A Sharing Analysis for SAFE

Clara Segura and Ricardo Peña and Manuel Montenegro[*]

Dpto. Sistemas Informáticos y Programación, Univ. Complutense de Madrid, Spain
csegura@sip.ucm.es,ricardo@sip.ucm.es, manuelmont@gmail.com

Abstract

We present a sharing analysis for the functional language *Safe*. This is a first-order eager language with facilities for programmer-controlled destruction and copying of data structures. It provides also *regions*, i.e. disjoint parts of the heap where the programmer may allocate data structures. The language and its associated type system guaranteeing that destruction facilities and region management are done in a safe way were presented in a previous paper [6]. That type system uses two functions, called *sharerec* and *shareall*, which are supposed to give upper approximations to the set of variables respectively sharing a recursive substructure, or any substructure, of a given variable.

In this paper we present the formal definition of such functions. In order to have a modular and efficient analysis, we provide signatures for functions, which summarize their sharing behaviour. The paper ends up describing the implementation of the analysis and some examples.

1 INTRODUCTION

Many imperative languages offer low level mechanisms to allocate and free heap memory, which the programmer may use in order to dynamically create and destroy pointer based data structures. These mechanisms give the programmer a complete control over memory usage but are very error prone. Well known problems that may arise when using a programmer-controlled memory management are dangling references, undesired sharing between data structures with complex side effects as a consequence, and polluting memory with garbage.

Functional languages usually consider memory management as a low level issue. Allocation is done implicitly and usually a garbage collector takes care of the memory exhaustion situation.

In a previous paper [6] we proposed a semi-explicit approach to memory control by defining a functional language, called *Safe*, in which the programmer cooperates with the memory management system by providing some information about the intended use of data structures. For instance, the programmer may indicate that some particular data structure will not be needed in the future and that, as a consequence, it may be safely destroyed by the runtime system and its memory recovered. The language uses regions to locate data structures. It also allows controlling the degree of sharing between different data structures. More interesting than the concrete facilities provided by the language, is the definition of a type system guaranteeing that all this can be done in a safe way. In particular, it guarantees that dangling pointers are never created in the live heap. An ill-constructed program is rejected by the type system.

[*]Work partially supported by the Spanish project TIN2004-07943-C04.

A garbage collector is not needed. Allocation and destruction of data structures are done as execution proceeds. The type system makes a heavy use of two functions, not defined there, called *shareall* and *sharerec*. Given a subexpression e of a function body and a free variable x such that e is included in the lexical scope of x,

- $\mathit{shareall}(x,e)$ returns the set of all the variables in scope in e which, at runtime, may share any substructure of the structure pointed to by x.
- $\mathit{sharerec}(x,e)$ returns the set of all the variables in scope in e which, at runtime, may share any recursive substructure of the structure pointed to by x.

In [6] we anticipated that (an upper approximation to) these two functions could be computed at compile time by an abstract interpretation like analysis. We provide such an analysis in this paper. Another contribution is the formal definition of the operational semantics of *Safe* which was not included in the above mentioned paper.

The structure of the paper is as follows: In Section 2, we provide a summary of the syntax, semantics and type system of *Safe*. Then, Section 3 presents in detail the sharing analysis. In Section 4, the implementation of the analysis is described and it is applied to some illustrative examples. Section 6 surveys some related work and concludes.

2 SUMMARY OF *SAFE*

2.1 Syntax

We start by reproducing some crucial definitions which underlie the language.

Definition 1 *A* **region** *is a contiguous memory area in the heap where data structures can be constructed, read, or destroyed. It is allocated and freed as a whole, in constant time.*

Definition 2 *A* **cell** *is a small memory space, big enough to hold a data constructor. In implementation terms, a cell contains the mark (or code pointer) of the constructor, and a representation of the free variables to which the constructor is applied. These may consist, either of basic values, or of pointers to non-basic values.*

Definition 3 *A* **data structure**, *in the following a DS, is the set of cells obtained by starting at one cell considered as the root, and taking the transitive closure of the relation* $C_1 \rightarrow C_2$*, where* C_1 *and* C_2 *are cells of the same type* T*, and in* C_1 *there is a pointer to* C_2.

That means that, for instance in a list of type `[[a]]`, we are considering as a DS all the cells belonging to the *outermost* list, but not those belonging to the individual innermost lists. Each one of the latter constitute a separate DS.

The following decisions were taken:

1. A DS completely resides in one region.
2. One DS can be part of another DS, or two DSs can share a third one.

$$
\begin{array}{lcll}
\mathit{prog} & \to & \mathit{dec}_1; \ldots ; \mathit{dec}_n; \mathit{expr} & \\
\mathit{dec} & \to & \;\; f\ \overline{x_i}^n\ r = \mathit{expr} & \{\text{single-recursive, polymorphic function}\} \\
 & & \mid f\ \overline{x_i}^n = \mathit{expr} & \\
\mathit{expr} & \to & \;\; a & \{\text{atom: literal } c \text{ or variable } x\} \\
 & & \mid x@r & \{\text{copy}\} \\
 & & \mid x! & \{\text{reuse}\} \\
 & & \mid (f\ \overline{a_i}^n)@r & \{\text{function application}\} \\
 & & \mid (f\ \overline{a_i}^n) & \{\text{function application}\} \\
 & & \mid (C\ \overline{a_i}^n)@r & \{\text{constructor application}\} \\
 & & \mid \mathbf{let}\ x_1 = \mathit{expr}_1\ \mathbf{in}\ \mathit{expr} & \{\text{non-recursive, monomorphic}\} \\
 & & \mid \mathbf{case}\ x\ \mathbf{of}\ \overline{\mathit{alt}_i}^n & \{\text{read-only case}\} \\
 & & \mid \mathbf{case}!\ x\ \mathbf{of}\ \overline{\mathit{alt}_i}^n & \{\text{destructive case}\} \\
\mathit{alt} & \to & C\ \overline{x_i}^n \to \mathit{expr} &
\end{array}
$$

FIGURE 1. First-order functional language *Safe*

3. The basic values —integers, booleans, etc.— do not allocate cells in regions. They live inside the cells of DSs, or in the stack.

4. A function of n parameters can access to:

   - Its n parameters, each one residing in a possibly different region.
   - Its **output region**, whenever it builds a DS as a result. There is at most one output region per function. Delivering this region identifier as a parameter is the responsibility of the caller. We force functions to leave their result in an output region belonging to the caller in order to safely delete the intermediate results computed by the function.
   - Its (optional) **working region**, referred to through the reserved identifier *self*, where it may create intermediate DSs. The working region has the same lifetime as the function call: It is allocated at each invocation and freed at function termination.

5. If a parameter of a function is a DS, it can be destroyed by the function. We will say that the parameter is **condemned** because this capability depends on the function definition, not on its use.

6. The capabilities a function has on its accessible DSs and regions are: a function may only read a DS which is a read-only parameter; a function may read (before destroying it), and must destroy, a DS which is a condemned parameter; a function may construct, read, or destroy DSs, in either its output or its working region.

The syntax of *Safe* is shown in Figure 1. This is a first-order eager functional language where sharing is enforced by using variables in function and constructor applications. We intend *Safe* to be a core language resulting from the desugaring of a higher level language similar to Haskell or ML. The analysis defined in this paper is however done at core level. This is a usual approach in many compilers.

$revD :: \forall a, \rho_1, \rho_2 . [a]!@\rho_1 \rightarrow \rho_2 \rightarrow [a]@\rho_2$
$revD\ xs\ r = (revauxD\ xs\ [\,]@r)@r$

$revauxD :: \forall a, \rho_1, \rho_2 . [a]!@\rho_1 \rightarrow [a]@\rho_2 \rightarrow \rho_2 \rightarrow [a]@\rho_2$
$revauxD\ xs\ ys\ r =$ **case**! xs **of**
$\qquad [\,] \rightarrow ys$
$\qquad x : xx \rightarrow (revauxD\ xx\ (x : ys)@r)@r$

FIGURE 2. Destructive list inversion

A program *prog* in *Safe* is a sequence of possibly recursive polymorphic function definitions[1] followed by a main expression *expr*, calling them, whose value is the program result. Function definitions building a new DS will have an additional parameter r, which is the output region, where the resulting DS is to be constructed. In the right hand side expression only r and its own working region *self* may be used. Polymorphic algebraic data types definitions are also allowed. We will assume they are defined separately through **data** declarations.

The program expressions include variables, literals, function and constructor applications, and also **let** and **case** expressions, but there are some additional expressions:

If x is a DS, the expression $x@r$ represents a copy in region r of the DS accessed from x. The DS x must live in a region $r' \neq r$. Both x and $x@r$ have the same recursive structure and they share their non-recursive substructures.

The expression $x!$ means the reusing of the destroyable DS to which x points. This is useful when we do not want to destroy completely a condemned parameter but instead to reuse part of it. In semantic terms, x and $x!$ are the same physical structure but, in language terms, once $x!$ is used x becomes unaccessible.

In function application we have a special syntax $@r$ to express the inclusion of the additional output region parameter. Using the same syntax, we express that a constructor application is to be allocated in region r.

The **case**! expression indicates that the outer constructor of x is disposed after the pattern matching so that x is not accessible anymore. The recursive substructures may be explicitly destroyed in the subsequent code via another **case**! or reused via $x!$. A condemned variable may be read but, once its content has been destroyed or reused in another structure, it may not be accessed again. This is what the type system in [6] guarantees.

We show now with several examples how to use the language facilities. In some of them we will write $x!$ or $(C\ \overline{a_i}^n)@r$ as actual parameters of applications in order to abbreviate, when a **let** binding would in fact be needed. In these examples we show also the types that the functions have in the type system previously mentioned. For the moment they can be ignored. The first example is the function that reverses a list and, at the same time, destroys it. The code is shown in Figure 2.

[1]The extension to mutual recursion would pose no special problems, but we restrict ourselves to single recursion in order to ease the presentation.

```
$insertD :: \forall a, \rho . a \rightarrow Tree\ a!@\rho \rightarrow \rho \rightarrow Tree\ a@\rho$
$insertD\ x\ t\ r =$ case! $t$ of
                    $Empty \rightarrow (Node\ Empty@r\ x\ Empty@r)@r$
                    $Node\ i\ y\ d \rightarrow$ let $c = compare\ x\ y$
                                    in case $c$ of
                                         $LT \rightarrow (Node\ (insertD\ x\ i)@r\ y\ d!)@r$
                                         $EQ \rightarrow (Node\ i!\ y\ d!)@r$
                                         $GT \rightarrow (Node\ i!\ y\ (insertD\ x\ d)@r)@r$
```

FIGURE 3. Destructive insertion with reuse in a binary search tree

We use the usual auxiliary function with an accumulator parameter. Notice that the differences with the usual functional version version are, on the one hand, the use of the region parameter r and, on the other, that a **case**! is used over the original list. The recursive application of the function destroys it completely. Those who call *revD* should know that the argument is lost in the inversion process, and should not try to use it anymore.

The next example illustrates the reuse of a condemned structure. It is the function, shown in Figure 3, that inserts an element in a binary search tree in such a way that the original tree is partially destroyed. Everything but the path from the root to the inserted element is reused to build the new tree but these parts can no longer be accessed from the original tree.

Notice that when the inserted element is already in the tree (EQ branch) the tree t that has just been destroyed is rebuilt. The purely functional version is obtained by removing the ! annotations and returning t in the EQ branch, as it would had not been destroyed.

2.2 Big-Step Operational Semantics

We have developed a big-step operational semantics for this language and a small-step operational semantics which have been proved equivalent. Here we only show the first one to provide *Safe* for a semantics.

In Figure 4 we show the big-step operational semantics for *Safe* expressions. A **judgment** of the form $\Delta, k : e \Downarrow \Theta, k' : v$ means that expression e is successfully reduced to normal form v under heap Δ with $k+1$ regions, ranging from 0 to k, and that a final heap Θ with $k'+1$ regions is produced as a side effect.

A **heap** Δ is a function from fresh variables p (in fact, heap pointers) to closures w of the form $(j, C\,\overline{a_i}^n)$, meaning that the closure resides in region j. If $[p \mapsto w] \in \Delta$ and $w = (j, C\,\overline{a_i}^n)$, we will say that $region(w) = j$ and also that $region(p) = j$.

A **normal form** v is either a basic value c or a construction $C\,\overline{a_i}^n @ j$ to be stored (but not stored yet) in region j. The actual parameters a_i are either basic values or pointers to other closures. Actual region identifiers j are just natural numbers. Formal regions appearing in a function body are either the formal parameter r or the constant $self$.

By $\Delta[p \mapsto w]$ we denote a heap Δ where the binding $[p \mapsto w]$ is highlighted. On the contrary, by $\Delta \cup [p \mapsto w]$ we denote the disjoint union of heap Δ with the binding $[p \mapsto w]$.

The semantics of a complete *Safe* program $d_1; \ldots ; d_n; e$ is the semantics of the main

$$\Delta, k : c \Downarrow \Delta : k : c \qquad [Lit]$$

$$\Delta, k : C\,\overline{a_i}^n @ j \Downarrow \Delta, k : C\,\overline{a_i}^n @ j \qquad [Cons]$$

$$\Delta[p \mapsto w], k : p \Downarrow \Delta, k : w \qquad [Var_1]$$

$$\frac{j \le k \quad l \ne j \quad (\Theta, C\,\overline{a'_i}^n) = copy(\Delta, j, C\,\overline{a_i}^n)}{\Delta[p \mapsto (l, C\,\overline{a_i}^n)], k : p@j \Downarrow \Theta, k : C\,\overline{a'_i}^n @ j}\ [Var_2]$$

$$\Delta \cup [p \mapsto w], k : p! \Downarrow \Delta, k : w \qquad [Var_3]$$

$$\frac{\Sigma \vdash f\ \overline{x_i}^n = e \quad \Delta, k+1 : \hat{e} \Downarrow \Theta, k'+1 : v}{\Delta, k : f\ \overline{a_i}^n \Downarrow \Theta\,|_{k'}, k' : v}\ [App_1]$$

$$\frac{\Sigma \vdash f\ \overline{x_i}^n\ r = e \quad \Delta, k+1 : \hat{e} \Downarrow \Theta, k'+1 : v}{\Delta, k : f\ \overline{a_i}^n @ j \Downarrow \Theta\,|_{k'}, k' : v}\ [App_2]$$

$$\frac{\Delta, k : e_1 \Downarrow \Theta, k' : c \quad \Theta, k' : e[c/x_1] \Downarrow \Psi, k'' : v}{\Delta, k : \textbf{let}\ x_1 = e_1\ \textbf{in}\ e \Downarrow \Psi, k'' : v}\ [Let_1]$$

$$\frac{\Delta, k : e_1 \Downarrow \Theta, k' : C\,\overline{a_i}^n @ j \quad j \le k' \quad fresh(p) \quad \Theta \cup [p \mapsto (j, C\,\overline{a_i}^n)], k' : e[p/x_1] \Downarrow \Psi, k'' : v}{\Delta, k : \textbf{let}\ x_1 = e_1\ \textbf{in}\ e \Downarrow \Psi, k'' : v}\ [Let_2]$$

$$\frac{C = C_r \quad \Delta, k : e_r[\overline{a_j/x_{rj}}^{n_r}] \Downarrow \Theta, k' : v}{\Delta[p \mapsto (j, C\,\overline{a_i}^{n_r})], k : \textbf{case}\ p\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^m \Downarrow \Theta, k' : v}\ [Case]$$

$$\frac{C = C_r \quad \Delta, k : e_r[\overline{a_j/x_{rj}}^{n_r}] \Downarrow \Theta, k' : v}{\Delta \cup [p \mapsto (j, C\,\overline{a_i}^{n_r})], k : \textbf{case!}\ p\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^m \Downarrow \Theta, k' : v}\ [Case!]$$

FIGURE 4. *SAFE* **big-step operational semantics**

expression e in an environment Σ containing the declarations $d_1, \ldots, d_n$ of all the functions.

Rules *Lit* and *Cons* just say that basic values and constructions are normal forms. Rule *Cons* does not create a closure. Closures are actually created by rule *Let*$_2$ which is the only one allocating fresh memory.

Rule *Var*$_1$ brings a copy of a closure into the main expression. Rule *Var*$_2$ makes a complete copy of the DS pointed to by a variable p into a new region j. Function *copy* follows the pointers in recursive positions of the original structure residing in region l and creates in region j a copy of all recursive closures except for the root closure $C\,\overline{a_i}^n$. Should *copy* find a dangling pointer during the traversal, the whole rule would fail and the derivation would be stuck at this rule. If there is no failure, then the main expression becomes a copy $C\,\overline{a'_i}^n$ of this root closure where the pointers a_i in recursive positions pointing to closures in region l have been replaced by pointers a'_i to the corresponding closures in region j. The pointers in non recursive positions of all the copied closures are kept identical in the new closures. This implies that both DSs, the old and the new, may share some sub-structures. For instance, if the original DS is a list of lists, the structure created by *copy* is a copy of the outermost list, while the innermost lists become shared between the old and the new list.

Rule Var_3 is similar to rule Var_1 except for the fact that the binding $[p \mapsto w]$ is deleted and p does not belong to the domain of the resulting heap. This action may create dangling pointers in the living heap as some closures may have free occurrences of p.

Rules App_1 and App_2 show when a new region is created. Notice that the body of the function is executed in a heap with $k+2$ regions. The occurrences of $\hat{e}$ in the rules respectively mean $e[\overline{a_i/x_i}^n, k+1/self]$ and $e[\overline{a_i/x_i}^n, k+1/self, j/r]$. That is, the formal identifier *self* is bound to the new region $k+1$ so that the function body may create DSs in this region or pass this region as a parameter to function calls. By $\Theta\mid_{k'}$ we denote the heap Θ restricted to closures belonging at most to region k'. In other words, before returning from the function, all closures created in region $k'+1$ are deleted. This action is another source of possible dangling pointers.

Rules Let_1 and Let_2 show the eagerness of the language: first, the auxiliary expression e_1 is reduced to normal form and then the main expression is evaluated. The occurrences of the program variable x_1 are replaced either by the normal form if it is a basic value, or by a pointer to it if it is a construction. Notice also that a construction is converted into a closure only if it is bound to a variable in a **let** expression.

Finally, rule *Case* is the usual one while rule *Case*! expresses what happens in a destructive pattern matching: the binding of the discriminant variable p disappears from the heap. This action is the last source of possible dangling pointers.

The type system briefly described below guarantees that all the destructive actions can be taken without danger provided the program is well-typed.

Proposition 1 *If $\Delta, k : e \Downarrow \Theta, k' : v$ is derivable, then $k = k'$.*
Proof: Straightforward, by induction on the depth of the derivation.

In the following, we will feel free to write the derivable judgments as $\Delta, k : e \Downarrow \Theta, k : v$.

By $fv(e)$ we denote the set of free variables of expression e, excluding function names and region variables, and by $frv(e)$, the set of free region variables of e. By *Fresh*, we denote the set of names from which the function *fresh* in rule Let_2 select fresh names, and by $\mathbb{N}$ the set of natural numbers. Also, by $dom(\Delta)$ and $range(\Delta)$ we denote the following sets:

$$\begin{array}{lcl} dom(\Delta) & \overset{\text{def}}{=} & \{p \mid [p \mapsto w] \in \Delta\} \\ range(\Delta) & \overset{\text{def}}{=} & \bigcup\{fv(w) \mid [p \mapsto w] \in \Delta\} \end{array}$$

Proposition 2 *If e is an expression satisfying $fv(e) \subseteq Fresh$ and $frv(e) \subseteq \mathbb{N}$, and $\Delta, k : e \Downarrow \Theta, k : v$ is derivable, and $range(\Delta) \subseteq Fresh$, then all judgments $\Delta_i, k_i : e_i \Downarrow \Theta_i, k_i : v_i$ of the derivation satisfy:*

1. *$fv(e_i) \cup fv(v_i) \subseteq Fresh$.*

2. *$frv(e_i) \cup frv(v_i) \subseteq \mathbb{N}$.*

Proof: By induction on the depth of the derivation.

For this reason, in the rules of Figure 4 we have systematically used letter p —intended to mean a pointer— when referring to free variables, and letter j —intended to mean a natural number— when referring to free region variables.

$$\frac{\begin{array}{ccc}\Gamma_1^P \vdash e_1 : s_1 & \Gamma_2^P + [x_1 : d_1] \vdash e : s & d_1 = \overline{s_1} \\ C = \mathit{shareall}(x_1, e) & R = \mathit{sharerec}(x_1, e) & P \cap R = \emptyset\end{array}}{\Gamma_1^P \rhd_{C \backslash R} \Gamma_2^P \vdash \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e : s}\ [\text{LET2}]$$

FIGURE 5. A sample rule for let expressions

2.3 The Type System

In this section we outline some aspects of *Safe*'s type system. For a complete view, the reader is addressed to [6]. As shown in the examples in Section 2.1 *Safe* types are similar to Hindley-Milner types where regions are explicit, and where a ! annotation represents a condemned structure.

In Figure 5 we show a sample type rule corresponding to a **let** expression in which the intermediate structure x_1 is destroyed in the main expression e. There, s and s_1 represents read-only types and d_1 is the condemned type corresponding to s_1 (denoted $d_1 = \overline{s_1}$). P is the set of read-only parameters of the function which the **let** belongs to. Roughly speaking, the rule establishes that:

- x_1 must not share any of its recursive substructures with the read-only parameters P.
- Neither the variables which are destroyed in e_1, nor any variable sharing a recursive substructure with them, must be referenced in the main expression e.
- None of the variables sharing a non-recursive substructure with x_1 should be destroyed either in e_1 or e.

The operator $\rhd_C$ over type environments takes care of enforcing the above rules. Its definition is such that:

- Those variables belonging to both environments must have the same underlying type, and in such a case the second environment prevails.
- If a variable is condemned in the first environment it may not appear in the second one.
- C is the set of variables that may not be condemned in any of the two environments.

Additionally, there is a general invariant of the type system telling that if a variable x is condemned by a **case**! expression then not only x but all the variables sharing a recursive substructure of x have a condemned type in the type environment.

Notice the use of functions *shareall* and *sharerec* in the rule. These are the functions we are defining in the next section.

3 SHARING ANALYSIS

In this section we define an analysis that approximates the sharing relations between the variables of a program.

3.1 Sharing relations

In order to capture sharing, we define four different binary relations between variables:

Definition 4 *Given two variables x and y, in scope in an expression,*

1. *$x \lhd\!\sim y$ denotes that x is a recursive descendant of y.*
2. *$x \bigtriangleup\!\sim y$ denotes that x shares a recursive descendant of y.*
3. *$x \lhd y$ denotes that x is any substructure of y.*
4. *$x \bigtriangleup y$ denotes that x shares any substructure of y.*

We make note that all the four relations are reflexive, $\bigtriangleup$ is also symmetric, and $\lhd\!\sim$ and $\lhd$ are transitive. Moreover, the following implications hold:

$$x \lhd\!\sim y \Rightarrow x \lhd y \Rightarrow x \bigtriangleup y$$
$$x \lhd\!\sim y \Rightarrow x \bigtriangleup\!\sim y \Rightarrow x \bigtriangleup y$$

but $\lhd$ and $\bigtriangleup\!\sim$ are not necessarily related.

The interpretation defined below does a top-down traversal of a program, accumulating these relations as soon as bound variables become free variables.

For the non-symmetric relations, the accumulator parameters are treated as functions $Var \rightarrow \{Var\}$, $R(x)$ giving the set of all y such that yRx (i.e. $(y,x) \in R$) . Whenever convenient we will write $R = [x \rightarrow S]$ to indicate that $S = R(x)$.

The symmetric relation $\bigtriangleup$ is kept in a set of sets of variables. If $S \in \bigtriangleup$ then $x \bigtriangleup y$ for all $x,y \in S$.

Based on the above considerations, we will define an abstract interpretation S (meaning *sharing*) which, given an expression e delivers the following seven sets:

$$(SubRP, ShRP, SubP, SubR, ShR, Sub, Sh)$$

which contain respectively all the variables z such that $e \lhd\!\sim z$, $e \bigtriangleup\!\sim z$, $e \lhd z$, $z \lhd\!\sim e$, $z \bigtriangleup\!\sim e$, $z \lhd e$ and $z \bigtriangleup e$[2].

3.2 Function signatures

In order to achieve a modular analysis, we decide to reflect the result of the analysis of a function f in a *function signature*. We keep these signatures in a function environment ρ. A function signature $\rho(f)$ has the following type: $(\{Int\},\{Int\},\{Int\},\{Int\},\{Int\},\{Int\},$ $\{Int\})$

The meaning of the seven sets is as above, except for the fact that these contain only parameter indexes instead of all the (free and bound) variables of the body expression. This is reasonable as the effect of a function should be completely reflected in the relationship between the parameters and the result.

In Figure 6 the interpretation S for expressions is defined. We explain it in detail later. When applied to a function definition $f\ x_1 \ldots x_n = e$, it is straightforward to extract the

[2]We use e here as an abbreviation of the anonymous variable pointing to the result after e is evaluated.

signature of the function while computing the least fixpoint, in case it is recursive. The interpretation of a definition adds the signature of the new definition to the signatures environment:

$$\begin{array}{l} S[\![f\ x_1 \dots x_n = e]\!]\ \rho = \mathit{fix}\ (\lambda\rho.\rho\ [f \rightarrow \mathit{extract}([x_1,\dots,x_n], S[\![e]\!]\ R_0\ R_0\ R_0\ R_0\ \rho)])\ \rho_0 \\ \quad \textbf{where}\ \ \rho_0 = \rho\ [f \rightarrow (\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset)] \\ \qquad\qquad\quad R_0 = \{(x_i,x_i) \mid i \in \{1..n\}\} \\ \qquad\qquad\quad \mathit{extract}(xs,(S_1,\dots,S_7)) = (\{i \mid x_i \in xs \cap S_1\},\dots,\{i \mid x_i \in xs \cap S_7\}) \end{array}$$

where $\rho\ [f \rightarrow s]$ either adds signature s for f or replaces it in case there was already one for it. As function S and function *extract* are monotone over a finite lattice, the least fixpoint exists and can be computed using Kleene's ascending chain.

Given a whole program $P = dec_1;\dots dec_k;\ e$ the analysis first builds an increasing function environment and then analyses the main expression given initially empty relations (there are no free variables but function names):

$$S[\![P]\!] = S[\![e]\!]\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ (S[\![dec_k]\!]\ (\dots(S[\![dec_1]\!]\ [\,])\dots))$$

Notice that the right hand sides of the definitions are analysed given relations where each parameter is only related to itself. This means that the signatures are calculated assuming that all the parameters are disjoint. When they are not, the function application computes the additional sharing.

3.3 Interpretation of expressions

We explain now the details of the interpretation S for expressions. By abuse of notation, we will write $Sh(x)$ even though Sh is not a function, with the following convention:

$$Sh(x) \stackrel{\text{def}}{=} \bigcup\{S \mid x \in S \wedge S \in Sh\}$$

A basic value c neither has substructures nor is part of any structure, so its interpretation is just seven empty sets.

If x is returned as the result of a function, we use the information in the accumulator parameters of S to extract all the relevant information about its sharing. Notice that, from the operational semantics point of view, $x!$ is just the same structure as x, hence its interpretation. The semantics of $x@r$ is the creation of a copy of the recursive part of x in a new region r. As a consequence, the first, third, fourth and fifth sets of the interpretation are empty, and the third set excludes those variables with the same (recursive) type as x. The non-recursive part of $x@r$ is shared with x and potentially with any variable sharing substructures with x, hence the seveth set. However only the non-recursive children of x may be children of $x@r$, hence the sixth set.

The interpretation of a function application $g\ \overline{a_i}^m @r$ returning a DS is rather involved. Regarding the first set, the recursive descendant relation is transitive. So, the result of g is a recursive descendant of a variable z if and only if an actual parameter a_j of g is a recursive descendant of z and the result of g is a recursive descendant of a_j. The same transitivity applies to the third set. Regarding the second set, the result of g shares a recursive descendant of a variable z if an actual parameter a_j of g shares a recursive descendant of z, and a_j is in sharing relation with the result of g. This probably will give us more

$$S\,[\![c]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho = (\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset)$$

$$\begin{aligned}S\,[\![x]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho = (&\{z \mid x \in \mathit{SubR}(z)\},\\ &\{z \mid x \in \mathit{ShR}(z)\},\\ &\{z \mid x \in \mathit{Sub}(z)\},\\ &\mathit{SubR}(x),\mathit{ShR}(x),\mathit{Sub}(x),\mathit{Sh}(x))\end{aligned}$$

$$S\,[\![x!]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho = S\,[\![x]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho$$

$$\begin{aligned}S\,[\![x@r]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho = (&\emptyset,\\ &\{z \mid x \in \mathit{ShR}(z) \wedge \mathit{type}(z) \neq \mathit{type}(x)\},\\ &\emptyset,\emptyset,\emptyset,\\ &\mathit{Sub}(x) - \mathit{SubR}(x),\mathit{Sh}(x))\end{aligned}$$

$$\begin{aligned}S\,[\![g\ \overline{a_i}^m @r]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho = (&\{z \mid \exists j \in \mathit{SubRPg}.a_j \in \mathit{SubR}(z)\},\\ &\{z \mid \exists j \in \mathit{ShRPg}.a_j \in \mathit{ShR}(z)\},\\ &\{z \mid \exists j \in \mathit{SubPg}.a_j \in \mathit{Sub}(z)\},\\ &\textstyle\bigcup_j\{\mathit{SubR}(a_j) \mid j \in \mathit{SubRg}\},\\ &\textstyle\bigcup_j\{\mathit{ShR}(a_j) \mid j \in \mathit{ShRg}\},\\ &\textstyle\bigcup_j\{\mathit{Sub}(a_j) \mid j \in \mathit{Subg}\},\\ &\textstyle\bigcup_j\{\mathit{Sh}(a_j) \mid j \in \mathit{Shg}\})\end{aligned}$$

where $(\mathit{SubRPg},\mathit{ShRPg},\mathit{SubPg},\mathit{SubRg},\mathit{ShRg},\mathit{Subg},\mathit{Shg}) = \rho(g)$

$$\begin{aligned}S\,[\![C\ \overline{a_i}^m @r]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho = (&\emptyset,\\ &\{z \mid \exists a_j \in \mathit{ShR}(z)\},\\ &\emptyset,\\ &\textstyle\bigcup_j\{\mathit{SubR}(a_j) \mid j \in \mathit{RecPos}(C)\},\\ &\textstyle\bigcup_j\{\mathit{ShR}(a_j) \mid j \in \mathit{RecPos}(C)\},\\ &\textstyle\bigcup_j\{\mathit{Sub}(a_j) \mid j \in \{1..m\}\},\\ &\textstyle\bigcup_j\{\mathit{Sh}(a_j) \mid j \in \{1..m\}\})\end{aligned}$$

$$S\,[\![\textbf{let}\ x_1 = e_1\ \textbf{in}\ e]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho = (S\,[\![e]\!]\ \mathit{SubR}_2\ \mathit{ShR}_2\ \mathit{Sub}_2\ \mathit{Sh}_2\ \rho)\backslash\{x_1\}$$

where $(\mathit{SubRP_1},\mathit{ShRP_1},\mathit{SubP_1},\mathit{SubR_1},\mathit{ShR_1},\mathit{Sub_1},\mathit{Sh_1}) = S\,[\![e_1]\!]\ \mathit{SubR}\ \mathit{ShR}\ \mathit{Sub}\ \mathit{Sh}\ \rho$

$$\begin{aligned}&\mathit{SubR}_2 = (\mathit{SubR} \cup [x_1 \mapsto \mathit{SubR_1}] \cup \{[z \mapsto \{x_1\}] \mid z \in \mathit{SubRP_1}\})^*\\ &\mathit{ShR}_2 = \mathit{ShR} \cup [x_1 \mapsto \mathit{ShR_1}] \cup \{[z \mapsto \{x_1\}] \mid z \in \mathit{ShRP_1}\} \cup \mathit{SubR}_2\\ &\mathit{Sub}_2 = (\mathit{Sub} \cup [x_1 \mapsto \mathit{Sub_1}] \cup \{[z \mapsto \{x_1\}] \mid z \in \mathit{SubP_1}\})^*\\ &\mathit{Sh}_2 = \mathit{Sh} \cup \{\{x_1\} \cup \mathit{Sh}_1\} \uplus (\mathit{Sub}_2 \cup \mathit{ShR}_2)\end{aligned}$$

$$S\,[\![\textbf{case}\,x\,\textbf{of}\,\overline{C_i\,\overline{x_{ij}}^{n_i} \to e_i}]\!]\,\mathit{SubR}\,\mathit{ShR}\,\mathit{Sub}\,\mathit{Sh}\,\rho = \textstyle\bigcup_i((S\,[\![e_i]\!]\ \mathit{SubR}_i\ \mathit{ShR}_i\ \mathit{Sub}_i\ \mathit{Sh}_i\ \rho)\backslash\{\overline{x_{ij}}^{n_i}\})$$

where $\mathit{SubR}_i = (\mathit{SubR} \cup [x \mapsto \{x_{ij} \mid j \in \mathit{RecPos}(C_i)\}]$
$\qquad \cup \{[x_{ij} \mapsto \mathit{SubR}(x)\backslash\{x\}] \mid j \in \mathit{RecPos}(C_i)\})^*$

$$\begin{aligned}&\mathit{ShR}_i = \mathit{ShR} \cup \{[x_{ij} \mapsto \mathit{ShR}(x)] \mid j \in \mathit{RecPos}(C_i)\} \cup \mathit{SubR}_i\\ &\mathit{Sub}_i = (\mathit{Sub} \cup [x \mapsto \{x_{ij} \mid j \in \{1..n_i\}] \cup \{[x_{ij} \mapsto \mathit{Sub}(x)\backslash\{x\}] \mid j \in \{1..n_i\}\})^*\\ &\mathit{Sh}_i = (\mathit{Sh} \cup \{\{y,x_{ij}\} \mid y \in \mathit{Sh}(x) \wedge j \in \{1..n_i\}\}) \uplus (\mathit{Sub}_i \cup \mathit{ShR}_i)\end{aligned}$$

FIGURE 6. Definition of the abstract interpretation S

variables than the ones actually sharing a recursive descendant of z, but it is a safe approximation. This is a place where signatures may lose information. The fourth and sixth sets are defined taking respectively into account the transitivity of the relations $\lhd\!\sim$ and $\lhd$. The fifth and seventh sets are safe, but may be imprecise, approximations to respectively the set of variables sharing a recursive substructure and sharing any substructure with the result of g. The interpretation of a function application $g\ \overline{a_i}^m$ of a function g not having an output region as a parameter is identical to the previous one.

In the interpretation of a data construction $C\ \overline{a_i}^m@r$, the first and third sets are empty because a newly created DS cannot be a substructure of any other. However, it will share a recursive descendant of a variable z if any of its substructures a_j already shared it. Any variable being a recursive descendant of a recursive parameter a_j of C will also be a recursive descendant of the construction. The set $RecPos(C)$ contains the recursive positions of the constructor C. A similar reasoning can be applied to the fifth set containing the variables which share a recursive descendant of the construction. The next set definition gets profit from the transitivity of the $\lhd$ relation. The last set consists also of a union over all the parameters of C, because the construction inherits the sharing of all its substructures.

The **let** expression introduces a new bound variable x_1 which may appear free in the main expression e. First, the interpretation of the auxiliary expression e_1 is launched and the sharing created by it is accumulated in the parameters. Then, the main expression e is interpreted taking into account the new sharing. If R represents a reflexive, non-symmetric, transitive relation, by R^* we mean its reflexive, transitive closure. Operator $\uplus$ computes the union of a reflexive, symmetric and non-transitive relation and a reflexive, non-symmetric transitive one. Notice that the addition of $SubR_2$ to ShR_2, and the addition of this latter set and that of Sub_2 to Sh_2 just implements the inclusion of the underlying relations, as explained above. Finally, the information related to x_1 is deleted as the variable will not be in scope in the context.

As usual, the interpretation of a **case** is the least upper bound of the interpretation of its alternatives, and this involves a loss of information. Before each alternative is interpreted, we accumulate the sharing of the bound variables x_{ij} introduced by it. Part of this sharing is straightforward: all these variables are descendants of the parent structure x and some of them are recursive descendants of it. Additionally, if we have $y \in SubR(x) \land y \neq x$, that means $y \lhd\!\sim x$. As there is no more information available, it may be the case that $y \lhd\!\sim x_{ij}$ for some recursive child of x. The only safe way to cope with this possibility is to include in $SubR_i$ the pairs $y \lhd\!\sim x_{ij}$ for all the recursive children of x. A similar reasoning applies to the rest of the sets.

The interpretation of **case!** is the same as the previous one. Although the discriminant variable is being condemned we cannot eliminate its sharing information as we do not know whether the rest of variables are safely used. For example, we could write $z =$ **case**! x **of** $C\ y \rightarrow x$. We should say that, even unsafely, variables x and z share a substructure.

4 IMPLEMENTATION AND EXAMPLES

In this section we present the implementation of the analysis and give some examples of functions to which it has been applied. We have defined a concrete sugared syntax for *Safe* in which programs look very much like Haskell programs, i.e. functions are defined by means of equations and pattern matching, guards and **where** clauses are allowed, as well as data type declarations and infix operators and constructors.

A complete front-end has been developed by using standard tools such as lexical analyzer and parser generators. In Figure 7 we show its phases. The renamer phase ensures that every identifier is well defined and that every bound variable is given a different name. A Hindley-Milner type inference is done at this level in order to reject ill-typed programs, and to provide report messages related to the sugared syntax. Also, the sharing analysis needs the underlying type of a variable and the recursive positions of data constructors (cf. Figure 6). This phase decorates each expression in the abstract syntax tree with its Hindley-Milner type.

The desugarer transforms the high-level syntax into the *Safe* core syntax presented in Section 2. During this transformation new bound variables may be introduced. They are given appropriate types and fresh names.

After these steps, the sharing analysis described in this paper is done. Its main function has the following type:

```
analyzeProg :: Prog TypeExp -> Prog (TypeExp, SharingInfo)
```

That is, given a program decorated with Hindley-Milner types, it returns a program additionally decorated with sharing information. This sharing information has different shapes depending on the entity being decorated:

- If it is a function definition, it consists of its signature.
- If it is an expression, it consists of the four relations accumulated from the beginning of the function body this expression belongs to, up to the root of the expression.
- Binding occurrences of variables are not decorated. Free occurences are decorated in the same way as any other expression.

In this way, it is easy for the following phase —the inference of *Safe* types— to extract the *shareall* and *sharerec* sets for any given variable x in any given context. Then, $ShR(x)$ and $Sh(x)$ give us the desired information.

The front-end and the analysis have been implemented in Haskell by using the GHC compiler. In order to improve efficiency, the analyzer stores the four relations in a single balanced tree, by using the modules `Map` and `Set` of the GHC library [1]. Also, the inverse of the three first relations are kept in the tree. In this way, the symmetric and/or transitive closures, the union, and some other operations on relations needed by the analysis, are done in a more concise and efficient way. In total, about 2.000 Haskell lines have been written.

When applied to the functions defined in Section 2, the analysis computes the following signatures:

$$\begin{aligned}
\rho(\mathit{revauxD}) &= (\{2\},\{2\},\{2\},\{2\},\{2\},\{2\},\{1,2\})\\
\rho(\mathit{revD}) &= (\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\{1\})\\
\rho(\mathit{insertD}) &= (\emptyset,\{1,2\},\emptyset,\emptyset,\{2\},\{1\},\{1,2\})
\end{aligned}$$

which are accurate descriptions of the input-output sharing relations of these functions.

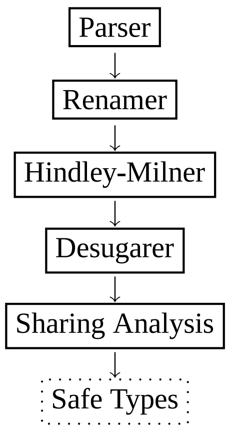

FIGURE 7. Phases of the Safe compiler implementation

Function *revauxD* appends the reverse of its first parameter to its second one. Since it does not reuse the recursive cells of its first parameter, the only remaining recursive sharing is related to its second parameter. Nevertheless the sharing with the non recursive elements of the first list is reflected in the last set of the signature. Function *revD* consists of a simple call to *revauxD* passing it an empty list as the second actual parameter, so the only remaining sharing is that between the non-recursive structures of the input and output lists.

Function *insertD* builds a new tree which shares with the original tree everything but the path from the root to the inserted element. This means that the resulting tree and the original one share both recursive and non-recursive parts. This is the reason why 2 appears in the second, fifth and seventh sets of the signature. Also the resulting tree has x as a non-recursive descendant, so the 1 in the second, sixth and seventh sets.

5 A MORE ELABORATED EXAMPLE

In this section we show a more involved example, a *mergesort* version that does not need additional heap space. In order to give a compact code, the functions shown in this section are sugared versions although the analysis is executed over their desugared versions.

First, we define the usual auxiliary functions to split the input list and merge two ordered lists in a single ordered list. In Figure 8 we show a destructive version of the splitting function. As in the previous examples, there are small differences with a purely functional version. In the base case ($n = 0$) we reuse the list in the ouput; in the recursive case we use a **case**! (written as a destructive pattern) over the argument list. We also have to add @r where necessary.

The sharing analysis produces the following signature for this function:

$$\rho(\mathit{splitD}) = (\emptyset, \{2\}, \emptyset, \emptyset, \emptyset, \{2\}, \{2\})$$

meaning that:

- The result of the function may share a recursive substructure of the argument list, which is obvious.

$$\begin{array}{l}
\mathit{splitD} :: \forall a,\rho.\mathit{Int} \to [a]!@\rho \to \rho \to ([a]@\rho,[a]@\rho)@\rho \\
\mathit{splitD}\ 0\ \mathit{xs}!\ r \qquad\quad = ([\,]@r,\mathit{xs}!)@r \\
\mathit{splitD}\ n\ [\,]!\ r \qquad\quad = ([\,]@r,[\,]@r)@r \\
\mathit{splitD}\ n\ (x:\mathit{xs})!\ r = ((x:\mathit{xs}_1)@r,\mathit{xs}_2)@r \\
\qquad\qquad\qquad\qquad\ \ \textbf{where}\ (\mathit{xs}_1,\mathit{xs}_2) = \mathit{splitD}\ (n-1)\ \mathit{xs}\ r
\end{array}$$

FIGURE 8. Destructive partition of a list

- The argument list may be a child of the result, which is true when n is 0.
- The argument list and the result share some substructure, which again is obvious.

Figure 9 shows the destructive version of the merging function. In the recursive calls to *mergeD* one of the parameters is one of the original lists. But the original list may not be referenced as its top cell has been destroyed by a **case**!, so the original list is rebuilt by reusing its components. This is the only detail to care about. The sharing analysis produces the following signature for this function:

$$\rho(\mathit{mergeD}) = (\{2\},\{1,2\},\{2\},\{2\},\{1,2\},\{2\},\{1,2\})$$

meaning that the argument lists and the result may share recursive and non-recursive substructures one of the other. Notice that only the second argument list may be a recursive child of the result (and viceversa) because we build a new cell for each cell of the first argument while we reuse the second argument list when the first one is empty.

Finally, in Figure 10 we show the destructive mergesort *msortD*, that uses the previously defined functions. Both the input list *xs* and the intermediate results are either destroyed or reused into the result. This allows us to conclude that this function consumes a constant additional heap space. In [6] we proved this by induction on the length of the argument list. The sharing analysis produces the following signature for this function:

$$\rho(\mathit{msortD}) = (\{1\},\{1\},\{1\},\{1\},\{1\},\{1\},\{1\})$$

meaning that the argument list and the result may share recursive and non-recursive substructures one of the other.

Recall that this sharing analysis does not take into account the fact that some substructures are destroyed because we do not know yet if the program is type-safe. In this sense the analysis is an upper approximation of the sharing.

6 RELATED WORK AND CONCLUSIONS

There are many works devoted to sharing analysis in functional and logic languages, some of them rather old. In the functional field, the aim of most analyses has been performing part of the garbage collection at compile time, or detecting when destructive updating of data structures could be done safely.

In [3] a reference count of shared data is done at compile time by using abstract interpretation on a first order, eager functional language with updatable arrays. The abstract domains consist of just natural numbers. In order to have a terminating analysis the

```
$mergeD :: \forall a,\rho.[a]!@\rho \rightarrow [a]!@\rho \rightarrow \rho \rightarrow [a]@\rho$
$mergeD\ [\,]!\ ys!\ r$              $= ys!$
$mergeD\ (x : xs)!\ [\,]!\ r$        $= (x : xs!)@r$
$mergeD\ (x : xs)!\ (y : ys)!\ r =$
          $|\ x \leq y$       $= (x : mergeD\ xs\ (y : ys!)@r\ @r)@r$
          $|\ otherwise = (y : mergeD\ (x : xs!)@r\ ys\ @r)@r$
```

FIGURE 9. Destructive ordered merge of two lists

```
$msortD :: \forall a,\rho.[a]!@\rho \rightarrow \rho \rightarrow [a]@\rho$
$msortD\ xs\ r$
  $|\ n \leq 1$        $= xs!$
  $|\ otherwise = mergeD\ (msortD\ xs_1\ @r)\ (msortD\ xs_2\ @r)\ @r$
   **where** $(xs_1,xs_2) = splitD\ (n\ `div`\ 2)\ \ xs\ @r$
          $n = length\ xs$
```

FIGURE 10. Destructive mergesort

domains are restricted to finite intervals $\{1\ldots n\}$, for an arbitrary n, and topped with ∞ meaning "too much sharing". An array based quicksort algorithm using updating in place is shown correct by the analysis.

Jones and Le Métayer [5] use also abstract interpretation on a first order, eager functional language with non-homogeneous lists in order to avoid allocation of fresh cells and to reuse instead cells that will not be needed by the rest of the computation. Their analysis is a combination of sharing and absence analyses and the abstract domains are nested tuples of booleans. Again, domains are forced to be finite by bounding the nesting depth of the tuples by an arbitrary number n. The analysis looks rather complex and not very efficient as it does several traversals of the same code. Also the authors do not show evidence of having implemented it.

In [4], the authors use non-standard techniques, such as context free languages and intersection between such languages, in order to perform garbage collection at compile time. The language analyzed is a first order subset of LISP. The idea is to detect cells created by a function and not belonging to the result. Such those cells are disposed at the end of the function body. They show good results for some LISP test programs.

In the logic programming field [2] provides a comprehensive survey of sharing analyses. Sharing is important here for much the same reasons than in the functional field but also to detect opportunities for parallel evaluation.

The main novelty of our approach is, in the one hand the context —a functional language with explicit destruction— and in the other its modular approach. In the previously described works, the analyses are done at the whole program level while ours is done function by function. Reflecting the result of a function analysis in a signature provides the connection between the different functions of the program. The subsequent type inference phase will also be done function by function, so the sharing signature can be seen as an annotation associated to the function type.

We find the approach to be precise enough for successfully analysing the examples we have tried so far, but more experimentation is needed in order to better asses the quality of the analysis. As future work, we will prove the correctness of the analysis with respect to the small-step operational semantics (not shown in this paper) of the language.

REFERENCES

[1] S. Adams. Efficient sets –a balancing act. *Journal of Functional Programming*, 3(4):553–561, 1993.

[2] G. Gudjónsson and W. H. Winsborough. Compile-time memory reuse in logic programming languages through update in place. *ACM TOPLAS*, 21(3):430–501, 1999.

[3] P. Hudak. A Semantic Model of Reference Counting and its Abstraction. In *Lisp and Functional Programming Conference*, pages 351–363. ACM Press, 1986.

[4] K. Inoue, H. Seki, and H. Yagi. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM TOPLAS*, 10(4):555–578, 1988.

[5] S. B. Jones and D. Le Metayer. Compile Time Garbage Collection by Sharing Analysis. In *Int. Conf. on Functional Programming and Computer Architecture*, pages 54–74. ACM Press, 1989.

[6] R. Peña and C. Segura. A First-Order Functional Language for Reasoning about Heap Consumption. In *16th International Workshop on Implementation of Functional Languages*, pages 64–80, 2004.