Improvements to a Resource Analysis for Hume*

Hans-Wolfgang Loidl[1] and Steffen Jost[2]

[1] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh EH14 4AS, Scotland, UK; Email: hwloidl@macs.hw.ac.uk
[2] School of Computer Science, University of St Andrews, St Andrews KY16 9SX, Scotland, UK; Email: jost@cs.st-andrews.ac.uk

Abstract. The core of our resource analysis for the embedded systems language Hume is a resource-generic, type-based inference engine that employs the concept of amortised costs to statically infer resource bounds. In this paper we present extensions and improvements of this resource analysis in several ways. We develop and assess a call count analysis for higher-order programs, as a specific instance of our inference engine. We address usability aspects in general and in particular discuss an improved presentation of the inferred resource bounds together with the possibility of interactively tuning these bounds. Finally, we demonstrate improvements in the performance of our analysis.

1 Introduction

In the past [22] we have developed an amortised cost based resource analysis for a higher-order, strict functional language, namely expression-level Hume [15]. Salient features of this analysis are its strong formal foundations, building on amortised program complexity and type systems, high performance due to employing efficient linear program solvers, and the possibility to express not only size-dependent but also data-dependent bounds on (generic) resource consumption. This analysis has been successfully used to infer upper bounds on the heap- and stack-space consumption and on the worst-case execution time of several embedded systems applications [23].

One of the main strengths of our analysis is its flexible design, which permits easy adaptation to model other quantitative resources. In essence, only a cost table, mapping abstract machine instructions to basic costs, needs to be modified. We use this flexibility to develop a call count analysis for higher-order programs. The bounds inferred by our analysis are in general data-dependent and we demonstrate this strength on a standard textbook example of insertion into a red-black tree, which is discussed in context of our automatic amortised resource analysis herein for the first time.

This paper also addresses practical aspects of our type-based analysis, with the goal of increasing acceptance. We have found that the presentation of resource bounds in the form of numeric annotations to the types is difficult to

* We acknowledge financial support by EU Framework VI projects IST-510255 (EmBounded) and IST-26133 (SCIEnce), and by EPSRC grant EP/F030657/1 (Islay).

understand for the non-expert user. We therefore produced an elaboration module, which translates the annotated types, produced by our analysis, into more easily digestible closed form expressions. Furthermore, the bounds shown to the user are just *one* of many possible solutions, picked by a heuristic. All solutions to the linear program yield valid cost bounds and there is no "best" solution. Starting with the solution presented by our heuristic, we now allow the user to interactively explore the solution space, which we describe here for the first time.

While our analysis was designed from the start to be highly efficient, we identified several possibilities of further improving its performance. These issues cover the tuning of the Haskell implementation as well as more tightly integrating the constraint solving phase into the overall analysis. As a result we achieve a speedup factor of up to 1.36.

The main *contributions* of this paper are:

- the development and assessment of a function call count analysis for higher-order programs, as an instance of our resource-generic amortised-cost-based resource analysis;
- concrete evidence of enhanced resource bounds due to the data-dependence, rather than only size-dependence, of our analyses;
- the development of an elaboration module providing interactive resource bounds phrased in natural-language as opposed to special type annotations;
- and the development and assessment of several improvements to the performance of the analysis.

The structure of the paper is as follows. In Section 2 we present a call count analysis. In Section 3 we discuss improvements made to the usability of our analysis, discussing an elaboration module for annotated types and the exploration of the solution space. In Section 4 we discuss and quantify performance improvements. Section 5 reviews related work. Finally, Section 6 summarises our results.

2 Call Count Analysis

In this section we use the flexibility of our resource analysis to instantiate a *call count analysis*. The goal of this analysis is to determine, within the inherent limitations to a static analysis, an upper bound on the number of function calls made in the program, possibly restricting the count to explicitly named functions. This metric is of particular interest for *higher-order programs*, where determining a bound on the number of calls to a certain function requires an inter-procedural analysis, since a call to one function may trigger further calls to itself or other functions.

Beyond being of just theoretical interest, call count information is of practical relevance for example on mobile devices, where the function of interest may be the transmission of a message, which is charged for by the mobile network provider. In this scenario the "costs" of a function call are very real and measurable in pounds. Therefore, this particular example has been studied in the Mobius project [2], where Java bytecode has been analysed.

Our cost table for the call count metric therefore features three parameters: a boolean value indicating whether or not calls to built-in functions (like vector operations) should be counted; a list of function identifiers not to be counted; and a list of function identifiers to be counted. Note that the latter two are mutually exclusive, since either all function calls except for the named ones are counted, or conversely, all function calls are ignored except for calls to the named functions. These parameters are useful in the above depicted usage scenario, where only certain functions use a chargeable service.

The resulting cost table is relatively simple,[3] with almost all entries in the cost table being zero, except for three: true function applications, built-in function application and closure creation overhead. Recall that the higher-order Hume language features under- and over-application, but not a general lambda abstraction[4]. We therefore distinguish between calling a true top-level function and a closure, since these generally have different costs. For the call count metric, the cost parameter for the application of closures is zero, since the actual function called depends on how the closure was created. Therefore at the time of closure creation, an overhead parameter is added to the cost of applying the created closure later, which thus accounts for each use of that closure. As a concrete example, we want to count the number of calls to the `add` function in the following definition of `sum`, using a higher-order `fold` function:

```
add :: num -> num -> num;
add x y = x + y;

fold :: (num -> num -> num) -> num -> [num] -> num;
fold f n []     = n;
fold f n (x:xs) = fold f (f n x) xs;

sum :: [num] -> num;
sum xs = fold add 0 xs;
```

Since we count only `add`, the type of the `add` closure created by under-application in the body of `sum` shows a cost of one per application of the closure. When folding this closure over a list, a cost proportional to the length of the list will be inferred. We get as a result the following type for `sum`

```
ARTHUR3 typing for Call Count sum: (list[Cons<2>:int,#|Nil]) -(2/0)-> int
```

which encodes a cost of 2 for each cons-cell of the input list plus a constant of 2, i.e. a bound of $2n + 2$, where n is the length of the input list.[5]

[3] See [22] for detailed cost tables with 15 entries each, showing the constants used to analyse for WCET, heap- and stack-space usage; the actual implementation has many more entries, roughly two per syntax construct and built-in operator.

[4] However, our prototype implementation also features lambda abstraction directly.

[5] The annotated function type A`-(`x`/`y`)->`B means that execution requires at most x resource units, of which y are free for reuse afterwards. Any constructor followed by `<`n`>` within type A means that up to n resources may be additionally required for *each* occurrence of that constructor within the input. Also see Section 3.1.

Table 1. Results from the Resource Analyses ($N = 10$)

Program	Cost Model	Analysis	Ratio	Cost Model	Analysis	Ratio
	Call Count			Heap Space		
`sum`	22	22	1.00	88	88	1.00
`zipWith`	21	21	1.00	190	192	1.01
`repmin`	60	60	1.00	179	179	1.00
`rbInsert`	10	20	2.00	208	294	1.41
	WCET			Stack Space		
`sum`	16926	21711	1.28	34	39	1.15
`zipWith`	27812	32212	1.16	139	140	1.01
`repmin`	47512	58759	1.24	81	222	2.74
`rbInsert`	27425	43087	1.57	82	155	1.89

Table 1 presents analysis results for call counts, heap- and stack-space consumption, and worst-case execution time (measured in clock cycles). The cost model results have been obtained from an instrumented version of the Hume abstract machine [14]. The cost model simply counts resource usage according to the cost table during an execution on some test input. The lists and trees used as test input for the cost model execution had a size of 10 each.

The `sum` example computes the sum over a list of integers, using a higher-order `fold` function, as shown above. The `zipWith` example implements a variant of the `zip` function parametrised with the function to apply to each pair of elements (which is `add` in the tested code). The `repmin` example replaces all leaves in a binary tree by the minimal element in the tree, using map and fold functions over trees. Finally, the `rbInsert` function inserts an element into a red-black tree, possibly re-balancing the resulting tree.

The results for heap- and stack-space consumption in Table 1 show generally good results. The tree-based programs, `repmin` and `rbInsert`, deliver poorer bounds for stack, since our analysis currently cannot express bounds in terms of the depth of a data-structure, as would be appropriate in this case. This problem is most pronounced for `repmin`, which performs two tree traversals. The `rbInsert` example will be discussed in more detail below. The time bounds are necessarily less accurate, since the costs for the basic machine instructions are already worst-case bounds, which we obtained through analysis of the generated machine code with the **aiT** tool [10]. In general we aim for bounds within 30% of the observed costs, which might not be the worst case. We achieve this goal for three of the four test programs. The results for the call counts show an exact match for the `sum`, `zipWith` and the `repmin` examples, all of which use higher-order operations.

In the following we take a closer look on the `rbInsert` example, with the source code given in Figure 1. This example is directly taken from Okasaki's text-

```
type num = int 16;
data colour = Red | Black;
data tree = Leaf | Node colour tree num tree;

balance :: colour -> tree -> num -> tree -> tree;
balance Black (Node Red (Node Red a x b) y c) z d =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black (Node Red a x (Node Red b y c)) z d =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black a x (Node Red (Node Red b y c) z d) =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black a x (Node Red b y (Node Red c z d)) =
   Node Red (Node Black a x b) y (Node Black c z d);
balance c a x b = Node c a x b;

ins :: num -> tree -> tree;
ins x Leaf = Node Red Leaf x Leaf;
ins x (Node col a y b) = if (x<y)      then balance col (ins x a) y b
                         else if (x>y) then balance col a y (ins x b)
                         else               (Node col a y b);

rbInsert :: num -> tree -> tree;
rbInsert x t = case ins x t of (Node _ a y b) -> Node Black a y b;
```

Fig. 1. Example `rbInsert`: insertion into a red-black tree

book [24]. A red-black tree is a binary search tree, in which nodes are coloured red or black. With the help of these colours, invariants can be formulated that guarantee that a tree is roughly balanced. The invariants are that on each path no red node is followed by another red node, and that the number of black nodes is the same on all paths. These invariants guarantee that the lengths of any two paths in the tree differs by at most a factor of two. This loose balancing constraint has the benefit that all balancing operations in the tree can be done locally. The `balance` function only has to look at the local pattern and restructure the tree if a red-red violation is found. The `rbInsert` function in Figure 1 performs the usual binary tree search, finally inserting the node as a red node in the tree, if it does not already exist in the tree, and balancing all trees in the path down to the inserted node.

The heap bound for the `rbInsert` function, inferred by our analysis is:

```
ARTHUR3 typing for HumeHeapBoxed:
 (int,tree[Leaf|Node<10>:colour[Red|Black<18>],#,int,#]) -(20/0)->
                                 tree[Leaf|Node:colour[Red|Black],#,int,#]
```

This bound expresses that the total heap consumption of the function is $10n + 18b + 20$, where the n is the number of nodes in the tree, and b is the number of black nodes in the tree. The latter demonstrates how our analysis is able to produce data-dependent bounds by attaching annotations to constructors of the input structure. This gives a more precise formula compared to one that only

refers to the size of the input structure. In this example the $18b$ part of the formula reflects the costs of applying the `balance` function, which restructures a sub-tree with a black root in the case of a red-red violation. The analysis assumes a worst-case, where every black node is affected by a balancing operation. Note that, due to the above invariants, this cannot occur for a well-formed red-black tree: any insertion into the tree will trigger at most two balancing operations (see [8][Chapter 13]). As expected, these (semantic) constraints are not captured by our analysis: our analysis must account for the worst-case of all well-typed programs. However, the *type* of red-black trees does not capture such semantic conditions and includes malformed trees (e.g. a tree with all nodes being red is still well-typed), whose processing must thus be accounted for.

Similarly the upper bound on the number of clock cycles required to compute the `rbInsert` function is associated with the black nodes in the input tree:

```
ARTHUR3 typing for Time:
(int,tree[Leaf|Node<2889>:colour[Red|Black<1901>],#,int,#]) -(2712/0)->
                                  tree[Leaf|Node:colour[Red|Black],#,int,#]
```

Finally, the call count analysis for `rbInsert` yields:

```
ARTHUR3 typing for Call Count:
 (int,tree[Leaf|Node<2>:colour[Red|Black],#,int,#]) -(1/0)->
                                  tree[Leaf|Node:colour[Red|Black],#,int,#]
```

This type encodes a bound of $2n+1$, where n is the number of nodes. By attaching costs to the constructors of the input it is possible to distinguish between nodes and leaves. However, it is currently not possible to express the fact that in the tree traversal the number of nodes visited on each path is at most $\log\ n$. In the extension of the amortised cost based analysis, developed by Campbell [5], such information on the depth of data structures is available, and his system is able to infer logarithmic bounds on space consumption for such examples.

3 Usability Improvements

3.1 Elaboration Module

Input dependent bounds on resource usage of programs are only useful if they easily allow one to distinguish large classes of inputs of roughly the same resource usage. Consider having a black box for a program that can compute the precise execution cost for any particular input. Even if this black box computes very fast, one still needs to examine all inputs one by one in order to determine the worst case or to establish an overview of the general cost-behaviour. Since the number of concrete inputs may be large or even infinite, this is generally infeasible.

The original amortised analysis technique as proposed by Tarjan [27], being very powerful, may generally produce such a precise "black box" cost oracle. This is not a hindrance for a *manual* technique, as the mathematician performing the method has direct control over the complexity and behaviour of the

"black box" that is created. However, for an automated analysis we must ensure that the outcome is always simple enough to be understood and useful. The cost bounds produced by our automated version of the amortised analysis technique are always simple. Namely, they are *linear in the sizes* of the input. This restriction to linearly dependent bounds is our chosen trade-off to obtain an *automated inference* for the amortised analysis. Recent research [16] shows how this restriction of the inference to linear bounds may be lifted. This design guarantees that we can easily divide all possible inputs into large classes having a similar cost. For example, for a program processing two lists we might learn instantly from the result of our efficient analysis that the execution cost can be bounded by a constant times the length of the second list, thereby collecting all inputs which only differ in the first argument in the same cost class. Furthermore we immediately know the execution cost for infinitely many such input classes.

We now exploit this even further to produce cost bounds expressed in natural language. Previously, the cost bound had only been communicated to the user using type annotations. While these allowed a concise and comprehensive presentation of the derived bounds, they also required a fair amount of expertise to understand, despite most derived bounds being actually rather simple. The new *elaboration module* helps to interpret the annotated types by ignoring irrelevant information, summing up weights in equivalent positions and producing a commented cost-formula, parametrised over a program's input.

We now revisit the results for the red-black tree insertion function from Section 2. We use the option `--speak` to immediately obtain:

```
ARTHUR3 typing for HumeHeapBoxed:
(int,tree[Leaf<20>|Node<18>:colour[Red|Black<10>],#,int,#]) -(0/0)->
                                 tree[Leaf|Node:colour[Red|Black],#,int,#]

Worst-case Heap-units required to call rbInsert in relation to its input:
  20*X1 + 18*X2 + 10*X3
    where X1 = number of "Leaf" nodes at 1. position
          X2 = number of "Node" nodes at 1. position
          X3 = number of "Black" nodes at 1. position
```

This makes it easy to see that the number of black nodes is significant for the cost formula. Furthermore the cost formula $20X_1 + 18X_2 + 10X_3$ is obviously much more compact and easy to understand. We are directly told that X_1 corresponds to the number of leaves in the first tree argument (there is only one tree argument here); that X_2 corresponds to the number of all nodes and that X_3 corresponds to the number of black nodes. Note that this bound is inferior to the one shown in Section 2, and we will address this in the second part of Section 3.2.

A further simplification implemented in our elaboration module is the recognition of list types and list-like types, i.e. all constructors are single recursive (e.g. `Cons`), except for precisely one constructor being non-recursive (e.g. `Nil`). In this case it is clear that each element of such a type must have precisely one such terminating constructor. Therefore the weight attached to the terminal

constructor may be moved outwards. For example, consider the annotated type of a function that receives a list of integer lists as its input:[6]

```
(list[Nil<1>|Cons<2>:list[Nil<3>|Cons<4>:int,#],#]) -(5/0)-> int]

Worst-case Heap-units required to call foo in relation to its input:
  6 + 5*X1 + 4*X2
    where X1 = number of "Cons" nodes at 1. position
          X2 = number of "Cons" nodes at 2. position
```

We see that the cost formula is much simpler than the annotated type, which features 5 non-zero annotations, whereas the cost formula has only three parameters. However, this useful simplification naturally incurs a slight loss of information. The annotated type expresses that 3 resource units are only needed once the end of the inner list is reached. If the program may *sometimes* choose to abort processing a list to the very end, those 3 resource units are not needed. This detail is almost always irrelevant and thus intentionally simplified. Nevertheless it is conceivable that a programmer might sometimes make use of this additional knowledge about the resource behaviour of the program.

3.2 Interactive Solution Space Exploration

Programs often admit several possible resource bounds and it is in general not clear which bound is preferable. For a simple example, we consider the standard list zipping, such as adding two lists of numerical values. Using a Haskell-style syntax we have:

$$\begin{aligned}
&\texttt{zipWith add}\ [\,] && [10,20] && = [\,]\\
&\texttt{zipWith add}\ [1,2,3,4] && [10,20] && = [11,22]\\
&\texttt{zipWith add}\ [1,2,3,4] && [10,20,30,40,50,60] && = [11,22,33,44]
\end{aligned}$$

We immediately see that the resource consumption, be it time or space, depends on the length of the *shorter* input list. Therefore, we have the following admissible annotated types for the closure created by `zipWith add`:

```
(list[Cons<6>:int,#|Nil<2>],list[Cons<0>:int,#|Nil<2>]) -(0/0)->
                                                  list[Cons:int,#|Nil]
Worst-case Heap-units required to call zipWith add in relation to input:
  2 + 6*X1
     where  X1 = number of "Cons" nodes at 1. position

(list[Cons<0>:int,#|Nil<2>],list[Cons<6>:int,#|Nil<2>]) -(0/0)->
                                                  list[Cons:int,#|Nil]
Worst-case Heap-units required to call zipWith add in relation to input:
  2 + 6*X1
     where  X1 = number of "Cons" nodes at 2. position
```

[6] The output was simplified to ease understanding. Our prototype implementation requires monomorphisation, so each list type would require unique constructors.

The first type says that the cost is proportional to six times the length of the first input list plus two whereas the latter type says that the cost is proportional to six times the length of the second input list plus two. Both bounds are equally useful, and it depends on external knowledge which one is preferable.

Our analysis is capable of expressing this choice within the constraints generated for the program. In fact, if we were to run the prototype analysis on a program involving the function `zipWith`, where the input for `zipWith` is generated in another part of the analysed program and in such a manner that one input list is often significantly shorter than the other one, the analysis would pick the type that admits the overall lower cost bound.

The problem here lies in communicating this choice to the user, when we analyse function `zipWith` all on its own. The analysis cannot guess which type would be preferable to the user, based on the intended use of the function. On the other hand, the overall meaning of a set of constraints is generally incomprehensible to a human due to sheer size, even after extensive simplification. A set of constraints describes an n-dimensional polytope, where n is the number of input sizes[7], i.e. the number of constructors per position in the input and output plus one for each annotation on the function arrows.

We resolve this dilemma through interaction. The analysis automatically presents a solution as usual. The user may then increase or decrease the "penalty" attached to a resource variable in the type. The constraints are then re-solved with an adjusted objective function, in which the modified penalties may cause another solution to be produced. This re-solving can be done almost instantaneously, thanks to the improvements described in Section 4, most notably due to keeping the pre-solved constraints and solution in memory. The new solution is printed on the screen again, and the user may then specify another cost variable to be altered, until the cost bound is satisfactory. Step-by-step, the user can thus explore the entire solution space for the analysed program.

Note that our implementation of the analysis has always employed a heuristic that was able to guess the "desired" result for many program examples right away. However, allowing the user to tune the solver's priorities is also a good way of understanding the overall resource behaviour of a program. So even in the many cases that are already properly resolved by the heuristic guessing a suitable objective function, this interaction may offer valuable insights.

Optimising the bound for red-black tree insertion. We again revisit the red-black tree example from Section 2, this time to show how the interactive optimisation of the solution works. Invoking our analysis for the heap space metric as before, but adding the command-line option for interaction, we obtain a prompt after the solution.

[7] Note that the solution space generally has a much higher dimension due to the necessary introduction of intermediate variables. Furthermore our experience showed that eliminating these intermediate variables is either best left to the LP solver, which is far more efficient at this task, or rather omitted entirely, since the intermediate variables have actually proven useful for the heuristic to pick a "good" solution.

```
ARTHUR3 typing for HumeHeapBoxed:
(int,tree[Leaf<20>|Node<18>:colour[Red|Black<10>],#,int,#]) -(0/0)->
                                tree[Leaf|Node:colour[Red|Black],#,int,#]
Worst-case Heap-units required in relation to input:
20*X1 + 18*X2 + 10*X3
    where X1 = number of "Leaf" nodes at 1. position
          X2 = number of "Node" nodes at 1. position
          X3 = number of "Black" nodes at 1. position

Enter variable for weight adjustment or "label","obj" for more info:
```

We are unhappy with the high cost associated with the leaves of the tree, since it seems unreasonable to require such a high cost for processing an empty leaf. Therefore we ask the analysis to lower this value considerably, by arbitrarily increasing the penalty of the associated resource variable from 6 to 36.

```
Enter variable for weight adjustment or "label","obj" for more info: X1
Old objective weight: 6.0 Enter relative weight change: 30
Setting CVar '351' to weight '36.0' in objective function.

(int,tree[Leaf|Node<10>:colour[Red|Black<18>],#,int,#]) -(20/0)->
                                tree[Leaf|Node:colour[Red|Black],#,int,#]
Worst-case Heap-units required in relation to input:
20 + 10*X1 + 18*X2
    where X1 = number of "Node" nodes at 1. position
          X2 = number of "Black" nodes at 1. position
```

This already results in the solution shown in Section 2. The fixed costs increase from 0 to 20, the costs associated with all leaves drop from 20 to 0, and the cost of each red node decreases by 8. Since every tree contains at least one leaf, this bound is clearly better for all inputs.

In this example the heuristic for choosing *a* solution picked a clearly inferior one. However, both solutions represent guaranteed upper bounds on the resource consumption. So it could be that the first solution was already precise enough. Furthermore, if we analyse a program that also constructs a red-black tree as input for the `rbInsert` function, then the LP-Solver automatically chooses the second solution in order to minimise the overall cost, which includes the cost of creating the input and all the potential associated with the input data-structure.

It is important to note that each and every function application will choose the most appropriate admissible annotated type for the function, albeit each function is analysed only once. This is achieved by simply copying the constraints associated with a function for each of its applications, using fresh variable names throughout. Since the generated LPs are sparse and easily solvable [17], this blow-up of constraints is of little concern. More information on this mechanism for resource parametricity can be found in [21]. This once more illustrates that the result for analysing a function is the set of all admissible annotations, rather than any single annotation.

4 Performance Improvements

The combined Hume prototype analyses delegate the solving of the generated linear programming (LP) problem to the LP-solver lp_solve [3], which is available under the GNU Lesser General Public Licence. In early versions of the analyses, this was done by writing all constraints in a human readable format to a file and then calling lp_solve to solve that file. The solution was then read via a Unix pipe and also recorded in a text file. This solution had the advantage that the generated LP was directly tangible. The file contained various comments, in particular the line and column of the source code that ultimately had triggered the generation of that particular constraint. This yielded very high transparency and was very useful in developing and validating the resource analysis. Furthermore, one could alter the LP by hand for experimentation and feed it to the solver again without any difficulties.

However, this solution also had several drawbacks, namely:

1. Communicating large data-structures, such as linear programming problems, via files on the hard-disk of a computer is very slow.
2. Altering the constraints just slightly, requires the full, slow repetition of transmitting the entire LP and solving it from scratch.
3. Running the analysis requires the user to install and maintain the lp_solve command-line tool separately.
4. lp_solve only allows very limited floating point precision when using file communication, causing rounding errors of minor significance.

We have thus added the option of calling the lp_solve library, written in C, directly through the foreign function interface (FFI) of the Glasgow Haskell Compiler (GHC) [11]. This solution now resolves all of the above issues. The library is linked into the combined Hume prototype analyses' executable file, producing an easy to use stand-alone tool. Furthermore, eliminating the first two problems was a direct prerequisite for realising the interactive solution space exploration described in Section 3.2. Interaction can only work if the time the user is required to wait between each step is very small. The solver lp_solve supports this by fast incremental solving, where the last solution and the pre-solved constraints are kept in the memory and can be adjusted for subsequent solving. Therefore in all program examples examined thus far on our contemporary hardware, *re-solving* the linear program could be done within a fraction of a second, for example less than 0.02 seconds for the biquad filter program example, as opposed to 0.418 seconds required for first-time solving as shown in Table 2.

Solving the linear programming problem via the foreign function interface is therefore the default setting now. However, the previous mechanism of calling lp_solve via the command-line is still available through option `--noapisolve`, since this is still quite useful when transparency is desired more than performance, which is often the case when studying the combined Hume analysis itself by applying it to small program examples.

Table 2. Run-time for Analysis and for LP-solving

Program	Constraints		Run-time non-FFI		Run-time FFI		Speedup	
	Number	Variables	Total	LP-solve	Total	LP-solve	Total	LP-solve
biquad	2956	5756	1.94s	1.335s	1.43s	0.418s	1.36	3.20
cycab	3043	6029	2.81s	2.132s	2.75s	1.385s	1.02	1.54
gravdragdemo	2692	5591	2.16s	1.605s	2.14s	1.065s	1.01	1.51
matmult	21485	36638	104.88s	101.308s	84.17s	21.878s	1.25	4.63
meanshift	8110	15005	11.32s	9.851s	11.01s	6.414s	1.03	1.54
pendulum	1115	2214	0.76s	0.479s	0.67s	0.260s	1.13	1.84

Table 2 summarises the run-times[8] of both versions of the combined Hume resource analysis on some program examples: the *non-FFI version*, which uses lp_solve via the command-line to solve the constraint set, and the *FFI version*, which calls the lp_solve library through the foreign-function-interface. For each version we show the total run-time of the analysis as well as the run-time for just the LP-solving component (both in seconds). The final two columns show the speedup of the FFI version over the non-FFI version.

The applications used in Table 2 to compare the run-times of the analysis are as follows. The `biquad` application is a biquadratic filter application. The `gravdragdemo` application is a simple, textbook satellite tracking program using a Kalman filter, developed in preparation for the `biquad` application. The `cycab` application is the messaging component of an autonomous vehicle. The `pendulum` application balances an inverted pendulum on a robot arm with one degree of freedom. The `meanshift` computer vision application is a simple lane tracking program. Finally, `matmult` is a function for matrix multiplication, which was automatically generated from low-level, C-like code. The generated program makes heavy use of higher-order functions and of vectors for modelling the state space, due to the design of the automatic transformation. This results in a high number of constraints and therefore in a compute-intensive analysis phase.

We see that the speedup for the LP-solving part is quite impressive (1.51–4.63). However, one should recall that the command-line version (non-FFI) is required to build the C data-structures holding the constraint set, whereas in the library version (FFI), this task is performed by our prototype analysis, delivering the constraints ready-to-use. This also explains why the overall run-time does not decrease by the same amount as the time spent on LP solving.

The overall speedup is largely varying for our program examples (1.01–1.36), but with the overall run-time being just around 1–3 seconds, it is hard to judge which is the dominating factor in processing. For the large `matmult` example, the only one where the analysis is actually working for a noticeable time, the overall run-time could be reduced by an impressive 20%, or roughly 20 seconds.

[8] The performance measurements in Table 2 have been performed on a 1.73GHz Intel Pentium M with 2MB cache and 1GB main memory.

5 Related Work

Type-based Resource Analysis: Using type inference to statically determine quantifiable costs of a program execution has a long history. Most systems use the basic type inference engine to separately infer information on resource consumption. In contrast, our analysis uses a tight integration of resource information into the type, by associating numeric values to constructors in the type. These values are the factors in a linear formula expressing resource consumption. Another notable system, which uses such a tight integration of resources into the type system, is the sized type system by Hughes et al. [20], which attaches bounds on data structure sizes to types. The main difference to our work is that sized types express bounds on the size of the underlying data structure, whereas our weights are factors of a linear bound on resource consumption. The original work was limited to type checking, but subsequent work has developed inference mechanisms [6,19,29]. Vasconcelos' PhD thesis [28] extended these previous approaches by using abstract interpretation techniques to automatically infer linear approximations of the sizes of recursive data types and the stack and heap costs of recursive functions. A combination of sized types and regions is also being developed by Peña and Segura [25], building on information provided by ancillary analyses on termination and safe destruction.

Amortised Costs: The concept of amortised costs has first been developed in the context of complexity analysis by Tarjan [27]. Hofmann and Jost were the first to develop an automatic amortised analysis for heap consumption [17], exploiting a difference metric similar to that used by Crary and Weirich [9]. The latter work, however, only *checks* bounds, and does not *infer* them, as we do. Apart from inference, a notable difference of our work to the work of Tarjan [27] is that credits are associated on a *per-reference* basis instead of the pure layout of data within the memory. Okasaki [24] also noted this as a problem, resorting to the use of *lazy evaluation*. In contrast, per-reference credits can be directly applied to strict evaluation. Hofmann and Jost [18] have extended their method to cover a comprehensive subset of Java, including imperative updates, inheritance and type casts. Shkaravska et al. [26] subsequently considered the inference of heap consumption for first-order polymorphic lists, and are currently studying extensions to non-linear bounds. Hoffmann and Hofmann [16] have recently presented an extension to the amortised resource analysis that can produce polynomial bounds for programs over lists and trees. Campbell [5] has studied how the Hofmann/Jost approach can be applied to stack analysis for first-order programs, using "give-back" annotations to return potential. This improves the quality of the analysis results that can be obtained for stack-like metrics. While, in order to keep the presentation clear, we have not done so here, there is no technical reason why "give-back" potential cannot also be applied to the higher-order analysis that we have described.

Other Resource Analyses: Another system that is generic over the resource being analysed is the COSTA system [1]. Its inference engine is based on abstract

interpretation. First a set of recurrence relations are generated, which are then solved by a recurrence solver that is tailored for the use of resource analysis and as such produces better results than general recurrence solvers.

Gómez and Liu [12] have constructed an abstract interpretation of determining time bounds on higher-order programs. This executes an abstract version of the program that calculates cost parameters, but which otherwise mirrors the normal program execution. Unlike our type-based analysis, the cost of this analysis therefore depends directly on the complexity of the input data. Gulwani et al.'s SPEED system [13] uses a symbolic evaluation approach to calculate non-linear complexity bounds for C/C++ procedures using an abstract interpretation-based invariant generation tool. Precise loop bounds are calculated for 50% of the production loops that have been studied.

Several systems aim specifically at the static prediction of heap space consumption. Braberman et al. [4] infer polynomial bounds on the live heap usage for a Java-like language with automatic memory management. However, unlike our system, they do not cover general recursive methods. Chin et al. [7] present a heap and a stack analysis for a low-level (assembler) language with explicit (de-)allocation. By inferring path-sensitive information and using symbolic evaluation they are able to infer exact stack bounds for all but one example program.

WCET Analysis: Calculating bounds on worst-case execution time (WCET) is a very active field of research, and we refer to [30] for a detailed survey.

6 Summary

This paper presented extensions and improvements of our amortised cost based resource analysis for Hume [22]. By instantiating our resource inference to the new cost metric of call counts, we obtain information on the number of (possibly specific) function calls in higher-order programs. While initial results from an early call count analysis where presented in [21], we here give the first discussion of the analysis itself and assess it for a range of example programs. In particular, we demonstrate for a standard textbook example of insertion into a red-black tree that the inferred bounds are in general data-dependent and therefore more accurate than bounds that are only size-dependent.

Furthermore, we presented improvements of our analysis in terms of usability, performance, and quality of the bounds. As an important new feature for the acceptance of our type based analysis, the resource bounds are now translated into closed-form cost formulae. Based on feedback from developers of Hume code in interpreting the resource bounds, encoded in annotated types, we consider the improvement in usability through the elaboration module as the biggest step in making our analysis available to a wider community. Although this improvement is the most difficult one to quantify, we believe that such presentation of resource bounds as closed-form formulae is essential for the acceptance of a type-based inference approach.

We have also reported on significant improvements made to the performance of the analysis. For the example programs used in this paper, we observe a

speedup factor of up to 1.36, mainly due to a tighter integration of the linear program solving through the FFI interface provided by GHC.

As future work we plan to investigate whether combining our approach with a sized-type analysis might also allow the inference of super-linear bounds, while still using efficient LP-solver technology, possibly multiple times. One challenge for the analysis will be to capture all future code optimisations that might be added to the Hume compiler. We are experimenting with approaches where resource usage is exposed in the form of explicit annotations to a high-level intermediate form, retaining a close correlation of the analysis with the source language, while being able to model a wider range of compiler optimisations.

The prototype implementation of our amortised analysis is available online at `http://www.embounded.org/software/cost/cost.cgi`. Several example Hume programs are provided, and arbitrary programs may be submitted through the web interface.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO'07)*, LNCS 5382, pages 113–132, Amsterdam, The Netherlands, October 24–26, 2007. Springer.
2. L. Beringer, M. Hofmann, and M. Pavlova. Certification Using the Mobius Base Logic. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO'07)*, LNCS 5382, pages 25–51, Amsterdam, The Netherlands, October 24–26, 2007.
3. M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve: Open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence). `http://lpsolve.sourceforge.net/5.5`.
4. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Intl. Symp. on Memory Management (ISMM'08)*, pages 141–150, Tucson, USA, June 2008. ACM.
5. B. Campbell. Amortised Memory Analysis Using the Depth of Data Structures. In *European Symp. on Programming (ESOP 2009)*, LNCS 5502, pages 190–204, York, UK, March 2009. Springer.
6. W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computing*, 14(2,3):261–300, 2001.
7. W.-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-level Programs. In *Intl. Symp. on Memory Management (ISMM'08)*, pages 151–160, Tucson, USA, June 2008. ACM.
8. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.
9. K. Crary and S. Weirich. Resource Bound Certification. In *Symp. on Principles of Programming Languages (POPL'00)*, pages 184–198, Boston, USA, 2000. ACM.
10. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Intl Workshop on Embedded Software (EMSOFT'01)*, LNCS 2211, pages 469–485, Tahoe City, USA, October 8–10, 2001. Springer.
11. The Glasgow Haskell Compiler. `http://haskell.org/ghc`.

12. G. Gomez and Y.A. Liu. Automatic Accurate Time-Bound Analysis for High-Level Languages. In *Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, LNCS 1474, pages 31–40, Montreal, Canada, June 1998. Springer.
13. S. Gulwani, K.K. Mehra, and T.M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, Savannah, USA, Jan. 2009. ACM.
14. K. Hammond. Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: The Hume Approach. In *First Central European Functional Prog. Summer School (CEFP'05)*, LNCS 4164, pages 100–134. Springer, 2005.
15. K. Hammond and G.J. Michaelson. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In *Intl. Conf. on Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pages 37–56. Springer, Sep. 2003.
16. J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential — A Static Inference of Polynomial Bounds for Functional Programs. In *European Symp. on Prog. (ESOP'10)*, LNCS. Springer, 2010. To appear.
17. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Symp. on Principles of Programming Languages (POPL '03)*, pages 185–197, New Orleans, USA, January 2003. ACM.
18. M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *European Symp. on Prog. (ESOP'06)*, LNCS 3924, pages 22–37. Springer, 2006.
19. R.J.M. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Intl. Conf. on Functional Programming (ICFP '99)*, pages 70–81, Paris, France, September 1999. ACM.
20. R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Symp. on Principles of Programming Languages (POPL'96)*, pages 410–423, St. Petersburg Beach, USA, January 1996. ACM.
21. S. Jost, H-W. Loidl, K. Hammond, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *Symp. on Principles of Prog. Langs. (POPL '10)*, pages 223–236, Madrid, Spain, Jan. 2010. ACM.
22. S. Jost, H-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In *Formal Methods (FM '09)*, LNCS 5850, pages 354–369, Eindhoven, 2009. Springer.
23. S. Jost, H-W. Loidl, N. Scaife, K. Hammond, G. Michaelson, and M. Hofmann. Worst-Case Execution Time Analysis through Types. In *Euromicro Conf. on Real-Time Systems (ECRTS'09)*, pages 13–16, Dublin, Ireland, July 1–3, 2009. ACM.
24. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
25. R. Pena, C. Segura, and M. Montenegro. A Sharing Analysis for Safe. In *Symp. on Trends in Functional Prog. (TFP'06)*, pages 109–128, Nottingham, 2006. Intellect.
26. O. Shkaravska, Ron van Kesteren, and Marko van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Typed Lambda Calculi and Applications (TLCA 2007)*, LNCS 4583, pages 351–365, Paris, France, June 26–28, 2007. Springer.
27. R.E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.
28. P.B. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, February 2008.
29. P.B. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Intl. Workshop on Implementation of Functional Languages (IFL 2003)*, LNCS 3145, pages 86–101, Edinburgh, UK, September 2003. Springer.
30. R. Wilhelm *et al.* The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.