Shape Analysis in a Functional Language by Using Regular Languages[☆]

Manuel Montenegro, Ricardo Peña, Clara Segura

Abstract

Shape analysis is concerned with the compile-time determination of the 'shape' the heap may take at runtime, meaning by this the pointer chains that may happen within, and between, the data structures built by the program. This includes detecting alias and sharing between the program variables.

Functional languages facilitate somehow this task due to the absence of variable updating. Even though, sharing and aliasing are still possible. We present an abstract interpretation-based analysis computing precise information about these relations. In fact, the analysis gives an information more precise than just the existence of sharing. It informs about the paths through which this sharing takes place. This information is critical in order to get a modular analysis and not to lose precision when calling an already analysed function.

The motivation for the analysis in our case is the need of knowing at compile time which variables are at risk of containing dangling pointers at runtime, in a language with explicit memory disposal primitives.

The main innovation with respect to the literature is the use of regular languages to specify the possible pointer paths from a variable to its descendants. This additional information makes the analysis much more precise while still being affordable in terms of efficiency. We have implemented it and give convincing examples of its precision.

Keywords: Functional languages, Abstract interpretation, Shape analysis, Points-to analysis, Regular languages.

[☆]Work partially funded by the projects TIN2013-44742-C4-3-R (CAVI-ART), TIN2008-06622-C03-01/TIN (STAMP), and S2009/TIC-1465 (PROMETIDOS).

1. Motivation

Shape analysis is concerned with statically determining the connections between program variables through pointers in the heap that may occur at runtime. As particular cases, it includes sharing and alias between variables. To know the shape of the heap for every possible program execution is undecidable in general, but the analysis computes an over-approximation of this shape. This means that it may include sharing relations that will never happen at runtime.

Much work has been done in imperative languages (see Section 7), specially for C. There, the sharing detection is aggravated by the fact that variables are mutable, and they may point to different places at different times. We have addressed the problem for a first-order functional language. This simplifies some of the difficulties since variables do not mutate. A consequence is that the inferred relations are immutable considering different parts of the program text. Another consequence is that the heap is never updated. It can only be increased with new data structures, or decreased by the garbage collector. But the latter cannot produce effects in its live part.

Our analysis puts the emphasis on three properties: (1) modularity; (2) precision; and (3) efficiency. For the sake of scalability, it is important for the analysis to be modular. The results obtained for a function should summarise the shape information so that the user functions should be able to compute all the sharing produced when calling it. Looked at from outside, and given that the language is functional, a function may only create sharing between its result and its arguments, or between the results themselves, but it can never create new sharing between the arguments. The internal variables become dead after the call, so the result of analysing a function only contains its input-output sharing behaviour. Differently from previous works, we compute the *paths* through which this sharing may occur in a precise way. This information is used to propagate to the caller the sharing created by a call. In this way, large programs need not to be analysed globally, but just a function at a time.

The motivation for our analysis is a safety type system we have developed for a functional language, called *Safe*, with explicit memory disposal [1]. This feature may create dangling pointers at runtime. The language also provides automatically allocated and deallocated heap regions, instead of having a runtime garbage collector. This feature can never create dangling pointers, so it plays no role in the current work and we will not mention it anymore.

```
unshuffle []!     = ([],[])
unshuffle (x:xs)! = (x:ys2, ys1)
                    where (ys1,ys2) = unshuffle xs
merge []!     ys!  = ys
merge (x:xs)! []!  = x:xs
merge (x:xs)! (y:ys)! | x <= y    = x : merge xs     (y:ys)
                      | otherwise = y : merge (x:xs) ys
msort []!  = []
msort [x]! = [x]
msort xs!  = merge (msort xs1) (msort xs2)
             where (xs1, xs2) = unshuffle xs
```

Figure 1: *mergesort* algorithm in constant heap space

The explicit memory disposal is achieved by means of a destructive pattern matching, denoted with symbol !, or a **case**! expression. By applying any of them, we can reuse the cell corresponding to the parameter or variable affected by it. This feature may be used in our language to implement data structures whose updating needs no additional heap space or constant heap space functions over data structures, see [2] for examples. As an example, in Figure 1 we show an implementation of the mergesort algorithm for sorting a list in constant heap space. Each cell of the original list is disposed by *unshuffle*; lists xs_1 and xs_2 are disposed by the recursive calls to *msort*; and finally the results of the recursive calls are disposed by *merge*.

The type inference algorithm [3] assigns the program variables safety marks: `d` for disposed, `s` for safe and `r` for in-danger variables. Each time a variable is marked as disposed, all those variables that may point to cells belonging to its recursive substructure are marked as in-danger, because they can potentially contain dangling pointers. The type rules propagate the marks and control how the variables are used. For instance, in a **let** expression in-danger and already disposed variables in the let-bound expression cannot be mentioned in the main expression. We have proved that passing successfully the type inference phase gives total guarantee that there will not be dangling pointers.

So, for typechecking a function, it is critical to know at compile time which variables may point to the disposed data structures, and for this we need a sharing analysis. Our prior prototype shape analysis done in [4] was correct but imprecise at some points. In particular, the type system rejected the mergesort algorithm shown in Figure 1, due to the imprecision of the sharing analysis results. As we will see in more detail in the following section, the reason for this is that it does not suffice knowing that two variables share a common descendant, but we should more precisely know through which paths this sharing occurs, and that is why in this work we introduce regular languages representing paths in the heap.

We believe that the sharing analysis presented here could be equally useful for other purposes, since it provides precise information about the heap shape. Note that some shapes, such as cyclic or doubly linked lists, cannot be created by a functional language, so they are out of the scope of our analysis. But, in some cases, the analysis is capable of asserting that a given structure is a tree, i.e. it does not have internal sharing (see Section 6 for an example).

The main contribution of this paper with respect to [4] is the incorporation of regular languages to our abstract domain. Each word of the language defines a pointer path within a data structure. Having regular languages introduces additional problems such as how to combine them during the analysis, how to compare them, and specially how to guarantee that a fixpoint will be reached after a finite number of iterations. We show that we have increased the precision of our prior analysis, and that the new problems can be tackled with a reasonable efficiency.

The plan of the paper is as follows: Section 2 provides a mild introduction to the analysis via a small example. Then, Sections 3, 4 and 5 contain all the technical material about the abstract domain, abstract interpretation rules, correctness, implementation, widening, and cost of the operations done on regular languages. Section 6 gives abundant examples of the sharing results obtained by our analysis and their corresponding running times. Finally, Section 7 concludes and discuss some related work.

This paper is an extended version of [5]. The additional material here mainly concerns sections 4, 5, and 6. In the first one, a much detailed proof of correctness is given. In the second one, we compare two alternative implementations (instead of the single one presented in [5]) in which regular languages are represented either by nondeterministic automata or by regular expressions. Section 6 provides more examples than [5], and additionally the detailed running times of the analysis. It compares three different implementations: the one using regular expressions, and other two based on automata but written in Haskell or C.

$prog$	$\rightarrow$	$\overline{data}; \overline{dec}; e$	{*Core-Safe* program}
			{function definition}
dec	$\rightarrow$	$f\ \overline{x}\ = e$	{recursive, polymorphic}
			{expressions}
e	$\rightarrow$	c	{basic type literal}
		$\mid x$	{variable}
		$\mid f\ \overline{a}$	{function application}
		$\mid C\ \overline{a}$	{constructor application}
		$\mid$ **let** $x_1 = e_1$ **in** e_2	{non-rec., monomorphic}
		$\mid$ **case** x **of** $\overline{alt}$	{case expression}
			{application argument}
a	$\rightarrow$	c	{basic type literal }
		$\mid x$	{variable}
alt	$\rightarrow$	$C\ \overline{x} \rightarrow e$	{case alternative}

Figure 2: Simplified *Core-Safe* syntax

2. Shape Analysis by Example

Our reference language *Safe* is a first-order eager language with a syntax similar to Haskell's. The destruction facilities explained in the previous section are not relevant to the shape analysis so we will not consider them in the rest of the paper.

The compiler's front-end processes *Full-Safe* and produces a bare-bones functional language called *Core-Safe*. This transformation desugars pattern matching into **case** expressions, transforms **where** clauses into **let** expressions, collapses several function-defining equations into a single one, and ensures unique names for the variables. In Figure 2 we show a simplified *Core-Safe*'s syntax. A program *prog* is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e whose value is the program result. The abbreviation $\overline{x}$ stands for a sequence $x_1 \cdots x_n$, and we denote its length n as $|\overline{x}|$. In Figure 3 we show the translation to *Core-Safe* of the *msort* function of Figure 1 without including the destructive pattern matching.

Our shape analysis infers for each function a sharing signature which captures the sharing generated by the function definition between the result of the function, represented as *res*, and its parameters. A signature is a set

```
msort xs = case xs of
             []   -> []
             x:xx ->
               case xx of
                []  -> x:[]
                _:_ -> let p  = unshuffle xs            in
                       let y1 = case p of (s1,s2) -> s1 in
                       let y2 = case p of (w1,w2) -> w2 in
                       let z1 = msort y1                in
                       let z2 = msort y2                in
                       merge z1 z2
```

Figure 3: Function *msort* in *Core-Safe*

of pairs like the following:

$$\{res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} xs_1, res \xrightarrow{2} \bullet \xleftarrow{2} xs_2\}$$

If the function has three parameters, xs_1, xs_2 and xs_3, this signature says that the result of the function may share with the first and the second parameters, but not with the third one.

The sharing information is represented by means of regular languages, here shown as regular expressions, that approximate all the possible paths in the heap through which two variables may point to the same heap cell. In order to write the paths we use natural numbers representing the positions in the constructor cell that point to the following cell in the path. For example, when considering list cells, number 1 represents the path leading to the first element of the list, while number 2 represents the path leading to the tail of the list. So, a regular expression 2^*1 represents all the paths that take the tail of the list any number of times and then take the head, i.e. all the paths leading to the elements of the list. So, the example signature above represents that the result shares its elements with the parameter xs_1 and its tail with the parameter xs_2.

Consider now, the `msort` example shown in Figure 1. Our shape analysis infers the following sharing information for the functions `unshuffle` and

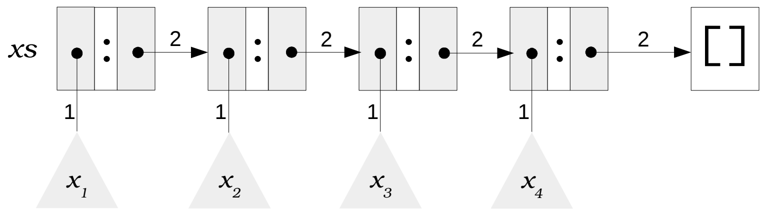


Figure 4: Heap state before executing `unshuffle` $[x_1, x_2, x_3, x_4]$

`merge`:

$$\begin{array}{lcl} \Sigma(\mathit{unshuffle}) & = & \{res \xrightarrow{12^*1+22^*1} \bullet \xleftarrow{2^*1} xs\} \\ \Sigma(\mathit{merge}) & = & \{res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} xs,\ res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} ys, \\ & & \ res \xrightarrow{2^*} \bullet \xleftarrow{2^*} xs,\ res \xrightarrow{2^*} \bullet \xleftarrow{2^*} ys\} \end{array}$$

The meaning for `unshuffle` is depicted in Figures 4 and 5. The resulting tuple *res* of calling the function with an input list *xs*, may share the elements of this list. Moreover, the path reaching a common descendant, in the case of *res*, begins either with a 1 or a 2 (this should be understood as descending to the left or to the right element of the tuple), and then follows by the path 2^*1, by this meaning that we should take the tail of the (left or right) list a number of times, and then take the head. From *xs*'s point of view of, the common descendant can be reached by a similar path 2^*1.

The meaning for `merge` is quite precise: the resulting list *res* may share its elements with any of the input lists *xs* and *ys*, and additionally one or more tails of *res* may be shared with one or more tails of both *xs* and *ys*. This is what the path 2^* means.

When analysing `msort`'s code of Figure 3, we have the information about `unshuffle` and `merge` available. By substituting the actual arguments for the formal ones, we get the following relations:

$$\begin{array}{lcl} R_1 & = & \{p \xrightarrow{12^*1+22^*1} \bullet \xleftarrow{2^*1} xs\} \\ R_2 & = & \{res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} z_1,\ res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} z_2, \\ & & \ res \xrightarrow{2^*} \bullet \xleftarrow{2^*} z_1,\ res \xrightarrow{2^*} \bullet \xleftarrow{2^*} z_2\} \end{array}$$

The **case** and **let** expressions in `msort` introduce more relations:

$$R_3 = \{x \xrightarrow{\epsilon} \bullet \xleftarrow{1} xs,\ y_1 \xrightarrow{\epsilon} \bullet \xleftarrow{1} p,\ y_2 \xrightarrow{\epsilon} \bullet \xleftarrow{2} p\}$$

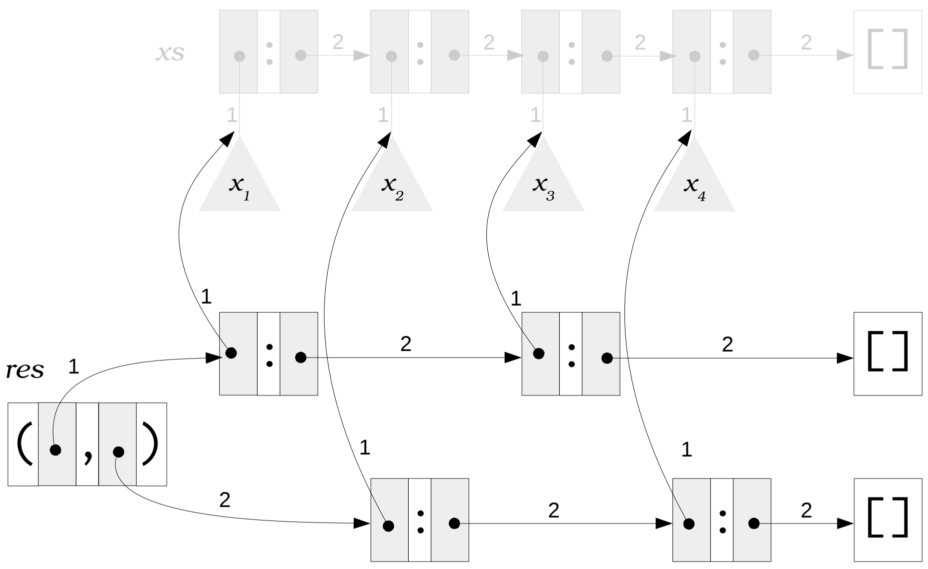


Figure 5: Heap state after executing `unshuffle` $[x_1, x_2, x_3, x_4]$

From these relations, we can derive others by reflexivity, symmetry and transitivity, such as:

$$R_4 = \{y_1 \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} xs,\ y_2 \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} xs,\ y_1 \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} y_2\}$$

In the first iteration of `msort`'s analysis, the only relation that can be inferred between its result and its argument is $xs \xrightarrow{1} \bullet \xleftarrow{1} res$. This is due to the singleton list case. The rest of the code gives us sharing information between the internal variables, but this cannot be propagated to the arguments, because in the internal recursive calls to `msort` we have nothing to start with. But, by interpreting the internal calls with the sharing information $\Sigma_1(msort) = \{xs \xrightarrow{1} \bullet \xleftarrow{1} res\}$, we get the following bigger result: $\Sigma_2(msort) = \{xs \xrightarrow{21+1} \bullet \xleftarrow{21+1} res\}$. By interpreting the code a third time by using this information when interpreting the internal calls, we get a still bigger result: $\Sigma_3(msort) = \{xs \xrightarrow{221+21+1} \bullet \xleftarrow{221+21+1} res\}$. At this point, the analysis makes the following conjecture: $\Sigma(msort) = \{xs \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} res\}$. By interpreting once more the code with this signature for *msort*, the analysis gets this same result for the *msort*'s body. So, a fixed point has been reached, and we consider this result as a correct approximation of the sharing created by `msort`.

Advantages of using paths. The rules of our safety type inference algorithm [3] demand sharing information at function applications and case expressions, so the sharing analysis not only infers sharing signatures for the functions involving only the result and the parameters of the function, but it also annotates the function definitions with information involving intermediate variables. In [2, Section 5.3] we presented 34 functions as case studies for the safety algorithm. Many of them are destructive versions of the case studies shown in Section 6. The sharing information used there was obtained from our previous work [4]. Two of them were rejected by the safety algorithm due to the imprecision of the sharing analysis results:

- A destructive version of function *msort* shown here.
- A destructive version of the function *joinAVL* shown in Section 6, used to perform AVL trees updates in constant heap space.

The same happens with a destructive version of quicksort algorithm, corresponding to the function *qsort* shown in Section 6. However, they were accepted by the algorithm if we manually removed the spurious sharing information. The shape analysis shown in this paper generates the appropriate sharing information for these three examples, shown in Figures 14 and 15, allowing them to be accepted by the safety analysis.

Now, we explain in more detail *msort* in order to show the advantages of using paths. Our prior analysis [4] of *msort* gave us the additional spurious sharing information $\{z_2 \longrightarrow \bullet \longleftarrow z_1\}$, meaning that a descendant of z_2 is shared by z_1 (the regular languages were absent in that analysis).

Having spurious relations is not incorrect, but just imprecise. Since we used this analysis to typecheck a destructive version of `msort`, using in turn a destructive version of `merge`, our type system rejected the function because of this spurious sharing information. One of the reasons of this imprecision was a worse analysis of **case** expressions:

- The analysis of the let bound variable `y1` yields the fact that it is a child of the pair `p`.
- The analysis of the let bound variable `y2` yields the fact that it is a child of the pair `p`. But, as the analysis does not distinguishes through

which paths, it is mandatory to infer that `y1` and `y2` may share some substructure in the heap.

- This information is propagated to the first recursive call of `msort` giving us that `y1` and `y2` may share a descendant with `z1`.
- Then, in the second call to `msort` we obtain that `z1` and `z2` may share a recursive descendant, which makes the safety analysis fail when the `merge` call is analysed: there is a risk that a recursive substructure is destroyed twice.

The current analysis of **case** expressions is more precise: the use of paths allows to avoid spurious relations between the variables pointing to separated substructures of a closure. So, the current analysis does not infer any relation between `y1` and `y2` , only between them and `p` through different paths, i.e. $\{p \xrightarrow{1} \bullet \xleftarrow{\epsilon} z_1, p \xrightarrow{2} \bullet \xleftarrow{\epsilon} z_2\}$. Also, the analysis of function applications is more precise because the use of paths improves the application of a sharing signature to the actual arguments.

3. The Analysis

We formally define here the analysis approximating the runtime sharing relations between the program variables. At this point, types have already been inferred, so the analysis can ask for type-related issues, such as the positions of constructor descendants, their types, and the like.

3.1. Sharing relation

In order to capture sharing, we define a binary relation between variables:

Definition 1. *Given two variables x and y, in scope in an expression, a sharing relation is a set of two pairs $\{(x, p_1), (y, p_2)\}$ specifying that x and y share a common descendant. Moreover, the regular languages denoted by p_1 and p_2 respectively define the possible pointer chains through which x and y reach their common descendant. We shall denote this sharing relation either by $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ or $y \xrightarrow{p_2} \bullet \xleftarrow{p_1} x$.*

For the sake of readability, we shall assume in the following p_1 and p_2 to be regular expressions that denote regular languages; although the actual implementation could represent the regular languages in other ways, as we

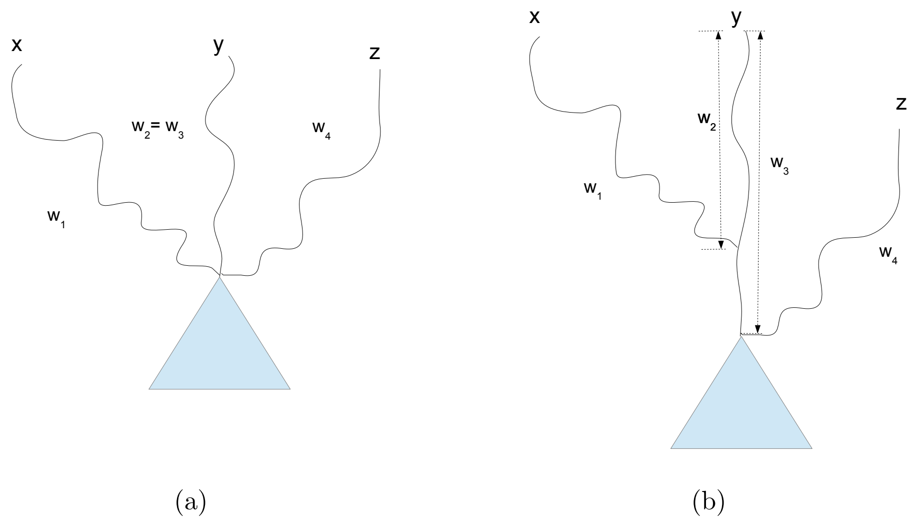


Figure 6: Particular cases of transitivity

will see in Section 5. Notice that, if $p_1 = \epsilon$, then x is a descendant of y, and symmetrically for p_2. If both $p_1 = p_2 = \epsilon$, then x and y are said to be aliases.

The regular languages have pairs i_C as alphabet symbols, where i is a natural number starting at 1, and C is a data constructor. The symbol i_C denotes a singleton pointer path in the heap passing through the i-th argument of constructor C. For instance, $x \xrightarrow{2^*_:} \bullet \xleftarrow{1_{(,)}} y$ indicates that a tail of the list x is pointed-to by the first element of the tuple y. In the examples, we shall usually omit the constructor.

The relation $\xrightarrow{p_1} \bullet \xleftarrow{p_2}$ is symmetric by definition and reflexive by writing $p_1 = p_2 = \epsilon$. But the transitivity does not hold, i.e. $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ and $y \xrightarrow{p_3} \bullet \xleftarrow{p_4} z$, with $p_2 \neq \epsilon$, does not necessarily imply $x \xrightarrow{p_1} \bullet \xleftarrow{p_4} z$. However, the transitivity holds in some cases, for example when y reaches its common descendant with x through *the same path* as it reaches its common descendant with z, as shown in Figure 6a.

More generally, we can investigate the languages denoted by p_2 and p_3, and decide whether a path in p_2 coincides with, or is a prefix of, a path in p_3 (as shown in Figure 6b), or the other way around. In these cases, there may exist a sharing path through y between x and z. Notice that both p_2 and p_3 are over-approximations to the actual runtime paths, so the risk of imprecision is still there, but if there are no such paths we are certain that

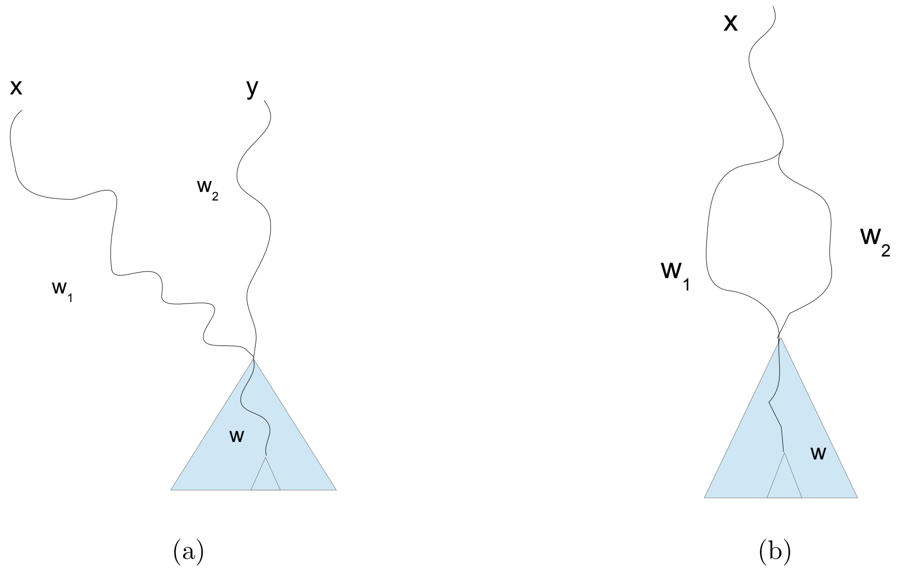


Figure 7: At least paths w_1 and w_2 must be recorded in a sharing relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ (a) or $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} x$ (b)

there will not be paths at runtime either, and we can safely omit a tuple relating x and z from the sharing relation. The rules computing the sharing derived by transitivity are explained in detail in Section 3.4.

3.2. The abstract interpretation

Based on the above considerations, we define an abstract interpretation S (meaning *sharing*) which, given an expression e and a set R containing an over-approximation to the sharing relations between the variables in scope in e, delivers another set R_{res} (*res* stands for result) containing (an over-approximation to) all the relations between the result of evaluating e, named *res*, and its variables in scope. To be precise, R and R_{res} must record at least the *minimum* information needed in order to compute all possible sharing, i.e. if we have $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ in R or R_{res}, and p_3 denotes all possible paths inside the data structure pointed-to by x and y, then we understand that $x \xrightarrow{p_1 \cdot p_3} \bullet \xleftarrow{p_2 \cdot p_3} y$ is implicitly included in the relation.

Notice that this means that:

- If two variables x and y share a substructure in the heap as in Figure 7a,

$$
\begin{array}{lcl}
S\ [\![c]\!]\ R\ \Sigma & = & R \\
S\ [\![x]\!]\ R\ \Sigma & = & R\ \uplus^*_{res}\ \{res \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x\} \\
S\ [\![C\ \overline{a}]\!]\ R\ \Sigma & = & R\ \uplus^*_{res}\ \{res \xrightarrow{j_C} \bullet \xleftarrow{\epsilon} a_j \mid\ j \in \{1..|\overline{a}|\},\ var(a_j)\} \\
S\ [\![f\ \overline{a}]\!]\ R\ \Sigma & = & R\ \uplus^*_{res}\ \Sigma(f)[\overline{a}/\overline{x}] \\
S\ [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!]\ R\ \Sigma & = & (S\ [\![e_2]\!]\ R_1\ \Sigma)\backslash\{x_1\} \\
\quad \mathbf{where}\ R_1 = (S[\![e_1]\!]\ R\ \Sigma)[x_1/res] & & \\
S\ [\![\mathbf{case}\,x\,\mathbf{of}\ \overline{alt}]\!]\ R\ \Sigma & = & \bigcup_i(S\ [\![e_i]\!]\ R_i\ \Sigma)\backslash\{x_{ij}\} \\
\quad \mathbf{where}\ alt_i = C_i\ \overline{x_i} \to e_i & & \\
\qquad\qquad R_i = R\ \overline{[\uplus^*_{x_{ij}}\{x \xrightarrow{j_{C_i}} \bullet \xleftarrow{\epsilon} x_{ij}\} \mid j \in \{1..|\overline{x_i}|\}]} & &
\end{array}
$$

Figure 8: Definition of the abstract interpretation S

there must exist a sharing relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ containing at least the paths w_1 and w_2, leading to the first point of confluence. Their extensions with a common path w need not.

- In case a variable x has internal sharing, as shown in Figure 7b, there must exist a sharing relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} x$ containing at least the paths w_1 and w_2 leading to the first point of confluence.

In order to achieve a modular analysis, it is very important to reflect the result of the analysis of a function f in a *function signature environment*, so that when the analysis finds calls to f in the body of another function g, it uses this knowledge to compute the sharing relations for g. We keep function signatures in a global environment Σ, so that $\Sigma(f)$ is a set R_{res} containing the sharing relations between the result of calling f and its arguments. The interpretation $S[\![e]\!]\ R\ \Sigma$ gives us the relations between (the normal form of) e and its variables in scope, provided Σ gives us correct approximations to the sharing relations of the functions called from e, and R is a correct approximation to the sharing relations between the variables in scope in expression e.

The rules for expressions will be explained in detail in Section 3.3. The interpretation S_d of a function definition $f\ x_1 \ldots x_n = e_f$ consists of the interpretation of its body e_f. It is straightforward to extract the signature of the function, which just describes the relations between the result of e_f and its formal arguments x_i, which are the only variables in scope in e_f. In case f is recursive, the interpretation is run several times, by starting with an empty signature for f and then computing the least fixpoint. Each iteration updates f's signature in the signature environment:

$$\begin{array}{l} S_d[\![f\ x_1 \ldots x_n = e_f]\!]\ \Sigma = \mathit{fix}\ (\lambda\Sigma\,.\,\Sigma[f \to S[\![e_f]\!]\ R_0\ \Sigma])\ \Sigma_0 \\ \quad \textbf{where}\ \ \Sigma_0 = \Sigma[f \to \emptyset] \\ \qquad\qquad\quad R_0 = \{x_i \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x_i \mid i \in \{1..n\}\} \end{array}$$

where $\Sigma[f \to R]$ either adds signature R for f or replaces it in case there was already one for it. Notice that the right hand side of the function definition is analysed starting with a initial relation R_0 in which each argument is only related to itself. This means that the signatures are computed assuming that all the parameters are disjoint and they do not present internal sharing in addition to the trivial sharing relation given by R_0. When they are not, the function caller knows the additional sharing of the actual arguments and the rule for application merges both information, as we will see in Section 3.3. In this way, the signatures only contain the information known at function definition while the whole sharing information can be computed when it is available, i.e. at function application. This makes the analysis modular. As function S is monotonic over a lattice, the least fixpoint exists and could be computed using Kleene's ascending chain if the chain were finite. We come back to this issue in Section 5.

3.3. Interpretation of expressions

The interpretation defined in Figure 8 does a top-down traversal of a function definition, accumulating these relations as soon as bound variables become free variables.

The notation $R[y/x]$ means the substitution of the variable y for the variable x in the relation R. In order to avoid name capture, y must be fresh in R. The operator $R\backslash\{x\}$ removes from R any tuple containing the variable x. The union operator $\cup$ is the usual set union. The closure operation $R_1 \uplus_x^* R_2$ takes a relation R_1 and completes it by adding R_2 and the tuples involving x that can be derived by transitivity. This operation also generates the reflexive relation $x \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x$. We explain this operator in detail in Section 3.4.

In order to express a sequence of closure operations like:

$$R \uplus_{x_1}^* R_1 \uplus_{x_2}^* R_2 \ldots \uplus_{x_n}^* R_n$$

we use the following notation:

$$R\ \overline{[\uplus^*_{x_i} R_i]}$$

If we want to add conditions C_i to define the R_i we will write $R\ \overline{[\uplus^*_{x_i} R_i \ \mid \ C_i]}$.

An important invariant of the rules presented in Figure 8 is that, in each occurrence of $S\ [\![e]\!]\ R\ \Sigma$, the set R contains an over-approximation of *all* the sharing relations that at runtime may happen between the variables in scope in e. This is the reason why the closure operator is defined in terms of a highlighted variable ($\uplus^*_x$), which in most of the rules is variable res. The set of relations is incrementally calculated: if set R already contains the sharing information for the variables in scope, we do not need to add any other spurious information between them; we just need to add the new relations between them and res which are generated by the expression being evaluated.

So, the set R_{res} returned by $S\ [\![e]\!]\ R\ \Sigma$ contains an over-approximation of *all* the sharing relations that at runtime may happen between the variables in scope in e, and the relations between those and res. It is easy to check that if the property holds for the original call $S\ [\![e_f]\!]\ R_0\ \Sigma$, where $R_0 = \{x_i \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x_i \mid i \in \{1..n\}\}$, then the rules preserve it. We recall that, according to the interpretation of a function, this means that the signature of a function contains an over-approximation of all the sharing relations that at runtime may happen between the result of the function and the parameters, assuming that the latter are disjoint. As we will see below, the function application rule deals with the additional sharing that may exist between the actual arguments.

The rule for a constant c introduces no new sharing. The rule of a variable x specifies that the result is an alias of x, and $\uplus^*_{res}$ propagates to the result the variables to which x is related.

When a constructor application $C\ \overline{a}$ is returned as a result, parent-child sharing relations are created with the constructor's children. These are added to the current set R, and then the closure computes all the derived sharing.

As an example, consider the following functions:

```
data BST a  = Empty | Node (BST a) a (BST a)
doubleE x = Node (Empty) x (Node (Empty) x (Empty))
doubleT x t = Node t x t
```

The signature[1] of `doubleE` is:

$$\{res \xrightarrow{2} \bullet \xleftarrow{32} res, res \xrightarrow{2} \bullet \xleftarrow{\epsilon} x, res \xrightarrow{32} \bullet \xleftarrow{\epsilon} x\}$$

saying that the argument element `x` is the root of the resulting tree and also the root of its right child, and consequently there is internal sharing in the result of the function. This signature can be transformed as follows (see Section 6):

$$\{res \xrightarrow{2} \bullet \xleftarrow{32} res, res \xrightarrow{2+32} \bullet \xleftarrow{\epsilon} x\}$$

In the following examples we shall apply directly this transformation whenever we have two sharing relations involving the same pair of variables. The signature of `doubleT` is:

$$\{res \xrightarrow{1} \bullet \xleftarrow{3} res, res \xrightarrow{2} \bullet \xleftarrow{\epsilon} x, res \xrightarrow{1+3} \bullet \xleftarrow{\epsilon} t\}$$

saying that the argument element `x` is in the root of the tree; and the argument tree `t` is shared between its left and right children, and consequently there is also internal sharing.

When a function application $f\ \overline{a}$ is returned as a result, first we get from f's signature the sharing relations between f's result and its formal arguments. These are copied by replacing the formal arguments by the actual ones, and then added to the current set. As before, the closure computation does the rest.

As an example, consider the following function:

```
swap l x r = Node r x l
```

whose signature is

$$\{res \xrightarrow{3} \bullet \xleftarrow{\epsilon} l, res \xrightarrow{2} \bullet \xleftarrow{\epsilon} x, res \xrightarrow{1} \bullet \xleftarrow{\epsilon} r\}$$

An application `swap t z t` first replaces in the signature the actual arguments:

$$\{res \xrightarrow{3} \bullet \xleftarrow{\epsilon} t, res \xrightarrow{2} \bullet \xleftarrow{\epsilon} z, res \xrightarrow{1} \bullet \xleftarrow{\epsilon} t\}$$

and then the closure operator additionally generates the relation $res \xrightarrow{3} \bullet \xleftarrow{1} res$ by combining $res \xrightarrow{3} \bullet \xleftarrow{\epsilon} t$ and $res \xrightarrow{1} \bullet \xleftarrow{\epsilon} t$ by transitivity.

[1]In most of the examples we will omit relation $res \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} res$.

The **let** rule is almost self-explanatory: first e_1 is analysed and the sharing computed for e_1's result is assigned to the new variable in scope x_1. Using this enriched set R_1 as assumption, the main expression e_2 is analysed, and its result is the result of the whole **let** expression. After that, the tuples involving x_1 are removed, since this variable is not in scope in the **let** expression.

The following two functions illustrate let expressions:

```
doubleDoubleSh x t = let aux = doubleT x t in Node aux x aux
doubleDouble x t = Node (doubleT x t) x (doubleT x t)
```

Both functions call `doubleT`, but in the first case the result of such application is shared in the result, while in the second case it is not. In fact, the second function is translated into two lets with different variables. The signature of `doubleDoubleSh` is:

$$\{\mathit{res} \xrightarrow{1+11+13+31} \bullet \xleftarrow{3+13+31+33} \mathit{res}, \mathit{res} \xrightarrow{2+12} \bullet \xleftarrow{2+32} \mathit{res},$$
$$\mathit{res} \xrightarrow{2+12+32} \bullet \xleftarrow{\epsilon} x, \mathit{res} \xrightarrow{1(1+3)+3(1+3)} \bullet \xleftarrow{\epsilon} t\}$$

which shows that `t` is shared at the second level of the resulting tree, and also that the left and right children of the resulting tree are shared (through variable `aux`, though this does not appear in the signature). The signature of `doubleDouble` is almost the same, but the sharing relation $\mathit{res} \xrightarrow{1} \bullet \xleftarrow{3} \mathit{res}$ is not contained.

Finally, a **case** expression introduces the pattern variables x_{ij} in the scope of a branch e_i. Their sharing relations are derived from the parent x's ones by first adding the child-parent relation between each x_{ij} and x, and then computing the closure. After analysing the branches, the least upper bound of all the analyses must be computed, expressing the fact that at compile time it is not known which branch will be taken at runtime. Finally, the tuples involving the patterns x_{ij} are removed, since these variables are not in scope in the **case**.

As an example, consider the following function:

```
fstSnd (x:[]) = x:[]
fstSnd (x:(y:xs)) = y:[]
```

From the first alternative we obtain $\{\mathit{res} \xrightarrow{1} \bullet \xleftarrow{1} \mathit{arg}\}$ and from the second one $\{\mathit{res} \xrightarrow{1} \bullet \xleftarrow{21} \mathit{arg}\}$, so the resulting signature is $\{\mathit{res} \xrightarrow{1} \bullet \xleftarrow{1+21} \mathit{arg}\}$.

$$
\begin{aligned}
&R \uplus_x \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y\} \stackrel{\text{def}}{=} \\
&\quad R \cup \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y\} \cup \{x \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x\} \\
&\qquad \cup \{x \xrightarrow{p_1 \cdot p_3|_{p_2}} \bullet \xleftarrow{p_4} z \mid y \xrightarrow{p_3} \bullet \xleftarrow{p_4} z \in R\} \\
&\qquad \cup \{x \xrightarrow{p_1} \bullet \xleftarrow{p_4 \cdot p_2|_{p_3}} z \mid y \xrightarrow{p_3} \bullet \xleftarrow{p_4} z \in R\} \\
&\qquad \cup \{x \xrightarrow{p_1 \cdot p_3|_{p_2}} \bullet \xleftarrow{p_1 \cdot p_4|_{p_2}} x \mid y \xrightarrow{p_3} \bullet \xleftarrow{p_4} y \in R\}
\end{aligned}
$$

$$
\begin{aligned}
&R \uplus_x^* R' \stackrel{\text{def}}{=} \\
&\quad \overline{R\ [\uplus_x \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y\} \mid x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y \in R', y \neq x]} \\
&\qquad \cup \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} x \mid x \xrightarrow{p_1} \bullet \xleftarrow{p_2} x \in R'\}
\end{aligned}
$$

Figure 9: Definition of the closure operation

3.4. The closure of a relation

As we explained in the previous section the closure of a relation is applied incrementally along the top-down traversal of a function body. It is only necessary to apply transitivity with respect to variable *res* or the case bound variables.

The closure operation $\uplus_x^*$ is defined in terms of the simpler one $\uplus_x$, which completes a relation set R with a new relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$, where $y \neq x$, by adding the relations that bind x to the variables contained in R, and are derived by transitivity. Both operators are defined in Figure 9.

The inclusion of R and the relations $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$, $x \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x$ are self-explanatory. We shall concentrate on the remaining relations shown in the last lines of the definition.

The second line corresponds to the case illustrated in Figure 6b, while the third one corresponds to the symmetric case. These relations involve the derivative operator $\_|\_$ whose meaning is:

$$p_1|_{p_2} = \{w_3 \mid \exists w_2 \in L(p_2). w_2 w_3 \in L(p_1)\}.$$

If p_1 and p_2 denote regular languages so does $p_1|_{p_2}$, and in Section 5 we explain how to compute it[2]. In the second line of the definition of $\uplus_x$ the language describing $p_1 \cdot p_3|_{p_2}$ might be empty. In this case we can discard the

[2]When $p_2 = \{a\}$, the language $p_1|_a$ is sometimes called the *derivative* of $L(p_1)$ with respect to a, and it is denoted $a\backslash L$, being $L = L(p_1)$.

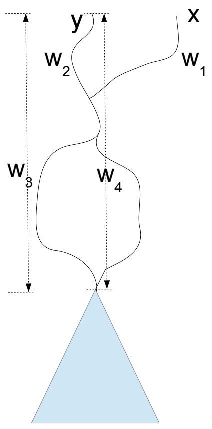


Figure 10: Transitivity with internal sharing.

corresponding sharing relation from the result of the closure operation. If it is not empty then there exists a word $w_2 \in L(p_2)$ such that it its a prefix of another word $w_3 \in L(p_3)$, so we can start from x, follow a path $w_1 \in L(p_1)$, and then follow the path w_3 without the prefix w_2 (which results in a path of $L(p_3|_{p_2})$) in order to reach the common descendant of x and z. The third line of $\uplus_x$ is applicable when a path of p_3 is a prefix of a path of p_2, and works similarly.

The fourth line deals with the case in which variable x gets internal sharing through variable y, as shown in Figure 10. This happens when the path w_2 through which y reaches its common descendant with x is a prefix of both paths w_3 and w_4 representing the internal sharing of y. Then $p_3|_{p_2}$ and $p_4|_{p_2}$ are not empty, and contain respectively the paths w_3 and w_4 without the prefix w_2, which prepended with $w_1 \in L(p_1)$ represent two paths of internal sharing from variable x.

The closure operator $\uplus^*_x$ is defined as a sequence of applications of the simpler one $\uplus_x$, plus a simple set union which takes care of the variables with internal sharing.

In spite of the restrictions of the $\uplus$ operator, we could replace the $\uplus^*$ operator by a sequence of $\uplus$ operations in all the rules of Figure 8 but in function application, because only in the application non-trivial reflexive relations may be added.

In fact, operation $R \uplus_x^* R'$ is used to define the confluence of information happening in a function call. R represents the context of the call, while R' represents the sharing generated by the function between the result and the arguments.

Its definition is divided into two parts:

1. First, we take each relation in R' of the form $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ such that $y \neq x$ and apply the previous transitivity operator incrementally. This is well defined because operator $\uplus_x$ is in a sense *commutative*, as we will prove in Section 4. So the order in which we add the relations of R' is not relevant: the final result may be different but *equivalent*, in the sense that it records the same information.
   This part reflects the interaction of the context with the function definition.
2. Second, we just add those reflexive relations $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} x \in R'$. In the abstract interpretation, this kind of relations only appear in the application of a function: it may happen that the result of a function f has internal sharing, so a relation $\mathit{res} \xrightarrow{p_1} \bullet \xleftarrow{p_2} \mathit{res} \in \Sigma(f)$. It is not necessary to apply transitivity here because the internal sharing of *res* either comes from the function itself (i.e. is reflected in R') or through a real argument which already has internal sharing (i.e. is reflected in R). As we will prove in Section 4, the transitivity closure would only add redundant information.

3.5. Type correctness

It is important to see whether the relations inferred by the analysis are well-typed. For instance, we could have a relation $x \xrightarrow{p_x} \bullet \xleftarrow{p_y} y$ in which the descendant reached from x and p_1 had a type t, while the descendant reached from y and p_y had a different type t'. This would obviously be a spurious relation since in well-typed programs, an ill-typed sharing may not occur at runtime.

The expression $\mathit{type}(t, p)$ returns the set of types computed starting at the type t, and then descending through the constructors of the words in p according to its type and to the child chosen at each step. In our language this set can be statically computed. Let t_x be the type computed by the compiler for the variable x. For instance, if $t_x = [\mathit{Int}]$, then $\mathit{type}(t_x, 1_:) = \{\mathit{Int}\}$. The set may be empty as in $\mathit{type}(t_x, 1_:1_:)$, because an integer has no constructors. Or it may yield more than one type, as in $\mathit{type}(t_x, 1_: + 2_:) = \{\mathit{Int}, [\mathit{Int}]\}$.

Definition 2. *We say that the relation* $x \xrightarrow{p_x} \bullet \xleftarrow{p_y} y$ *is well-typed if both* $type(t_x, p_x)$*, and* $type(t_y, p_y)$ *are singleton and* $type(t_x, p_x) = type(t_y, p_y)$*.*

Lemma 3. *If the relations in* R *and* Σ *are well-typed, then for every expression* e*, the relations in* $S[\![e]\!]\ R\ \Sigma$ *are well-typed.*

Proof. By induction on the rules used to compute $R' = S[\![e]\!]\ R\ \Sigma$, we will prove a stronger property, namely that if $x \xrightarrow{p_x} \bullet \xleftarrow{p_y} y \in R'$, then for all $w_x \in L(p_x), w_y \in L(p_y)$ the relation $x \xrightarrow{w_x} \bullet \xleftarrow{w_y} y$ is well-typed, and $type(t_x, w_x) = type(t_x, p_x)$, and $type(t_y, w_y) = type(t_y, p_y)$.

By inspection of the rules of Figure 8, it is easy to check that every relation explicitly introduced there is well-typed, and that the regular languages are just basic ones such as j_C or ϵ. So, they satisfy the desired property. Also, substituting a variable for another one with the same type, preserves the property. We concentrate then on the closure operator of Figure 9.

Let us assume in line 3 that for every $w_1 \in L(p_1), w_2 \in L(p_2), w_3 \in L(p_3), w_4 \in L(p_4)$, the relations $x \xrightarrow{w_1} \bullet \xleftarrow{w_2} y$ and $y \xrightarrow{w_3} \bullet \xleftarrow{w_4} z$ are well-typed. Let $w_5 \in L(p_3|_{p_2})$. Let us assume this language to be non-empty, since otherwise the Lemma trivially holds. Let $\{t_1\} = type(t_x, p_1) = type(t_y, p_2)$ and $\{t_2\} = type(t_y, p_3) = type(t_z, p_4)$.

Given w_5, there must exist words $w_2 \in L(p_2), w_3 \in L(p_3)$ such that $w_3 = w_2 \cdot w_5$. Let $w_1 \in L(p_1)$. We have then the following equalities:

$$\begin{array}{l} type(t_x, p_1 \cdot p_3|_{p_2}) = type(t_x, w_1 \cdot w_5) = type(t_1, w_5) = \\ type(type(t_y, w_2), w_5) = type(t_y, w_3) = type(t_y, p_3) = type(t_z, p_4) \end{array}$$

Then, for all $w_x \in L(p_1 \cdot p_3|_{p_2}), w_y \in L(p_4)$, the relation $x \xrightarrow{w_x} \bullet \xleftarrow{w_y} z$ is well-typed. The reasoning is very similar for the lines 4 and 5 of Figure 9. □

4. Correctness

In this section we provide the main results needed to prove the analysis is well-defined and correct. Full proofs can be found in [6].

First, in Section 4.1, we provide for some notation about paths in a heap and prove closure preservation in *Safe*'s operational semantics, i.e. the immutability of heap data structures along execution.

In Section 4.2 we give some auxiliary lemmas about the abstract interpretation, like monotonicity and commutativity of the closure operator. We also prove some properties about substitutions over relations, which occur in function applications.

Then, in Section 4.3 we define the notion of correct approximation to the sharing in a real heap. A function signature is considered correct when it allows to compute the sharing of any application of that function by combining it with the actual arguments' sharing relations.

Finally, in Section 4.4 we prove the correctness of the abstract interpretation, i.e. that the interpretation of expressions is correct and that interpretation of functions give correct signatures.

4.1. Heap properties

In what follows, we will denote by $\mathcal{V}$ the alphabet of our regular languages. Its symbols are pairs (i, C), written i_C, where C is a data constructor and i is a natural number starting at 1, denoting the C's argument taken by a heap path when arriving at a closure having constructor C applied to its arguments.

In order to prove the correctness of the analysis, we will need the precise meaning of executing a *Core-Safe* program. The *Core-Safe* operational semantics can be found at [1]. It is a standard big-step operational eager semantics: judgement $E \vdash h, e \Downarrow h', v$ means that expression e, starting with runtime environment E and initial heap h, evaluates to value v and the heap changes to h'. Environment E maps variables in scope in e to values, heaps map pointers to closures of the form $(C\ \overline{v})$, where each v_j ($j \in \{1..|\overline{v}|\}$) is a value, and a value is either a basic constant or a heap pointer.

Actually, we prove the correctness of our analysis with respect to a slightly modified semantics in which destructive pattern matching is replaced by a standard pattern matching, and then trivially the result of the analysis over-approximates the sharing when the full semantics is considered.

Definition 4. *Let h be a heap, $p, q \in \operatorname{dom} h$, and $v = i_C \in \mathcal{V}$. We say that q is an immediate v-successor of p (written $p \overset{v}{\dashrightarrow}_h q$) iff $h(p) = (C\ \overline{v})$, for some C and $\overline{v}$, and $q = v_i$. Analogously, assume a word w in $\mathcal{V}^*$. A pointer $q \in \operatorname{dom} h$ is a w-successor of p (written $p \overset{w}{\Longrightarrow}_h q$) if there exists a sequence of pointers $p_0, \ldots, p_n$ ($n \geq 0$) and a sequence of positions $v_1, \ldots, v_n \in \mathcal{V}$ such that $w = v_1 \cdots v_n$ and:*

$$p = p_0 \overset{v_1}{\dashrightarrow}_h p_1 \overset{v_2}{\dashrightarrow}_h \ldots \overset{v_n}{\dashrightarrow} p_n = q$$

We are mostly interested in the fact that two given variables are pointing to a common pointer p, rather than in the p itself. That is why we shall use the notation

$$p_1 \overset{w_1}{\Longrightarrow} \bullet \overset{w_2}{\Longleftarrow} p_2 \text{ (in } h\text{)}$$

to denote the existence of a pointer p such that $p_1 \overset{w_1}{\Longrightarrow}_h p$ and $p_2 \overset{w_2}{\Longrightarrow}_h p$.

If in a heap h there exists an actual sharing between two variables x and y through respective pointer paths w_1 and w_2, we say that there exists a *sharing condition* in h and denote it by $E(x) \overset{w_1}{\Longrightarrow} \bullet \overset{w_2}{\Longleftarrow} E(y)$ (in h).

The following lemma asserts that live heap data structures do not change along execution.

Lemma 5 (Closure preservation)**.** *Let us assume an execution $E \vdash h, e \Downarrow h', v$. For every pointer $p \in \text{dom } h'$, variable $x \in \text{dom}\, E$, and $w \in \mathcal{V}^*$:*

$$E(x) \overset{w}{\Longrightarrow}_h p \textit{ if and only if } E(x) \overset{w}{\Longrightarrow}_{h'} p$$

As a consequence of this, $E(x) \overset{w}{\Longrightarrow}_{h'} p$ implies $p \in \text{dom}\, h$.

Proof. By induction on the $\Downarrow$-derivation, and by cases on the expression e when the last rule is applied. □

4.2. Properties of the abstract interpretation

First, we prove that the closure operator is well-defined. For this we need two auxiliary lemmas. We start with some properties of the derivative of regular languages.

Lemma 6. *Let p_1, p_2 and p_3 be path expressions. Then:*

1. $L((p_1 \cdot p_2)|_{p_3}) = L(p_1|_{p_3} \cdot p_2) \cup L(p_2|_{(p_3|_{p_1})})$.
2. $L(p_1|_{p_2 \cdot p_3}) = L((p_1|_{p_2})|_{p_3})$.

Proof. By set inclusion in both directions. □

And then we prove monotonicity of the closure operation.

Lemma 7 (Monotonicity of the closure operation)**.** *Let R and R' be two sets of relations, and $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ a sharing relation such that $y \neq x$. If $R \subseteq R'$ then:*

$$R \uplus_x \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y\} \subseteq R' \uplus_x \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y\}$$

An immediate consequence is that $R \uplus_x^* R'' \subseteq R' \uplus_x^* R''$.

Proof. It follows trivially from the definition of $\uplus_x$. □

We prove now that the operator $\uplus_x^*$ is well defined. As we said in Section 3, the order in which we apply transitivity to the rules belonging to R' may lead to different but equivalent results. First, we give the notion that two sets of relations contain the same information in terms of the sharing paths. Then, we prove well-definedness of the operator.

Definition 8. *A set of sharing relations R is included in R' (written $R \sqsubseteq R'$) if for every sharing relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y \in R$ and every pair of words $w_1 \in L(p_1)$, $w_2 \in L(p_2)$ there exists a sharing relation $x \xrightarrow{p'_1} \bullet \xleftarrow{p'_2} y \in R'$ such that $w_1 \in L(p'_1)$ and $w_2 \in L(p'_2)$. Two sets of relations R and R' are said to be* equivalent *(written $R \equiv R'$) if both $R \sqsubseteq R'$ and $R' \sqsubseteq R$ hold.*

Lemma 9 (Commutativity of closure operation). *Let R be a set of sharing relations and $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$, $x' \xrightarrow{p'_1} \bullet \xleftarrow{p'_2} y'$ a pair of sharing relations such that $y \neq x$ and $y' \neq x'$. Let us define $R_{x,x'}$ and $R_{x',x}$ as follows:*

$$\begin{array}{lcl} R_{x,x'} & \stackrel{def}{=} & (R \uplus_x \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y\}) \uplus_{x'} \{x' \xrightarrow{p'_1} \bullet \xleftarrow{p'_2} y'\} \\ R_{x',x} & \stackrel{def}{=} & (R \uplus_{x'} \{x' \xrightarrow{p'_1} \bullet \xleftarrow{p'_2} y'\}) \uplus_x \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y\} \end{array}$$

If $y' \neq x$ and $x' \neq y$ then $R_{x,x'} \equiv R_{x',x}$.

Proof. By case distinction and using Lemmas 6 and 7. □

Consequently, following different orders in adding the relations of R' lead to equivalent sets of relations.

The following notion has to do with the parameter passing mechanism. In a function application $f\ \overline{a}$, the actual arguments $\overline{a}$ replace the formal ones $\overline{x}$, and this replacement may be no-injective, since two or more formal ones may be replaced by the same actual one. We call this replacement a *generalised substitution.*

Definition 10. *A* generalised substitution *θ is a set of pairs of variables, where the pair (x, y) specifies that x is going to be replaced by y. The* domain *of θ (denoted* dom θ*) is the set of variables x such that $(x, z) \in \theta$ for some z. The range of θ (denoted* ran θ*) is the set of variables z such that $(x, z) \in \theta$ for some x. The notation $[z/x] \in \theta$ is defined as follows:*

$$[z/x] \in \theta \Leftrightarrow_{def} (x,z) \in \theta \vee (x \notin \text{dom } \theta \wedge x = z)$$

If R is a set of sharing relations and θ is a generalised substitution, the set $R\theta$ is defined as follows:

$$R\theta = \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y \mid x' \xrightarrow{p_1} \bullet \xleftarrow{p_2} y' \in R, [x/x'] \in \theta, [y/y'] \in \theta\} \qquad (1)$$

A generalised substitution θ is said to be injective *whenever $[x/z_1], [x/z_2] \in \theta$ implies $z_1 = z_2$. The inverse of a generalised substitution θ^{-1} is defined by $\theta^{-1} = \{(x,y) \mid (y,x) \in \theta\}$.*

It can be easily shown that $[x/y] \in \theta$ iff $[y/x] \in \theta^{-1}$. By abuse of notation we denote $[\overline{y}/\overline{x}]$ the substitution $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. It is easy to see that the notation $R[\overline{a}/\overline{x}]$ used above is a particular case of (1).

As an example, assume a function definition $f\ a_1\ a_2\ a_3\ a_4 = \ldots$ and that we want to analyse the function application $f\ x\ x\ y\ y$. The corresponding substitution is $\theta = [a_1/x, a_2/x, a_3/y, a_4/y]$. If we define R as follows:

$$R = \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y, y \xrightarrow{p_3} \bullet \xleftarrow{p_4} z\}$$

We obtain:

$$\begin{aligned} R\theta = \{ \quad & a_1 \xrightarrow{p_1} \bullet \xleftarrow{p_2} a_3, a_2 \xrightarrow{p_1} \bullet \xleftarrow{p_2} a_3, a_1 \xrightarrow{p_1} \bullet \xleftarrow{p_2} a_4, a_2 \xrightarrow{p_1} \bullet \xleftarrow{p_2} a_4, \\ & a_3 \xrightarrow{p_3} \bullet \xleftarrow{p_4} z, a_4 \xrightarrow{p_3} \bullet \xleftarrow{p_4} z\} \end{aligned}$$

Lemma 11 (Properties of substitution)**.** *Let R, R_1, R_2 be sets of relations and θ a generalised substitution. Then:*

1. $(R_1 \cup R_2)\theta = R_1\theta \cup R_2\theta$.
2. *If $x \notin \text{dom}(\theta) \cup \text{ran}(\theta)$, then $(R \uplus_x^* R')\theta = R\theta \uplus_x^* R'\theta$.*
3. $R\theta\theta^{-1} \supseteq R$.
4. *If θ is injective, then $R\theta\theta^{-1} = R$.*

Proof. (1) is proved by equational reasoning. (2) and (4) are proved by set inclusion in both directions and by case distinction in each one. (3) is straightforward. □

Another auxiliary property we will use later says that the abstract interpretation add new relations to the initial set of relations.

Lemma 12 (Conservative abstract interpretation). *Let e be an expression, Σ a signature environment and R a set of sharing relations. Then $R \subseteq S\ [\![e]\!]\ R\ \Sigma$.*

Proof. By structural induction on e. All cases are straightforward. $\square$

4.3. Notion of correct approximation

Now we define when a set of relations correctly approximates the real sharing in a heap and the notion of correct signature. The first definition reflects the fact that at least the minimum sharing must be recorded in the relations, i.e. the paths leading to the first point of confluence must be recorded, while their extensions with a common path need not. Notice that this means that in case of internal sharing, each point of internal confluence must also be recorded.

A correct function signature must record enough sharing information to be able to approximate each possible call to that function, i.e. each possible execution of the body.

Definition 13. *A sharing relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ is said to* approximate *a sharing condition $E(x) \overset{w_1}{\Longrightarrow} \bullet \overset{w_2}{\Longleftarrow} E(y)$* (in h) *iff there exists a word w such that $w_1 \in L(p_1 w)$ and $w_2 \in L(p_2 w)$.*

Definition 14. *Let R be a set of sharing relations, E a runtime environment, and h a heap. We say that R is a correct approximation of E and h, denoted $R \succeq (E, h)$, iff for every pair of variables $x, y \in \operatorname{dom} E$, and pair of words $w_1, w_2 \in \mathcal{V}^*$ if the condition $E(x) \overset{w_1}{\Longrightarrow} \bullet \overset{w_2}{\Longleftarrow} E(y)$* (in h) *holds, it is approximated by a sharing relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ in R.*

Lemma 15 (Properties of correct approximations). *For any R, R', E, h, x, z, v of their respective types:*

1. *If $R \succeq (E, h)$ and $R \equiv R'$, then $R' \succeq (E, h)$.*
2. *If $R \succeq (E, h)$ and $R \sqsubseteq R'$ then $R' \succeq (E, h)$.*
3. *If $R \succeq (E \uplus [\overline{x_i \mapsto v}^n], h)$ then $R[\overline{z_i/x_i}^n] \succeq (E \uplus [\overline{z_i \mapsto v}^n], h)$.*
4. *If $R \succeq (E \uplus [x \mapsto v], h)$ then $R \backslash \{x\} \succeq (E, h)$.*

Proof. All the properties trivially follow from the definition of $\succeq$. $\square$

Definition 16 (Correct signature). *A set R of relations is a correct signature for a function definition $f\ \overline{x} = e_f$ iff for each execution $E_f \vdash h, e_f \Downarrow h', v$ of the body of the function and every set of relations R' such that $R' \succeq (E_f, h)$ it holds that $R' \uplus^*_{res} R \succeq (E_f \uplus [res \mapsto v], h')$. A signature environment Σ is said to be correct iff every signature it contains is correct.*

4.4. Correctness

Correctness of the analysis is divided into two steps. First, we prove that given correct signatures of the functions which are called from an expression, the interpretation of the expression is correct. Then, we prove that the interpretation of a function generates a correct signature. For both theorems we need to prove that the transitive closure operator is correct, which we show in the following two lemmas. The first one deals with the normal case in which variables do not present internal sharing, i.e. with the first three lines of operator $\uplus_x$ definition. The second lemma concerns the case in which a variable gets internal sharing through another variable having internal sharing, i.e. with the fourth line of operator $\uplus_x$ definition.

Lemma 17 (Transitive closure lemma)**.** *Let us assume a runtime environment E, a heap h, a set of sharing relations R, some variables x, y, z, (with $y \neq z$) words w_x, w_y, w_z, and paths p_{xy}, p_{yx}, p_{yz}, p_{zy} such that the following holds:*

$$E(x) \overset{w_x}{\Longrightarrow} \bullet \overset{w_y}{\Longleftarrow} E(y) \text{ (in } h\text{)}, \textit{approximated by } x \xrightarrow{p_{xy}} \bullet \xleftarrow{p_{yx}} y \in R$$
$$E(z) \overset{w_z}{\Longrightarrow} \bullet \overset{w_y}{\Longleftarrow} E(y) \text{ (in } h\text{)}, \textit{approximated by } z \xrightarrow{p_{zy}} \bullet \xleftarrow{p_{yz}} y$$

Then there exists a sharing relation $x \xrightarrow{p_{xz}} \bullet \xleftarrow{p_{zx}} z \in R \uplus_z \{y \xrightarrow{p_{yz}} \bullet \xleftarrow{p_{zy}} z\}$ approximating $E(x) \overset{w_x}{\Longrightarrow} \bullet \overset{w_z}{\Longleftarrow} E(z)$ (in h).

Proof. It is easy to show that there exist words $w_{yx} \in L(p_{yx})$, $w_{yz} \in L(p_{yz})$, w, and w' such that $w_y = w_{yx}w = w_{yz}w'$, so either w_{yx} is a prefix of w_{yz}, or w_{yz} is a prefix of w_{yx}. The proof distinguishes these two cases, and essentially applies the definition of the $\uplus_z$ operator, and of the derivation of a regular language w.r.t. another one. □

Lemma 18 (Transitive self-closure lemma)**.** *Let us assume a runtime environment E, a heap h, a set of sharing relations R, some variables x, y (with $x \neq y$), words w_x, w_y, w_1, w_2 and paths p_{x1}, p_{x2}, p_{xy}, p_{yx} such that the following holds:*

$$E(x) \overset{w_x w_1}{\Longrightarrow} \bullet \overset{w_x w_2}{\Longleftarrow} E(x) \text{ (in } h\text{)}, \textit{approx. by } x \xrightarrow{p_{x1}} \bullet \xleftarrow{p_{x2}} x \in R$$
$$E(x) \overset{w_x}{\Longrightarrow} \bullet \overset{w_y}{\Longleftarrow} E(y) \text{ (in } h\text{)}, \textit{approximated by } x \xrightarrow{p_{xy}} \bullet \xleftarrow{p_{yx}} y$$

Then there exists a sharing relation $y \xrightarrow{p_{y1}} \bullet \xleftarrow{p_{y2}} y \in R \uplus_y \{x \xrightarrow{p_{xy}} \bullet \xleftarrow{p_{yx}} y\}$ approximating $E(y) \overset{w_y w_1}{\Longrightarrow} \bullet \overset{w_y w_2}{\Longleftarrow} E(y)$ (in h).

Proof. We show that there exist words w, w', $w_{yx} \in L(p_{yx})$, $w_{xy} \in L(p_{xy})$, $w_{x1} \in L(p_{x1})$, and $w_{x2} \in L(p_{x2})$, such that $w_y = w_{yx}w'$, $w_x w_1 = w_{x1}w$, $w_x w_2 = w_{x2}w$ and $w_x = w_{xy}w'$. So $w_{xy}w'w_1 = w_{x1}w$ and $w_{xy}w'w_2 = w_{x2}w$. The proof proceeds by distinguishing four cases: (1) w_{xy} is a prefix of both w_{x1} and w_{x2}; (2) w_{x1} and w_{x2} are prefix of w_{xy}; (3) w_{x1} is prefix of w_{xy} and w_{xy} is prefix of w_{x2}; and (4) w_{x2} is prefix of w_{xy} and w_{xy} is prefix of w_{x1}. $\square$

So far we have used the notation $R[\overline{z}/\overline{x}]$ to denote the substitution of the x_i variables appearing in each side of R by their corresponding z_i. In this context, the $[\overline{z}/\overline{x}]$ will be interpreted as a generalised substitution θ.

The following theorem establishes the correctness of the abstract interpretation modulo the correctness of function signatures.

Theorem 19. *Assume an expression e, a set of sharing relations R and a correct signature environment Σ. If $S\ [\![e]\!]\ R\ \Sigma = R'$, then for every execution $E \vdash h, e \Downarrow h', v$ in which $R \succeq (E, h)$, it holds that $R' \succeq (E \uplus [res \mapsto v], h')$.*

Proof. Let us denote the environment $E \uplus [res \mapsto v]$ by E'. We have to prove that for every $x, y \in \mathrm{dom}\, E'$ the runtime sharing condition $E'(x) \overset{w_1}{\Longrightarrow} \bullet \overset{w_2}{\Longleftarrow} E'(y)$ (in h') implies the existence of a sharing relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y \in R'$ and a word w such that $w_1 \in L(p_1 w)$ and $w_2 \in L(p_2 w)$. If x and y are distinct from *res*, then by Lemma 5 we know that $E(x) \overset{w_1}{\Longrightarrow} \bullet \overset{w_2}{\Longleftarrow} E(y)$ (in h), and since $R \succeq (E, h)$, there exists a relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y \in R$ and a word w satisfying the same conditions, but since R is a subset of R' (by Lemma 12), the Theorem holds when x and y are distinct from *res*, so henceforth we shall assume that at least one x and y is the *res* variable. We shall assume without loss of generality that $y = res$. We proceed by induction on the size of the $\Downarrow$-derivation. We distinguish cases on the structure of e. We show only two of them:

- **Case** $e \equiv f\ \overline{a}$

  Let us assume that $f\ \overline{y} = e_f$ is the function definition of f, and assume the following execution:

$$\frac{E_f \vdash h, e_f \Downarrow h', v}{E \vdash h, f\ \overline{a} \Downarrow h', v} \qquad \text{where } E_f = [\overline{y} \mapsto E(\overline{a})]$$

  i.e. $E_f(y_i) = E(a_i)$ for each $i \in \{1..|\overline{y}|\}$.

Let E_0 be a runtime environment such that $E = E_0 \uplus [\overline{a} \mapsto E(\overline{a})]$. By assumption the following relation holds:

$$R \succeq (E_0 \uplus [\overline{a} \mapsto E(\overline{a})], h)$$

By Lemma 15 we can replace each a_i by its y_i so as to get:

$$R[\overline{y}/\overline{a}] \succeq (E_0 \uplus [\overline{y} \mapsto E(\overline{a})], h)$$

Notice that $[\overline{y}/\overline{a}]$ is not a standard substitution, but a generalised one. Moreover, since all the y_i are distinct, this substitution is injective. We can leave out the E_0 from this approximation relation so as to get:

$$R[\overline{y}/\overline{a}] \succeq ([\overline{y} \mapsto E(\overline{a})], h)$$

which follows trivially from the previous one. From the definition of correct signature, we obtain:

$$R[\overline{y}/\overline{a}] \uplus^*_{res} \Sigma(f) \succeq ([\overline{y} \mapsto E(\overline{a})] \uplus [res \mapsto v], h')$$

Now we substitute the a_i for the y_i in the environment of the right-hand side by using Lemma 15:

$$(R[\overline{y}/\overline{a}] \uplus^*_{res} \Sigma(f))[\overline{a}/y] \succeq ([\overline{a} \mapsto E(\overline{a})] \uplus [res \mapsto v], h') \tag{2}$$

and we use the properties of Lemma 11 in order to transform the left-hand side:

$$\begin{array}{ll} & (R[\overline{y}/\overline{a}] \uplus^*_{res} \Sigma(f))[\overline{a}/\overline{y}] \\ = & \qquad \{\text{by Lemma 11 (2) as } res \neq \overline{y}, \overline{a}\} \\ & R[\overline{y}/\overline{a}][\overline{a}/\overline{y}] \uplus^*_{res} \Sigma(g)[\overline{a}/\overline{y}] \\ = & \qquad \{\text{by Lemma 11 (4), since } [\overline{y}/\overline{a}] \text{ is injective }\} \\ & R \uplus^*_{res} \Sigma(f)[\overline{a}/\overline{y}] \end{array}$$

Therefore we can rewrite (2) so as to get:

$$R \uplus^*_{res} \Sigma(f)[\overline{a}/\overline{y}] \succeq ([\overline{a} \mapsto E(\overline{a})] \uplus [res \mapsto v], h')$$

Notice that R is a subset of the left-hand side, and R correctly approximates E_0, so we can add E_0 to the right-hand side:

$$R \uplus^*_{res} \Sigma(f)[\overline{a}/\overline{y}] \succeq (E_0 \uplus [\overline{a} \mapsto E(\overline{a})] \uplus [res \mapsto v], h')$$

which is equivalent to $R' \succeq (E', h')$.

- **Case** $e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2$

  We get the following execution:

$$\frac{E \vdash h, e_1 \Downarrow h_1, v_1 \quad E \uplus [x_1 \mapsto v_1] \vdash h_1, e_2 \Downarrow h', v}{E \vdash h, \textbf{let } x_1 = e_1 \textbf{ in } e_2 \Downarrow h', v}$$

  Since $R \succeq (E, h)$ we can apply the induction hypothesis on the $\Downarrow$-derivation of e_1 and obtain:

$$S \llbracket e_1 \rrbracket \; R \; \Sigma \succeq (E \uplus [res \mapsto v], h_1)$$

  By Lemma 15 we can substitute x_1 for res in order to get:

$$(S \llbracket e_1 \rrbracket \; R \; \Sigma)[x_1/res] \succeq (E \uplus [x_1 \mapsto v_1], h_1)$$

  Let us denote the left-hand side by R_1. Now we can apply the induction hypothesis on the derivation of e_2,

$$S \llbracket e_2 \rrbracket \; R_1 \; \Sigma \succeq (E \uplus [x_1 \mapsto v_1] \uplus [res \mapsto v], h')$$

  and apply Lemma 15 again,

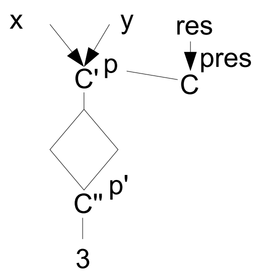
$$(S \llbracket e_2 \rrbracket \; R_1 \; \Sigma) \backslash \{x_1\} \succeq (E \uplus [res \mapsto v], h')$$

  which is what we wanted to prove.

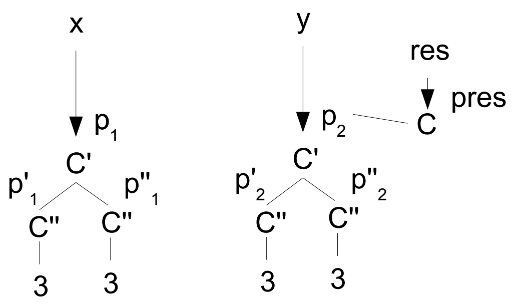
$\square$

Now we prove that the interpretation of a function returns a correct signature. A signature records the sharing between the result and the arguments of the function assuming these are disjoint and without internal sharing. However, a real call to the function may not satisfy such assumption. Given the real configuration (E, h), we define an hypothetical execution where both the environment $\widehat{E}$ and the heap $\widehat{h}$ contain the same information as (E, h) but meeting the separation property. The signature of the function captures the sharing information corresponding to this hypothetical execution.

Definition 20. *Let (E, h) and $(\widehat{E}, \widehat{h})$ be two configurations such that dom $E =$ dom $\widehat{E}$. A mapping γ : dom $\widehat{h} \to$ dom h is said to be an entanglement from $(\widehat{E}, \widehat{h})$ to (E, h), iff:*



(a)



(b)

Figure 11: Final heaps in the real execution (a), and the untangled one (b).

1. *For every pointer $\widehat{p} \in dom\ \widehat{h}$, if $\widehat{h}(\widehat{p}) = C\ \widehat{v}_1 \cdots \widehat{v}_n$, then $h(\gamma(\widehat{p})) = C\ \gamma(\widehat{v}_1) \cdots \gamma(\widehat{v}_n)$.*
2. *For every variable $x \in dom\ \widehat{E}$, $\gamma(\widehat{E}(x)) = E(x)$.*

As an example, assume a function definition $f\ x\ y = C\ y$. Its signature consists of the following relations: $\{res \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} res, res \xrightarrow{1_C} \bullet \xleftarrow{\epsilon} y\}$. Assume we execute a call $f\ z\ z$ where $E(z) = p$, $h(p) = C'\ p'\ p'$, $h(p') = C''\ 3$, i.e. $E_f = [x \mapsto p, y \mapsto p]$. In this case x and y are not disjoint and also contain internal sharing. We can define $(\widehat{E}_f, \widehat{h})$ such that $\widehat{E}_f(x) = p_1$, $\widehat{E}_f(y) = p_2$, $\widehat{h}(p_1) = C'\ p'_1\ p''_1$, $\widehat{h}(p_2) = C'\ p'_2\ p''_2$ and $\widehat{h}(p'_1) = \widehat{h}(p''_1) = \widehat{h}(p'_2) = \widehat{h}(p''_2) = C''\ 3$. Then $\gamma(p_1) = \gamma_(p_2) = p$, $\gamma(p'_1) = \gamma(p''_1) = \gamma(p'_2) = \gamma(p''_2) = p'$ is an entanglement from $(\widehat{E}_f, \widehat{h})$ to (E_f, h).

The following lemma proves that both the hypothetical and the real execution proceed in parallel and that the information inside the heap is the same although with a different shape. In Figure 11 we show the final heaps of the executions corresponding to the previous example.

Lemma 21. *Assume an execution $E \vdash h, e \Downarrow h', v$ and a configuration $(\widehat{E}, \widehat{h})$. For every entanglement γ from $(\widehat{E}, \widehat{h})$ to (E, h) there exist some $\widehat{h}'$, $\widehat{v}'$ and γ' such that:*

1. $\widehat{E} \vdash \widehat{h}, e \Downarrow \widehat{h}', \widehat{v}$.

2. *γ' is a conservative extension of γ. That is, $\gamma \subseteq \gamma'$.*
3. *γ' is an entanglement from $(\widehat{E}, \widehat{h}')$ to (E, h').*
4. *$\gamma'(\widehat{v}) = v$.*

Proof. By induction on the $\Downarrow$-derivation of e. □

For the same heap several entanglements may be defined, but we are interested in a configuration $(\widehat{E}, \widehat{h})$, where everything is untangled, as shown in the previous example. This is because, then $R_0 = \{x_i \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x_i \mid i = 1..n\}$ correctly approximates its sharing.

Lemma 22. *For any configuration (E, h) there exists another configuration $(\widehat{E}, \widehat{h})$ and an entanglement γ from $(\widehat{E}, \widehat{h})$ to (E, h) such that the set $\{x \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x \mid x \in dom\ E\}$ is a correct approximation of $(\widehat{E}, \widehat{h})$.*

Proof. (Sketch) We define the following function $\psi(v, h) = (\widehat{v}, \widehat{h}, \gamma)$ which untangles the closure of a pointer p in a heap h and yields the corresponding entanglement γ:

$$\begin{array}{lcl}
\psi(c, h) & = & (c, \emptyset, \emptyset) \\
\psi(p, h) & = & (\widehat{p}, [\widehat{p} \mapsto C\ \widehat{v}_1 \ldots \widehat{v}_n] \uplus \widehat{h}_1 \uplus \cdots \uplus \widehat{h}_n, [\widehat{p} \mapsto p] \uplus \gamma_1 \uplus \cdots \uplus \gamma_n) \\
& & \quad \textbf{where} \quad C\ v_1 \ldots v_n = h(p) \\
& & \qquad\qquad\quad (\widehat{v}_i, \widehat{h}_i, \gamma_i) = \psi(v_i, h) \text{ for all } i \in \{1..n\} \\
& & \qquad\qquad\quad \widehat{p} \text{ is a fresh pointer}
\end{array}$$

And we define $(\widehat{E}, \widehat{h})$ as

$$\widehat{h} = \biguplus_{x \in \text{dom } E} \widehat{h}_x \qquad \widehat{E} = [x \mapsto \widehat{v}_x \mid x \in \text{dom } E] \qquad \gamma = \biguplus_{x \in \text{dom } E} \gamma_x$$

where $(\widehat{v}_x, \widehat{h}_x, \gamma_x) \overset{\text{def}}{=} \psi(E(x), h)$. Then, we prove the lemma by induction on the length of the longest pointer chain that can be followed starting from v. □

In the example above, signature is $R' = S\ [\![e_f]\!]\ R_0\ \Sigma = \{res \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} res, res \xrightarrow{1_C} \bullet \xleftarrow{\epsilon} y\}$. The environment of the call is approximated by $R = \{x \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} y, x \xrightarrow{1_{C'}} \bullet \xleftarrow{2_{C'}} x, y \xrightarrow{1_{C'}} \bullet \xleftarrow{2_{C'}} y\}$. So the final sharing is approximated by $R \uplus^*_{res} R'$ which merges the context of the call with the signature, and contains $\{\mathit{res} \xrightarrow{1_C} \bullet \xleftarrow{\epsilon} y, \mathit{res} \xrightarrow{1_C} \bullet \xleftarrow{\epsilon} x, \mathit{res} \xrightarrow{1_C \cdot 1_{C'}} \bullet \xleftarrow{1_C \cdot 2_{C'}} \mathit{res}, \mathit{res} \xrightarrow{1_C \cdot 1'_C} \bullet \xleftarrow{2_{C'}} x, \mathit{res} \xrightarrow{1_C \cdot 1'_C} \bullet \xleftarrow{2_{C'}} y, \mathit{res} \xrightarrow{1_C \cdot 2'_C} \bullet \xleftarrow{1_{C'}} x, \mathit{res} \xrightarrow{1_C \cdot 2'_C} \bullet \xleftarrow{1_{C'}} y\}$. This happens for each R approximating a context call, so R' is a correct signature for f. We prove this in the following theorem.

Theorem 23. *Assume a function definition $f\ \overline{x} = e_f$, a set of relations $R_0 = \{x_i \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} x_i \mid i = 1..n\}$, and an environment Σ with correct signatures. If $R' = S\ [\![e_f]\!]\ R_0\ \Sigma$, then R' is a correct signature for f.*

Proof. (Sketch) Assume a configuration (E, h) with dom $E = \{\overline{x}\}$ and a set of relations R such that $R \succeq (E, h)$. If we execute e_f under the configuration (E, h) we get $E \vdash h, e_f \Downarrow h', v$ for some h', v. By Lemma 22 there exists a mapping γ which entangles a configuration $(\widehat{E}, \widehat{h})$ into (E, h), where $(\widehat{E}, \widehat{h})$ is correctly approximated by R_0. Assume we execute e_f under the untangled configuration so as to get, by Lemma 21, that $\widehat{E} \vdash \widehat{h}, e_f \Downarrow \widehat{h}', \widehat{v}$ for some $\widehat{h}'$ and $\widehat{v}$. By correctness theorem (Theorem 19) we know that $R' \succeq (\widehat{E}', \widehat{h}')$, where $\widehat{E}' \stackrel{\text{def}}{=} \widehat{E} \uplus [\mathit{res} \mapsto \widehat{v}]$. Then, we prove that $R \uplus^*_{\mathit{res}} R' \succeq (E', h')$. In order to prove this we need two auxiliary properties:

1. For every variable $z \in \text{dom } E$ such that $E(z) \xRightarrow{w_z} \bullet \xLeftarrow{w_v} v$ (in h') there exists a variable $y \in \text{dom } E$ and a word w_y such that $\widehat{E}(y) \xRightarrow{w_y} \bullet \xLeftarrow{w_v} \widehat{v}$ (in $\widehat{h}'$) and $E(z) \xRightarrow{w_z} \bullet \xLeftarrow{w_y} E(y)$ (in h). This means, by Lemma 17, that the sharing between the result and a variable is captured by $R \uplus^*_{\mathit{res}} R'$.
2. For every w_1, w_2 such that $v \xRightarrow{w_2} \bullet \xLeftarrow{w_1} v$ (in h') holds, but $\widehat{v} \xRightarrow{w_2} \bullet \xLeftarrow{w_1} \widehat{v}$ (in $\widehat{h}'$) does not, either
   - there exist two variables $y \neq z \in \text{dom } E$ and two words w_y, w_z such that:
     (a) $E(y) \xRightarrow{w_y} \bullet \xLeftarrow{w_z} E(z)$ (in h).
     (b) $\widehat{E}(y) \xRightarrow{w_y} \bullet \xLeftarrow{w_1} \widehat{v}$ (in $\widehat{h}'$).
     (c) $\widehat{v} \xRightarrow{w_2} \bullet \xLeftarrow{w_z} \widehat{E}(z)$ (in $\widehat{h}'$).
   - or, there exists a variable $z \in \text{dom } E$ and words w_v, w_z, w'_1, w'_2 such that
     (a) $E(z) \xRightarrow{w_z w'_1} \bullet \xLeftarrow{w_z w'_2} E(z)$ (in h).
     (b) $\widehat{v} \xRightarrow{w_v} \bullet \xLeftarrow{w_z} \widehat{E}(z)$ (in $\widehat{h}'$).
     (c) $w_1 = w_v w'_1$ and $w_2 = w_v w'_2$

This means that the internal sharing of *res* which is not created inside the function, can only come from an argument with internal sharing or from two arguments sharing between them and with the result in the appropiate way. The definition of $R \uplus^*_{res} R'$ also covers this situations, as Lemmas 17 and 18 show.

$\square$

5. Implementation Issues and Cost

The analysis presented in Section 3 contains some tests and operations on regular languages that deserve a detailed comment in order to see whether all of them are decidable, and what their costs are. An efficient implementation of these operations is crucial, as the abstract interpretation function makes intensive use of them. We need the following operations:

1. To test whether a regular language L is empty, i.e. $L = \emptyset$. This is necessary in order to discard those relations $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ where p_1 or p_2 are empty.
2. Given regular languages L_1 and L_2, to compute their concatenation $L_1.L_2$.
3. Given regular languages L_1 and L_2, to compute their derivation $L_1|_{L_2}$.
4. Given regular languages L_1 and L_2, to test whether $L_1 \subseteq L_2$. This is used to check whether a fixpoint has been reached.
5. Given regular languages L_1 and L_2, to compute their union $L_1 + L_2$. This is needed at the end of each analysis iteration (see Section 5.3).

For our implementation we have explored two alternatives for representing regular languages: nondeterministic finite automata with ϵ transitions and regular expressions. The next two sections are devoted to each of these. A detailed comparison of their respective execution times for some case studies is deferred to Section 6.

5.1. Regular languages via non-deterministic finite automata

In this section we represent regular languages by non-deterministic finite automata with ϵ transitions (in what follows, NFA). We will denote them by $A = (\Sigma, Q, i, F, \delta)$, where, as usual, Σ is the alphabet ($\mathcal{V}$ in our setting), Q is the set states, i is the initial state, F contains the final states, and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation. As usual in automata theory, we denote by δ^* the extension of δ to a subset of $Q \times \Sigma^* \times Q$. In order to achieve efficiency, we do not convert NFAs to other representations unless it becomes unavoidable. As we will see, this happens when checking whether a fixpoint has been reached. There we convert the NFAs to DFAs (deterministic finite automata). The rest of operations needed by the analysis are performed in the NFA 'world'. Trying to use DFAs for them would result in a less efficient implementation. In particular, concatenation and derivation of regular languages would be much more expensive with DFAs than with NFAs.

The emptiness test in a NFA can be achieved [7] by looking for a final state that is reachable from the initial one. If $n = |Q|$ is the number of states of A, the algorithm costs $O(n^2)$.

Given NFA automata A_1 and A_2, the computation of the automaton recognising $L(A_1).L(A_2)$ can be constructed with a cost $O(n)$, just by connecting with ϵ-transitions the final states of A_1 to the initial state of A_2.

Given NFA automata $A_1 = (\Sigma, Q_1, i_1, F_1, \delta_1)$ and $A_2 = (\Sigma, Q_2, i_2, F_2, \delta_2)$, the automaton recognising $L(A_1)|_{L(A_2)}$ is more involved. In fact, we have not found in the literature an algorithm to compute it and have invented our own, which involves a parallel traversal of A_1 and A_2 whilst collecting a set I of states of Q_1 that will be the initial states of the resulting automaton. The algorithm Derivative is described informally as follows:

1. Start with an empty set I and an empty queue $\mathcal{Q}$.
2. Insert the pair (i_1, i_2) into $\mathcal{Q}$.
3. While $\mathcal{Q}$ is not empty:
   (a) Take the first element (q_1, q_2) of $\mathcal{Q}$.
   (b) If $q_2 \in F_2$ then add q_1 to I.
   (c) For each $v \in \Sigma$, let Q'_1 denote the set of $q'_1 \in Q_1$ such that $(q_1, v, q'_1) \in \delta_1^*$ and Q'_2 the set of $q'_2 \in Q_2$ such that $(q_2, v, q'_2) \in \delta_2^*$. Insert at the end of $\mathcal{Q}$ those elements of $Q'_1 \times Q'_2$ that have not been inserted in $\mathcal{Q}$ before.
4. Return $(\Sigma, Q_1 \cup \{q_0\}, q_0, F_1, \delta_R)$, being $\delta_R = \delta_1 \cup \{(q_0, \epsilon, q) \mid q \in I\}$

Lemma 24. *The automaton returned by* Derivative *recognises the language* $L(A_1)|_{L(A_2)}$.

Proof. (Sketch) It is easy to see that whenever we insert a pair (q_1, q_2) into $\mathcal{Q}$ there exists a word $w \in \Sigma^*$ such that $(i_1, w, q_1) \in \delta_1^*$ and $(i_2, w, q_2) \in \delta_2^*$. In particular, when $q_2 \in F_2$ it holds that $w \in L(A_2)$. Reciprocally, for every word w such that $(i_1, w, q_1) \in \delta_1^*$ and $(i_2, w, q_2) \in \delta_2^*$ then (q_1, q_2) is inserted in $\mathcal{Q}$.

Given this invariant, assume that $w \in L(A_1)|_{L(A_2)}$, implying the existence of $w_1 \in L(A_1)$ and $w_2 \in L(A_2)$ such that $w_1 = w_2w$. Therefore there exists some $q_2 \in F_2$ such that $(i_2, w_2, q_2) \in \delta_2^*$. Since w_2 is a prefix of w_1 there must exist a $q_1 \in Q_1$ and a $q_1' \in F_1$ such that $(i_1, w_2, q_1), (q_1, w, q_1') \in \delta_1^*$. As a consequence of this, the pair (q_1, q_2) is inserted into $\mathcal{Q}$ during the algorithm. Moreover, since $q_2 \in F_2$ it holds that $q_1 \in I$. From the fact $(q_1, w, q_1') \in \delta_1^*$ it follows that w is recognised by the resulting automaton.

Reciprocally, if w is recognised by the automaton returned by Derivative there exists some $q' \in F_1$ and $q \in I$ such that $(q, w, q') \in \delta_R$. Therefore, there exists a $q_2 \in F_2$ such that (q_2, q) is inserted into $\mathcal{Q}$, which implies the existence of a word $w_2 \in L(A_2)$ such that $(i_2, w_2, q_2) \in \delta_2^*$, and $(i_1, w_2, q) \in \delta_1^* \subseteq \delta_R$. With the fact that $(q, w, q') \in \delta_R$ and $q' \in F_1$ it holds that $w_2w \in L(A_1)$. Therefore, $w \in L(A_1)|_{L(A_2)}$.

□

The algorithm Derivative inserts, in the worst case, the whole set $Q_1 \times Q_2$ into the queue. For each element (q_1, q_2) in the queue it tries to insert $Q_1' \times Q_2'$ into the queue, which may be equal to $Q_1 \times Q_2$. Therefore, the overall worst-case cost of Derivative is in $O(n^4)$, although this case corresponds to an automaton in which all the states are directly interconnected. These kind of automata are seldom generated in our analysis.

Given NFA automata A_1 and A_2, $L(A_1) \subseteq L(A_2)$ if and only if $L(A_1) \cap L(A_2) = L(A_1)$, so inclusion is a particular case of equality. Unfortunately, equality cannot be directly computed on NFA's. They must be converted to DFA, and then their equality tested with the well-known table-filling algorithm [7], which has a cost $O(n^2)$. But the conversion from NFA to DFA has a worst-case cost in $O(n^3 2^n)$. This is because the states of the DFA are subsets of the NFA set of states, and can in theory be up to 2^n. In practice, however, the DFA has about the same number of states than the NFA it comes from.

We have developed two different implementations of each of these operations (see Section 6 for a performance comparison). The first implementation manages automata in a purely functional fashion. That is, each automata is represented as a tuple $(\Sigma, Q, i, F, \delta)$, and whenever we apply one of the operators explained above we still preserve the original automaton to which the operator is applied. This implies that, given two automata A_1 and A_2, the computation of $L(A_1)\cdot L(A_2)$ may generate a copy of A_1 before computing the result. In order to avoid this duplication we have developed an alternative implementation of automata, in which we have a single state space shared by all automata. In this case, an automaton is represented as a tuple (i, F) containing only the initial state and the accepting states, whereas the states themselves and their transitions are represented in the shared state space. By using this implementation we can avoid generating copies of automata whenever we apply an operator on them, but this is an 'impure' implementation in the sense that the application of these operators may partly mutate the operands. In the example above, assume that $A_1 = (i_1, F_1)$ and $A_2 = (i_2, F_2)$ if we want to compute $L(A_1).L(A_2)$ we just have to add ϵ-transitions in the shared state space from the accepting states of A_1 to the initial state of A_2, and the result would be represented by (i_1, F_2). This comes, however, at the cost of modifying the operand A_1. As another example, given an automaton A_3, the concatenation of $L(A_3)$ with itself would yield $L(A_3)^*$. We can ensure that the concatenation of the languages of two automata in a shared state space is sound provided their respective initial states do not reach a common state. Fortunately, each iteration of our abstract interpretation function only applies the concatenation operator to automata satisfying this disjointness property. Between iterations (see Section 5.3) the simplification operation builds a new state space from scratch, so the automata resulting from different iterations of the abstract interpretation are pairwise disjoint as well.

5.2. Regular languages via regular expressions

As an alternative to finite automata, we assume that the analysis manages regular expressions for representing regular languages. In this section we consider the set of regular expressions given by the following grammar:

$$p ::= \emptyset \mid \epsilon \mid v \in \mathcal{V} \mid p \cdot p \mid p + p \mid p^*$$

With this representation the operations of language concatenation and union are straightforward. The emptiness of the language given by a regular expression can be checked as follows:

- $L(\emptyset) = \emptyset, L(\epsilon) \neq \emptyset$, and $L(v) \neq \emptyset$ for any $v \in \mathcal{V}$.
- $L(p_1 \cdot p_2) = \emptyset$ iff $L(p_1) = \emptyset$ or $L(p_2) = \emptyset$.
- $L(p_1 + p_2) = \emptyset$ iff $L(p_1) = \emptyset$ and $L(p_2) = \emptyset$.

- $L(p^*) = \emptyset$ iff $L(p) = \emptyset$.

As in the case of finite automata the derivation operation is more involved. Assume we have two expressions p_1, p_2 and we want to compute a regular expression representing the language $L(p_1)|_{L(p_2)}$. The case in which p_2 is a symbol $v \in \mathcal{V}$ is widely covered in the literature [8, 9, 10]:

$$\begin{array}{lcllcll}
\emptyset|_v & = & \emptyset & (p_1 \cdot p_2)|_v & = & p_1|_v \cdot p_2 + p_2|_v & \text{if } \epsilon \in L(p_1) \\
\epsilon|_v & = & \emptyset & (p_1 \cdot p_2)|_v & = & p_1|_v \cdot p_2 & \text{if } \epsilon \notin L(p_1) \\
v|_v & = & \epsilon & (p_1 + p_2)|_v & = & p_1|_v + p_2|_v & \\
v'|_v & = & \emptyset \text{ if } v' \neq v & (p^*)|_v & = & p|_v \cdot p^* &
\end{array}$$

The rest of the cases, except when $p_2 = p^*$ for some p, can be deduced from the definition of the derivative operator:

$$p|_\emptyset = \emptyset \qquad p|_\epsilon = p \qquad p|_{p_1+p_2} = p|_{p_1} + p|_{p_2} \qquad p|_{p_1 \cdot p_2} = (p|_{p_1})|_{p_2}$$

In order to derive with respect to an expression p^* we take advantage of the fact that p^* and $\epsilon + pp^*$ denote the same language. Therefore:

$$\begin{array}{rcl}
p_1|_{p^*} & = & p_1|_{\epsilon+pp^*} \\
 & = & p_1 + (p_1|_p)|_{p^*} \\
 & = & p_1 + p_1|_p + ((p_1|_p)|_p)|_{p^*} \\
 & = & \ldots
\end{array}$$

Let us define $q^0 = p_1$, and $q^i = (q^{i-1})|_p$ for $i > 0$. We can rewrite $p_1|_{p^*}$ as follows:

$$p_1|_{p^*} = \sum_{i=0}^{\infty} q^i \tag{3}$$

Brzozowski proved [8] that the set of languages $\{L(q^i)\}_{i \in \mathbb{N}}$ is finite when p is a letter of the vocabulary, but this does not necessarily imply that the set $\{q^i\}_{i \in \mathbb{N}}$ of regular expressions is finite. However, if we consider equality of regular expressions modulo the following equations,

$$p + p = p \qquad p_1 + p_2 = p_2 + p_1 \qquad (p_1 + p_2) + p_3 = p_1 + (p_2 + p_3) \tag{4}$$

it is proved in [8] that we can get a finite number of distinct q^i. This result can be extended by structural induction so as to include the case in which p is a regular expression not containing the closure operator $(^*)$. Moreover, by induction on the number of nested $(^*)$-expressions we can extend this result to the case in which p is an arbitrary regular expression. As a consequence of this, the number of terms in the sum (3) is finite, and we can stop generating q^i terms as soon we reach some q^j which is equal (modulo (4)) to some q^k $(k < j)$ computed previously.

In our implementation we use a larger number of equality rules owed to Owens et al. [9]. This set of rules leads, in many cases, to minimal expressions. In the worst case the application of the derivative operator as defined in (3) may yield an expression which is exponentially larger than p_1, especially when dealing with expressions with several nested closure operations in p_2. Fortunately, these kind of regular expressions are hardly ever generated in the context of our shape analysis.

The only operation remaining is subset inclusion. Given two expressions p_1 and p_2 we check whether $L(p_1) \subseteq L(p_2)$ by transforming these expressions to DFA by using derivative-based techniques [9], which directly lead to minimal automata in many cases. The inclusion of these automata is checked as done in the previous section. Another approach consists in finding a simulation between the states of the generated DFA (see [11]).

5.3. Analysing function definitions

When interpreting the body of a recursive function f we start by setting an empty signature for f, i.e. $\Sigma(f) = \emptyset$. It is easy to show that the interpretation is monotonic in the lattice:

$$\langle \mathcal{M}(\mathit{Var}_f \times \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \times \mathit{Var}_f), \emptyset, \top, \sqsubseteq, \cup, \cap \rangle$$

where $\mathcal{M}$ stands for 'multiset of', Var_f are the bound variables of f, Σ^* is the top regular language, and $\top$ is the maximum relation. We need to ensure that no two tuples with the same type exist relating the same variables. So, at the end of each iteration, the following collapsing rule is used:

$$\frac{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y \in R \qquad x \xrightarrow{p_3} \bullet \xleftarrow{p_4} y \in R \qquad \mathit{type}(x, p_1) = \mathit{type}(x, p_3)}{\text{replace in } R \text{ the two tuples by } x \xrightarrow{p_1+p_3} \bullet \xleftarrow{p_2+p_4} y}\ \mathit{OR}$$

Should not we use this rule, the abstract domain, regarding only the relations between program variables, would be infinite.

For pragmatic reasons, in the actual implementation we do not keep the tuples $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ separated by types. During the interpretation, we keep at most one single tuple for each pair of variables (x, y), and we use the collapsing rule *OR* even if $\mathit{type}(x, p_1)$ and $\mathit{type}(x, p_3)$ are different. This means that, in general, $\mathit{type}(x, p_1)$ and $\mathit{type}(y, p_2)$ may contain more than one type. We do not lose precision by doing this because we can always separate the different languages by types and remove the ill-typed sublanguages. In fact, we do this when presenting the signatures to the user. Having at most one tuple per variable pair has advantages from the efficiency point of view, and it also makes our algorithms simpler.

The order relation between two tuples relating the same pair of variables, and having the same type, is as follows:

$$x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y \;\sqsubseteq\; x \xrightarrow{p_1'} \bullet \xleftarrow{p_2'} y$$

if $L(p_1) \subseteq L(p_1')$ and $L(p_2) \subseteq L(p_2')$. Let us denote by $\mathcal{I}_f\ \Sigma$ the interpretation of e_f with current signature environment Σ, returning Σ with f's signature updated. By monotonicity, we have:

$$\emptyset \sqsubseteq \mathcal{I}_f\ \emptyset \sqsubseteq \mathcal{I}_f\ (\mathcal{I}_f\ \emptyset) \sqsubseteq \ldots \sqsubseteq (\mathcal{I}_f)^i\ \emptyset \sqsubseteq \ldots$$

Disregarding the regular languages, this chain is finite because so is Var_f, and the number of different types of the program. Then, the least fixpoint can be reached after a finite number of iterations. If n is the number of f's formal arguments, then at most n iterations are needed. This is because functional languages have no variable updates, and then there never may arise sharing relations between the formal arguments as a consequence of the function body actions. The only possible relations will be between the function's result and its arguments.

Considering now the regular languages, infinite ascending chains are possible, i.e. one can obtain infinite chains $L_1 \subseteq L_2 \subseteq L_3 \subseteq \ldots$.

The least upper bound of such a sequence of regular languages needs not to be a regular one. But, at least, there always exists the regular language Σ^* greater than any other one. In order to ensure termination of the fixpoint computation, we use the following *widening* technique [12]:

1. Based on the form of the automata or regular expression denoting the increasing language sequence, and by using some heuristics, we guess a regular language L such that $\bigcup_i L_i \subseteq L$. Then, we iterate the interpretation by using this automaton as an assumption in f's signature.

2. If A is a fixpoint or a post-fixpoint, then we are done. Otherwise, we use Σ^* as the upper bound of the sequence. In terms of precision, $x \xrightarrow{\Sigma^*} \bullet \xleftarrow{\Sigma^*} y$ is completely uninformative about the paths through which x and y share their common descendant.

In the case of finite automata, the heuristic consists of comparing the automata sequence obtained for a given relation $x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y$ in the successive iterations, and discovering growing sequences reaching three or more states related by the same alphabet symbol. For example q_1, q_2, q_3, with $(q_1, a, q_2), (q_2, a, q_3) \in \delta$. These sequences are collapsed into a single state class q, with a single iterative transition $(q, a, q) \in \delta$. The resulting automata is compared with the non-widened one, to ensure that they are equivalent regarding the remaining transitions. In all the examples we have tried this heuristic appears to be enough to reach a fixed point.

We pay now attention to the asymptotic cost of the whole interpretation. We choose the size n of a function to be its number of bound variables. This figure is linearly related to the size of its abstract syntax tree, and to the number of lines of its source code. How is n related to the size of the inferred automata in terms of their number of states? It is easy to check that every bound variable y introduces a relation $x \xrightarrow{j_C} \bullet \xleftarrow{\epsilon} y$ with a prior bound variable x. This increases by one the number of states of the y relations with respect to those of the x relations. So, the automata number of states grow from one to the abstract syntax tree height, when going from the initial expression to the deepest ones. Assuming a reasonably balanced syntax tree, we consider $\log n$ to be an accurate bound to the automata size.

If a function definition has n bound variables, and considering as a constant the number of different types, in the worst case there can be up to $O(n^2)$ tuples in the current relation R. The computation of a single closure operation $R \uplus_x \{x \xrightarrow{p_1} \bullet \xleftarrow{p_2} y\}$ (see Figure 9) introduces as many relations $x \xrightarrow{p_x} \bullet \xleftarrow{p_z} z$ as prior relations $y \xrightarrow{p_y} \bullet \xleftarrow{p_z} z$ are there in R, i.e. $O(n)$ in the worst case. A single iteration of the abstract interpretation will compute one such closure for every bound variable, giving an upper bound of $O(n^2)$ new relations per iteration. For each one, two languages $A_1|_{A_2}. A_3$ must be computed, giving a total cost of $O(n^2 \log^4 n)$ per iteration.

It has been said that the number of iterations is at most the function's number of arguments, which is usually small. Even if it is not, in practice it suffices to perform only three iterations of the analysis before applying the widening, and then an additional iteration in order to check that the

```
last xs = case xs of
            x:xx -> case xx of
                      []   -> {* R1 *} x
                      y:yy -> {* R2 *} last xx
```

Figure 12: Definition of the function `last`

fixpoint has been reached. This checking is the most expensive operation of the analysis. A maximum of $O(n^2)$ languages are tested for equality, giving a total theoretical cost of $O(n^2 2^{\log n} \log^3 n)$ in the worst case, i.e. $O(n^3 \log^3 n)$. This is bigger than the prior cost of $O(n^2 \log^4 n)$ per iteration.

A worst-case theoretical cost of $O(n^3 \log^3 n)$ is by no means a low one, but we consider it to be rather pessimistic. We remark that we are assuming each variable to be related to each other, and all conversions from NFA to DFA to produce an exponential blow-up of states. This leads us to think that this theoretical cost is almost never reached. Also, in functional programming it is common to write small functions. So, the number n of bound variables can be expected to remain below 20 for most of the functions (the reader is invited to check this assertion for the functions presented in this paper).

In practice, our analysis is affordable for medium-sized functions. More importantly, it is modular, because even if analysing a single function of size n takes a time in $O(n^3 \log^3 n)$, once it is analysed all its relevant information is recorded in the signature environment. Hence, analysing a big program just costs the addition of the costs of analysing each individual function.

5.4. A Small Example

In order to illustrate the analysis, we present in Figure 12 the code of a function `last` computing the last element of a non-empty list. By iterating once the interpretation, and in the places marked in the text, we get the following two sets:

$$\begin{array}{rl}
R_1 = & \{xs \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} xs\} \uplus_x \{xs \xrightarrow{1} \bullet \xleftarrow{\epsilon} x\} \uplus_{xx} \\
& \{xs \xrightarrow{2} \bullet \xleftarrow{\epsilon} xx\} \\
R_2 = & R_1 \uplus_y \{xx \xrightarrow{1} \bullet \xleftarrow{\epsilon} y\} \uplus_{yy} \{xx \xrightarrow{2} \bullet \xleftarrow{\epsilon} yy\}
\end{array}$$

Then $\Sigma_1 = \mathcal{I}_{last}\ \{last \mapsto \emptyset\} = \{res \xrightarrow{\epsilon} \bullet \xleftarrow{1} xs\}$, where we omit the reflexive relations. By applying again the interpretation, we get:

$$\begin{array}{rcl} \Sigma_2 = \mathcal{I}_{last}\ \{last \mapsto \Sigma_1\} & = & \{res \xrightarrow{\epsilon} \bullet \xleftarrow{21} xs\} \ \cup \\ & & \{res \xrightarrow{\epsilon} \bullet \xleftarrow{1} xs\} \\ & = & \{res \xrightarrow{\epsilon} \bullet \xleftarrow{21+1} xs\} \end{array}$$

The language 21 is obtained by the transitive closure $\{res \xrightarrow{\epsilon} \bullet \xleftarrow{1} xx\} \uplus_{res} \{xs \xrightarrow{2} \bullet \xleftarrow{\epsilon} xx\}$. In the next round, we get $\Sigma_3 = \{res \xrightarrow{\epsilon} \bullet \xleftarrow{2(21+1)+1} xs\}$ Applying now the widening step, we get $\Sigma_3 = \{res \xrightarrow{\epsilon} \bullet \xleftarrow{2^*1+21+1} xs\}$, and by applying the interpretation once more:

$$\begin{array}{rcl} \mathcal{I}_{last}\ \{last \mapsto \Sigma_3\} & = & \{res \xrightarrow{\epsilon} \bullet \xleftarrow{2(2^*1+21+1)} xs\} \ \cup \\ & & \{res \xrightarrow{\epsilon} \bullet \xleftarrow{1} xs\} \\ & = & \{res \xrightarrow{\epsilon} \bullet \xleftarrow{2(2^*1+21+1)+1} xs\} \end{array}$$

The final test is $2(2^*1+21+1)+1 \subseteq 2^*1+21+1$ which returns *true* because all the words in the left language are also in the right one. Notice that both expressions could be further simplified to 2^*1. This language clearly expresses that the result of `last` is a descendant of the argument list that can be reached by taking the tail of the list a number of times and then by taking the head.

6. Case Studies

We have implemented the analysis presented in this paper as a part of our *Safe* compiler, written in Haskell [13]. While the implementation of the abstract interpretation rules of Figure 8 was rather straightforward, the closure operation defined in Figure 9 was much more involved. Regarding the manipulation of regular languages, we started by extending *HaLeX* [14], a Haskell library for automata manipulation, with new operations such as language intersection, derivation and equality. Since this library was targeted towards educational purposes, the obtained performance measures were poor, so we have considered three alternative implementations of the operations on regular languages:

[NFA pure] A rewrite of *HaLeX* with more efficient underlying data structures, such as `Data.Map` and `Data.Set` from Haskell's Hierarchical Libraries. This is a purely functional library. The application of an operation may generate a copy of its operands.

[**NFA C**] An implementation of NFA written in C and its corresponding Haskell bindings. In this version our automata are represented in a shared state space (as described at the end of Section 5.1), so purity is sacrificed for efficiency.

[**RegExp**] A small purely functional library for regular expression manipulation, as described in Section 5.2.

Besides the examples already shown in the paper we have applied our analysis to some case studies that involve list and binary tree manipulations. The following functions show how our analysis can also detect internal sharing in the data structure given as a result. This is useful to know whether a given data structure is laid out in memory without overlapping.

```
buildTree x 0 = Empty
buildTree x n = Node (buildTree x (n-1)) x (buildTree x (n-1))

buildTreeSh x 0 = Empty
buildTreeSh x n = let t = buildTreeSh x (n-1) in Node t x t
```

The shape analysis yields the following results:

$$\begin{array}{rcl} \mathit{buildTree}\ x\ n & : & \{\mathit{res} \xrightarrow{(1+3)^{*}2} \bullet \xleftarrow{\epsilon} x \\ & & ,\ \mathit{res} \xrightarrow{(1+3)^{*}2} \bullet \xleftarrow{(1+3)^{*}2} \mathit{res}\} \\ \mathit{buildTreeSh}\ x\ n & : & \{\mathit{res} \xrightarrow{(1+3)^{*}2} \bullet \xleftarrow{\epsilon} x \\ & & ,\ \mathit{res} \xrightarrow{(1+3)^{*}2} \bullet \xleftarrow{(1+3)^{*}2} \mathit{res} \\ & & ,\ \mathit{res} \xrightarrow{(1+3)^{*}} \bullet \xleftarrow{(1+3)^{*}} \mathit{res}\} \end{array}$$

These two functions are included in a set `basic` of small functions that generate, in a somewhat artificial way, sharing between the results of the function and its parameters. Some of them have been already shown in this paper. We have also tried our analysis with several medium-sized examples, such as implementations of *Quicksort* (`quicksort`), *Mergesort* (`mergesort`), libraries for dealing with balanced binary trees (`AVLTrees`), bitmap images represented by quad trees (`quadTrees`) and priority queues implemented with

Module	F/L	NFA pure (ms)	NFA C (ms)	RegExp (ms)
`basic`	15/47	53.6	24.8	19.2
`qsort`	3/14	209.4	107.8	96.4
`mergesort`	3/15	424.0	94.2	75.6
`AVLTrees`	10/57	5981.0	274.2	145.2
`quadTrees`	9/48	212.0	56.9	33.5
`priorityQueues`	6/33	1178.6	85.2	32.8
`compiler`	30/278	36786.3	7921.0	5741.7

CPU: Intel$^{(R)}$ Quad Core^{TM} i7-2640M CPU @ 2.80GHz 64-bit / Mem: 7.7GB
Compiled with GHC 7.4.2

Figure 13: Execution times with the three implementations.

leftist trees (`priorityQueues`). We have included, as a more complex example, a compiler for a a simple imperative language (`compiler`). The running time of the shape analysis for each of these examples is shown in Figure 13. The column F/L respectively contains the number of functions and lines of code in each example. Figures 14 and 15 contain the results of the analysis when applied to the most relevant functions of each case study. The numbers in the right hand side of each sharing relation denote parameter positions. For the sake of clarity we have omitted the name of the constructors in the sharing paths. The `AVLTree a` type is defined as follows:

```
data AVLTree a = Empty | Node Int (AVLTree a) a (AVLTree a)
```

where the `Int` parameter in the `Node` constructor contains the height of the tree. The `joinAVL` function builds a tree from its parameters while maintaining the balancedness properties. The result points to both the structure and the elements of the input trees. The relations $res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 1$ and $res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 3$ report that any of the subtrees in the first and third parameters may share with any of the subtrees in the result. These relations may actually occur at runtime, as a consequence of the rotations that are performed on the input trees. For instance, a subtree of the left child of the tree passed as first parameter might be part of the right child of the result of `joinAVL`. With the functions `insertAVL` and `deleteAVL` we obtain similar results.

The `quadTrees` module defines the following data type:

`mergesort`

`unshuffle :: [a] -> ([a],[a])`

$\{res \xrightarrow{12^*1+22^*1} \bullet \xleftarrow{2^*1} 1\}$

`merge :: [a] -> [a] -> [a]`

$\{res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} 1,\ res \xrightarrow{2^*} \bullet \xleftarrow{2^*} 1,\ res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} 2,\ res \xrightarrow{2^*} \bullet \xleftarrow{2^*} 2\}$

`msort :: [a] -> [a]`

$\{res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} 1\}$

`AVLTrees`

`joinAVL :: AVLTree a -> a -> AVLTree a -> AVLTree a`

$\{res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 1, res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{(2+4)^*3} 1, res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{\epsilon} 2,$
$res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 3, res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{(2+4)^*3} 3\}$

`insertAVL :: a -> AVLTree a -> AVLTree a`

$\{res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{\epsilon} 1, res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 2, res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{(2+4)^*3} 2\}$

`deleteAVL :: a -> AVLTree a -> AVLTree a`

$\{res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 2, res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{(2+4)^*3} 2\}$

`quadTrees`

`consQTree :: QTree -> QTree -> QTree -> QTree -> QTree`

$\{res \xrightarrow{1} \bullet \xleftarrow{\epsilon} 1, res \xrightarrow{2} \bullet \xleftarrow{\epsilon} 2, res \xrightarrow{3} \bullet \xleftarrow{\epsilon} 3, res \xrightarrow{4} \bullet \xleftarrow{\epsilon} 4\}$

`rotate :: QTree -> QTree`

No sharing detected

`flipH :: QTree -> QTree`

No sharing detected

`overlay :: QTree -> QTree -> QTree`

$\{res \xrightarrow{(1+2+3+4)^+} \bullet \xleftarrow{(1+2+3+4)^+} 1, res \xrightarrow{(1+2+3+4)^*} \bullet \xleftarrow{(1+2+3+4)^*} 2\}$

Figure 14: Analysis results of some of the case studies.

`priorityQueues`

`cons :: Leftist a -> a -> Leftist a`

$\{res \xrightarrow{(2+4)(2+4)} \bullet \xleftarrow{(2+4)} 1, res \xrightarrow{(2+4)3} \bullet \xleftarrow{3} 1, res \xrightarrow{3} \bullet \xleftarrow{\epsilon} 2,$
$res \xrightarrow{(2+4)} \bullet \xleftarrow{\epsilon} 3\}$

`join :: Leftist a -> Leftist a -> Leftist a`

$\{res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 1, res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{(2+4)^*3} 1, res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 3,$
$res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{(2+4)^*3} 3\}$

`minPQueue :: Leftist a -> a`

$\{res \xrightarrow{\epsilon} \bullet \xleftarrow{3} 1\}$

`delMinPQueue :: Leftist a -> Leftist a`

$\{res \xrightarrow{(2+4)^*} \bullet \xleftarrow{(2+4)^*} 1, res \xrightarrow{(2+4)^*3} \bullet \xleftarrow{(2+4)^*3} 1\}$

`qsort`

`append :: [a] -> [a] -> [a]`

$\{res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} 1, res \xrightarrow{2^*} \bullet \xleftarrow{\epsilon} 2\}$

`partition :: a -> [a] -> ([a],[a])`

$\{res \xrightarrow{(1+2)2^*1} \bullet \xleftarrow{2^*1} 2\}$

`qsort :: [a] -> [a]`

$\{res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} 1\}$

`compiler`

`constantFold :: Stm a -> Stm a`

$\{res \xrightarrow{S^*S_EE^*O} \bullet \xleftarrow{S^*S_EE^*O} 1, res \xrightarrow{S^*S_EE^*D_E} \bullet \xleftarrow{S^*S_EE^*D_E} 1, res \xrightarrow{S^*D_S} \bullet \xleftarrow{S^*D_S} 1\}$

`typeCheck :: Env -> Stm a -> (Env, Stm (Maybe Type))`

$\{res \xrightarrow{2S^*(D_S+S_EE^*D_E)V} \bullet \xleftarrow{2S^*(D_S+S_EE^*D_E)V} res, res \xrightarrow{2S^*(D_S+S_EE^*D_E)V} \bullet \xleftarrow{V} 1,$
$res \xrightarrow{2S^*S_EE^*O} \bullet \xleftarrow{2S^*S_EE^*O} 2\}$

`translate :: Stm a -> ([PInst Label], Table Label Int)`

No sharing detected

`patch :: Table Label Int -> [PInst Label] -> [PInst Int]`

No sharing detected

Figure 15: Analysis results of some of the case studies.

```
data QTree = White | Black | Node QTree QTree QTree QTree
```

This representation provides a space efficient way to store bitmap images. The `rotate`, and `flipH` functions are used to perform transformations in the input image that construct the result from scratch, so no sharing is reported in the result of the analysis. The `overlay` function produces a result that might share with the images being overlaid. With respect to our last example `priorityQueues`, the data type `Leftist a` is defined in a similar way as `AVLTree a` but, in this case, the constructor function `cons` builds a height-biased leftist tree which is not balanced, in general.

The `compiler` module implements a small compiler that translates an imperative language into a sequence of P-machine instructions. The source and target languages are given by the **data** definitions in Figure 16. Notice that the elements of the source language abstract syntax tree (AST) are decorated with an additional parameter (of type `a`) that may hold additional information, such as types. The resulting machine instructions are parametric on the type of the jump addresses: firstly these are symbolic labels and then a patching phase translates them into integer program locations. A detailed description of the compiler is beyond the scope of this paper, so we give a brief description of the results of the main functions performing the four phases of the compiler (constant propagation, type checking, translation, and patching). These are shown in Figure 15, in which we use the following abbreviations:

$$
\begin{array}{lcl}
E & = & 2_{\texttt{AppBinArithOp}} + 3_{\texttt{AppBinArithOp}} + 2_{\texttt{AppUnArithOp}} + 2_{\texttt{AppRelOp}} + 3_{\texttt{AppRelOp}} + \\
 & & 2_{\texttt{AppBinBoolOp}} + 3_{\texttt{AppBinBoolOp}} + 2_{\texttt{AppUnBoolOp}} \\
O & = & 1_{\texttt{AppBinArithOp}} + 1_{\texttt{AppUnArithOp}} + 1_{\texttt{AppRelOp}} + 1_{\texttt{AppBinBoolOp}} + 1_{\texttt{AppUnBoolOp}} \\
D_E & = & 4_{\texttt{AppBinArithOp}} + 4_{\texttt{AppUnArithOp}} + 4_{\texttt{AppRelOp}} + 4_{\texttt{AppBinBoolOp}} + 4_{\texttt{AppUnBoolOp}} + \\
 & & 2_{\texttt{Const}} + 2_{\texttt{Var}} \\
S & = & 1_{\texttt{Seq}} + 2_{\texttt{Seq}} + 2_{\texttt{If}} + 3_{\texttt{If}} + 2_{\texttt{While}} \\
S_E & = & 2_{\texttt{Assign}} + 1_{\texttt{If}} + 1_{\texttt{While}} \\
D_S & = & 1_{\texttt{Skip}} + 3_{\texttt{Assign}} + 3_{\texttt{Seq}} + 4_{\texttt{If}} + 3_{\texttt{While}}
\end{array}
$$

Besides this, we use V to denote the path leading to the types in the typing environments of type `Env`. The results show that, after the constant folding phase, the result may share the operators and the decorations of the input AST. These sharing relations may actually occur at runtime, as the `constantFold` function reconstructs the structure of the program, but

Source language:

```
data Exp a = Const Int a  | Var Int  a           -- constants and variables
    | AppBinArithOp BinArithOp (Exp a) (Exp a) a -- binary arithmetic ops
    | AppUnArithOp  UnArithOp (Exp a) a          -- unary arithmetic ops
    | AppRelOp      RelOp (Exp a) (Exp a) a      -- relational ops
    | AppBinBoolOp  BinBoolOp (Exp a) (Exp a) a  -- binary logic ops
    | AppUnBoolOp   UnBoolOp (Exp a) a           -- unary logic ops

data Stm a = Skip a
    | Assign Int (Exp a) a                       -- variable assignment
    | Seq (Stm a) (Stm a) a                      -- sequence
    | If (Exp a) (Stm a) (Stm a) a               -- conditional
    | While (Exp a) (Stm a) a                    -- loops
```

Target language:

```
data PInst a = Push Int | Load | Store | Jmp a | Jfalse a | Add | Sub | ...
```

Figure 16: Source and target languages of the compiler example.

it reuses the operators and the decorations of the AST given as input. Regarding the type checking phase, the *typeCheck* function also rebuilds the input program in order to include the type decorations. However, there is a type environment that stores the type of each variable in scope. Whenever a variable is found during the AST traversal, the *typeCheck* function updates the decoration of this variable with a reference to a type stored in the environment. As a consequence of this, there may be sharing between the decorations and the type environment given as input, and also between the decoration themselves (e.g. when the same variable occurs twice in a program, both occurrences share the same type decoration). That is why the analysis reports internal sharing between the decorations of the result, and sharing between these decorations and the input environment.

The results of Figures 14 and 15 show that, given the current choice of the abstract domain (i.e. sets of sharing relations), accuracy is hardly lost due to the application of the abstract interpretation function. However, we could achieve better results by considering other abstract domains. For instance, when analysing the `merge` function, the analysis returns, among others, the relations $res \xrightarrow{2^*} \bullet \xleftarrow{2^*} 1$ and $res \xrightarrow{2^*} \bullet \xleftarrow{2^*} 2$, meaning that the result may share with the tails of the input lists. However, only one of these tails is actually shared at runtime, so we can refine our abstract domain by allowing *disjunctions* of sharing relations. Whilst this would result in a more precise analysis, it would come at the cost of efficiency. Another example of innaccuracy is the `append` function, which reports the sharing relation $res \xrightarrow{2^*1} \bullet \xleftarrow{2^*1} 1$, meaning that the first elements of the result are the same that those of the list given as first parameter. This can be improved by considering relations such as $res \xrightarrow{2^n1} \bullet \xleftarrow{2^n1} 1$ (where n ranges over natural numbers) implying that the elements of the result are also in the same order as the elements of the input list. It is, however, unclear how the widening strategy should be modified in order to include these constraints.

From Figure 13 it follows that the use of regular expressions instead of NFAs lead to better execution times, even if we consider a specific highly-optimized automata library such as NFA C. Moreover, the functions manipulating regular expressions are simpler than their counterparts in NFAs and lead to more readable results.

7. Related Work and Conclusions

There exist many different analyses dedicated to extracting information about the heap, mainly in imperative languages where pointers are explicitly used and may be reassigned. *Alias analysis* is one of the most studied. It tries to detect program variables that point to the same memory location. *Pointer analysis* aims at determining the storage locations a pointer can point to, so it may be also used to detect aliases in a program. These analyses are used in many different applications such as live variable analysis for register allocation and constant propagation. In [15, 16, 17] we can find surveys about pointer analysis applied to imperative languages from the 80's. Related to these analyses, an *escape analysis* tries to determine statically the dynamic scope of the data structures that will be created at runtime, whereas *shape analysis* [18, 19, 20] tries to approximate the 'shape' of the heap-allocated structures. That information has been used, for example, for binding time optimisations.

The level of detail of the information about the heap that these analyses provide mainly depends on the needs of the "user" of the analysis. Our analysis tries to capture a kind of sharing information more refined than alias and pointer analysis may provide, and in fact both are subsumed in our relations: if $x \xrightarrow{\epsilon} \bullet \xleftarrow{\epsilon} y$, then, x and y are aliases; if $x \xrightarrow{j} \bullet \xleftarrow{\epsilon} y$, then x points to y (i.e. y is the j-th child of the data structure x). In the area of escape analysis, Blanchet [21] applies the concept of paths in order to determine which pointers in a data structure survive the current execution scope. The sets of paths are subsequently abstracted by integer numbers denoting escape contexts, whereas in this work we use regular expressions for abstracting those sets. Moreover, our analysis aims to infer sharing relations between our structures. That is why shape analysis is nearer to our needs.

Jones and Muchnick [19] associate sets of k-limited graphs to each program point in order to approximate the sharing relations between variables. The k limits the length of the paths in the graphs modelling the heap in order to make the domain finite and obtain the minimal fixpoint by iteration. The graphs obtained after the abstract execution of a program instruction must be transformed in order to maintain themselves k-limited. Our widening operator resembles this operation. Our path relations are in general uncomparable in precision to these sets of limited graphs. First, having sets of graphs may provide more precision because our union operation loses information: adding $x \xrightarrow{p_1+p_3} \bullet \xleftarrow{p_2+p_4} y$ introduces combinations of paths $x \xrightarrow{p_1} \bullet \xleftarrow{p_4} y$ and $x \xrightarrow{p_3} \bullet \xleftarrow{p_2} y$ which did not exist previously. Second, paths longer than k may be more precise that k-limited graphs: $x \xrightarrow{2221} \bullet \xleftarrow{\epsilon} y$ indicating that y is the fifth element of the list x is more precise than saying in a 2-limited graph that y shares in an *unknown* way with x after the path 22. Additionally, the cost of having sets of graphs is doubly exponential in the number of variables.

In order to reduce the cost to polynomial, Reps [20] formulated the analysis as a graph-reachability problem over the dependence graph generated from the program. The reachability is defined in terms of those (context-free) paths one is interested in. The fixpoint calculation in this case is also finite because he just records the information about the variables, not the exact paths. We need the paths in order to make the analysis more precise as shown in the *mergesort* example, that is why we need the widening. The use of context-free paths in our framework would make undecidable most of our tests.

The logic programming field has produced plenty of analyses in which sharing plays an important role. A pioneering one is [22], whose aim is to approximate the set of terms a logic variable may be bound to at runtime. To this purpose, it introduces the so-called type graphs, whose power is similar to that of context free languages. The abstract domain is made finite by introducing a *depth restriction*, limiting the number of times a functor may occur in any graph path. Sharing information is kept in abstract substitutions by an environment mapping variables to specific nodes of the type graphs. This corresponds to our internal sharing. The nodes are selected by selector paths very similar to our paths. The sharing property is of a *must* type, meaning that *all* the concrete substitutions represented by the abstract one must share these sub-terms. Ours is a *may* sharing, meaning that the actual property we are seeking for is the absence of sharing.

In [23], the above analysis is made practical for bigger programs. Instead of the depth restriction, they use a widening operator in an infinite domain of type graphs. To prevent that the graphs keep growing while computing the fixpoint, a cycle is introduced in the graph type. This resembles our widening operator.

Other related works are those devoted to compile-time garbage collection, such as [24] whose aim is to detect dead cells in Prolog programs, and to produce code to reuse them. This analysis reduces the number of runtime garbage collections. In this case, the sharing property sought for is, as in our case, the absence of sharing. The abstract domain consists, as in [22], of depth-restricted type graphs, and the sharing information is also kept as a mapping from variables to specific nodes in the type graphs. In a sense, this representation is dual to ours: their type graphs abstract away infinite sets of terms, while the sharing decoration points to precise nodes of this abstract structure. By contrast, we do not explicitly represent the terms, and sharing is expressed as abstract paths, usually denoting an infinite number of them, through the concrete terms. All the examples of sharing shown in [24] could also be expressed with our formalism but, in general, it is difficult to assess which representation gives a more precise sharing information.

In [25] these ideas are applied to Mercury, a typed and annotated logic language. In this framework, the compiler has more information about types, instantiation modes, and deterministic behaviour, so a more precise sharing information can be obtained than in Prolog. Also selector paths are used here in order to denote specific nodes inside a term. What they call an alias (a pair of paths), corresponds to what we call sharing in this paper. They do not try to manipulate these paths as we do. When the number of detected pairs grows too much, the compiler triggers a widening operator. Its effect is to collapse many pairs in a more compact and less precise information.

A final set of works [26, 27] try to detect at compile time which cells are dead at a program point, but in this case the cells live in disjoint heap regions, and all the cells in a region are deallocated at the same time. The abstract interpretation domain are points-to graphs, a sort of type graphs enriched with more information. Nodes in the graph represent local variables, and edges represent the subterm relation. An edge is just a selector, i.e. a pair ⟨functor name, argument position⟩. Additionally, a node keeps a set of variables that will be allocated in the same region, and may share a subterm with the node variable. So, while the internal sharing recorded in the graph seems to be precise enough, this is not the case with the sharing between variables living in a region.

In the functional field, there are also some works devoted to compile-time garbage collection such as [28, 29]. The first one tries to save creating a new array when updating an array that is only referenced once. The second one provides an analysis also detecting when a cell is referenced at most once by the subsequent computation. Its aim is to destroy the cell after its last use so that it can be reused by the runtime system. Both analyses are done on a first-order eager functional language. After these ones, there have been many similar analyses, usually known as *usage analyses* (e.g. [30, 31, 32, 33]) whose aim is to detect when a cell is used at most once and then, either to recover or to avoid to update it, when the language is lazy. These analyses do not try to know *which* other data structures points to a particular cell, but rather *how many* of them do it, and in this sense they are simpler. The problem closer to ours is treated in [29] since it pursues an aim similar to that of *Safe*: to save memory. The main difference is that, in our case, it is the programmer who decides to destroy a cell and the compiler just analyses whether doing this is safe or not. So, the programmer may have destructive and non-destructive versions of the same function and uses the first one in contexts where it is safe to do it. In [29] it is the compiler who decides to destroy the cell, when it is safe to do it in *all* the contexts in which the function is called. A single unsafe context will avoid to recover the cell in all the safe ones. Another important difference is that our analysis is modular, while theirs needs to analyse the program as a whole. This makes it impractical for big programs.

Our analysis is done at the *Core-Safe* level, our internal representation of source programs. That means that small changes in the source may produce big changes in its *Core-Safe* representation, and hence big changes in its sharing properties. This implies that programs rejected by our type system due to excessive sharing, could be admitted after a slight change in the source, and the other way around. We have not done enough experimentation in order to learn which changes are beneficial and which ones are not. This issue remains an open question for future work.

The extension of this analysis to higher-order functions maintaining modularity can be done by applying the standard techniques shown in [34, 35], in which the signature of a higher-order function is an abstract function (also called suspension) whose argument is a function signature which is unknown until function application occurs. This means that, if the functional argument is applied inside the function body, the substitution and closure process cannot happen until the higher-order function is applied to a concrete function, i.e. it is symbolically kept in the abstract function. This is not a problem if the analysis is implemented in a lazy language such as Haskell. However, in order to extend the full development of our language Safe, we would have to extend some other, probably more involved, features such as region inference and safe types inference.

Acknowledgements: We thank the anonymous reviewers for the numerous comments made to a previous version of this paper. In particular, we thank them for making us aware of the huge amount of work about sharing done in the logic programming field.

References

[1] M. Montenegro, R. Peña, C. Segura, A Type System for Safe Memory Management and its Proof of Correctness, in: ACM Principles and Practice of Declarative Programming, PPDP'08, Valencia, Spain, July. 2008, 2008, pp. 152–162.

[2] M. Montenegro, Safety properties and memory bound analysis in a functional language without a garbage collector, Ph.D thesis, Univ. Complutense of Madrid, Computer Science Faculty, Spain, 2011.

[3] M. Montenegro, R. Peña, C. Segura, An Inference Algorithm for Guaranteeing Safe Destruction, in: Selected papers of *Logic-Based Program Synthesis and Transformation, LOPSTR'08*, LNCS 5438, Springer., 2009, pp. 135–151.

[4] R. Peña, C. Segura, M. Montenegro, A Sharing Analysis for SAFE, in: Selected Papers Trends in Functional Programming, TFP'06, Intellect, 2007, pp. 109–128.

[5] M. Montenegro, R. Peña, C. Segura, Shape analysis in a functional language by using regular languages, in: 15th International Symposium on Principles and Practice of Declarative Programming, PPDP 2013, ACM Press, 2013, pp. 251–262.

[6] M. Montenegro, R. Peña, C. Segura, Shape Analysis in a Functional Language by Using Regular Languages (Extended Version), Technical Report, TR-8-13. Dpto. de Sistemas Informáticos y Computación. Universidad Complutense de Madrid, 2013. Available at: `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/informes-tecnicos`.

[7] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, 2nd ed., Addison Wesley, 2001.

[8] J. A. Brzozowski, Derivatives of regular expressions, Journal of the ACM 11 (1964) 481–494.

[9] S. Owens, J. Reppy, A. Turon, Regular-expression derivatives reexamined, Journal of Functional Programming 19 (2009) 173–190.

[10] A. Krauss, T. Nipkow, Proof pearl: Regular expression equivalence and relation algebra, Journal of Automated Reasoning 49 (2012) 95–106.

[11] J. Rutten, Automata and coinduction (an exercise in coalgebra), in: D. Sangiorgi, R. Simone (Eds.), CONCUR'98 Concurrency Theory, volume 1466 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1998, pp. 194–218.

[12] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points, in: Proc. 4th ACM Symp. on Principles of Prog. Languages, ACM, 1977, pp. 238–252.

[13] S. L. Peyton Jones, J. Hughes (Eds.), Report on the Programming Language Haskell 98, URL http://www.haskell.org, 1999.

[14] J. Saraiva, HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages, in: Proc. ACM Workshop on Functional and Declarative Programming in Education, University of Kiel. Tech. Report 0210, 2002, pp. 133–140.

[15] V. Raman, Pointer analysis – a survey, CS203 UC Santa Cruz, `http://www.soe.ucsc.edu/~vishwa/publications/Pointers.pdf`, 2004.

[16] M. Hind, Pointer analysis: Haven't we solved this problem yet?, in: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, ACM Press, 2001, pp. 54–61.

[17] D. Rayside, Points–to analysis, `http://www.cs.utexas.edu/~pingali/CS395T/2012sp/lectures/points-to.pdf`, 2005.

[18] J. C. Reynolds, Automatic computation of data set definitions, in: IFIP Congress (1), 1968, pp. 456–461.

[19] N. D. Jones, S. S. Muchnick, Flow analysis and optimization of lisp-like structures, in: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '79, ACM, 1979, pp. 244–256.

[20] T. Reps, Shape analysis as a generalized path problem, in: Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM '95, ACM, 1995, pp. 1–11.

[21] B. Blanchet, Escape analysis for Java^{TM}: Theory and practice, ACM Transactions on Programming Languages and Systems 25 (2003) 713–775.

[22] G. Janssens, M. Bruynooghe, Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation, J. Log. Program. 13 (1992) 205–258.

[23] P. V. Hentenryck, A. Cortesi, B. L. Charlier, Type Analysis of Prolog Using Type Graphs, J. Log. Program. 22 (1995) 179–209.

[24] A. Mulkers, W. H. Winsborough, M. Bruynooghe, Live-Structure Dataflow Analysis for Prolog, ACM Trans. Program. Lang. Syst. 16 (1994) 205–258.

[25] N. Mazur, P. Ross, G. Janssens, M. Bruynooghe, Practical Aspects for a Working Compile Time Garbage Collection System for Mercury, in:

P. Codognet (Ed.), ICLP, volume 2237 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 105–119.

[26] Q. Phan, G. Janssens, Towards Region-Based Memory Management for Mercury Programs, in: S. Etalle, M. Truszczynski (Eds.), ICLP, volume 4079 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 433–435.

[27] Q. Phan, G. Janssens, Z. Somogyi, Region-based memory management for Mercury programs, TPLP 13 (2013) 959–1024.

[28] P. Hudak, A Semantic Model of Reference Counting and its Abstraction (Detailed Summary), in: ACM Symposium on Lisp and Functional Programming, ACM, 1986, pp. 351–363.

[29] T. P. Jensen, T. A. Mogensen, A Backwards Analysis for Compile-Time Garbage Collection, in: European Symposium on Programming, LNCS 432, Springer, 1990, pp. 227–239.

[30] D. N. Turner, P. L. Wadler, C. Mossin, Once upon a type, in: 7'th International Conference on Functional Programming and Computer Architecture, ACM Press, La Jolla, California, 1995, pp. 1–11.

[31] E. Barendsen, S. Smetsers, Uniqueness typing for functional languages with graph rewriting semantics, Mathematical Structures in Computer Science 6 (1996) 579–612.

[32] K. Wansbrough, S. L. P. Jones, Once upon a polymorphic type, in: The Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, 1999.

[33] J. Gustavsson, J. Sveningsson, A Usage Analysis with Bounded Usage Polymorphism and Subtyping, in: Selected Papers of the 12th International Workshop on Implementation of Functional Languages, IFL'00, volume 2011 of *LNCS*, Springer-Verlag, 2001, pp. 140–157.

[34] G. L. Burn, C. L. Hankin, S. Abramsky, The Theory of Strictness Analysis for Higher Order Functions, in: H. Ganzinger, N. D. Jones (Eds.), Programs as Data Objects, volume 217 of *LNCS*, Springer-Verlag, 1986, pp. 42–62.

[35] G. L. Burn, The abstract interpretation of higher-order functional languages: From properties to abstract domains, in: R. Heldal, C. H. Kehler, P. Wadler (Eds.), Glasgow functional programming workshop, 91, pp. 56–72. URL: `http://theory.doc.ic.ac.uk/tfm/papers/BurnGL/Glasgow91.ps.gz`.