe function definition For a function definition $f(x_1, \ldots, x_n) = e$ we construct an optimized Ar-safe definition f^{Ar} by translating it into language L_1 in the following way:

– variables $x_1, \ldots, x_n$: if x_i is in $Un(f)$ then its type is changed from $[T_1, \ldots, T_m \to T]$ to $[[T_1, \ldots, T_m] \to T]$, otherwise it is left unchanged.
– occurrence of an application $e_l(e_1, \ldots, e_n)$:
  • $e_l \equiv x$ and $x \in Mult(f)$, expression is unchanged
  • $e_l \equiv g$ expression is unchanged
  • otherwise, expression becomes $e_l((e_1, \ldots, e_n))$ - arguments have been coupled

Theorem 3. *The destructive definition $f^{Ar}(x_1, \ldots, x_n) = e^{Ar}$ obtained from the nondestructive definition $f(x_1, \ldots, x_n) = e$, is AR-safe.*

Proof.

5 Operational semantics

We present operational semantics for the languages with destructive updates. We then exhibit a bisimulation between evaluation steps on a nondestructive expression e and its safe destructive counterpart e^D. The concepts used in defining the operational semantics are quite standard, but we give the details for the language used here.

The expression domain is first expanded to include

1. Explicit arrays: $\#(e_0, \ldots, e_{n-1})$ is an expression representing an n-element array.
2. References: $\mathit{ref}(i)$ represents a reference to reference number i in the store. Stores appear in the operational semantics.

A *value* is either a boolean constant, integer numeral, a closed lambda-abstraction $(\lambda x_1, \ldots, x_n : e)$ or a reference $\mathit{ref}(i)$. The metavariable v ranges over values.

An *evaluation context* [**?**] E is an expression with an occurrence of a hole $[]$ and is of one of the forms

1. $[]$
2. $[](e_1, \ldots, e_n)$
3. $v(v_1, \ldots, v_{j-1}, [], e_{j+1}, \ldots, e_n)$
4. $\mathtt{IF}([], e_2, e_3)$, $\mathtt{IF}(\mathtt{TRUE}, [], e_3)$, or $\mathtt{IF}(\mathtt{FALSE}, e_2, [])$.
5. $e_1[([]) := e_3]$, $e_1[(v_2) := []]$, or $[][(v_2) := v_3]$.
6. $e_1[([]) \leftarrow e_3]$, $e_1[(v_2) \leftarrow []]$, or $[][(v_2) \leftarrow v_3]$.
7. $E_1[E_2]$, if E_1 and E_2 are evaluation contexts.

A *redex* is an expression of one of the following forms

1. $p(v_1, \ldots, v_n)$.
2. $f(v_1, \ldots, v_n)$.
3. $(\lambda(x : [0 < n]) : e)$.
4. $(\lambda(x_1, \ldots, x_n) : e)(v_1, \ldots, v_n)$.
5. $\#(v_0, \ldots, v_{n-1})$.
6. $\mathit{ref}(i)(v)$.
7. $\mathtt{IF\ TRUE\ THEN}\ e_1\ \mathtt{ELSE}\ e_2$.
8. $\mathtt{IF\ FALSE\ THEN}\ e_1\ \mathtt{ELSE}\ e_2$.
9. $\mathit{ref}(i)[(v_2) := v_3]$.
10. $\mathit{ref}(i)[(v_2) \leftarrow v_3]$.

A *store* is a mapping from a reference number to an array value. A store s can be seen as a list of array values $[s[0], s[1], \ldots,]$ so that $s[i]$ returns the $(i{+}1)$'th element of the list. Let $s[i]\langle i \mapsto v_i \rangle$ represent the array value $\#(w_0, \ldots, v_i, \ldots, w_{n-1})$, where $s[i]$ is of the form $\#(w_0, \ldots, w_i, \ldots, w_{n-1})$. List concatenation is represented as $r \circ s$.

A *reduction* transforms a pair consisting of a redex and a store. The reductions corresponding to the redexes above are

1. $\langle p(v_1, \ldots, v_n), s\rangle \to \langle v, s\rangle$, if the primitive operation p when applied to arguments $v_1, \ldots, v_n$ yields value v.
2. $\langle f(v_1, \ldots, v_n), s\rangle \to \langle [v_1/x_1, \ldots, v_n/x_n](e), s\rangle$, if f is defined by $f(x_1, \ldots, x_n) = e$.
3. $\langle (\lambda(x : [0 < n]) : e), s\rangle \to \langle \#(e_0, \ldots, e_{n-1}), s\rangle$, where $e_i \equiv (\lambda(x : [0 < n]) : e)(i)$, for $0 \le i < n$.
4. $\langle (\lambda(x_1 : T_1, \ldots, x_n : T_n) : e)(v_1, \ldots, v_n), s\rangle \to \langle [v_1/x_1, \ldots, v_n/x_n](e), s\rangle$.
5. $\langle \#(v_0, \ldots, v_{n-1}), s\rangle \to \langle \mathit{ref}(m), s'\rangle$, where $s \equiv [s[0], \ldots, s[m-1]]$ and $s' \equiv s \circ [\#(v_0, \ldots, v_{n-1})]$.

6. $\langle ref(i)(v), s\rangle \to \langle s[i](v), s\rangle$.
7. $\langle \texttt{IF TRUE THEN } e_1 \texttt{ ELSE } e_2, s\rangle \to \langle e_1, s\rangle$.
8. $\langle \texttt{IF FALSE THEN } e_1 \texttt{ ELSE } e_2, s\rangle \to \langle e_2, s\rangle$.
9. $\langle ref(i)[(v_2) := v_3], s\rangle \to \langle ref(m), s'\rangle$, where

$$\begin{aligned} s &\equiv [s[0], \ldots, s[i], \ldots, s[m-1]] \\ s' &\equiv [s[0], \ldots, s[i], \ldots, s[m]] \\ s[m] &= s[i]\langle v_2 \mapsto v_3\rangle. \end{aligned}$$

10. $\langle ref(i)[(v_2) \leftarrow v_3], s\rangle \to \langle ref(i), s'\rangle$, where $s_1 \equiv [s[0], \ldots, s[i], \ldots, s[m-1]]$ and $s_2 \equiv [s[0], \ldots, s[i]\langle v_2 \mapsto v_3\rangle, \ldots, s[m-1]])$.

An evaluation *step* operates on a pair $\langle e, s\rangle$ consisting of a closed expression and a store, and is represented as $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$. If e can be decomposed as a $E[a]$ for an evaluation context E and a redex a, then a step $\langle E[a], s\rangle \longrightarrow \langle E[a'], s'\rangle$ holds if $\langle a, s\rangle \to \langle a', s'\rangle$. The reflexive-transitive closure of $\longrightarrow$ is represented as $\langle e, s\rangle \xrightarrow{*} \langle e', s'\rangle$. If $\langle e, s\rangle \xrightarrow{*} \langle v, s'\rangle$, then the result of the computation is $s'(v)$, i.e., the result of replacing each reference $ref(i)$ in v by the array $s'[i]$. The computation of a closed term e is initiated on an empty store as $\langle e, []\rangle$. The value $eval(e)$ is defined to be $s(v)$, where $\langle e, []\rangle \xrightarrow{*} \langle v, s\rangle$.

6 Correctness

Proofs of correctness.

Idea:

- Notion of safety.
- Safety is preserved during evaluation.
- For safe expressions, there is a bisimulation between the evaluation of optimized and unoptimized expression.

More precisely, what we want to prove is the following theorem:

Theorem 4. *For every closed, reference-free, non-destructive expression e, we have*

$$eval(e) \equiv eval(e^{D\,Dv\,Ar})$$

.

where e^D denotes constructed safe destructive expression, e^{Dv} denotes constructed DV-safe expression, and e^{Ar} denotes constructed AR-safe expression.

In order to prove the theorem, we will need to introduce some auxiliary notions and prove some lemmas. The proof has the following structure:

- We lift the analyse from variables to references, since the expressions being evaluated do not contain free variables. The definitions of Fl, Av, etc. have to be extended to include references. See [**?**].

– We need the notions of normality of expressions and well-formedness of configurations, as well as the lemmas that states that normality and well-formedness are preserved during evaluation. See [**?**], Section 4.
– We will need some properties of the functions such as Fl, Av, Ov with respect to evaluation - i.e. properties that state the semantics of these functions.
– For each of the analysis, we prove that the corresponding safety notion is preserved by evaluation of the optimized expressions. Note that for the arity optimization, the preservation of AR-safe notion is a 'subject reduction' for the type system.
– We prove that there is a bisimulation with respect to evaluation between optimized and unoptimized expressions. Thus it will be possible to conclude the proof of the main theorem.

Definition. The definition of a function $f(x_1, \ldots x_n) = e$, is called a *valid definition* with respect to a list of functions $g_1, \ldots, g_n$, if it is expressed only in terms of $f, g_1, \ldots g_n$, the free variables of e are among $x_1, \ldots, x_n$, and e contains no references.

Lemma 1. *Let $f(x_1, \ldots x_n) = e$ be a valid definition and $v_1, \ldots, v_n$ be values. Then if $\forall\ s : \exists\ E, s', w_1, \ldots, w_n : \langle f(v_1, \ldots, v_n), s\rangle \xrightarrow{*} \langle E[f(w_1, \ldots, w_n)], s'\rangle$, then*
$\forall\ i, j_1, j_2\ \forall\ r\ \mathit{ref}(r) = v_{j_1} \wedge \mathit{ref}(r) \neq v_{j_2} \wedge \mathit{ref}(r) = w_{j_2} \Rightarrow x_{j_2} \in FA(f)(x_{j_1})$.

Proof. We prove a somewhat stronger assertion by induction on the length of the derivation. We need simultaneous induction on the function Av.

Lemma 2. *For every reduction $\langle a, r\rangle \to \langle a', r'\rangle$, $Ov(a') \cap dom(r) \subseteq Ov(a)$. Hence, for each evaluation step $\langle d, r\rangle \longrightarrow \langle d', r'\rangle$, $Ov(d') \cap domr \subseteq Ov(d)$.*

Proof.

Theorem 5. *If $\langle d, r\rangle$ is a well-formed configuration, $\langle d, r\rangle \longrightarrow \langle d', r'\rangle$, and d is safe, then d' is safe.*

Proof. The proof from [**?**] can be adapted.

Theorem 6. *If $\langle d, r\rangle$ is a well-formed configuration, $\langle d, r\rangle \longrightarrow \langle d', r'\rangle$, and d is DV-safe with respect to S, then d' is DV-safe with respect to S.*

Proof. Let A be one of the expressions mentioned in the definition of DV-safe, i.e. $e_1[(e_2) := e_3]$, $f(e_1, \ldots, e_n)$, $e(e_1, \ldots, e_n), e \neq g$. Let d be of the form $E[a]$ for some evaluation context E and redex a. Given any occurrence of A in d', the residual a' either occurs within A, A occurs within a', or a' occurs within E. This is because none of the redexes can partially overlap an update expression or an application.

If a' occurs properly in A, the following cases arise:

1. A is of the form $e_1'[(e_2') := e_3']$. Then if a' occurs in either e_2 or e_3, we have that $e_1' = e_1$, and $Ov(e_1') = Ov(e_1)$, and since d was DV-safe with respect to S, so is d'.
2. A is of the form $g(e_1', \ldots, e_n')$, where A occurs in e_i' for some i. But, by Lemma **??**, we have $Ov(e_1') = Ov(e_1)$, and since d was DV-safe with respect to S, so is d'.
3. A is of the form $e'(e_1', \ldots, e_n'), e \neq g$. No reference from S occurred in $e(e_1, \ldots, e_n)$, so if d was DV-safe with respect to S, so is d'.

If a' occurs within E, we can conclude immediately.

If A occurs (properly or not) in a', then, by the syntax of a redex a, one of the following four cases is possible:

1. Redex a is a conditional expression, and a' must be either the `THEN` or `ELSE` part of a. In any case, A occurred in d already, and since d was DV-safe with respect to S, so is d'.
2. Redex a is of the form $\lambda z_1, \ldots, z_n : e_m(v_1, \ldots, v_n)$. In this case, by the definition of DV-safety, no reference from S occurred in a, thus we can conclude.
3. Redex a is of the form g, for some defined function g. Since g was defined by a valid definition, no reference from S occurred in a, thus we can conclude.
4. Redex a is of the form $g(v_1, \ldots, v_n)$ and is reduced to a' of the form $\sigma(b)$, where g is defined as $g(x_1 \ldots, x_n) = b$ and substitution σ is of the form $\langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \rangle$. We know that $g(v_1, \ldots, v_n)$ is DV-safe with respect to S. If A is of the form $\sigma(b_1[(b_2) := b_3])$, we have that $Ov(b_1) \subset NDv(g)$. Thus for all x_i in $Ov(b_1)$, v_i is not in S, so we can conclude. Similar argument can be made for the other two cases of A.

Theorem 7. *If $\langle a, r \rangle$ is a well-formed typable configuration, $\langle a, r \rangle \longrightarrow \langle a', r' \rangle$, then $\langle a', r' \rangle$ is a well-formed typable configuration.*

Proof. Note that typable refers to type system T_1 defined above. Note that this lemma states that AR-safety is preserved during the course of evaluation. The proof is a 'subject reduction' type proof for type system T_1.

7 Performance comparisons

Example:

```
insert(A, a, k): RECURSIVE ARR =
  IF (k = 0 OR a >= A(k - 1))
    THEN A WITH [(k) := a]
  ELSE insert(A WITH [(k) := A(k - 1)], a, k - 1)
  ENDIF
   MEASURE k

insort_rec(A, (n: upto(N))): RECURSIVE ARR =
   IF n < N THEN insort_rec(insert(A, A(n), n), n + 1)
   ELSE A ENDIF
   MEASURE N - n

insort(A): ARR = insort_rec(A, 0)

J(k): nat = N - k - 1

jsort: nat = insort(J)(0)
```

Simple sorts.

	Insertion sort [s]	Bubble sort [s]
PVS 2.4	17.880	31.340
PVS 2.4 LI	7.920	28.330
PVS NV	**3.170**	**5.450**
C	1.760	2.980

Notations:
LI - improvements in Lisp coding
NV - new version, with improved static analyses

The times suggest that the overhead for each update is constant with respect to C (in both cases, factor is 1.8).

Parametrized insertion sort in PVS

```
insertion_sort   [T : TYPE, < : (total_order?[T]), N : nat]: THEORY
 BEGIN
  INDEX:  TYPE = below(N)
  ARR: TYPE = ARRAY[INDEX -> T]
  insert(i)((k: upto(i)), (A : ARR): RECURSIVE ARR =
   (IF k = 0 THEN A
      ELSIF A(k) < A(k-1)
      THEN insert(i)(k-1, A WITH [(k-1) := A(k),
                                  (k)   := A(k-1)])
      ELSE A ENDIF)
   MEASURE k
  insort_rec(n, (A : sorted(n, n))): RECURSIVE ARR=
    (IF n < N
      THEN insort_rec(n + 1, insert(n)(n, A))
      ELSE A ENDIF)
   MEASURE N - n
END insertion_sort
```

Parameterized sorts.

	Par. ins. sort [s]	Par. bubble sort [s]
PVS 2.4	49.160	87.480
PVS 2.4 LI	41.820	71.130
PVS NV DA	28.750	52.910
PVS NV AA	21.340	29.740
PVS NV	10.240	18.420
C	4.670	6.140
C++	2.880	4.120

Notation:
LI - improvements in Lisp coding
NV DA - new version, with only destructive variable analysis
NV AA - new version, with only arity analysis
NV - new version

Space usage for parameterized sorts.

	Par. ins. sort	Par. bubble sort
PVS 2.4	1.2 MB	2.4 MB
PVS 2.4 LI	1.2 MB	2.4 MB
PVS NV DA	1.2 MB	2.4 MB
PVS NV AA	40,040 B	40,016 B
PVS NV	40,040 B	40,016 B

Notation:
LI - improvements in Lisp coding
NV DA - new version, with only destructive variable analysis
NV AA - new version, with only arity analysis
NV - new version

Here it is clear that arity analysis helped to reduce the space usage, because it prevented unnecessary coupling/decoupling of arguments of functions.

Technical note: The tests were performed on a Pentium III 550MHz machine, with 256MB RAM.

We chose insertion sort as an example for two reasons: Firstly, it is simple, containing updates, possible genericity, so that use of analyses has visible effect. Secondely, it is a "Counter-example" to the claim that programming should be done in logic, since it is possible to write best possible insertion sort by hand. This might not be true for more complicated algorithms; so use of automatic static analyses might yield better results when compared to hand-written programs.

8 Related work

There is a vast literature on static analyses of functional programs, classical example being strictness analysis (that can be formulated in the abstract interpretation framework).

Flow analysis introduce here is similar in nature to data-flow analysis, commonly used by optimizing compilers for imperative languages.

A lot of work has been done on the problem of safe destructive updates. The key insight about variables being the channels through which updates interact is due to Draghicescu and Purushothaman [**?**]. Other references to works on this topic can be found in [**?**].

9 Conclusion

The results of these relatively simple static analyses are encouraging: the typical performance of a simple optimized functional program is within a linear factor of 2 of the corresponding C program, with the same space behavior, and relatively small cost of exploiting parametricity.

Important extensions can be done:

- Nested array updates: Structure sharing within a variable violates assumptions of the update analysis. Theorem proving could discern independent updates.
- Phase distinction: Code generation is done separately for each theory When actual type parameters are known, more efficient code generation might be possible (or non-executable operations might become executable) — this can be exploited by staged optimizations.