Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

Markus Mohnen
Lehrstuhl für Informatik II, Aachen University of Technology
Ahornstraße 55, 52056 Aachen, Germany
email: markusm@zeus.informatik.rwth-aachen.de

May 9, 1995

Abstract

This paper describes a *compile-time garbage collection* (ctgc) method in the setting of a first-order functional language with data structures. The aim is to obtain information on positions in a program where certain heap cells will become obsolete during execution. Therefore we develop an abstract interpretation for the detection of *inheritance information* which allows us to detect whether the heap cells of an argument will be propagated to the result of a function call. The abstract interpretation itself is independent of the evaluation strategy of the underlying language. However, since the actual deallocations take place after on termination of functions, the information obtained by the abstract interpretation can be only be applied in an eager context, which may be detected using strictness analysis in a lazy language. In order to increase efficiency we show how the number of recomputations can be decreased by using only parts of the abstract domains. The worst case time complexity is essentially quadratic in the size of the program. We illustrate the method developed in this paper with several examples and we demonstrate how to use the results in an eager implementation. Correctness of the analysis is considered, using a modified denotational semantics as reference point. A main goal of our work is to keep both the run-time and the compile-time overhead as small as possible.

1 Introduction

One of the advantages of modern functional languages is the ability to work with dynamic data structures without the necessity to control memory allocations explicitly. On the other hand, this prevents a programmer from expressing the reusability of heap cells of intermediate data structures. However, since the available memory is restricted in concrete computers, there must be some means of deallocating obsolete memory cells. But well-known garbage collection mechanisms make no (or only little) use of the special structure of the underlying functional programs.

We present an *abstract interpretation* which exploits this special structure. The underlying observation is that data cells which were created specially for a certain function call become obsolete, if they are not inherited to the *function result*, i.e. if they are not reachable from the top cell of the result. In contrast to other ctgc approaches (like for instance [JM89]) we do not try to reuse obsolete cells as soon as possible, i.e. during function execution, because this would cause severe changes in the abstract machine and some runtime overhead. However, application of our technique is therefore only possible in 'eager' situations, i.e. either in an eager language or at positions in lazy programs which were detected as eager by a strictness analysis.

Furthermore, we show that our method can be implemented *very efficiently* by using test arguments instead of the full domains. The worst case complexity for general programs is essentially $O(n^2)$ in the length of the program. Especially for realistic program we will show that the computation of the fixpoint can be done almost in *linear time*.

We finish this introduction by an outline of the contents of the paper. The next section gives a more detailed intuitive description of our method. Section 3 formalises this intuition and emphasises the problem of arbitrary data structures and their finite representation. After that, we give a brief discussion of the correctness of our method and explain how to use the results of the abstract interpretation in an eager implementation. The next section contains a discussion of efficiency issues. We show how to decrease the time spent for computing the fixpoint and give experimental results. The paper concludes with a comparison with related work and some prospects for further research.

2 Intuitive Description

We want to address the problem in the setting of a first-order functional language with constructors. For the moment, we will assume eager evaluation, although the abstract interpretation itself is also applicable to programs of a lazy language. A constructor on the right hand side on a function definition causes the allocation of a corresponding heap cell, if it is encountered during the execution. We assume *boxed* representation of basic types, i.e. the heap cell representing a constructor contains just pointers to other heap cells, which may represent constructors or basic values. The loosening of some of these restrictions will be discussed in Section 8.

The general idea of the optimisation can be summarised in the following way: if there is a subterm $f(g(t))$ on the right hand side of a function definition and we know that f does not inherit parts of the result of the evaluation of $g(t)$, then deallocate all cells belonging to these parts which were allocated during the execution of $g(t)$. The deallocation takes place when the call of f terminates. In an eager implementation this can easily be done, since the termination point of a function is fixed.

In order to illustrate which kind of information we want to derive, we consider the following example:

```
datatype ListOfInt ::= Nil | Cons(int,ListOfInt)
append : ListOfInt x ListOfInt -> ListOfInt
append(Nil, L) := L
append(Cons(a,L1), L2) := Cons(a,append(L1,L2))
```

An obvious observation is that `append` sets up copies of all `Cons` cells contained in the first argument, but not of the `int` values stored in there. We say that `append` *inherits* the second level of its first argument and all levels of its second argument, but not the first level of its first argument. The term *level* corresponds to the definition of the underlying type. For instance, the type

```
datatype ListOfList ::= LNil | LCons(ListOfInt, ListOfList)
```

has three levels: one for the list built up from LNil and LCons, one for the ListOfInt constructors and one for for the actual int entries.

The basic idea of the abstract interpretation, is to represent the (*infinite*) set with elements of *variable* size which corresponds to a type by means of a *finite* domain of elements with a fixed size, where each element has one component for each level of the type. This component is a binary value, where the value 1 is used to indicate that the corresponding level may contain a cell which is part of one of the parameters of the function call. The value 0 guarantees that there is no such cell, i.e. all cells are new.

In the above example we choose $\mathfrak{A}_{\texttt{ListOfInt}} = \{0, 1\}^2$ as abstract domain for lists over integers. The components represent all constructors resp. all entries in a list.

The corresponding abstract functions are essentially built from the abstract constructors. The formal parameters of a constructor can be divided into two classes: those which are of the same type as the constructor and those which are not. E.g. the constructor Cons for lists has the second argument in the first class and the first argument in the second. The abstract interpretation of a constructor is performed in two stages:

1. All parameters of class two are put (at appropriate positions) into a new value. All other positions in this value are set to 0. These are the binary value indicating the inheritance of the constructors and the positions for parameters of class two of other constructors for this sort. This step copes with the creation of a new heap cell.

2. The maximum (wrt. the order within the abstract domain) of this value and all parameters of class one is returned as result. In this way the information already contained is those parameters in preserved.

If one of those parameters contains a nonzero entry at the position for the constructors, then the result will also. This captures our intuition, that if at least one cell of a list is inherited, we approximate that all cells are inherited. In our example, the constructors are abstractly interpreted by:

$$\begin{array}{ll} \mathrm{A}[\![\texttt{Nil}]\!] : \mathfrak{A}_{\texttt{ListOfInt}} & \mathrm{A}[\![\mathsf{Cons}]\!] : \mathfrak{A}_{\texttt{int}} \times \mathfrak{A}_{\texttt{ListOfInt}} \rightarrow \mathfrak{A}_{\texttt{ListOfInt}} \\ \mathrm{A}[\![\texttt{Nil}]\!] = (0,0) & \mathrm{A}[\![\mathsf{Cons}]\!](a,l) = (0,a) \sqcup l \end{array}$$

In Section 6 we will show that we do not need to determine the values of the abstract functions for all possible argument combinations. Instead, we will use a single *test argument* for each level of each argument to determine whether this particular level has an impact on the result, i.e. if there are elements from this level, which are part of the result. The test arguments are those with exactly one nonzero entry at a position representing this level. The result can only contain nonzero entries if a nonzero entry in the initial test argument is propagated. For example, the abstract values of append for all possible test arguments are:

$$\begin{array}{ll} \mathrm{A}[\![\mathsf{append}]\!]((0,0),(0,1)) = (0,1) & \mathrm{A}[\![\mathsf{append}]\!]((0,0),(1,0)) = (1,0) \\ \mathrm{A}[\![\mathsf{append}]\!]((0,1),(0,0)) = (0,1) & \mathrm{A}[\![\mathsf{append}]\!]((1,0),(0,0)) = \underline{(0,0)} \end{array}$$

We can see that the abstract value of append for the test argument $((1,0),(0,0))$ is $(0,0)$. This means that no Cons or Nil of the first level of the first argument is inherited to the result of append

In order to demonstrate the intended use, we expand our example from above by definitions for the functions `filterle`, `filtergt` and `quicksort`:

```
filterle : ListOfInt x int -> ListOfInt
filterle(Nil,b) := Nil
filterle(Cons(a,L),b) := if a<=b then Cons(a,filterle(L,b))
                                 else filterle(L,b)
filtergt (* analogous *)
quicksort : ListOfInt -> ListOfInt
quicksort(Nil) := Nil
quicksort(Cons(a,L)) := append(quicksort(filterle(L,a)),
                               Cons(a,quicksort(filterge(L,a))))
```

We can infer that all these functions do not inherit the constructors of their `ListOfInt`–typed argument. During the execution of a call to `quicksort` with a parameter not equal to `Nil`, assuming left-to-right eager evaluation, the expression `filterle(L,a)` is evaluated first, creating an intermediate data structure which is fed into a recursive call of `quicksort`. All constructors of the intermediate list are copied by `quicksort`, which means that we can deallocate everything which was allocated for the intermediate list *after* the termination of `quicksort`. Accordingly, every intermediate result can be deallocated after the surrounding function call has terminated, except for the outermost `append` (this will maybe be deallocated on the level of the calling function) and the second `quicksort` (because `append` does not copy its second argument).

3 Abstract Interpretation

In this section we will formalise what we have described informally in the previous section. First, the abstract syntax of our language will be presented. In order to simulate pattern matching, a set of selector and test functions will be be associated with each constructor. Thereafter, we will discuss how to choose the abstract domains, where we focus on the problems arising due to arbitrary data structures. The abstract interpretation is basically determined by the way the constructors and selectors are interpreted w.r.t. the abstract domains.

3.1 Abstract Syntax

Let $S = BS \dot\cup CS$ be the set of *sorts*, where BS and CS are finite, disjoint sets of *(basic) sorts*, containing at least a sort $\mathsf{bool} \in BS$, and *constructed sorts*. Correspondingly, we assume that there are disjoint families of *variables* $X = (X^s \mid\mid s \in S)$, *defined functions* $DF = (DF^{s_1,\ldots,s_n \to s} \mid\mid n \in \mathbb{N}_0,\ s_1, \ldots, s_n, s \in S)$, *basic functions* $\Omega = (\Omega^{bs_1,\ldots,bs_n \to bs} \mid\mid n \in \mathbb{N}_0,\ bs_1, \ldots, bs_n,\ bs \in BS)$ and *constructors* $C = (C^{s_1,\ldots,s_n \to cs} \mid\mid n \in \mathbb{N}_0,\ s_1, \ldots, s_n \in S$ and $cs \in CS)$. From the constructors we can derive

1. the family of *selectors*

$$CSel := (CSel^{cs \to s} \mid\mid cs \in CS \text{ and } s \in S)$$

   where the set of selectors of type $cs \to s$ is defined by

$$CSel^{cs \to s} := \{c^j_{\mathtt{sel}} \mid\mid \exists c \in C^{s_1,\ldots,s_n \to cs},\ 1 \leq j \leq n\colon\ s_j = s\}$$

2. the family of *constructor tests*

$$CTest := (CTest^{cs \to \mathtt{bool}} \mid\mid cs \in CS)$$

where the set of constructor tests of type $cs \to \mathtt{bool}$ is defined by

$$CTest^{cs \to \mathtt{bool}} := \{\mathtt{is}\text{−}c \mid\mid \exists c \in C^{s_1,\ldots,s_n \to cs}\}$$

We use these auxiliary functions to simulate pattern matching. We define the family of all *expressions* over X, DF, Ω, C, $CSel$ and $CTest$ by $E := (E^s \mid\mid s \in S)$, where the sets E^s are defined by:

1. $X^s \subseteq E^s$
2. $f \in (\Omega \cup C \cup CSel \cup CTest)^{s_1,\ldots,s_n \to s}$ and $e_1 \in E^{s_1}, \ldots, e_n \in E^{s_n} \Longrightarrow f(e_1, \ldots, e_n) \in E^s$
3. $F \in DF^{s_1,\ldots,s_n \to s}$ and $e_1 \in E^{s_1}, \ldots, e_n \in E^{s_n} \Longrightarrow F(e_1, \ldots, e_n) \in E^s$
4. $e \in E^{\mathtt{bool}}, e_1, e_2 \in E^s \Longrightarrow \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \mathtt{fi} \in E^s$

A *program* is a finite set of *definitions*

$$F(x_1, \ldots, x_n) := e \quad \text{with } F \in DF^{t_1,\ldots,t_n \to t},\ x_i \in X^{t_i} \text{ and } e \in E^t \text{ with variables } \{x_1, \ldots, x_n\}$$

one for each defined function.

The intended declarative semantics is the usual fixpoint semantics. The data representation we have in mind is a graph reduction machine where the graph is represented by means of a heap. If a constructor is evaluated during the execution, a heap cell representing this constructor is created on the heap. Arguments are already represented in the heap, and so references to the arguments are simply copied into the cell. Note, that we assume a *boxed* representation of basic values. Accordingly, variables always refer to heap cells.

3.2 Abstract Domains

Assume that we have sets V_{bs} for each basic sort $bs \in BS$. Denotational semantics uses sets V_{cs}, which are fixpoints of the equations:

$$V_{cs} = \bigcup_{c \in C^{s_1,\ldots,s_n \to cs}} \{c(v_1, \ldots v_n) \mid\mid v_1 \in V_{s_1}, \ldots, v_n \in V_{s_n}\} \qquad \forall cs \in CS$$

This reflects the free interpretation of constructors. In the strict case, these sets consists of the finite terms. Infinite terms are added to obtain the sets for the non-strict semantics. But, of course, as we want to do an abstract interpretation we have to guarantee termination, so we need at least domains with only finite ascending chains instead. Furthermore, since abstract values with variable size can only be implemented inefficiently, we prefer finite domains, where values can be represented with a fixed size. The first (obvious) step is to replace V_{bs} by $I_{bs} := \{0, 1\}$, since our interest is focused on whether parts of the arguments are inherited to the result, and not on the actual value of the result. Later on, we will use test values with a single '1' for a particular level of an argument. If we can observe that this '1' propagates through the computation, we have detected (possible) inheritance w.r.t. this level. Additionally, we add a value to each constructor, indicating its inheritance behaviour. This yields sets I_{cs} in a similar same way as fixpoints of equations:

$$I_{cs} = \bigcup_{c \in C^{s_1,\ldots,s_n \to cs}} \{c(b; v_1, \ldots, v_n) \mid\mid b \in \{0,1\}, v_1 \in I_{s_1}, \ldots, v_n \in I_{s_n}\} \forall cs \in CS$$

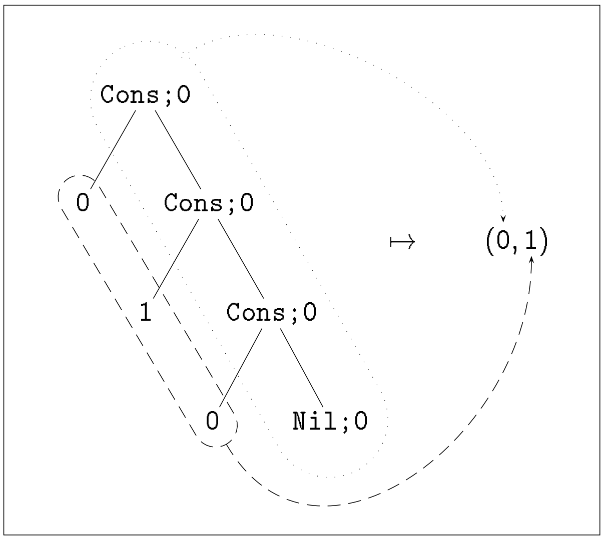

Figure 1: Mapping $I_{\texttt{ListOfInt}}$ to $A_{\texttt{ListOfInt}}$

We still have the problem of infinity, since we have essentially only reduced, for instance, the set of lists over integers to the set of lists over $\{0,1\}$. One possibility for obtaining finite sets, we call it the *horizontal* approach, is to build the elements of the abstract domain by gratuitously restricting the *height* of the elements of I_s, for instance only the list of length 7 or less. This approach is used in [JM89]. The obvious drawback is the selection of the threshold, since there is no possibility to derive a good choice from the program itself.

We propose a different approach, which has a more *vertical* appearance. If I_s has n levels, we define $A_s \simeq \{0,1\}^n$ and we map an element of I_s to an element of A_s by taking the maximum of all entries of level i as entry for the i-th component (see Figure 1). There is an obvious simplification of the domains, which could be applied here. Instead of $\{0,1\}^n$ we could choose $\{0,\ldots,n\}$, with a value k representing the fact that levels k and above are not inherited. This approach is taken in [Hug92] and [PG92]. The underlying observation is that it is not possible to define a functional program which inherits a certain constructor c and simultaneously does not inherit a constructor d below c. This is due to the fact that functional programs can not change parameters stored in existing heap cells for constructors. But there is a drawback: this approach works for list structures, but *not* for the general case. We will come back to this point later.

Our first attempt to define the abstract sets starts with setting $A_{bs} := \{0,1\}$ for all basic sorts

$bs \in BS$. We then can define the sets A_{cs} for the constructed sorts as the fixpoints of

$$A_{cs} = \{0,1\} \times \prod_{c \in C^{s_1,\ldots,s_n \to cs}} \prod_{\substack{1 \le i \le n \\ s_i \ne cs}} A_{s_i} \qquad \forall cs \in CS$$

In order to evaluate the first Cartesian product $\prod$ we need an order on all constructors of target sort cs. The second product filters out all direct recursiveness since we want to combine all constructors of cs in one level. If we apply this to our example, we can compute that

$$\begin{aligned} A_{\texttt{ListOfInt}} &= \{0,1\} \times \prod_{c \in \{\texttt{Nil}^{\to \texttt{ListOfInt}}, \texttt{Cons}^{\texttt{int} x \texttt{ListOfInt} \to \texttt{ListOfInt}}\}} \prod_{\substack{1 \le i \le n \\ s_i \ne cs}} A_{s_i} \\ &= \{0,1\} \times A_{\texttt{int}} \\ &= \{0,1\}^2 \end{aligned}$$

Note that we assume that a $\prod$ over an empty index creates a set, which is a neutral element for subsequent Cartesian products. Therefore, these parts do not contribute to the resulting domain. E.g. the constructor `Nil` has no influence.

But there is a drawback: it is possible that we have a constructed sort cs, where we get $|A_{cs}| = \infty$. A minimal example for this consists of the data definitions:

```
datatype T1 ::= c1 T2     datatype T2 ::= c2 T1
```

which are valid definitions in Miranda or Haskell, and which can be expressed in our abstract syntax by choosing $C^{T2 \to T1} = \{\texttt{c1}\}$ and $C^{T1 \to T2} = \{\texttt{c2}\}$.The corresponding sets are empty for strict semantics and both have exactly one element of infinite length for non-strict semantics:

$$V_{\texttt{T1}} = \{\texttt{c1}(\texttt{c2}(\texttt{c1}(\cdots)))\} \quad \text{and} \quad V_{\texttt{T2}} = \{\texttt{c2}(\texttt{c1}(\texttt{c2}(\cdots)))\}$$

Accordingly, we get the abstract set equations:

$$A_{\texttt{T1}} = \{0,1\} \times A_{\texttt{T2}} \quad \text{and} \quad A_{\texttt{T2}} = \{0,1\} \times A_{\texttt{T1}}$$

which have the unique solution $A_{\texttt{T1}} = A_{\texttt{T2}} = \{0,1\}^\omega$, the set of all infinite sequences of binary numbers.

Of course, sets of this kind are not useful for abstract interpretation, because in order to guarantee termination of the abstract interpretation we need sets with only finite ascending chains under an appropriate order.

Therefore, we must enhance our notion. The problem of our first attempt is that *direct recursion* and *indirect recursion* are handled in different ways. In the above example, it would be convenient to choose $A_{\texttt{T1}} = A_{\texttt{T2}} = \{0,1\}^1$: one level for all occurrences of `c1` and `c2`.

The indirect recursion, or, to be more precise, the fact that the constructed sorts need not form a proper hierarchy is also the reason for the failure of the simplified approach. It assumes that the abstract levels of the type can be ordered in a single chain. Consider the example

```
datatype ITree ::= ILeaf int  | INode int  CTree CTree
datatype CTree ::= CLeaf char | CNode char ITree ITree
```

which defines types of trees consisting of alternating layers of int resp. char entries. We choose $\mathfrak{A}_{\texttt{ITree}} = \mathfrak{A}_{\texttt{CTree}} = \{0,1\} \times (\{0,1\}^2)^2$, which means that we have three levels. These levels can not be ordered linearly, since the levels for the int and for the char entries can not be ordered, they are kind of incomparable.

An extension of the simplified approach would be to choose a domain, where several incomparable levels are positioned as brothers of a common father. Since all incomparable levels can be handled independently we have to add values for all possible combinations of inheritance of levels. In our case, this would be the domain in Figure 2 where the '2' represents a situation where both char and int entries are inherited to the result but not the constructors. These domains, however,

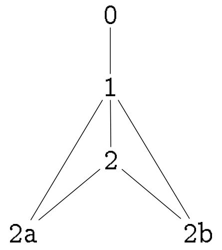

Figure 2: Alternative domain

have lost the simplicity of a linear chain, so there is no advantage anymore. Furthermore it is not possible to use these domains if we consider an extension of the language with non-functional procedures (see related work section).

The next definition will enable us to express indirect recursion of constructed sorts: Let $cs_1, cs_2 \in CS$. We say that cs_1 *depends on* cs_2 ($cs_1 \twoheadleftarrow cs_2$) iff there exists a constructor $c \in C^{s_1,\ldots,s_{i-1},cs_2,s_{i+1},\ldots,s_n \to cs_1}$. As usual, $\stackrel{\star}{\twoheadleftarrow}$ denotes the transitive and reflexive closure of $\twoheadleftarrow$. If we have $cs_1 \stackrel{\star}{\twoheadleftarrow} cs_2$ and $cs_2 \stackrel{\star}{\twoheadleftarrow} cs_1$, then we say that cs_1 and cs_2 are *mutually recursive dependent*. By definition, this is an equivalence relation, and we denote the equivalence class of $cs \in CS$ by

$$[cs] := \{cs' \in CS \mid\mid cs' \stackrel{\star}{\twoheadleftarrow} cs \text{ and } cs \stackrel{\star}{\twoheadleftarrow} cs'\}$$

Additionally, we define $[bs] := \{bs\}$ and $A_{[bs]} := \{0,1\}$ for all basic sorts $bs \in BS$. Again, we can obtain sets $A_{[cs]}$ by the equations:

$$A_{[cs]} = \{0,1\} \times \prod_{cs' \in [cs]} \prod_{c \in C^{s_1,\ldots,s_n \to cs'}} \prod_{\substack{1 \leq i \leq n \\ [s_i] \neq [cs]}} A_{[s_i]} \qquad \forall cs \in CS$$

These are suitable sets for abstract interpretation, since we have the following lemma:

Lemma 1 $|A_{[cs]}| < \infty$ for all $cs \in CS$.

Proof We can use the notion of dependency to define a relation of the equivalence classes:

$$[cs_1] \preceq [cs_2] \quad :\Longleftrightarrow \quad \exists cs_1' \in [cs_1], cs_2' \in [cs_2] : cs_1' \stackrel{\star}{\twoheadleftarrow} cs_2'$$

By definition, $\preceq$ is transitive and reflexive. Additionally, it is antisymmetric: assume $[cs_1] \preceq [cs_2]$ and $[cs_2] \preceq [cs_1]$ holds, i.e. $\exists cs_1', cs_1'' \in [cs_1], cs_2', cs_2'' \in [cs_2] : cs_1' \stackrel{\star}{\twoheadleftarrow} cs_2'$ and $cs_2'' \stackrel{\star}{\twoheadleftarrow} cs_1''$. By

definition of $[cs_i]$ we have $cs'_i \overset{\star}{\twoheadleftarrow} cs''_i$ and $cs''_i \overset{\star}{\twoheadleftarrow} cs'_i$ for $i = 1, 2$. Therefore, transitivity implies: $cs'_2 \overset{\star}{\twoheadleftarrow} cs''_2 \overset{\star}{\twoheadleftarrow} cs''_1 \overset{\star}{\twoheadleftarrow} cs'_1$, which means that $[cs_1] = [cs_2]$.
Let $\prec$ denote the strict part of $\preceq$, i.e. $[cs_1] \prec [cs_2]$ iff $[cs_1] \preceq [cs_2]$ and $[cs_1] \neq [cs_2]$. We can observe that for cs_1 and cs_2 with $[cs_1] \preceq [cs_2]$ holds:

$$[cs_1] \prec [cs_2] \iff |A_{[cs_1]}| < |A_{[cs_2]}| \tag{$*$}$$

Now assume, we have $|A_{[cs]}| = \infty$ for $cs \in CS$. This can only happen if the equations imply that $|A_{[cs]}| < |A_{[cs]}|$. By definition $A_{[cs]}$ does not depend directly on itself, i.e. there must be a second class $[cs'] \neq [cs]$ with $|A_{[cs]}| < |A_{[cs']}| < |A_{[cs]}|$. This leads with $(*)$ to $[cs_1] \prec [cs_2] \prec [cs_1]$ $\lightning$. q.e.d.

If we use this definition for the example, we get $[T1] = [T2] = \{T1, T2\}$ and $A_{[T1]} = \{0, 1\}$. Additionally, this notion is compatible with our first attempt, if we define $A_{cs} := A_{[cs]}$.

Now we are in the position to define our abstract domains. Note that, for each $s \in CS \cup BS$, A_s can be "flattened" to a set $\{0,1\}^n$ for some $n \in \mathbb{N}$. Thus we can define a partial order $\leq_s$ by "bitwise" comparison. The resulting structure $\mathfrak{A}_s := \langle A_s, \leq_s \rangle$ is isomorphic to $\langle \mathfrak{P}(\{1, \ldots, n\}), \subseteq \rangle$, which means that we have a complete lattice with bottom element $\bot_s = (0, \ldots, 0)$, top element $\top_s = (1, \ldots, 1)$ and lub-operator which is a bitwise or

$$(a_1, \ldots, a_n) \sqcup_s (b_1, \ldots, b_n) = (a_1 \vee b_1, \ldots, a_n \vee b_n)$$

3.3 Interpretation of Constructors

In order to formalise which parts of an abstract value are affected by a *particular constructor (or selector)* of a *particular type* of an equivalence class $[cs]$, we need arbitrary, but fixed orderings on all constructed types of class $[cs]$ and on all constructors of a target type in $cs' \in [cs]$. More formally, we assume that $[cs] = \{cs_1, \ldots, cs_{l_{cs}}\}$ and $C^{cs} := (C^{s_1,\ldots,s_n \to cs} \mid\mid n \in \mathbb{N}_0,\ s_1, \ldots, s_n \in S) = \{c_1, \ldots, c_{m_{cs}}\}$ for all $cs \in CS$. Implicitly, we have already used these orderings in the definition of $A_{[cs]}$.

In order to locate those parameters which are not directly recursive, we need the following auxiliary functions. Let $c \in C^{s_1,\ldots,s_n \to cs}$ be such that cs has index l in the order of the sorts and c has index m in the order of constructors. We define $\varphi_{[cs]}$ such that $\varphi_{[cs]}(l, m, k) = j$ iff s_k is the non-recursive parameter of c which has the position j in the second component of the abstract values of $A_{[cs]}$. We define $\varphi_{[cs]} : \mathbb{N}_0^3 \to \mathbb{N}_0$ such that

$$\varphi_{[cs]}(l, m, k) = j \quad \text{iff} \quad \begin{array}{l} t_k \text{ is the non-recursive parameter of } c \text{ which has the position } j \\ \text{in the second component of the abstract values of } A_{[cs]}. \end{array}$$

The definition of $\varphi_{[cs]}$ (see Figure 3) is done by simultaneous induction on l, m and k.
These preparations are sufficient to give a definition for the abstract meaning of selectors:

$$\mathrm{A} : CSel^{cs \to s} \to (\mathfrak{A}_{cs} \to \mathfrak{A}_s)$$

$$\mathrm{A}[\![c^k_{\mathtt{sel}}]\!]((b, \bar{a})) := \begin{cases} (b, \bar{a}) & \text{if } [cs] = [s_k] \\ \mathtt{proj}_{\varphi_{[cs]}(l,m,k)}(\bar{a}) & \text{otherwise} \end{cases}$$

if $c \in C^{s_1,\ldots,s_n \to cs}$ such that cs has index l in the order of the sorts and c has index m in the order of constructors

$$
\begin{array}{rcl}
\varphi_{[cs]} & : & \mathbb{N}^3 \rightarrow \mathbb{N} \\
\varphi_{[cs]}(0, m, k) & := & 0 \\
\varphi_{[cs]}(l+1, 0, k) & := & \varphi_{[cs]}(l, m_l, n'_l) \\
 & & \text{where } m_l \text{ is the number of constructors of the sort with index } l\text{, and} \\
 & & n'_l \text{ the number of arguments of the last constructor of this sort} \\
\varphi_{[cs]}(l, m+1, 0) & := & \varphi_{[cs]}(l, m, n_m)) \\
 & & \text{where } n_m \text{ is the number of arguments of the constructor with index} \\
 & & m \\
\varphi_{[cs]}(l, m, k+1) & := & \mu(i > \varphi_{[cs]}(l, m, k)).([t_i] \neq [cs])
\end{array}
$$

Figure 3: Definition of $\varphi_{[cs]}$

With this definition we now can easily define the abstract meaning of constructors:

$$
\begin{array}{l}
\mathrm{A} : C^{s_1,\ldots,s_n \rightarrow cs} \rightarrow (\mathfrak{A}_{s_1} \times \cdots \times \mathfrak{A}_{s_n} \rightarrow \mathfrak{A}_{cs}) \\
\mathrm{A}[\![c]\!]((b_1, \bar{a}_1), \ldots, (b_n, \bar{a}_n)) := (b, \bar{a}) \sqcup \bigsqcup_{[s_i]=[cs]} (b_i, \bar{a}_i)
\end{array}
$$

if $c \in C^{s_1,\ldots,s_n \rightarrow cs}$ such that cs has index l in the order of the sorts and c has index m in the order of constructors and a is a minimal value such that for all $1 \leq k \leq n$ holds: $\mathrm{A}[\![c^k_{\mathtt{sel}}]\!]((b, \bar{a})) = (b_{\varphi_{[cs]}(l,m,k)}, \bar{a}_{\varphi_{[cs]}(l,m,k)})$

Firstly, a new vector $(b, \bar{a})$ is created by placing those arguments, which do not represent references to structures of the same class, into a vector filled with 0. Secondly, the information on references to the same class, which is contained in the remaining arguments are taken into account by building the maximum.

In order to illustrate these definitions, we will examine two examples:

1. Recall the definition of lists over integers from the previous section. We have $\mathfrak{A}_{\mathtt{ListOfInt}} = \{0, 1\}^2$ and with the above definitions we get the abstract functions

$$
\begin{array}{ll}
\mathrm{A}[\![\mathtt{Nil}]\!] : \mathfrak{A}_{\mathtt{ListOfInt}} & \mathrm{A}[\![\mathrm{Cons}]\!] : \mathfrak{A}_{\mathtt{int}} \times \mathfrak{A}_{\mathtt{ListOfInt}} \rightarrow \mathfrak{A}_{\mathtt{ListOfInt}} \\
\mathrm{A}[\![\mathtt{Nil}]\!] = (0, 0) & \mathrm{A}[\![\mathrm{Cons}]\!](a, l) = (0, a) \sqcup l \\
\mathrm{A}[\![\mathrm{Cons}^1_{\mathtt{sel}}]\!] : \mathfrak{A}_{\mathtt{ListOfInt}} \rightarrow \mathfrak{A}_{\mathtt{int}} & \mathrm{A}[\![\mathrm{Cons}^2_{\mathtt{sel}}]\!] : \mathfrak{A}_{\mathtt{ListOfInt}} \rightarrow \mathfrak{A}_{\mathtt{listOfInt}} \\
\mathrm{A}[\![\mathrm{Cons}^1_{\mathtt{sel}}]\!]((x, y)) := y & \mathrm{A}[\![\mathrm{Cons}^2_{\mathtt{sel}}]\!]((x, y)) := (x, y)
\end{array}
$$

   The selection of the head of the list is abstractly represented by $\mathrm{A}[\![\mathrm{Cons}^1_{\mathtt{sel}}]\!]$, which extract the information for all entries from the abstract value for the list. Note that $\mathrm{A}[\![\mathrm{Cons}^1_{\mathtt{sel}}]\!]$ is *not* the projection of the first component of its argument. The tail of the list is assumed to have the same inheritance behaviour as the whole list, and therefore $\mathrm{A}[\![\mathrm{Cons}^2_{\mathtt{sel}}]\!]$ is the identity function.

2. Our next example is a little bit more intricate. In the example which defines types of trees consisting of alternating layers of int resp. char entries. Obviously, we have $\mathfrak{A}_{\mathtt{ITree}} = \mathfrak{A}_{\mathtt{CTree}} =$

$\{0,1\} \times (\{0,1\}^4)$. With the above order, we have the first half of the second component of an element of the abstract domain reserved for the information referring to the int layers and the second half for the information referring to the char layers. The abstract functions for the constructors are in Figure 4.

$$
\begin{array}{ll}
\mathrm{A}[\![\mathtt{ILeaf}]\!] : \mathfrak{A}_{\mathtt{int}} \to \mathfrak{A}_{\mathtt{ITree}} & \mathrm{A}[\![\mathtt{INode}]\!] : \mathfrak{A}_{\mathtt{int}} \times \mathfrak{A}_{\mathtt{ITree}} \times \mathfrak{A}_{\mathtt{ITree}} \to \mathfrak{A}_{\mathtt{ITree}} \\
\mathrm{A}[\![\mathtt{ILeaf}]\!](a) := (0,(a,0,0,0)) & \mathrm{A}[\![\mathtt{INode}]\!](a,t_1,t_2) := (0,(0,a,0,0)) \sqcup t_1 \sqcup t_2 \\
\mathrm{A}[\![\mathtt{CLeaf}]\!] : \mathfrak{A}_{\mathtt{char}} \to \mathfrak{A}_{\mathtt{CTree}} & \mathrm{A}[\![\mathtt{CNode}]\!] : \mathfrak{A}_{\mathtt{char}} \times \mathfrak{A}_{\mathtt{CTree}} \times \mathfrak{A}_{\mathtt{CTree}} \to \mathfrak{A}_{\mathtt{CTree}} \\
\mathrm{A}[\![\mathtt{CLeaf}]\!](a) := (0,(0,0,a,0)) & \mathrm{A}[\![\mathtt{CNode}]\!](a,t_1,t_2) := (0,(0,0,0,a)) \sqcup t_1 \sqcup t_2
\end{array}
$$

Figure 4: Abstract constructors for CTree and ITree

Obviously, the only way to set an entry to 1 is that there is a 1 in the parameters of the abstract function.

3.4 The Complete Abstract Interpretation

We are now able to extend the interpretation of constructors to the interpretation of a program. Firstly, we define the abstract meaning of each basic function by:

$$
\begin{aligned}
&\mathrm{A} : \Omega^{bs_1,\ldots,bs_n \to bs} \to (\mathfrak{A}_{bs_1} \times \cdots \times \mathfrak{A}_{bs_n} \to \mathfrak{A}_{bs}) \\
&\mathrm{A}[\![f]\!](a_1,\ldots a_n) := 0
\end{aligned}
$$

This is reasonable since we can assume that the result of a basic function is always represented in a newly created heap cell. Similarly, we can define the abstract meaning of the constructor test functions:

$$
\begin{aligned}
&\mathrm{A} : CTest^{cs \to \mathtt{bool}} \to (\mathfrak{A}_{cs} \to \mathfrak{A}_{\mathtt{bool}}) \\
&\mathrm{A}[\![\mathtt{is}-c]\!](a) := 0
\end{aligned}
$$

The definition of the abstract semantic of an expression is as usual; let

$$\mathrm{Env}_X := (X^s \dashrightarrow \mathfrak{A}_s \mid\mid s \in S)$$

be the family of *variable assignments* and

$$\mathrm{Env}_{DF} := (DF^{s_1,\ldots,s_n \to s} \dashrightarrow (\mathfrak{A}_{s_1} \times \cdots \times \mathfrak{A}_{s_n} \to \mathfrak{A}_s) \mid\mid n \in \mathbb{N}, s_1,\ldots,s_n, s \in S)$$

be the family of *function assignments*. The meaning of an expression is inductively defined by:

$$
\begin{aligned}
&\mathrm{A} : E^s \times \mathrm{Env}_X \times \mathrm{Env}_{DF} \dashrightarrow \mathfrak{A}_s \quad \forall s \in S \\
&\mathrm{A}[\![x]\!](\chi,\varrho) := \chi(x) \quad \text{if } x \in X \\
&\mathrm{A}[\![f(e_1,\ldots,e_n)]\!](\chi,\varrho) := \mathrm{A}[\![f]\!](\mathrm{A}[\![e_1]\!],\ldots,\mathrm{A}[\![e_n]\!]) \quad \text{if } e_i \in E^{s_i}, 1 \le i \le n \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad f \in (\Omega \cup C \cup CSel \cup CTest)^{s_1,\ldots,s_n \to s} \\
&\mathrm{A}[\![F(e_1,\ldots,e_n)]\!](\chi,\varrho) := \varrho(F)(\mathrm{A}[\![e_1]\!],\ldots,\mathrm{A}[\![e_n]\!]) \quad \text{if } e_i \in E^{s_i}, F \in DF^{s_1,\ldots,s_n \to s} \\
&\mathrm{A}[\![\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \mathtt{fi} \in E^s]\!](\chi,\varrho) := \mathrm{A}[\![e_1]\!](\chi,\varrho) \sqcup \mathrm{A}[\![e_2]\!](\chi,\varrho) \quad \text{if } e \in E^{\mathtt{bool}}, e_1,e_2 \in E^s
\end{aligned}
$$

Given a program $P = \{F_i(x_{i,1}, \ldots, x_{i,n_i}) := e_i \mid\mid 1 \leq i \leq p\}$ with $F_i \in DF^{s_{i,1},\ldots,s_{i,n_i} \to s_i}$, $x_{i,j} \in X^{s_{i,j}}$ and $e_i \in E^{s_i}$ with free variables $x_{1,i}, \ldots, x_{i,n_i}$ we define the semantics of each F_i as a function

$$\mathrm{A}[\![F_i]\!] : \mathfrak{A}_{s_{i,1}} \times \cdots \times \mathfrak{A}_{s_{i,n_i}} \to \mathfrak{A}_{s_i}$$

These functions can be computed as the least fixpoint of the equations

$$\mathrm{A}[\![F_i]\!](a_{i,1}, \ldots, a_{i,n_i}) = \mathrm{A}[\![e_i]\!]([x_{i,1}/a_{i,1}, \ldots, x_{i,n_i}/a_{i,n_i}], [F_1/\mathrm{A}[\![F_1]\!], \ldots, F_p/\mathrm{A}[\![F_p]\!]])$$

Since the underlying domains do not have infinitely ascending chains and all abstract functions are monotonous, we can compute this fixpoint according to the Theorem of Knaster and Tarski.

4 Correctness

Due to the restrictions in space we can only give an outline of the correctness proof. The justification of the abstract interpretation A is done w.r.t. a modified non-strict denotational semantics. It is sufficient to proof the correctness for a non-strict semantics only, since the property of inheritance is only of interest if the function terminates. Since a function which terminates for strict semantics yields the same value with non-strict semantics, we can directly reuse the theorem.

In order to formalise that the result of a function contains parts of the arguments, we must be able to distinguish between components of the original parameters and copies of these components. This is not possible within the domains of the usual denotational semantics $\mathrm{I}[\![.]\!]$, which are the sets of finite and infinite partial terms $V^s_{\perp,\infty}$. Therefore the domains $V^s_{\perp,\infty}$ are annotated with binary values at each node. This leads to new domains $\bar{V}^s_{\perp,\infty}$, and we can define a modified semantics $\bar{\mathrm{I}}[\![.]\!]$ on these domains, which actually does not use the annotations, except that all data created by the modified semantics is annotated with 0. Therefore it is easy to show that the original semantics is equivalent to the modified one, if only arbitrary annotations are added.

If we annotate parts of the input with '1' we can observe whether this tag is propagated to the result of the function. This makes it possible to distinguish between original input and copies of it. We relate the annotated domains $V^s_{\perp,\infty}$ with the abstract domains $\mathfrak{A}_s$ via a family of functions

$$\mathsf{abst} = (\mathsf{abst}^s : V^s_{\perp,\infty} \to \mathfrak{A}_s \mid\mid \forall t \in S)$$

These functions create a component of the abstract value by collecting all corresponding annotations in the annotated value and combines them via the maximum. It is easy to extent these functions to functions $\mathsf{abst}^{\mathrm{env}_X}$ and $\mathsf{abst}^{\mathrm{env}_{DF}}$ which map annotated environments to abstract environments.

For all expression $e \in E^s$ and all environments $\bar{\chi}$, $\bar{\varphi}$ we can prove that

$$\mathsf{abst}^s(\bar{\mathrm{I}}[\![e]\!](\bar{\chi}, \bar{\varphi})) \leq_s \mathrm{A}[\![e]\!](\mathsf{abst}^{\mathrm{env}_X}(\bar{\chi}), \mathsf{abst}^{\mathrm{env}_{DF}}(\bar{\varphi}))$$

Especially, if we have $\mathrm{A}[\![e]\!](\chi, \varphi) = \perp_s$ for some abstract environments χ and φ, we know that there can not be corresponding annotated environments $\bar{\chi}$ and $\bar{\varphi}$ such that $\bar{\mathrm{I}}[\![e]\!](\bar{\chi}, \bar{\varphi})$ contains a non-zero annotation. With other words, the result of the evaluation of e does not contain parts of the input.

5 How to Use the Results

We adopt a notion known from strictness analysis (see for instance [Myc80]) in order to use the abstract interpretation.

Given a sort $s \in S$ we define the set of *test vectors* for s

$$T_s := \{(1,0,\dots,0),(0,1,\dots,0),\dots,(0,0,\dots,1)\} \subset \mathfrak{A}_s$$

as those vectors from $\mathfrak{A}_s$ with exactly one 1 entry. The set of test arguments T_F for a function $F \in DF^{s_1,\dots,s_n \to s}$ consists of those arguments where there is exactly one 1 in a single argument position:

$$(T_{s_1} \times \{\perp_{s_2}\} \times \cdots \times \{\perp_{s_n}\}) \cup (\{\perp_{s_1}\} \times T_{s_2} \times \cdots \times \{\perp_{s_n}\}) \cup \cdots \cup (\{\perp_{s_1}\} \times \{\perp_{s_2}\} \times \cdots \times T_{s_n})$$

If $\mathrm{A}[\![F]\!](t_{i,j}) = \perp_s$, where $t_{i,j} \in T_s$ is the test argument with a 1 at the j-th position for the i-th argument, we can be sure, that there is no member of level i of the j-th argument inherited to the result of the function F, since this would propagate the 1 to the result of the abstract function.

Recall the `quicksort` example from Section 2. The evaluation of $\mathrm{A}[\![\mathrm{append}]\!]$ for the four elements of T_{append} yields:

$$\begin{array}{ll} \mathrm{A}[\![\mathrm{append}]\!]((0,0),(0,1)) = (0,1) & \mathrm{A}[\![\mathrm{append}]\!]((0,0),(1,0)) = (1,0) \\ \mathrm{A}[\![\mathrm{append}]\!]((0,1),(0,0)) = (0,1) & \mathrm{A}[\![\mathrm{append}]\!]((1,0),(0,0)) = (0,0) \end{array}$$

We can conclude that the constructors of the first argument are not inherited to the result. Similarly, we obtain that the functions `filterle`, `filterge` and `quicksort` do not inherit the constructors of their argument. With this knowledge we can insert commands for deallocation into the code for `quicksort` immediately after the recursive calls. At those positions every constructor created by `filter*` can be deallocated, since it not be referenced from the result of `quicksort`.

The general scheme is to detect positions in a given program, where we have a subexpression $F(e_1,\dots,e_n)$ on the right hand side of a function definition such that $\mathrm{A}[\![t_{i,j}]\!] = \perp$. Therefore we know that all cells of level j created during evaluation of e_i is garbage after the execution of F. We can now insert commands for the deallocation of all these cells. This could easily be done by traversing the graph representing of the intermediate result e_i. However, this method is quite time consuming. Another possibility is to trace all allocations of cells of the appropriate level during execution of e_i. The advantage is that deallocation can be done without searching the result. Of course, there is some overhead during the allocation and we need additional memory. A more sophisticated method would be to allocate intermediate results in a more stack-like way on a frame which is associated to the function which is currently evaluated. All intermediate results can then be deallocated by releasing the stack frame. Of course, we need additional analyses in order to determine at compile-time how much intermediate space in terms of the input size is needed for the evaluation of a function definition. This is due to the fact that during the evaluation of a function f, which will create intermediate data, we may need space for 'inter-'intermediate data. This will be allocated in the stack frame associated with f. But this stack frame is on top of the stack frame of the function calling f, where the result of f will be located. . Therefore we need to reserve enough space for the result of f before evaluating f.

6 Efficiency

One of the key targets of this work was to keep both the compile-time and run-time overhead as small as possible. The compile-time efficiency is represented by the complexity of the computation of the abstract information. On the other hand, the run-time efficiency is characterised by two parameters: the time lost by the execution of the additional commands for the deallocation and the gain in memory utilisation.

6.1 Compile-Time Efficiency

In general, we need to evaluate all abstract functions associated with a given program for all possible arguments and recompute until stability. The finiteness of all abstract domains guarantees that a stable state is reached after a finite number of iterations. Firstly, we will estimate the number of iterations for this naïve approach.

Let $n \in \mathbb{N}$ be the number of defined functions in a program P, $m \in \mathbb{N}$ the maximal arity of a defined function and $l \in \mathbb{N}$ the maximal number of levels of a sort in P. This is also the length of the longest ascending chain in the abstract domain for this sort. An upper bound for the number of values computed in each iteration is $2^l nm$. The longest chain contains at most l elements, which means that each computed value can change at most l times during the computation until the maximal element is reached. Altogether, the worst case are $2^l mnl$ iterations, which is far from being practicable.

Since we are only interested in the value of the functions for the test arguments, it is convenient to compute only these. This optimisation reduces the number of iterations to $l^2 mn$, because there are only lmn test arguments. But we may need the value of a function for a non-test argument. Luckily, we can compute such values from the values of the functions for the test arguments.

Lemma 2 (Distributivity of A and $\sqcup$)
Let $F \in DF^{s_1,\ldots,s_k \to s}$ and $a_i, a_i' \in \mathfrak{A}_i$ for $1 \leq i \leq k$. The abstract semantics distributes with the lub-operator, i.e.

$$\mathrm{A}[\![F]\!](a_1 \sqcup a_1', \ldots, a_k \sqcup a_k') = \mathrm{A}[\![F]\!](a_1, \ldots, a_k) \sqcup \mathrm{A}[\![F]\!](a_1', \ldots, a_k')$$

Together with the next corollaries, this enables us to represent the values for a non-test argument as a 'linear' combination of test argument value.

Corollary 1 (Consistency of A)
Let $F \in DF^{s_1,\ldots,s_k \to s}$: $\quad \mathrm{A}[\![F]\!](\perp_{s_1}, \ldots, \perp_{s_n}) = \perp_s$

Corollary 2 (Generating set T_s)
Let $s \in S$. The set of test values T_s is a **generating set** for $\mathfrak{A}_s$ in the following sense:

$$\mathfrak{A}_s = \bigcup_{\{T \subseteq T_s\}} (\bigsqcup T)$$

If we examine the number of function evaluations instead of iterations, the advantage is even more dramatical. While the naïve implementation requires $2^l mn$ function evaluations per iteration, the better version only needs lmn, which implies a total of $2^{2l}m^2n^2l$ evaluations versus $l^3m^2n^2$ evaluations.

Since the number of function definitions in a program is the most characteristic value, we can interpret this result as the proof for a quadratic worst case complexity of our method.

Realistic programs, however, do not reach this worst case, since the function definitions are not that enigmatic. Especially those functions, which only depend on themselves, like for instance the append function, are interesting. If we look at the corresponding transformation, we can observe, that the value of A⟦append⟧ for an argument (a_1, a_2) in the $i+1$-th iteration depends only on the value for (a_1, a_2) in the i-th iteration and not on the value for any other argument.

$$\begin{aligned} \mathrm{A}[\![\mathrm{append}]\!](a_1, a_2) &= \mathrm{A}[\![\mathtt{Nil}]\!] \sqcup \mathrm{A}[\![\mathtt{Cons}]\!](\mathtt{Cons}^1_{sel}(a_1), \mathrm{A}[\![\mathrm{append}]\!](\mathtt{Cons}^2_{sel}(a_1), a_2)) \\ &= \mathrm{A}[\![\mathtt{Nil}]\!] \sqcup \mathrm{A}[\![\mathtt{Cons}]\!](\mathtt{Cons}^1_{sel}(a_1), \mathrm{A}[\![\mathrm{append}]\!](a_1, a_2)) \end{aligned}$$

The reason for this is that the abstract interpretation of $\mathrm{A}[\![\mathtt{Cons}^2_{sel}]\!]$ is the identity function, since this corresponds to the access to a substructure of type `ListOfInt`. Functions which are defined via *structural recursion* on the underlying type have this property. The impact on the complexity of the computation is enormous. Because the value for each argument is independent from the other values the computation of the fixpoint is stable after at most l steps. For a program where all definitions have this form the number of evaluations drops to l^2mn, which is essentially linear in the size of the program.

A single evaluation can be done very efficiently, since it consists of a sequence of lub-operations which can be implemented by a "bitwise or".

6.2 Run-Time Efficiency

We have done some experiments with programs which were enriched with deallocation commands. The programs are compiled by hand from our example language to C. The impact of the deallocations on the run-time was small, but since a real implementation is missing we can not make fair experiments on this topic. Especially, since additional deallocations reduce the need for normal garbage collection, it may even be possible that the run-time is decreased in situations where much intermediate data is created. But the impact on memory consumption can be quantified very well.

n	1	10	50	100
input	2	11	51	101
w/o ctgc	7	196	3376	15451
ctgc (end)	2	11	51	101
ctgc (max)	4	58	1278	5053

Table 1: Statistics for `qsort`

The first example (see Table 1) is the `qsort` program. The points of deallocation are those which were described above. The first line of the table shows the number of heap cells used for the input, which is the list from n down to 1. The remaining three lines show additional amount of heap used by the program without ctgc, at the end of the ctgc version, and the maximal heap usage during execution of the ctgc version, respectively. Note, that the maximal heap usage of the ctgc version is only a third of the memory consumption of the version without optimisation. The ctgc version of the program is optimal in the sense, that all intermediate data is deallocated; only the result is still in memory at the end. If `qsort` would be called from a position where the input could not be accessed any more, the ctgc version would effectively be an in-place version.

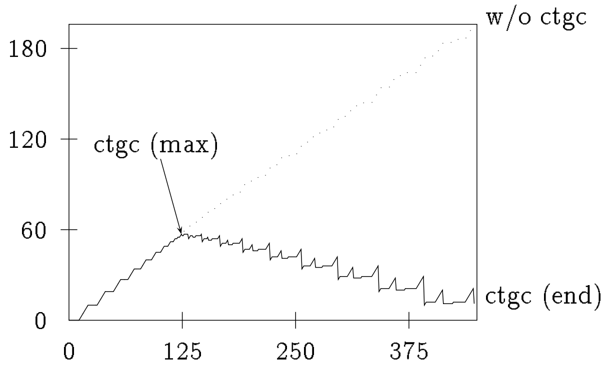

Figure 5: `qsort(10)` with and w/o ctgc

In Figure 5 we have compared the different memory consumptions during the evaluation of `qsort(10)`. The diagram shows the number of heap cells used at each function call performed. After an initial phase, where only alternating calls to `filter*` and `qsort` are executed until an empty list is reached, the two curves start to differ. The ctgc version continuously allocates and deallocates intermediate data. The point of maximal memory consumption of the ctgc version is the end of the initial phase. Until then no deallocation is performed.

Our second example (Table 2) is an implementation of queens. This particular example is remarkable, because our interpretation can not infer any information for deallocation, since everything is inherited by the functions. The only point where a deallocation can be incorporated is a single call to append. Surprisingly, the effect on memory usage is immense.

n	1	2	4	8
w/o ctgc	5	10	92	20446
ctgc (end)	4	4	20	2150
ctgc (max)	5	6	23	2239

Table 2: Statistics for queens

These results show clearly, that ctgc is worth the (small) effort.

7 Related Work

Several methods have been proposed to reduce the run-time memory consumption by ctgc. A simple classification can be done by two characteristics:

1. Which kind of information is approximated by the underlying abstract interpretation(s).

By definition, a heap cell is garbage if it is (part of a structure) not reachable from the main expression. Consequently, there are four actions which influence the state of a cell:

generation by a constructor,

sharing by non-linear function bodies,

inheriting by returning as (part of) the result of a function, and

dereferencing by pattern matching.

Accordingly, ctgc methods use abstract interpretation in order to retrieve information on some of these properties.

2. How is the memory management strategy altered by the information obtained. Especially, the *location of memory reuse* and the *way of memory reuse* is of importance.

   The location of memory reuse determines at which point of execution a garbage cell is reused. An *immediate* reuse (like in [JM89]) has the advantage of keeping the number of garbage cells small at the price of frequent interruptions of the actual computation. It is only possible for functions, where sharing of the argument does not occur. Otherwise, either the function must be altered to receive additional arguments indicating an "unshared situation", or be special versions of this function must be used in the appropriate situations. The first case requires additional test within the function, which increase the time spent on memory management operations. The second case avoids this but can increase the code size exponentially in the number of arguments of the function. A more *delayed* approach (like in [Hug92]) deallocates at the end of the corresponding function call. The advantage is that more deallocations are performed at the same time, which may be done more efficiently. Also, there is no need for modifications within the function, which circumvents the above problems.

   The way of memory reuse can be either *deallocating*, i.e. adding the cell to the free part of the heap, or *direct reuse*. The latter can be used in situations where the deallocation is immediately followed by an allocation, like for instance the append function. The result is a "in-place" version of append. Again, there is the problem whether an argument is shared or not.

The approach taken in [Hug92] is the closest work to ours. It uses a combination of generation analysis and inheritance analysis of (nested) lists of atomic values in the setting of a higher order language. Arbitrary data structures or structures containing functional parameters are not considered. Garbage is detected if heap cells are created "locally" and are not inherited. The abstract interpretation uses domains which are also based on the notion of levels of the corresponding list structure. In contrast to our work, the levels are not handled independently, i.e. all results have the shape "all levels above and including are not inherited". As we have pointed out, this is sufficient in the setting of a purely functional language with lists, but for arbitrary data structures the reduced approach is not applicable at all.Moreover, for non-functional procedures, like the `SET-CAR!` in Scheme, it makes optimisations impossible, since there is no abstract value which can be associated with those procedures. Our approach would even optimise those calls if the corresponding abstract values were given correctly. Of course, it is not possible for our method to derive this information. Furthermore, there is no gain by using this simpler domains. Each abstract value can be represented by less memory, but the worst case number of iterations is not affected, since the domains of Hughes have the same height. A single lub-operation is the maximum of two numbers which is even a bit more expensive than our bitwise or.

Park and Goldberg [PG92] describe a similar approach, which they call *escape analysis*, indicating whether (parts of) the arguments can escape from the function call.

In [JM89] a different approach is taken: list constructors which are not shared are collected as soon as they are dereferenced, i.e. as soon as they match a constructor on the left hand side of a function definition. The abstract domains are introduced as (infinite) domains $I_{\mathtt{list}}$. For practical application, the height of the abstract list is restricted, the choice of a threshold is left to the user. Additionally, if it is detected that a deallocation is immediately followed by an allocation, update-in-place is performed.

Two different approaches which use backward analysis are [HJ90] and [JM90]. While the latter is essentially abstract reference counting the first infers information on whether a particular cell is going to be accessed in the future. The cell will be deallocated if this is not the case.

8 Conclusions and Future Work

In this paper we have presented a method for statically estimating positions in a functional program, where unreferenced data will occur at run-time. Our method is based on the abstract interpretation of the underlying language.

We have described how to use our method in the context of eager evaluation. Experimental results demonstrate that the storage use used in a much better way. We have quantified the gain of this optimisation by applying it to two examples: The first example, `quicksort`, perfectly fits this approach. Not surprisingly, the result was a program with an optimal space consumption. On the other hand, the second example, `queens`, only offers one point for optimisation. Nevertheless, the result was quite good. This is evidence to assume that our method works well in practice.

Apart from a optimised memory utilisation, it was a major concern to keep the overhead at compile-time and run-time as small as possible. Of course, due to the fact that ctgc decreases the time spent for normal gc, it can even be possible that the run-time decreases.

Some of the aspects of our work, distinguishing it from other work on ctgc are:

1. We can handle arbitrary data structures instead of only lists.

2. The abstract domains can be inferred directly from the types, whereas others approaches essentially use approximations of list structures which are obtained by gratuitously restricting the height of a list.

3. We have considered correctness w.r.t. to a denotational semantics, which is more general than a proof w.r.t. a particular implementation represented as an abstract machine.

4. We do not try to handle destructive updates, because this would increase the complexity of the implementation immensely, whereas the gain is only of temporary nature. Additionally, destructive updates require enhanced or specialised versions of the functions, which may lead

to an increase of either run-time or code-size, respectively. The increase of code-size may be exponentially in the worst case, since it may be necessary to create a special version for all possible combinations of shared and unshared arguments.

5. The 'point of optimisation' is *after* the return from a function, which has the advantage that we only need the abstract information on a particular function and not its definition. This may be of interest in the context of modules, where the definition is not available.

6. We compute the abstract function only for their test arguments, which makes the analysis fast.

More work needs to be done in the following directions:

Higher-order functions: Currently, we are developing a higher-order version of our analysis. The representation of functions with functional argument or result within the abstract interpretation has two aspects: one one hand, the interpretation must be extended to handle function as arguments and results, and on the other hand, we want to infer informations on partial applications, which are also represented in the heap in the shape of *closures*. For instance, if there is a call to `filter` with a partial application as functional argument, we can infer that the heap cell associated with this partial application can be deallocated after termination of `filter`. Alternatively, those closures can be allocated on the stack frame associated with the surrounding function. In some cases, it is even possible to allocate the closure statically, since there is always only one active *incarnation* of this closure. This can be approximated with an additional analysis of the *call structure* of the program.

Polymorphism: With easy extensions our approach is able to handle this. The major idea is that a polymorphic function can not affect the levels of data with a, so that the abstract interpretation can use the smallest set to represent those elements [BH89]. In this case we can associate the set $\{0,1\}^2$ with this polymorphic type definition and evaluate each polymorphic function once with this type. Actually, our inheritance analysis is polymorphically invariant [Abr86]. This means that given a polymorphic function, our analysis will return the same results on any two monotyped instances of that function. Therefore we can analyse a polymorphic function by analysing the simplest monotyped instance of this function.

Lazy evaluation: Since the abstract interpretation itself does not depend on the evaluation strategy, the informations obtained by the analysis are still valid for lazy languages. Since in a lazy language there is no particular point in the program associated with the termination of a function, our approach is not directly applicable. But in combination with a strictness analysis, it will be possible to determine 'eager' situations, i.e. positions in a lazy program where we have $f(g(e))$ such that f is strict. In these cases we can insert deallocation commands for those heap cells created in e, which are not inherited by g.

Unboxed values: Until now, we have assumed basic values to be stored in separate heap cells. Of course, it it more reasonable to store basic values directly in constructor cells. In general, this refers to a merge of several levels of the abstract domains.

Additionally, it would be interesting to investigate the relationship with other ctgc approaches and other approaches to decrease memory consumption like deforestation ([Wad90, GLP93]) in more detail.

References

[Abr86] S. Abramsky. Strictness analysis and polymorphic invariance. In G. Goos and J. Hartmanis, editors, *Workshop on Programs as Data Objects*, number 217 in LNCS, pages 1—24, 1986.

[BH89] G. Baraki and J. Hughes. Abstract interpretation of polymorphic functions. In K. Davis and J. Hughes, editors, *Functional Programming, Glasgow 1989*, Workshops in Computing, 1989.

[GLP93] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of FPCA*, 1993.

[HJ90] G. W. Hamilton and S. B. Jones. Compile-time garbage collection by necessity analysis. In S. L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow*, 1990.

[Hug92] S. Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, 1992.

[JM89] Simon B. Jones and Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of FPCA*, 1989.

[JM90] T. P. Jensen and T.Æ. Mogensen. A backward analysis for compile-time garbage collection. In G. Goos and J. Hartmanis, editors, *Proceedings of ESOP 90*, number 432 in LNCS, pages 227—239, 1990.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the International Symposium on Programming*, number 83 in LNCS, pages 269–281, 1980.

[PG92] Y. G. Park and B. Goldberg. Escape analysis on lists. In *PLDI 92*, ACM SIGPLAN, pages 116—127, 1992.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, (73):231—248, 1990.