IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

# D43 (WP7): Analysis Robustification

| | |
|---|---|
| Due date of deliverable: | February 2009 |
| Actual submission date: | March 2009 |

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: St Andrews University Revision: 1.40

**Purpose:** Robustifying and enhancing the prototype implementation of the combined Hume analyses in order to render them more usable outside the EmBounded project.

**Results:** The combined Hume prototype analysis is now much easier to use due to producing results in clear and concise human readable format. Auxiliary tools have been more tightly integrated into the analysis. Furthermore the bounds produced on resource consumption are of higher quality by refining the treatment of several special cases and of built-in functions, such as primitive vector operations.

**Conclusion:** Making the improvements described in this document is highly useful to facilitate the dissemination of the combined Hume analysis, and also allows us to make the package easily available through the project website.

| Project co-funded by the European Commission within the 6th Framework Programme (2002-06) | | | |
|---|---|---|---|
| **Dissemination Level** | | | |
| PU | Public | | ⋇ |
| PP | Restricted to other programme participants | (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium | (including the Commission Services) | |
| CO | Confidential only for members of the consortium | (including the Commission Services) | |

# Analysis Robustification

Steffen Jost `<jost@cs.st-andrews.ac.uk>`
School of Computer Science, Univ. of St Andrews, St Andrews, Scotland

Hans-Wolfgang Loidl `<hwloidl@tcs.ifi.lmu.de>`
Institut für Informatik, Theoretische Informatik, Ludwig-Maximilians Universität, München

Kevin Hammond `<kh@cs.st-andrews.ac.uk>`
School of Computer Science, Univ. of St Andrews, St Andrews, Scotland

**Abstract**

This document describes the enhancements and improvements that we implemented for the prototype implementation of the parameterised stack-, heap-space and worst-case execution time analyses for Hume. We assess the improvements made on the quality of the inferred bounds, in particular we obtain exact matches for heap and stack bounds on vector programs, on the reduction in runtime of the analysis, which shows performance improvements of up to 115%, and on the increase in overall user-friendliness, in particular the presentation of parameterised resource bounds as human readable, closed-form formulae.

The positioning of this deliverable is discussed in Section 1. In addition, we created a user manual for the combined Hume analyses prototype implementation, which is contained in Appendix A for reference and also available on-line [9].

| Major Revisions | | |
|---|---|---|
| Revision | Date | Changes |
| *1.40* | 14 Aug. 2009 | positioning in Sec 1; link to on-line manual |
| *1.37* | 31 March 2009 | initial version |

# Contents

## 1 Overview and Positioning of Deliverable D43

We developed a prototype implementation of the Hume Space- and Worst-Case Execution Time analyses in Deliverables D13 (WP4) [5] and D15 (WP3b) [6] respectively. Both analyses were verified against the program examples developed in Deliverables D07, D27 and D33 (WP8abc) [16, 15, 13], which was reported in Deliverables D30 (WP4c) [8]and D16 (WP3c) [7].

We have now robustified and enhanced the prototype analyses in order to increase their usefulness outside the EmBounded project. In particular, we performed the following tasks, closely matching the outline of the deliverable:[*]

a) Enhancing the outputs of our analyses by implementing an elaboration module, which turns the result, i.e. an annotated type, into closed form cost-formulae that are accompanied by an elaboration in natural language. This makes the combined Hume analyses outcome more accessible. See Section 2.

b) Extending the analysis to directly deal with the primitive vector operations provided by Hume, as well as base-type coercions. See Section 5.

c) Improving the ancillary analyses, such as those performed by the Hume compiler, the **aiT** tool and preliminary steps performed by the prototype implementation itself. See Section 6.

d) Allowing the user to interactively explore the solution space, if more than one cost bound can be ascribed to a specific program. See Section 3.

e) Calling the LP-solver through the Haskell foreign function interface, thereby allowing to link the solver library directly into the prototype implementation's executable, lifting the requirement for an external commandline tool. See Section 4.

All of these tasks contribute to an increased quality on the resources bounds determined by the combined Hume analyses. The work performed had been directly beneficial for the production of Deliverable D44 (WP8d)[14], for example d) had been used to obtain a more precise cost bound. We report on this example in more detail in Section 2.

After completing these improvements made to the analysis, described herein, we are in the process of creating binary distributions of the analysis, and all related tools (in particular, the translation of Hume to Schopenhauer (Core-Hume), and the now included `lp_solve` library for constraint solving). The packages will be made available via the EmBounded web page. We will support a range of architectures to ensure broad compatibility with common-place systems. Furthermore, we are in the process of updating our web-based interface to an on-line version of the analysis. This provides an easy-to-use interface for exploring the possibilities of the analysis, without requiring local installation. We consider both steps as important dissemination activities, to make in particular the academic community aware of the results of the EmBounded project, in the area of automated resource analysis.

## 2 Elaboration Module

The most visible improvement of the prototype implementation of the combined Hume analyses is the capability to print the result of the analysis in a more descriptive format. Thus far, all information had been communicated through type annotations, which allow a concise and comprehensive presentation at the expense of requiring some experience and certain degree of familiarity with types.

[*]The task listing follows the order given in the description of Deliverable D43 (WP7f). For presentation purposes a different order was chosen for the following sections, depending on the concepts required to describe the performed task.

```
ARTHUR3 typing for HumeHeapBoxed:
photoaction:
  (?tuple4intintlist1_oplist1tuple2intint[
    ?T4intintlist1_oplist1tuple2intint:
      int,
      int,
      ?list1_op[
        ?C1_op<10>:_op[L|R|P<4>|U|D],#
        |?N1_op<20>],
      ?list1tuple2intint[
        ?C1tuple2intint:?tuple2intint[?T2intint:int,int],#
        |?N1tuple2intint]])
  -(0/0)->
  ?tuple2list1tuple2intintint[
      ?T2list1tuple2intintint:?list1tuple2intint[
        ?C1tuple2intint:?tuple2intint[?T2intint:int,int],#
        |?N1tuple2intint],
      int]
```

Figure 1: Old-style annotated type for the Photographer program example. For readability some line-breaks were manually added.

The new elaboration module helps to interpret the annotated types by ignore irrelevant information, summing up weights in equivalent positions and producing a commented cost-formula, parameterised over a program's input. We illustrate this by revisiting the heap-space analysis result of the first program example from Deliverable D30 (WP4c), the Photographer program, which simulated a controller for an electric pan & tilt camera. Recall that the input of this program was a list of simple instructions (Left,Right,Photo,Up,Down). The raw result of the heap-space analysis for function `photoaction` is reproduced for convenience in Figure 1.

We now rerun the analysis, using the new option `--speak`, which immediately yields the much compacter heap-space cost formula $20 + 10X_1 + 4X_2$ for the function `photoaction`. We are further informed by the analysis, using natural language, that $X_1$ denotes the number of `C_op` nodes and that $X_2$ is the number of `P` nodes among the function's input. Hence we received a closed cost formula, depending on the sizes of the input.

Using information from the specification of the program and the intentions for the used datatypes, we know that there is one `C_op` node per instruction in the input, and one `P` per "take photo"-instruction. In other words, running the photographer program requires *at most* 20 heap cells plus 10 heap cells per instruction plus 4 heap cells per photo. Unfortunately this last step cannot be performed by the analysis, as it has no chance to guess our intentions for specifying the datatypes and the meaning of the names given to them by the programmer accordingly. However, the formula $20 + 10X_1 + 4X_2$ is of course much easier to understand for the average user than the full annotated type shown in Figure 1.

Note that the original annotated type actually said something slightly different. Precisely, the annotated type conveys that running the photographer needs at most 10 heap cells per instruction, plus 4 heap cells per photo and plus 20 heap cells per `N_op`. The latter being the nil-constructor of the list. Each finite list has precisely one end. The elaboration module recognises this and adds the cost of 20 heap cell units to the constant part of the cost formula, further simplifying the meaning. This simplification is performed for all list-like datatypes, i.e. datatypes whose constructors have precisely one recursive argument, except for a single non-recursive constructor.

```
ARTHUR3 typing for HumeHeapBoxed:

photoaction:
  (tuple_oplist[T_oplist:int,int,list_op[C_op<10>:_op[L|R|P<4>|U|D],#|N_op<20>],
                                                      list[C:tuple[T:int,int],#|N]])
 -(0/0)->
           tuple[T:list[C:tuple[T:int,int],#|N],int]

Worst-case Heap-units required to call photoaction in relation to its input:
  20 + 10*X1 + 4*X2
    where
        X1 = number of "C_op" nodes at 1. position
        X2 = number of "P" nodes at 1. position
```

Figure 2: Shortened annotated type and commented cost formula for the Photographer example.

Another minor improvement is the shortening of the long datatype and constructor names. The long names are introduced by the translation from Hume to Core-Hume and ensure that there are no name clashes. However, for on-screen printing purposes only, some ambiguity is tolerable, especially since it is unlikely to happen at all. Therefore the analysis offer options `--shrtcon` to abbreviate all generated names for the sake of readability. This minor improvement turned out to be quite helpful before the elaboration module became operational, so the shortcut option `-s` exists which is equivalent to both `--shrtcon` and `--speak`. The full output produced by option `-s` for the Photographer example program is shown in Figure 2.

## 3 Interactive Solution Space Exploration

It is often the case that programs admit several possible resource bounds and that it is not clear which bound is preferable. For a simple example, we consider the standard list zipping, i.e. merging two different lists into a single list of pairs. So we have

$$\begin{array}{llllll}
\texttt{zip} & [] & [A,B] & = & [] \\
\texttt{zip} & [1,2,3,4] & [A,B] & = & [(1,A),(2,B)] \\
\texttt{zip} & [1,2,3,4] & [A,B,C,D,E,F] & = & [(1,A),(2,B),(3,D),(4,E)]
\end{array}$$

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 (shortened) annotated types for zip:

```
zip: (list[Cons<5>:int,#|Nil] -&-> list[Cons:int,#|Nil])
                                                -(2/0)-> plist[PCons:int,int,#|PNil]

zip: (list[Cons:int,#|Nil] -&-> list[Cons<5>:int,#|Nil])
                                                -(2/0)-> plist[PCons:int,int,#|PNil]
```

The first type saying that the cost is proportional to five times the length of the first input list and the latter saying that is proportional to five time the length of the second input list. Both types are of equal use.

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

However, the problem lies within communicating this choice to the user, when we analyse function `zip` all on its own. The analysis cannot guess which type would be preferable to the user, based on his intended use of the function. In general, we can only present the complete constraint system to the user, since min's and max's can be arbitrarily nested with arbitrary cost factors added in between. While it is generally possible to simplify the constraints, they nevertheless become quickly indigestible to humans. The reason being that solution space is in fact an $n$-dimensional polytope, where $n$ is the number of input sizes,[†] i.e. number of constructors per position in the input and output, which would be 8 for `zip` and 17 for `photoaction` from the example discussed in Section 2. We cannot reduce the dimension any further without loosing precision.[‡]

We propose to resolve this using the elaboration module from Section 2. If the analysis receives the option `--askobjective`, it then presents a prompt to the user after showing the cost formula. The user may then specify a cost variable, whose factor can be freely adjusted. We then resolve the constraints for the altered objective function, which can be done almost instantaneously thanks to the improvement described in Section 4, and print the new solution again. The user may then specify another cost variable to be altered until he is satisfied.

Step-by-step, the user can thus explore the solution space for the analysed program. Note however that the analysis has always employed a clever heuristic to guess the most optimal result for many program examples right away. However, allowing the user to tweak 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.

## 3.1 Improving the Biquadratic filter

The bounds for the Biquadratic filter application presented in D44 (WP8d) already made use of this new feature of interactively exploring the solution space. As an illustrative example we will now describe the steps taken to arrive at the improved cost bound shown in D44 (WP8d).

We start the analysis with options `--askobjective` and `-s` and receive the following worst-case execution time for executing box `compute_filter` on the Renesas M32 processor:

```
Worst-case Time-units required to compute box compute_filter in relation to its input:
  359 + 2089*X1 + 1553*X2 + 23944*X3
    where
        X1 = number of "BPF" nodes at 1. position
        X2 = number of "HPF" nodes at 1. position
        X3 = one if 2. wire is live, zero if the wire is void

Enter variable for weight adjustment or "label","obj" for more info;
leave blank to proceed:
```

We enter `X3`, since we are unhappy with the high fixed cost associated to the wire, and are asked how we would like to change the weight.

[†] Note that internally to the analysis, the solution space has a much higher dimension due to the necessary introduction of intermediate variables. However, we have found that performing variable elimination, which would reduce the dimension again, is either best left to LP solver, which is far more efficient at this task, or rather omitted at all, since the intermediate variables have actually proven useful for the heuristic to pick a "useful" solution.

[‡]Except for improving list-like types as observed in Section 2, which would reduce the dimension of the (significant) solution space for `zip` down to five.

```
Enter variable for weight adjustment or "label","obj" for more info;
leave blank to proceed: X3
Old objective weight: 2.0 Enter relative weight change: 6
Setting CVar '1042' to weight '8.0' in objective function.
Calling "lp_solve" via foreign function interface now......
Done. 1.0001659393310547e-3s required for solving.

Worst-case Time-units required to compute box compute_filter in relation to its input:
  359 + 2089*X1 + 1553*X2 + 23944*X3
    where
        X1 = number of "BPF" nodes at 1. position
        X2 = number of "HPF" nodes at 1. position
        X3 = one if 3. wire is live, zero if the wire is void

Enter variable for weight adjustment or "label","obj" for more info;
leave blank to proceed:
```

So the solver responded by shifting the fixed cost of 23944 clock cycles from the second to the third wire. This is natural, since the box only runs if all wires are live and contain data, therefore it is ambiguous which wire should be attributed with the cost.

Note that since the second wire has now zero cost contributed to it, the elaboration module chooses to omit it in the cost formula, causing the variable name `X3` to be reused for the third wire. While this may seem ambiguous, we have made the experience that users are more comfortable with this re-enumeration of variables for the short cost formula. However, the analysis also offers the option `--noreenum` to avoid this re-enumeration, using unique variable names. Even without this option, one can enter `label` to immediately see the unique variable identifiers internally used by the analysis. The output "`Setting CVar '1042' to weight '8.0' in objective function.`" shows this internal unique number. In fact, we could have just entered `1042` instead of `X3` right away, whichever the user is more comfortable with. In addition, this unique name also allows us to lower the weight factor again, once it has become zero and is thus omitted from the cost formula.

It is also interesting to note that the output contains the time required for re-solving the entire constraint set. Less than 0.0011 seconds[§] were required for re-solving after adjusting the objective function, so this process is indeed fast enough so that the user can interactively use it. We will comment more on the runtime of the LP-solver in Section 4.

Back to our example at hand. The attribution of the fixed cost of 23944 clock cycles has been shifted from the second to the third wire. Therefore we now ask the analysis to the reduce the cost associated with the third wire:

```
Enter variable for weight adjustment or "label","obj" for more info;
leave blank to proceed: X3
Old objective weight: 2.0 Enter relative weight change: 6
Setting CVar '1043' to weight '8.0' in objective function.
Calling "lp_solve" via foreign function interface now......
Done. 9.999275207519531e-4s required for solving.
```

As can be expected by the symmetry of the wire weights, this causes the cost attribution once more to be shifted to the fourth and last wire, for which we again ask the analysis to reduce it. This leads eventually to the following final output:

[§]Measurement performed on a contemporary 2.53GHz Intel Core 2 Duo laptop with 6MB cache and 4GB main memory.

```
Enter variable for weight adjustment or "label","obj" for more info;
leave blank to proceed: X3
Old objective weight: 2.0 Enter relative weight change: 6
Setting CVar ’1044’ to weight ’8.0’ in objective function.
Calling "lp_solve" via foreign function interface now......
Done. 9.999275207519531e-4s required for solving.

Worst-case Time-units required to compute box compute_filter in relation to its input:
  359 + 9374*X1 + 16659*X2 + 16123*X3 + 14570*X4
    where
        X1 = one if 1. wire is live, zero if the wire is void
        X2 = number of "BPF" nodes at 1. position
        X3 = number of "HPF" nodes at 1. position
        X4 = number of "LPF" nodes at 1. position
```

We have now arrived at the cost formula as presented in D44 (WP8d). Note that the overall worst-case execution time is still the same, since we know that there is always exactly one of the mutually exclusive constructors `BPF`, `HPF` and `LPF`. Hence executing box `compute_filter` requires at most 26392, for the most expensive input `BPF`.

However, compared with the first cost formula given, we can now see that the fourth case of input `NULLF` has a much lower cost bound of only $359 + 9374 = 9733$ clock cycles. The original cost formula did not distinguish between the constructors `NULLF` and `LPF` attributing a cost of 24303 to both cases, thereby over-estimating the `NULLF` case.

## 4 Integration of LP-Solve

The combined Hume prototype analyses delegated the solving of the generated linear programming problem to the LP-solver lp_solve [1], which is available under the GNU Lesser General Public License.

Technically 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 Unix pipe and also protocolled into a 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. 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:

a) Communicating large datastructures, such as linear programming problems, via files on the hard-disk of a computer is very slow.

b) Altering the constraints just slightly, requires the full, slow repetition of transmitting the entire LP and solving it from scratch.

c) Running the analysis requires the user to install and maintain the lp_solve commandline tool separately.

d) lp_solve only allows very limited floating point precision in file communication, causing rounding errors of minor significance.

We have thus *added* the option of calling the lp_solve library directly through the foreign function interface (FFI) of the Glasgow Haskell Compiler[3]. This was a purely technical implementation problem,

| 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 1: Run-time for Analysis and for LP-solving

that essentially required the understanding of extensive documentations that were readily available, and we therefore refrain from reporting further on this task.

This solution now resolves all of the above issues. The library is now 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. For this interaction to work, the user must be willing to wait for the solution in each step of the process, of which there are usually quite a few. So, fast incremental solving, which is supported by lp_solve, is essential to reduce the waiting time to an acceptable few seconds. As we can actually see in the verbatim output included in Section 3.1, where we used interactive solving for the biquadratic filter application, the time for resolving has always been below 0.0011 seconds, a quite acceptable value.

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 commandline 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 toy program examples.

Table 1 summarises the run-times[¶] of both versions of the combined Hume resource analysis on some program examples: the *non-FFI version* using option `--noapisolve`, which uses lp_solve via the command line to solve the constraint set and an *FFI version* using option `--apisolve`, which calls the lp_solve library through the foreign-function-interface. There is no other difference between the two versions, and the analysis is the current one (March 2009). 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 1 to compare the run-times of the analysis are as follows. The `biquad` application is the biquadratic filter application, discussed in detail in Deliverable D44 (WP8d) [14], and `gravdragdemo` 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 the cycab application, and `pendulum` is the inverted pendulum application, both discussed in Deliverable D33 (WP8c) [13]. The `meanshift` computer vision application is a simple lane tracking program. Finally, `matmult` is a function for matrix multiplication. This Hume code is unusual in that it has been automatically generated from miniC code as discussed in Deliverable D35 (WP9b) [4]. This code makes heavy use of higher-order functions and of vectors for modelling the state space in the original miniC code. 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 (51–363%). However, one should recall that the commandline version (non-FFI) is required to build the C datastructures holding the

[¶]The performance measurements in Tables 1 and 4 have been performed on a 1.73GHz Intel Pentium M with 2MB cache and 1GB main memory.

| Program | Cost model | *old* Analysis | | *improved* Analysis | |
|---|---|---|---|---|---|
| | | absolute | ratio | absolute | ratio |
| | | *Heap Space Analysis* | | | |
| vector addition | 84 | 196 | 2.33 | 84 | 1.00 |
| vector scan | 193 | 233 | 1.21 | 193 | 1.00 |
| matrix transpose | 288 | 963 | 3.34 | 288 | 1.00 |
| | | *Stack Space Analysis* | | | |
| vector addition | 41 | 63 | 1.54 | 41 | 1.00 |
| vector scan | 42 | 62 | 1.48 | 42 | 1.00 |
| matrix transpose | 69 | 279 | 4.04 | 69 | 1.00 |
| | | *WCET Analysis* | | | |
| vector addition | 9555 | 29084 | 3.04 | 11101 | 1.16 |
| vector scan | 25454 | 38008 | 1.49 | 25976 | 1.02 |
| matrix transpose | 48517 | 117405 | 2.42 | 54531 | 1.12 |

Table 2: Analysis Results for Vector-based Programs

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 increase by the same amount as the time spent on LP solving.

The overall speedup is largely varying for our program examples (1–36%), but with the overall runtime 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 runtime could be reduced by an impressive 25%, or roughly 20 seconds.

We therefore conclude that the calling the lp_solve library through the FFI is also beneficial for programs where LP solving actually requires a significant amount of time, in addition to making interactive solving possible at all, as mentioned earlier.

## 5 Improving bounds for vector-based programs

One shortcoming, identified for the old version of the analysis, was the representation of some builtin data structures in the Schopenhauer (Core-Hume) intermediate language. This resulted in a conservative, over-estimation for costs, in particular of heap consumption, for applications generating vector structures. In the improved version of the analysis the data-type representation has been modified to exactly model the representation in the Hume Abstract Machine (HAM). All builtin functions operating on these data structures, which are represented as automatically generated, explicit Schopenhauer code, have been adjusted accordingly. For important cases, such as the builtin function for vector indexing, time bounds have been inferred directly from the generated machine code through the **aiT** tool, thus further improving the overall resource bounds. For most cases, the resource bounds of builtin functions are inferred through analysis of the explicit Schopenhauer code.

Table 2 shows the improvements in the resource bounds for simple vector based programs: vector addition performs component-wise addition of two vectors; vector scan implements the higher-order scan function (a fold that also records intermediate values) and applies it to the addition function; and finally a matrix transpose function. The columns in the table first record the program, then the costs as predicted by the cost model and then compare the results of the following two versions of the analysis: the *old* version (as of October 2008) and the *improved* version (incorporating all changes described in this deliverable). As can be seen from the absolute values, all bounds have been reduced in the improved version of the analysis. Most notably, the bounds obtained for vector addition and matrix transposition have been improved drastically. For the heap bounds this is mainly due to the change in representing vectors, which now directly reflects the representation in the HAM. For time, additionally the improved bounds on individual HAM instructions causes a reduction in the inferred bound. The biggest improvement can be observed for the matrix transpose function, which uses 2-dimensional vectors, and repeated applications of the builtin function `update`, which has a tighter bound in the improved analysis. Both factors contribute to the reduction of space and time bounds. The ratio of the improved analysis results over the values obtained from the cost model, shown in the last column of the table, confirm the validity of the improved bounds: for heap and stack the bounds are exact; for time the analysis shows an over-estimation between 2 and 16% relative to the cost model, which is in turn a safe bound for the exact costs.

## 6 Enhancing ancillary analyses

Several minor analyses have been added in order to extract specific program information that helps to improve the quality of the resource inference. These did not only involve the prototype analysis itself, but also the Hume compiler as well as more fine grained used of the **aiT** tool.

### 6.1 New analyses performed by the Hume compiler

Some of ancillary analyses are performed by the Hume compiler itself, since they depend on knowledge that is only accessible during the compilation process. The information in then passed on through the analysis by annotations added to the Core-Hume format. There are three such additions.

a) An analysis of all call sites in the program provides an important input for the WCET analysis, since the WCET of the HAM return operation varies significantly with the overall number of call sites. This number is only known to the compiler and cannot be deduced from the Core-Hume file. Therefore, this number is now included in the header of the Core-Hume file, making it accessible to the analysis, which uses this value to establish a tighter bound on HAM return operations.

b) A frame-depth analysis determines the stack frame, in which a variable is located. This information is used by the WCET analysis to determine the costs of variable access, which may vary with the frame-depth accessed. Each variable within the Core-Hume file is now annotated with a frame-depth, which then allows the WCET analysis to select the corresponding cost parameter as determined by the **aiT** tool.

c) A syntactic analysis of the structure of case-patterns provides information on the maximal number of variables needed in the pattern binding. This affects the size of the stack frames allocated, allowing the analysis to account the accurate stack frame sizes in all cases now. Again, the information is conveyed through an annotation in the Core-Hume code.

d) The compiler now also marks special function applications, such as tail calls, within the Core-Hume file. The analyses have been improved to take advantage of the function call kind, correctly attributing the cost for tail calls, under- and over-applications. Previously, safe over-approximations have been applied by the analysis.

### 6.2 Value range analysis

The experimental numeric value range analysis, as part of the resource inference, has been strengthened considerably, since this experimental analysis has proved useful in order to cost program examples which otherwise would have super-linear resource consumption.[‖]

Previously, the value range analysis only covered floating point numbers and the addition and subtraction operations on them. Any other operation, such as multiplication caused the value ranges to be undetermined. Now a much wider range of language constructs is covered: all numeric base types and booleans can now carry a value range and these range annotations can now transcend through operations such as exponentiation or pattern matching.

Another important aspect that was improved was the safety of ascribing potential to numeric types in this manner. Several bugs were removed in the process, as well as allowing the programmer to specify primitives within the source code to guide the distribution of potential by numeric types. See Section A.2.3 for more details.

However, one must conclude that it is highly desirable to replace this simple value range analysis by a much more advanced (an complicated) sized type analysis, such as described in [17], or an even more general dependent type analysis, such as [2], as an ancillary analysis for the main amortised analysis. Our experiments thus far has led us to strongly believe that the amortised analysis could make good use of the information provided by such methods, especially since the information provided is of a complementary nature, and amortised analysis and sized types excel on different classes of algorithms. In particular, it seems that a marriage of both methods would allow the inference of super-linear resource bounds, without abandoning the highly efficient LP solver technology. However, this extensive work must be left for future research

## 6.3 Increased precision for floating point operations

Operations on floating point numbers on the Renesas M32C/85U processor are implemented in software, rather than primitive hardware operations. Thus the worst-case execution time for floating point operations calculated by the **aiT** tool had been unsatisfactory, leading to an over-prediction of 88% as compared to the worst measurable runtime, observed in Deliverable D16 (WP3c) [7], Section 4.1.

Of course, the worst measurable runtime can be far away from the actual WCET, since it is generally infeasible to try all possible machine states of a program to determine its WCET. After all, this is our motivation for creating this analyses! Nevertheless, we tried a traditional manual analysis of the generated machine code to examine the over prediction. This inspection was able to find tighter loop bounds as predicted by the **aiT** tool. Furthermore, improvements made by AbsInt GmbH on the **aiT** tool also allowed to predict tighter loop bounds.

Overall, this resulted in an improvement for the bound on the WCET for floating point operations. In particular, the bound on WCET for floating point addition could be reduced from 1106 clock cycles to 934 clock cycles and for floating point subtraction from 1112 to 938 clock cycles. The bound on WCET for the other operations remained unchanged, but the lower absolute values, e.g. 356 clock cycles for multiplication, indicate that the bounds are actually already quite accurate.

We therefore revisit the List folding example from Deliverable D16, Section 4. The program example simply sum up a list of floating point numbers, implemented through higher-order list folding. Of course, the number of floating point operations in this program is relatively small, therefore the overall impact is likely to be small too.

The new cost formula provided is $1228 + 2532 * n$, where is $n$ is the length of the list of floating point numbers to be added up. Compared to the previous bound of $1071 + 2610 * n$ we see the expected decrease in runtime proportional to the length of the list, but also an increase of the fixed cost by 157 clock cycles. This increase is due to the fact that the new analysis also accounts for the overhead of wrapping a Hume expression within pseudo-box. Recall that Hume programs consist of a number of boxes, which are wired to each other. If the Hume compiler is given an expression instead, an invisible

[‖] An input size is usually the number of nodes in the input data structure. For numeric values, this is always one. The value range analysis allows us to take the actual numeric value as a “size” that can be referred to in the cost formula.

| list- | | old analysis | | new analysis | |
|---|---|---|---|---|---|
| size | measured | absolute | ratio | absolute | ratio |
| 0 | 1008 | 1071 | 1.06 | 1071 | 1.06 |
| 1 | 2305 | 3681 | 1.60 | 3603 | 1.56 |
| 2 | 3696 | 6291 | 1.70 | 6135 | 1.66 |
| 3 | 5083 | 8901 | 1.75 | 8667 | 1.71 |
| 4 | 6477 | 11511 | 1.78 | 11199 | 1.73 |
| 5 | 7874 | 14121 | 1.79 | 13731 | 1.74 |
| 6 | 9271 | 16731 | 1.80 | 16263 | 1.75 |
| 7 | 10668 | 19341 | 1.81 | 18795 | 1.76 |
| 8 | 12072 | 21951 | 1.82 | 21327 | 1.77 |
| 9 | 13479 | 24561 | 1.82 | 23859 | 1.77 |
| $\infty$ | | 1 | $\leq 1.88$ | | $\leq 1.80$ |

Table 3: WCET results for summation by list folding on the M32C/85U

box is created around it. This had not been taken into account in the earlier analysis, but is accounted for now, thanks to further improvements to the prototype implementation. The additional cost incurred is exactly 157 clock cycles. Since the measurement performed for Deliverable D16 did also exclude this cost, we have subtracted it from the comparison between the old and new analysis shown in Table 3.

## 7 Conclusion

We now address each point that we had made in the proposal for Deliverable D43:

***enhance the outputs of our analyses ... :*** The analyses now produce closed cost-formulae that express the worst-case resource cost in a linear dependence on the input sizes. These formulae are further elaborated in natural language. This relieves the user from understanding the much more complicated annotated types used before. (Section 2)

***extend the analysis to deal directly with vectors and other Hume data structures:*** The analyses now produce tight bounds for the primitive vector operations of Hume. The bounds on heap- and stack-space usage are now exact for the example programs studied. An over-estimation of 16% for WCET over the worst-case observed by the cost-model is achieved, a quite encouraging result. (Section 5)

***improve ancillary analyses ...:*** The Hume compiler now embeds additional resource relevant information known to the compiler in the header of the file generated by the translation to Core-Hume. Resource relevant information such as the number of return labels, frame-depth annotations and frame sizes have thus become accessible to the analysis, therefore improving the overall quality of the bounds produced.

Furthermore, we greatly enhanced the value range analysis performed in the preprocessing step of the prototype implementation, both in capabilities and safety.

Also, we refined the WCET bounds for floating point operations obtained through the **aiT** tool, again decreasing the over-estimation of the worst-case. (Section 6)

***modify our tools to allow the user to guide the solution of generated constraints:*** The user is now capable of interactively browsing through the full solution space as described by the raw constraint set. This allows the user to obtain even tighter bounds on resource usage by making

| Program-example | Constraints Number | Constraints Variables | Old run-time | New run-time | Speedup (old/new) |
|---|---|---|---|---|---|
| biquad | 2956 | 5756 | 1.43s | 1.43s | 1.00 |
| cycab | 3043 | 6029 | 3.36s | 2.75s | 1.22 |
| gravdragdemo | 2692 | 5591 | 3.29s | 2.14s | 1.54 |
| matmult | 21485 | 36638 | 181.11s | 84.17s | 2.15 |
| meanshift | 8110 | 15005 | 13.88s | 11.01s | 1.26 |
| pendulum | 1115 | 2214 | 0.76s | 0.67s | 1.13 |

Table 4: Change of Analysis run-time between October 2008 and March 2009

use of external knowledge that cannot be gleaned from the program code itself, such as which inputs can be expected to be smaller than others. (Section 3)

***improve the quality of the resource bounds that are produced by the analysis:*** This overall goal has been achieved by the various improvements in concert, as clearly demonstrated by the improvements shown for the revisited program examples in Sections 3.1, 5 and 6.3.

**Additional measures for robustification:** Additionally, the stability of the analysis has been improved in several ways. The type checker for the Schopenhauer intermediate language now performs more checks, and thus errors are caught earlier. Here, our choice of defining a typed intermediate-language as the input language for the analysis pays off in terms of robustness of the entire system.

In order to ensure the robustness of the analysis, we have added an extensive test-suite of Hume programs, together with expected values for the resource analysis. On the one hand, this helps as a regression-test-suite to quickly identify problems introduced by changes to the compiler or the analysis. On the other hand it is a sanity check of the analysis results, at least for those programs where the results have been validated by code examination or measurements.

While the current version of the analysis has been made significantly more robust, some restrictions to the input language, not specified in the Hume report, remain. Most of these are coding guidelines, which we want to promote anyway, such as providing types for all top-level definitions or naming conventions. They are discussed in Section A.4.1 of the appendix.

The integration of the LP-solver, 4, also enhances the robustness of the prototype analysis and moreover decreases the runtime of the analysis significantly. All improvements together lead to the performance increase shown in Table 4, comparing the prototype analysis at the beginning of the work performed for Deliverable D43 and now, March 2009, showing an impressive performance improvement of 115% for the only program example having a significant runtime, the `matmult` program.

## References

[1] 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

[2] Edwin Brady and Kevin Hammond. A dependently typed framework for static analysis of program execution costs. In *Implementation of Functional Languages (IFL) 2005*, volume 4015 of *Lecture Notes in Computer Science*, pages 74–90, Berlin/Heidelberg, 2006. Springer. 6.2

[3] The Glasgow Haskell Compiler. `http://haskell.org/ghc`. 4

[4] G. Grov and G. Michaelson. Destructive Update and Dynamic Data Structures. EmBounded Project Deliverable, August 2008. Deliverable D35. 4

[5] S. Jost. Prototype Implementation of Space Analyses. EmBounded Project Deliverable, February 2007. Deliverable D13. 1

[6] S. Jost. Prototype Implementation of Time Analysis. EmBounded Project Deliverable, February 2007. Deliverable D15. 1

[7] S. Jost and K. Hammond. Validation of the Prototype Worst-Case Executation Time (WCET) analysis. EmBounded Project Deliverable, October 2007. Deliverable D16. 1, 6.3

[8] S. Jost and K. Hammond. Validation of Space analyses. EmBounded Project Deliverable, October 2008. Deliverable D30. 1

[9] S. Jost and H-W. Loidl. *Art3 Manual.* St Andrews University and Ludwig-Maximilians University, Munich, 2009. `http://www-fp.cs.st-andrews.ac.uk/embounded/software/space-analysis/art3Manual.pdf`. (document)

[10] S. Jost, H-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D11. A

[11] S. Jost, H-W. Loidl, and K. Hammond. Report on Stack-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D5. A

[12] S. Jost, H-W. Loidl, and K. Hammond. Report on WCET Analysis. EmBounded Project Deliverable, February 2007. Deliverable D14. A

[13] N. Scaife, H-W. Loidl, G. Michaelson, and J. Sérot. Evaluation of Hume and the Hume Methodology. EmBounded Project Deliverable, September 2008. Deliverable D33. 1, 4

[14] N. Scaife, H-W. Loidl, G. Michaelson, and J. Sérot. Costing-by-construction Exemplar. EmBounded Project Deliverable, March 2009. Deliverable D44. 1, 4

[15] J. Sérot and N. Scaife. Real-time Computer Vision Algorithms. EmBounded Project Deliverable, January 2008. Deliverable D27. 1

[16] Jocelyn Serot and Norman Scaife. Real-time testbed applications. EmBounded Project Deliverable, September 2006. Deliverable D7. 1

[17] Pedro Baltazar Vasconcelos. *Space cost analysis using sized types.* PhD thesis, School of Computer Science, University of St Andrews, November 2008. 6.2

# A art3 Manual

This is the user manual for the art3 prototype Amortised Analysis, produced during the EmBounded research project and implemented in Haskell for the Glasgow Haskell Compiler (version 6.4.2 through 6.10.1).

This document is not meant to teach anything about the underlying principles of the analysis or how to interprete its result (for these things see the EmBounded deliverables on the analyses: D5 for stack space [11], D11 for heap space [10], D14 for WCET [12]). This document will only talk about the technical issues arising in the implementation, such as command line options, etc.

## A.1 List of Options

`--listRK` List all known resource metrics and exit

`--infoRK p1..pn` List detailed information about a given resource metric and exit

Use this option to obtain more information about the particular sub-metric codes reported by option `--listRK`. The output given when using this function is also stored inside the files constraints.lp and constraints.solved.

`--RK p1..pn` Analyse for specified cost metric, with given parameters.

If parsing your model does not work correctly, specify option `--end` directly afterwards as a delimiter.

`-H, --RKH` Analyse for default heap-space metric, HumeHeapBoxed.

`-S, --RKS` Analyse for default stack-space metric, HumeStackBoxed.

`-T, --RKT` Analyse for default time metric, HumeTimeM32.

`--end` Ignored option. Useful as a delimiter after options taking parameters.

`--retlab p1..pn` Specifies number of return labels in HAM code. Default: 50. **WARNING: Affects time costs! This value must be correct!** It can be gleaned from the .c file, the number of cases in the switch statement found under "`_KY16_humeReturn`". In the future, this number is supposed to be transmitted by the phamc-ann in the .art3 file.

`-z` Attribute cost of zero everywhere, unless inline signal received.

This option is only useful in conjunction with signals, described in Section A.2.2. The cost metric is replaced by an all-zero metric, and signals can be used to apply the specified metric to certain code fragments to analyse the cost of the fragment on its own, but within the surrounding context. For example, if all input is wired into box A and all output is wired from box C, but we are only interested in the costs of box B inbetween, then using signals and `-z` can be used to express the cost of running box B in the terms of the input of A and the leftover potential in terms of the output of box C; except for all scheduling costs.

`--nnp` Do not assign potential to numeric values (default) The opposite of option `--nnp`, see comment there.

`--ap` Assign potential to numeric values (experimental) Assigning potential to numerical types is sometimes needed to successfully analyse some programs that cannot be analysed otherwise. The easiest examples that benefit from this option would be `repeat :: A -> Int -> [A]` and `length :: [A] -> Int`. However, the numeric potential is still experimental and not yet fully tested. See section A.2.3 for more information.

`--bb` One big box, i.e. a matching potential must be transmitted on each internal wire. This is useful if more than one box run is to be considered.

`--igbo` Ignore all box declarations

`--igex` Ignore main expression

`--jfun` Analyse functions only. Equivalent to –igbo and –igex. This is highly useful to understand why the LP for an entire program is infeasible. Each global function definition is analysed on its own (together with all called functions, which are individually re-analysed in each step to yield the best possible typing; "best" still being best by some heuristic guess, since there is no clear definition of "best").

`--sim` Simultaneously solve all constraints in a single LP-solver call, i.e. all common variables must have the same value.

`--nosim` Solve all constraint set individual with separate LP-solver calls (default). This means that each function/box receives its best type, regardless of the call graph. Be aware that function that uses another function might require a worse type for that subfunction than the one printed for that subfunction (harmless, but might cause confusion in understanding).

`--trcmerge` Produce only one merged tracefile for all traces

`--ngw` Disable ghost-variable warnings

`--nsl` Do not insert slack variables, allows faster feasibility test for LP

**RECOMMENDATION:** Use this option by default, as it will speed up the analysis significantly. Whether or not the analysis is able to determine an annotated typing, is not at all affected by this option, as the constraints fed to the LP-solver are essentially the same (mainly the objective functions is changed). However, the particular result reported will usually be of lesser quality and only occasionally better than without this option.

`-X` Enable miscellaneous experimental features

`--zeroes` Shows "¡0¿" within annotated types; otherwise suppressed

`--nocon` Suppresses constructor names within annotated types

`--nodupwarn, --ndw` Suppress immediately repeated identical warning messages

`-v, -V` Verbose messages

`-h, -?` Print a short help message

`--help` Print an extended help message listing all options

`--version` Print version info

More detailed descriptions should appear here...

## A.2 Tweaking the analysis through the art3-file

### A.2.1 Destructive Matches

The implementation allows art3-programs to be annotated with deallocation directives. The deallocation directives must be part of a `case` pattern and consist of appending `@_` at the end of a constuctor pattern. The deallocation directives are taken for granted, their validity is not checked. However, their use will cause warning messages, unless the resource metric *HeapBoxedDestructive* or *HeapUnboxedDestructive* are used.

### A.2.2 Signals

Signals are expressions within the art3-sourecode, which may influence the analysis. A signal is always written in this manner:

`<>`*code*`;`*string*`<>`*expression*

A signal must always precede a normal art3-expression and comes into effect before and/or after the expression it is bound to is examined by the analysis.

The code is a number, whose digits are interpreted independently. Currently the following signal codes are recognised:

| Code | Type of String | Digit $10^x$ | encodes Type & Meaning |
|---|---|---|---|
| 0 | (Ignored) | All | Ignored |
| 1 | (Ignored) | 0,1 | Permanently switch on normal costing |
| 2 | (Ignored) | 0,1 | Permanently switch all costs to zero |
| 3 | (Ignored) | 0,1 | Toggle whether all costs are zero |
| 5 | Integer | 0,1 | Insert fix cost of $n$ resource units |
| 5 | Integer | 2 | Replace shallow cost for expression |
| 6 | Constraint | 2 | Insert specified constraint |
| 7 | (Cvar,Int) | 2 | Insert specified equation |
| 8 | [(Double,Cvar)] | 2 | Insert Sum = 0 |
| 9 | Any | 0,1,2 | Issue an onscreen warning |
| 6 | Double | 3 | Set lower bound on value for numeric type |
| 7 | Double | 3 | Set upper bound on value for numeric type |
| 9 | (Ignored) | 3 | Expression type forced to 0 potential |

The position of codes 1 through 5 determines whether the switch comes into effect before or after analysing the attached expression, where digit 0 means before and digit 1 after. Note that the meaning of “after analysing the expression” has nothing to do with the control flow of the program: if the signal was send attached to the body of a global function, then “afterwards” refers to all global functions which happen to be analysed after this particular function. Usually the use of signal codes 1,2 and 3 is in conjunction with option `-z` to turn off all costs and then tracking the cost of a certain function or box (in conjunction with option `--bb`) on its own, or vice versa to attribute zero costs to a certain function. Note that it is not possible to only affect certain calls of a function, since each function is only analysed once: Either all calls to a function will incur a cost or none. Sending the signal right before a function call will only affect the standard cost overhead for the call.

**Example** Inserting a cost of 42 after evaluating an expression: `<>50;"42"<>`*expr*.

Ensuring that a numeric type does not carry potential despite option `--ap` was given, e.g. because we now that the value drops below zero: `<>9000,"ignored"<>`*expr*. Note that this also works for expressiosn of non-numeric types.

Codes may be combined, e.g. `<>712;"(\"b007\",69)"<>`*expr*, means that all cost-constants are considered to be zero while analysing the attached expression, *excluding* function calls, and then we revert to the normal cost metric afterwards. Furthermore the Equation $b007 = 69$ is added to the constraints for the expression.

Another useful example is `<>9500;"48"<>`*expr* meaning that the cost of the expression is ignored and assumed to be 48. Note the combination with 9000, for otherwise the potential in the resulting type will always be infinite, since the surface potential for the expression becomes unbounded when signal 500 is received.

### A.2.3 Numeric Potential

As an experimental feature through option `--ap`, we support potential for numeric values. This means that the potential depends on the actual numeric runtime value, which requires a value range analysis. The included value range analysis is very primitive. It does not work across boxes nor function calls. It is incomplete, which means in particular that it is *not safe* to use. In order to ensure safety, the user has to investigate that all values that were assigned a positive potential never reach a value below zero at any time during execution. *Otherwise the analysis' result is meaningless!*

However, the analysis allows the manual insertion of pragmas (signals, see section A.2.2) to help guiding the value range analysis. Signal 9000 can be used to deny potential to an expression of a numeric type. This is always safe to do, so numeric variables, especially function arguments, which cannot be expected to be non-negative. The disadvantage is of course, that the analysis may not succeed anymore due to super-linear costs.

Signals 6000 and 7000 can be used to manually set the bounds for a numeric type. This should be done very carefully, as the validity of the result depends on the correctness of the specified bounds. Note that a numeric value with a negative lower bound can never be assigned any potential, hence a negative value attached to signal 6000 or 7000 will automatically trigger 9000.

Another pitfall lies in increasing the upper bound for a value that already has potential assigned to it, for an increase in value would mean an unjustified increase in potential, rendering the entire analysis useless. However, the analysis will allow it and issue a warning in this case. This can again be avoided by first sending the signal 9000, i.e. `<>7000,"45.5"<><>9000,"nopot"<>`.

## A.3 Tracing and Debugging

By default, the analysis generates several files during a successful run:

**constraints.lp** The LP-solver is given the constraints to solve through a real file and this is it. Hence this is exactly the data that the LP-solver sees and it can be run again on this file directly.

However, the format is human readable and contains a great deal of information. For example, each constraint is labelled by the line and column number of the instruction wihtin the Schopenhauer code that triggered its generation. The numbers are followed by a 3-4 letter code, detaling the annotated type rule. Furthermore comments yield more information on sharing and function application.

However, during the run of the analysis multiple files with this name may be produced, but only the last instantiation is accessible (each call to LP-solve will require its own incarnation, but these are announced by onscreen warnings).

**constraints.solved** This is similar to constraints.lp, but with the solution of the LP-solver filled in. The original constraints are repeated alongside in comments for convenience. Note that the LP-solver can also be run on this file directly for verifying the solution.

Unlike for constraints.lp, multiple calls to LP-solve will only generate a single file, for all constraints are simply merged together at the end of the analysis, if possible. (If this is not possible, a warning is given.)

**DBGTRACE** The general trace. This file contains a varying wealth of information, but is very difficult to read.

**DBGTRACE_ResConstAll** This file shows all access to the resource metric: For each instruction whose analysis required a lookup in the resource metric its line and column number are logged here, together with the value the cost metric produced. The labels are actually identical to the ones found in constraints.lp. Furthermore, the cost-codes given in detailed as well, together with symbolic cost information (currently only available for Time-Resources).

**DBGTRACE_ResConst** This file is identical to DBGTRACE_ResConstAll, except that all line showing a zero cost are deleted. A highly useful file.

**DBGTRACE_Symbolic and DBGTRACE_TimeCons** Obsolete. These files once showed symbolic resource metric information. They are not produced anymore, since they are now subsumed by DBGTRACE_ResConstAll.

### A.3.1 Tracking Time Constants

It is not only possible to see where which Time Constants was used (through DBGTRACE_Symbolic), but also to track the overall contribution of a single time constant to the overall annotated type. This can be done by using the resource metric "Time Debug", which we describe by example:

`--RK Time Debug`
: All Time Constants are mapped to 1.

`--RK Time Debug Treturn`
: The time constant `Treturn` is mapped to one, all others to zero.

`--RK Time Debug 42 Tcopyarg`
: Time constant `Tcopyarg` is mapped to 42, all others to zero.

`--RK Time Debug 69 Tcopyinput 3`
: Time constant `Tcopyinput` with parameter 3 is mapped to 42, all others zero.

`--RK Time Debug 1 100 Tgoto`
: Time constant `Tgoto` is mapped to 100, all others are mapped to one.

`--RK Time Debug 2 99 Tcallbprim +. FloatTyp FloatTyp`
: Time constant for float addition is 99, all others are mapped to 2.

Currently all parameters for time constants have to be fully specified, but this could be addressed on request.

## A.4 Translation from Hume to Schopenhauer

### A.4.1 Input language restrictions

The translation of Hume code to Schopenhauer code, as a pre-requisite to resource analysis, imposes some restrictions on the input program that are not part of the Hume specification. Here we document these restrictions.

| | | | | | |
|---|---|---|---|---|---|
| min | max | and | or | div | mod |
| lshl | lshr | ashl | ashr | rotl | rotr |
| bittest | bitset | bitclr | program | module | expression |
| val | box | out | type | data | union |
| int | nat | word | bool | float | string |
| char | true | false | let | glet | in |
| rec | match | with | match | endmatch | bcase |
| fcase | case | of | esac | begin | end |
| if | then | else | raise | within | time |
| stack | heap | | | | |

Figure 3: Schopenhauer Keywords

**Upper-case constructors:** Constructors of algebraic data-types, and *only these*, must start with an upper-case letter.

**No type polymorphism:** In order to yield a suitable format for the resource analysis, all translated functions must have a monomorphic type. The translation generates monomorpic instances for all polymorphic, built-in types. However, it will fail on polymorphic, user-defined types. These have to be manually instantiated to their monomorphic types in the source program.

**Type declarations for functions:** In principle, all top-level defintions have to be accompanied by an explicit type declaration. In many cases Hume's type inference can infer the monomorphic type, and in these cases the translation will succeed. Common error messages from the translation, due to missing type declarations, complain about an unknown type or about having found a polymorphic type.

**No vector size polymorphism:** All vectors must have a fixed, explicitly declared type including a fixed size i.e. size polymorphism over vectors is not supported.

**Deprecated 0-ary functions:** 0-ary functions and constants are deprecated, since these can be problematic for the analysis. Instead, macros should be used when defining constants. In fact, the Hume compiler treats constants like macros, so the costs will be preserved.

**Exceptions:** Exceptions are supported in principle, but have not been tested thoroughly or validated, yet. In particular, raising an exception in a function, which is called from a box, will cause problems. By default, all `raise` expressions are translated to calls of special functions, which encode the exception handler code associated to the box. For uncaught, or expression-level exceptions, generic code is generated.

**Pre-defined keywords:** The symbols in Figure 3 are keywords in Schopenhauer and must not be used as identifiers in the source program. Note that some of these symbols are keywords only in Schopenhauer but not in Hume.

**Pre-defined builtin functions:** The symbols in Figure 4 are built-in functions in Schopenhauer and they must not be used as user-defined functions in the source program. Each function as appended with the monomorphic type of its use. See Section A.5 for a discussion on how builtins are translated in general. Additionally, for each pair of types $\tau_1$ and $\tau_2$, a conversion function $\tau_1$`To`$\tau_2$ will be generated, if needed.

| | | | | | |
|---|---|---|---|---|---|
| `nth` | `tuple2bundle` | `length` | `lengthAcc` | `len` | `append` |
| `mkVector` | `vecmapn` | `vecmap` | `vecmapn` | `vecmaken` | `vecmake` |
| `vecdef` | `vecdefn` | `vecfoldr` | `vecfoldrn` | `update` | `unsafeUpdate` |

Figure 4: Schopenhauer Built-in Functions

**Limitations of the translation:** Some known problems with the translation of general Hume code to Schopenhauer are documented here:

- Nested patterns in functions and boxes can cause problems in the transformation, due to missing type information. Using explicit case expressions should get around this problem.
- Groups of mutually recursive functions have to be placed in the same `{ ... }` block in Schopenhauer. The experimental option `-X` to the `phamc` should group the functions appropriately. If this grouping doesn't succeed, and the type-checker of the analysis reports an "unknown function" which is part of the set of mutually recursive functions, the generated code must be edited manually to put all these functions into one group.
- Using `vecdef` to define a vector can be difficult to cost, since it is a higher-order function. For the special case, that the initialisation function doesn't depend on the index, e.g. it is a constant for all slots in the vector, the option `--vecdef-hack` can be used with the `phamc` to produce tighter bounds.

### A.4.2 Supported features

a) Let-normal form. The let-normal-form translation takes the AST of the Hume program, and introduces bindings via the ghost-let construct to ensure that arguments to function calls, the header of a conditional etc. are always variables. A ghost-let is semantically equivalent to a let, but differs in the costs.

b) Specialised datatypes are translated into monomorphic standard algebraic types. Builtin lists are translated into the data type `?list`$\tau$, where $\tau$ is the element type. The constructors are `?C1`$\tau$ for cons and `?N1`$\tau$ for nil. Builtin vectors are translated into the data type `?vector1`$\tau$, with the single constructor `?V1` which takes all vector elements as arguments.

c) Distinguished cases are introduced for top-level function and box definitions (definitions by equations). Introduced cases are always single-patterns, i.e. `f(x,y) = e` translates to `f = fcase 2 x of x' -> fc`

   These cases, fcase and bcase, are cascaded into a top-level case, annotated with the maximum number of free variables in all patterns and unannotated ghost-cases.

   All expression-level cases are top-level cases, since the analysis now deals with nested patterns directly.

d) Applications marked as either Over- (`$>`), Under- (`$<`), Exact- (`$$`) and Tail-call-application (`$!`)

e) Wire/Bundle-Wrappers for Box-inputs used, where `Wire a = Maybe a`. A wire constructor is wrapped around each input and output argument of a box, in order to distinguish it from arguments to functions. A bundle constructor is wrapped around all wires of the input and output of a box. These wrappers are only introduced for proper costing of boxes and do not exist in the generated HAM code.

f) Function lifting: Functions are expanded to wrap their results in Wires/Bundles as required. Example:

```
Box
(x,y,*) -> f(x,y)                    -- f delivers bundle
(*,y,z) -> (*,f(y,z))                -- f delivers wire!
(x,*,z) -> let a=f(x,z) in (*,a)     -- f delivers wire!
```

Note if `f` is used in different contexts, then two versions are made (see example `tests/boxes/pump.hume`).

g) Built-in functions are expanded into proper Schopenhauer-code. Known built-in functions are: `length,  vecfoldr, vecdef, vecmap,` ... See Table 4 for all builtin functions.

h) Schopenhauer signals within Hume-code are preserved and passed into Schopenhauer. See Section A.2.2 for a discussion of signals.

### A.4.3 Arity encoding Function Types

It is important to note that, in general, un-/currying alters the cost of the execution:

$$f :: A \to (B \to C) \tag{1}$$

$$g :: (A, B) \to C \tag{2}$$

Given an element of type $A$, we can execute the code for $f$ right away to obtain a function of type $B \to C$; whereas for function $g$ we would need to create a closure storing the element of type $A$ in the heap somewhere. Furthermore, when the element of type $B$ is eventually supplied, the element of type $A$ must be first retrieved from the heap memory again.

Hence, in order to cost the HUME language, we must distinguish the following functions

$$f :: A \to (B \to C) \tag{3}$$

$$f\ a = \ldots; \tag{4}$$

$$\tag{5}$$

$$f :: A \to (B \to C) \tag{6}$$

$$f'\ a\ b = \ldots; \tag{7}$$

since $f'$ will trigger closure creation when it is under-applied, where under-application is determined with respect to the definiting body of a function, but not through the type.

However, in the intermediate language Schopenhauer we must distinguish between the two, since the type is all there is to distinguish arguments of function types in higher-order functions. Otherwise we would loose precision of the analysis.

Therefore we express the arity of functions within their type as follows

$$f :: A \to B \to C \tag{8}$$

$$f' :: A \overset{\&}{\to} B \to C \tag{9}$$

Note that this rigid type distinction only affects Schopenhauer, but not HUME. Therefore the translation from HUME to Schopenhauer has the problem of deciding which type is required for functions like

$$app\ h\ x = hx; \tag{10}$$

which allows either type of $f$ or $f'$.

The current solution is to simply duplicate the definition and provide the function twice during the translation process. See `tests/t/ho14.hume` for a program example.

### A.4.4 Planned features

a) Grouping functions according to call-graph (experimental option -X available).

   **WORKAROUND:** Edit the Schopenhauer code to wrap each function definition wihtin its own pair of curly braces. This will allow a larger range of programs to be successfully analysed, speed up the analysis and produce better quality results. If the analysis reports an error, then some functions must be put into the same block of curly braces or the blocks must be reorderd. Only mutual recursive functions should be within the same block of curly braces. All functions used by another function must be in a preceding block.

b) Experimental destructive patterns are not allowed in Hume, but only in Schopenhauer.

   **WORKAROUND:** Append "`@_`" to all patterns that should be destroyed after a successful match. Note that this is currently only affecting constructor patterns, but this might be easily extended if there is any demand for such a feature.

## A.5 Built-in functions

Built-in functions and primitives are treated as follows:

a) Builtin functions are made explicit in Schopenhauer by the phamc-ann and are thus analysed as all other normal functions (e.g. `length, vecdef, vecmap,`...). See Table 4 for a list of all Schopenhauer builtin-functions.

b) Primitives remain as *exact* function applications in Schopenhauer, but are parsed as unary (prefix) or binary (infix) operators (e.g. `sqrt, sin, tan,`...). The analysis will see those exactly like other unary or binary operators (e.g. `+,-,*,>,not,==,`...), which do not involve the exact application symbol `$$` within their syntax.

The latter require a special treatment within the analysis, such as a bound for WCET on `sqrt`, while the costs of the former can be derived by the analysis.

Furthermore, the latter kind is not allowed to deal with constructor types and is greatly restricted in dealing with potential assigned to numeric types. In addition, repeated use of these functions must be permitted.