Users Guide: a Resource Analysis for Hume

Steffen Jost

Hans-Wolfgang Loidl

Revision History

Revision $Revision: 1.3 $ 28 Aug 2009 Revised by: hwl
Version for Summer School on Advances in Programming Languages

This document (http://www-fp.cs.st-andrews.ac.uk/embounded/pubs/manuals/users-guide/index.html) gives guidance on how to use the amortised-cost based resource analysis for Hume. *This is a very early draft.*

1. Introduction

A central design goal for Hume is the predictability of resources. The main tool that is provided to the developer of Hume programs is a static analysis, inferring bounds on resource consumption. Resources that can be handled are heap, stack, execution-time and function calls.

This document aims to give guidance on how to use the resource analysis for Hume. It is not an introduction to Hume programming, nor does it discuss the techniques used in the analysis to deliever these bounds.

Other useful resources on Hume are:

- An overview paper on the Hume language (http://dx.doi.org/10.1007/b13639).
- Papers on Hume (http://www-fp.cs.st-andrews.ac.uk/hume/papers/index.shtml).
- An on-line version of the resource analysis (http://www-fp.cs.st-andrews.ac.uk/embounded/software/cost/cost.cgi).
- The Hume web page (http://www.hume-lang.org).
- Hume example programs (http://www-fp.cs.st-andrews.ac.uk/hume/hume-examples/).

In this document we first give a quick introduction how to analyse Hume programs (Section 2). More detailed worked examples explain how to interpret the results of the analysis and how to use them in developing the Hume code (Section 3). Advanced features of the resource analysis, in particular helping

to explain the result of the analysis, are discussed in Section 4. The options available to the analysis are discussed in Section 5. Internal aspects of the analysis that might be of relevance to the user are discussed in Section 6. Finally, we give information on the system including profiling, tools etc (Section 7). Information on how to install the analysis is available in Section 8.

2. A Quick Introduction to Using the Resource Analysis for Hume

2.1. Basic Usage

Assume you have written a Hume program for calculating the sum of a list of floating-point numbers:

```
type _float = float 32;

data flist = Cons _float flist | Nil;

sum11 :: flist -> _float;
sum11 (Nil) = 0.0 ;
sum11 (Cons f fs) = f + (sum11 fs);

expression sum11;
```

To perform an analysis of the heap consumption of the program in `sum11.hume` type the following command: **phamc-an -H -R sum11.hume**

As a result you should see the following:

```
ARTHUR3 typing for HumeHeapBoxed:
 0, (flist[Cons<2>:float,#|Nil<2>]) -(0/0)-> float ,0
```

This is the extended type of the main expression in sum11.hume, which encodes resource information by weights attached to the constructors. This output should be read, "for a list of length n, the heap consumption of the main expression is 2*n+2".

While it is possible to do resource analysis in one go, usually, it is done in two steps, first generating an intermediate program in ARTHUR3 (.art3) or Schopenhauer, notation and then calling the resource analysis, proper:

- **phamc-an -H -r sum11.hume > sum11.art3**
- **art3 -H sum11.art3**

This yields the same type as above.

We can also infer WCET bounds on the program by calling: **art3 -T sum11.art3**

The resource analysis, i.e. the art3 executable, has many options to tune its behaviour. Check the ART3 Manual (http://www-fp.cs.st-andrews.ac.uk/embounded/software/DISTS/art3Manual.pdf) (also available in `docu/art3Manual.pdf` of the distribution) for details. Both phamc-an and art3 have a --help option.

One of the most useful options is **--speak** (see Section Section 4.1), which delivers an explanation of the resource consumption encoded in the annotated type: **art3 -H --speak sum11.art3** which yields the following output:

```
ARTHUR3 typing for HumeHeapBoxed:
0, (flist[Cons<2>:float,#|Nil<2>]) -(0/0)-> float ,0

Worst-case Heap-units required in relation to input:
2 + 2*X1
  where
      X1 = number of "Cons" nodes at 1. position
```

immediately delivering the linear bound, in terms of the input list length, discussed above.

3. Worked Examples of Hume Resource Analysis

In this section we present several worked examples of using the resource analysis, thereby showing various aspects of the analysis.

3.1. Integer Examples

The resource analysis attaches cost information to constructors of data types in the program. For programs iterating over integers this basic design is not sufficient because integers have no constructors. The workaround is to use the flag **--ap**, which attaches size information to integers in the program and propagates this information. To work properly it must be guaranteed that the integers, over which iterations are performed, are never negative. Section 5.2 discusses safety issues realted to this option.

Another important issue to consider is that the integer variable, over which iteration is performed should always *count down* so that the analysis can find a bound. This might require some restructuring of the code, as in the following example, where an explicit counter `z` is introduced for this purpose.

```
type num = int 16;
```

```
gcd' :: num -> num -> num -> num;
gcd' x y z = if (y==1)
               then x
               else gcd' y (x mod y) (z-1);

gcd :: num -> num -> num;
gcd x y = if (x < y)
            then gcd' y x y
            else gcd' x y x;
```

For the straightforward `gcd` computation above we get the heap bound below, after typing **phamc-an -H -r gcd12.hume > gcd12.art3** and **art3 -H --ap --speak gcd12.art3**

```
ARTHUR3 typing for HumeHeapBoxed:
0, (int<10> -&-> int<10>) -(8/2)-> int ,0
Worst-case Heap-units required in relation to input:
8 + 10*Z1 + 10*Z2
  where
      Z1 = runtime value of the 1. NTint
      Z2 = runtime value of the 2. NTint
```

3.2. List Examples

Consider the following simple list reversal function (available in the examples of the distribution as `ra16.hume`).

```
type num = int 16;

data num_list = Nil | Cons num num_list;

revApp :: num_list -> num_list  -> num_list;
revApp acc Nil = acc;
revApp acc (Cons x xs) = revApp (Cons x acc) xs;

reverse :: num_list  -> num_list;
reverse xs = revApp Nil xs;

expression reverse;
```

The first step is to generate Schopenhauer code from the Hume code. This is done like this: **phamc-an -H -r ra16.hume > ra16.art3**

The second step is to run the resource analysis, proper, on the Schopenhauer code, like this: **art3 -H --speak ra16.art3**

The result of the analysis will be an annotated type, encoding resource information, together with an explanation:

```
ARTHUR3 typing for HumeHeapBoxed:
 0, (num_list[Nil|Cons<4>:int,#]) -(2/0)-> num_list[Nil|Cons:int,#] ,0

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

In this example the heap consumption is linear in the input list: if the input list has n elements, heap consumption will be 2 + 4*n.

3.3. Operations on Trees

3.3.1. Insertion into a Red-Black Tree

The resource 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. The following code is taken from Okasaki's textbook (http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#cup98).

```
program -- RedBlack

type num = int 16;
data colour = Red | Black;
data tree = Empty | 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 Bla
balance Black (Node Red a x (Node Red b y c)) z d = Node Red (Node Black a x b) y (Node Bla
balance Black a x (Node Red (Node Red b y c) z d) = Node Red (Node Black a x b) y (Node Bla
balance Black a x (Node Red b y (Node Red c z d)) = Node Red (Node Black a x b) y (Node Bla
balance c a x b = Node c a x b;

ins :: num -> tree -> tree;
ins x Empty = Node Red Empty x Empty;
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);

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


expression insert;
```

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 `insert` function in the above code 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.

In a first step we translate the Hume program into a Schopenhauer program, using the command **phamc-an -H -r redblack11.hume > redblack11.art3**. The program in `redblack11.art3` is the one we analyse in the following steps.

To infer the heap bound for the `insert` function we use the command **art3 -H --noapisolve --oldarity redblack11.art3**. The important option is **-H** for "heap bound". The following 2 options make the output slightly easier to read but can be omitted in general: **--noapisolve** in this case avoids floating point numbers in the annotated type, and **--oldarity** does not distinguish between two forms of function arrows. All available options are discussed in the art3 manual (http://www-fp.cs.st-andrews.ac.uk/embounded/software/DISTS/art3Manual.pdf). The result of the heap analysis is:

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

This bound expresses that the total heap consumption of the function is 10 n + 18 b + 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 18 b 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 2 balancing operations (see Chapter 13 of Cormen, Leiserson, Rivest, Stein's textbook (http://projects.csail.mit.edu/clrs/)). As expected, capturing these (semantic) constraints is beyond the power of our type system.

In Section Section 4.1.1 we will revisit this example and show how to get even better analysis results for its heap consumption.

The stack bound for the `insert` function, inferred by our analysis is (using the command **art3 -S --noapisolve --oldarity redblack11.art3**):

```
ARTHUR3 typing for HumeStackBoxed:
 7, (int,tree[Empty<8>|Node<13>:colour[Red|Black],#,int,#]) -(14/18)-> tree[Empty|Node<2>:c
```

Similarly the time bound for the `insert` function associates costs to the black nodes in the input tree (using the command **art3 -T --noapisolve --oldarity redblack11.art3**):

```
ARTHUR3 typing for HumeTimeM32:
 298, (int,tree[Empty<2288>|Node<3010>:colour[Red|Black<1830>],#,int,#]) -(474/0)-> tree[Em
```

Finally, the call count analysis for `insert`, using the command **art3 --RK CC --noapisolve -- oldarity redblack11.art3**, yields:

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

This type encodes a bound of 2 n +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 in his PhD Thesis, 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.

3.4. Box-level examples

Resource analysis of box-leve code proceeds in the same way as analysis of expression-level code. In particular, resources are encoded as numeric annotations attached to the input wires of a box. One important difference to the expression level is that on box level an input may not be available at all. This is reflected in the elaboration of the inferred type.

3.4.1. Multiplication by recursion

The following example realises a box calculating the product of two integer numbers. The first version performs the entire calculation on expression level, using a recursive function `mult` that multiplies two numbers using an accumulating parameter as first argument.

```
program -- recmult

-- stream stdin  from "std_in";
stream stdout to "std_out";

type integer = int 64;

mult :: integer -> integer -> integer -> integer;
mult r x 0 = r;
mult r x y = mult (r+x) x (y-1);

box mult_box
in  (i :: integer)
out (i' :: integer, o :: integer)
```

```
match
 (x) -> (x+1, mult 0 x x);

wire mult_box (mult_box.i' initially 0) (mult_box.i, stdout);
```

We analyse the time consumption of this program by first converting the code into Schopenhauer code, using **phamc-an -H -r recmult.hume > recmult.art3**, and then calling the analysis like this **art3 -T --ap --shrtcon --speak recmult.art3**. The output of the analysis is:

```
ARTHUR3 typing for HumeTimeM32:

@BOX mult_box:
    mult_box.i: wire[W<1308>:int<828>|NOVAL]
  ---137/0--->
    mult_box.i':
                wire[W:int|NOVAL]
    mult_box.o: wire[W:int|NOVAL]
Worst-case Time-units required to compute box mult_box once in relation to its input:
  137 + 1308*X1 + 828*Z2
    where
        X1 = one if 1. wire is live, zero if the wire is void
        Z2 = runtime value of the 1. NTint
```

First of all, with the help of the --ap option, it is possible to infer a bound on the overall time consumption. The bound delieverd has three components: a fixed cost of 137, an optional cost of 1308, which is incurred only if an input is available on the first argument, representing the accumulating parameter, and finally a linear component (828) in terms of the value of the second component, over which the recursion is performed.

We can check the concrete time consumption of the `mult`, without the overhead on box level, by using the additional option --jfun to the analysis (**art3 -T --ap --shrtcon --speak --jfun recmult.art3**). This will deliever resource bounds for all functions in the code:

```
ARTHUR3 typing for HumeTimeM32:
mult_LIFTED: (int -&-> int -&-> int<854>) -(544/0)-> wire[W:int|NOVAL]

Worst-case Time-units required to call mult_LIFTED in relation to its input:
  544 + 854*Z1
    where
        Z1 = runtime value of the 3. NTint
```

3.4.2. Multiplication by iteration

As a second version we examine the resource consumption of box-level implementation of multiplication. In this version, we use 2 boxes: the `mult_box` sets up the computation, by initialising the three input wires of the `itermult` box; the latter performs only one iteration of multiplication, using its

fourth input and its first output wire for accumulating the result. First we call **phamc-an -H -r itermult.hume > itermult.art3**.

```
program -- itermult

type integer = int 64;

stream output to "std_out";

-- controls input stream (x) feeding input vals (0,x,x) to itermult box
-- sending result (r) to output, once it's returned from itermult
box mult2
in  (i::integer,  iter'::integer)
out (i'::integer, iter1 ::integer, iter2 ::integer, iter3 ::integer, r::integer)
match
(x,r) -> (x+1,0,x,x,r);

-- computing (*,x,y,*,*,*,*) -> x*y,
-- using the last 3 inputs and first 3 outputs as state via feedback wires
box itermult
in  (i1::integer, i2::integer, i3::integer, iter1::integer, iter2::integer, iter3::integer)
out (iter1'::integer, iter2'::integer, iter3'::integer, r::integer)
match
 (r,x,y,*,*,*) -> (r,x,y,*) |
 (*,*,*,r,x,0) -> (*,*,*,r) |
 (*,*,*,r,x,y) -> (r+x, x, y-1, *);

wire mult2
 (mult2.i' initially 0,itermult.r)
 (mult2.i,itermult.i1,itermult.i2,itermult.i3,output);

wire itermult
 (mult2.iter1,mult2.iter2,mult2.iter3,itermult.iter1',itermult.iter2',itermult.iter3')
 (itermult.iter1,itermult.iter2,itermult.iter3,mult2.iter');
```

We analyse the time consumption of this code by calling **art3 -T --ap --shrtcon --speak itermult.art3** .

```
ARTHUR3 typing for HumeTimeM32:

@BOX mult2:
    mult2.i:    wire[W<1034>:int|NOVAL]
    mult2.iter':
                wire[W:int|NOVAL]
  ---211/0--->
    mult2.i':   wire[W:int|NOVAL]
    mult2.iter1:
                wire[W:int<ANY>|NOVAL]
    mult2.iter2:
                wire[W:int|NOVAL]
    mult2.iter3:
                wire[W:int|NOVAL]
    mult2.r:    wire[W:int|NOVAL]
```

```
Worst-case Time-units required to compute box mult2 once in relation to its input:
  211 + 1034*X1
    where
        X1 = one if 1. wire is live, zero if the wire is void


ARTHUR3 typing for HumeTimeM32:

@BOX itermult:
    itermult.i1:
                wire[W:int|NOVAL]
    itermult.i2:
                wire[W:int|NOVAL]
    itermult.i3:
                wire[W:int|NOVAL]
    itermult.iter1:
                wire[W<859>:int|NOVAL]
    itermult.iter2:
                wire[W:int|NOVAL]
    itermult.iter3:
                wire[W:int|NOVAL]
  ---1322/0--->
    itermult.iter1':
                wire[W:int|NOVAL]
    itermult.iter2':
                wire[W:int|NOVAL]
    itermult.iter3':
                wire[W:int|NOVAL]
    itermult.r: wire[W<687>:int|NOVAL]
Worst-case Time-units required to compute box itermult once in relation to its input:
  1322 + 859*X1
    where
        X1 = one if 4. wire is live, zero if the wire is void
```

As expected, the time bound for the `itermult` box is now fixed, not depending on an input value but only on the availability of data along the fourth input wire, which holds the accumulated result.

3.4.3. Vending machine

```
type cash = int 8;

type _smallint = int 16;
type _int = int 32;

data coins =  Nickel | Dime;
data drinks = Coffee | Tea;
data buttons = BCoffee | BTea | BCancel;

showdrink :: drinks -> char;
```

```
showdrink Coffee = 'C';
showdrink Tea = 'T';

vend :: drinks -> cash -> cash -> (drinks, cash, cash);
vend drink cost v =
   if v >= cost then
        ( drink, v-cost, * )
   else ( *,     v,      * );

-- input handling box

box inp
in ( c :: char )
out ( coin :: coins, button :: buttons )
match
    'N' -> ( Nickel, * )
  | 'D' -> ( Dime,   * )
  | 'C' -> ( *,      BCoffee )
  | 'T' -> ( *,      BTea )
  | 'X' -> ( *,      BCancel )
  | _   -> ( *,      * )
  ;

-- vending machine control box
box control
in  ( coin :: coins, button :: buttons, value :: cash )
out ( drink :: drinks, value' :: cash,  return :: cash )
match
    ( Nickel, *,  v )  ->  ( *, v + 5, * )
  | ( Dime,   *,  v )  ->  ( *, v + 10, * )
  | ( *, BCoffee, v )  ->  vend Coffee 10 v
  | ( *, BTea,    v )  ->  vend Tea 5 v
  | ( *, BCancel, v )  ->  ( *, 0, v )
  ;

box outp
in ( drink :: drinks, return :: cash )
out (d :: char, r :: cash) -- s :: string 15)
match
  ( d, * ) -> (showdrink d, *) -- "Vending " ++ showdrink d ++ "\n"
| ( *, r ) -> (* , r)          -- "Returning " ++ r as string ++ "\n"
;

stream stdout to "std_out";
stream stderr to "std_err";
stream stdin  from "std_in";

wire inp    ( stdin )                          ( control.coin, control.button );
wire control ( inp.coin, inp.button, control.value' initially 0 )( outp.drink,  control.val
wire outp   ( control.drink, control.return )  ( stdout, stderr );
```

We will now illustrate Hume with a simple, but more realistic, example from the reactive systems literature, suggested to us at CEFP 2005 (http://plc.inf.elte.hu/cefp/) by Pieter Koopman: the control logic for a simple vending machine. A system diagram is shown in the figure above. We will show Hume code only for the most important part of the system: the `control` box. This box responds to inputs from the keypad box and the cash holder box representing presses of a button (for tea, coffee, or a refund) or coins (nickels/dimes) being loaded into the cash box. In a real system, these boxes would probably be implemented as hardware components. If a drinks button (tea/coffee) is pressed, then the controller determines whether a sufficient value of coins has been deposited for the requested drink using the `vend` function. If so, the vending unit is instructed to produce the requested drink. Otherwise, the button press is ignored. If the cancel button is pressed, then the cash box is instructed to refund the value of the input coins to the consumer.

First we define some basic types representing the value of coins held in the machine (`Cash`), the different types of coins (`Coins`), the drinks that can be dispensed (`Drinks`) and the buttons that can be pressed (`Buttons`).

Now we can define the `control` box itself. This box uses *asychronous* constructs to react to each of the two input possibilities: either an inserted coin or a button-press. As with the `inc` box, state is maintained explicity through a feedback wire: the `value'` output will be wired directly to the `value` input. For simplicity, the box uses unfair matching, which will prioritise coin inputs over simultaneous button presses. If the cancel button (`BCancel`) is pressed, no drink will be dispensed (shown by `*`), but the internal cash value will be reset to zero and the cash holder instructed to refund the current value of coins held (`value`) through the `return` wire. A timeout has the same effect as explicitly pressing the cancel button.

The control box logic makes use of two auxiliary functions: `vend` calculates whether sufficient coins have been deposited and instructs the `outp` box accordingly; and `showdrink` displays the chosen drink as a single character. Note the use of `*` as a return value in the `vend` function: this is permitted only in positions which correspond to top-level outputs. Note also that the box corresponds to the FSM-Hume level: it uses first-order non-recursive functions as part of its definition.

Finally, we wire the control box to the other boxes shown in the system diagram.

We convert the Hume program to a Schopenhauer program: **phamc-an -H -r vending.hume > vending.art3**. Then we perform an analysis of the heap consumption of the program: **art3 -H --shrtcon --speak vending.art3**.

```
ARTHUR3 typing for HumeHeapBoxed:

@BOX inp:
    inp.c:      wire[W<2>:char|NOVAL]
  ---0/0--->
    inp.coin:   wire[W:coins[Nickel|Dime]|NOVAL]
    inp.button: wire[W:buttons[BCoffee|BTea|BCancel]|NOVAL]
Worst-case Heap-units required to compute box inp once in relation to its input:
  2*X1
```

```
    where
        X1 = one if 1. wire is live, zero if the wire is void


ARTHUR3 typing for HumeHeapBoxed:

@BOX control:
    control.coin:
                wire[W:coins[Nickel|Dime]|NOVAL]
    control.button:
                wire[W<4>:buttons[BCoffee|BTea|BCancel]|NOVAL]
    control.value:
                wire[W<4>:int|NOVAL]
  ---0/0--->
    control.drink:
                wire[W:drinks[Coffee|Tea]|NOVAL]
    control.value':
                wire[W:int|NOVAL]
    control.return:
                wire[W<6>:int|NOVAL]
Worst-case Heap-units required to compute box control once in relation to its input:
  4*X1 + 4*X2
    where
        X1 = one if 2. wire is live, zero if the wire is void
        X2 = one if 3. wire is live, zero if the wire is void


ARTHUR3 typing for HumeHeapBoxed:

@BOX outp:
    outp.drink: wire[W:drinks[Coffee<2>|Tea<2>]|NOVAL]
    outp.return:
                wire[W:int|NOVAL]
  ---0/0--->
    outp.d:     wire[W:char|NOVAL]
    outp.r:     wire[W:int|NOVAL]
Worst-case Heap-units required to compute box outp once in relation to its input:
  2*X1 + 2*X2
    where
        X1 = number of "Coffee" nodes at 1. position
        X2 = number of "Tea" nodes at 1. position
```

We notice that for these simple boxes the heap consumption is in all case fixed, depending only on the availability but not on the size of the input. For the `outp` we know that only one value, either `Coffee` or `Tea`, can be supplied, since they are 0-ary constructors of the same type. Therefore, the maximal heap consumption of the `outp` box is 2. Note that in the `control` box a value of 6 is attached to the output wire `return`. This indicates, that in the case of producing an output along this wire, 6 heap cells, of the maximal bound of 8, are not used, reducing the bound in this case to only 2.

4. Advanced Features of the Resource Analysis

4.1. Elaboration Module

Input dependent bounds on resource usage of programs are only useful if they easily allow to distinguish large classes of inputs having 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 over 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, 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. Of course, this is a byproduct of enabling an *automated inference* in the first place. 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 throwing all inputs which only differ in the first argument into the same cost class. Furthermore we immediately know the execution cost for infinitely many of such input classes.

We now exploit this even further to produce cost bounds expressed in natural language. Thus far, 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, parameterised over a program's input.

We now revisit the results for the red-black tree insertion function from Section Section 3.3.1. Performing heap analysis of the **insert** with the additional option **--speak** produces the following output:

```
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 20*X1 + 18*X2 + 10*X3 is obviously much more compact and easy to understand. We are directly told that X1 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 Section 3.3.1 and we will address this in the following subsection.

A further optimisation implemented in our elaboration module is the recognition of list-like types, that have the property that all constructors have precisely one self-recursive argument, except for a single constructor having none. 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 can be moved outwards, e.g.

```
(llist[Nill<1>|Consl<2>:ilist[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 "Consl" nodes at 1. position
        X2 = number of "Cons" nodes at 1. 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. Note that this useful simplification naturally incurs a slight loss of information. The annotated type expresses that 3 resource units are needed whenever the end of the inner list is reached. However, the program may *sometimes* choose to abort processing a list to the very end, in which case those 3 resource units are not needed. While our analysis must produce a guarantee on the resource usage for all cases, this simplification means no loss for the bound produced. Nevertheless it is conceivable that a programmer might sometimes make use of this additional knowledge about the resource behaviour of the program. However, we believe that the majority of cases benefits from this simplification.

4.1.1. Optimizing the bound for red-black tree insertion

We again revisit the red-black tree example from Section Section 3.3.1, this time to show how the interactive optimisation of the solution works. Invoking our analysis for the heap space metric as follows **art3 -H -a redblack11.art3** delivers the following prompt:

```
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 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 desired solution. 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.

5. Options to the Resource Analysis for Hume

This section shortly summarises the main options for the resource analysis. A full discussion is given in the ART3 Manual (http://www-fp.cs.st-andrews.ac.uk/embounded/software/DISTS/art3Manual.pdf).

The **--help** option gives an up-to-date list of all available options.

5.1. 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 \texttt{--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. \textbf{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 “\verb+_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~\ref{signals}. 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 \texttt{-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 `\verb+length :: [A] -> Int`. However, the numeric potential is still experimental and not yet fully tested.
- **--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 \textbf{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

5.2. Notes on 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 Sction Section 5.3) 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"<>`.

5.3. Signals

Signals are expressions within the Schopenhauer 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.

A table of available signals, together with examples of their use, is given in the ART3 Manual (http://www.hume-lang.org).

6. Internal aspects of the Resource Analysis

6.1. Translation from Hume to Schopenhauer

This section discusses aspects of the internal program transformation from Hume to Schopenhauer, that are relevant for the user of the resource analysis.

6.1.1. 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.

- *Upper-case constructors:* Constructors of algebraic data-types, and \emph{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 \verb+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 and functions:* Be aware of The symbols in Figure~\ref{fig:keywords} are Keywords and builtin function in Schopenhauer 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. See the ART3 Manual (http://www-fp.cs.st-andrews.ac.uk/embounded/software/DISTS/art3Manual.pdf) for full details

6.1.2. 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-an` 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-an` to produce tighter bounds.

6.2. Cost tables

The inference engine of the resource analysis is generic in the resource to be analysed. It is parameterised with a cost table, that models basic costs of the instruction in the Hume Abstract Machine. This section will eventually discuss the available cost tables.

6.3. Hume Abstract Machine

Hume code is compiled to code of the (Hume Abstract Machine (http://www-fp.cs.st-andrews.ac.uk/hume/papers/HAM.ps)).

7. System

Empty

8. Getting the Resource Analysis

The Hume Resource Analysis is distributed in binary format separately from the Hume compiler itself. Versions for different architectures are available: Hume language our web page (http://www-fp.dcs.st-and.ac.uk/hume/downloads)

8.1. On-line resource analysis

An on-line version of the resource analysis (http://www-fp.cs.st-andrews.ac.uk/embounded/software/cost/cost.cgi) is available. You can select from a number Hume example programs, or paste in your own Hume program, and directly perform resource analysis.

8.2. Installing

You can download the Hume Resource Analysis from the *Hume software web page* (http://www-fp.cs.st-andrews.ac.uk/embounded/software/).

Currently we ship the analysis in the form of binary distributions, wrapped up as a `.tgz` (tar-ed, gziped binaries) bundles. To download and unpack a distribution with the name `art3-release`, do

```
wget art3-release
tar xvfz art3-release
```

Go into the directory `bin` of the release and create links to the binaries from one of your directories in your path, e.g.

```
cd art3-release/bin
ln -s * ~/bin
```

To test the analysis, go into `examples` and follow the instructions in Section 2.

8.3. Emacs Hume Mode

An emacs mode for Hume programs (http://www.hume-lang.org) is available online. Download it and extract it into a directory in your emacs path, e.g. `~/Elisp`. To install it put the following into your `.emacs` file:

```
(setq load-path
      (append
`(,(expand-file-name "~/Elisp/hume-mode"))
load-path))

 (setq auto-mode-alist
(append '(("\\.hume$" . hume-mode) ("\\.art3$" . hume-mode))
   auto-mode-alist))
 (autoload 'hume-mode "hume-mode"
    "Major mode for editing Hume programs." t)
```

8.4. Hume example programs

A set of example programs, included in the binary distribution, is also available on-line: Hume example programs (http://www-fp.cs.st-andrews.ac.uk/embounded/pubs/manuals/examples/)