Chapter 1

An application of Hume analysis to imperative programs with pointers

Gudmund Grov & Greg Michaelson[1], Hans-Wolfgang Loidl [2], Christoph Herrmann & Steffen Jost & Kevin Hammond [3]
Category: Research

Abstract: The worst case execution time (WCET) analysis of an imperative language (miniC*) with pointers and memory allocation, through direct translation to a functional subset of Hume, is discussed. The formal translation rules are presented and comparisons of empirical miniC* measurements with corresponding translated Hume WCET predictions are made.

1.1 INTRODUCTION

This paper aims to provide WCET bounds for a minimal C language with explicit dynamic memory management. This is achieved by translation into our Hume language, for which tried and tested (validated) analyses of resource consumption are available.

Our claim is that the WCET bounds obtained from the generated Hume code give a realistic time profile of the original program.

Hume [9] is a contemporary language which seeks to reconcile high expressivity with strong resource use guarantees. As is well known, Turing Complete

[1]Heriot-Watt University,Riccarton,Scotland,EH14 4AS,UK;
{G.Grov,G.Michaelson}@hw.ac.uk
[2]Ludwig-Maximilians University, D-80339 Munich, Germany;
hwloidl@tcs.ifi.lmu.de
[3]University of St. Andrews, North Haugh, St. Andrews, UK;
{ch,jost,kh}@cs.st-and.ac.uk

(TC) languages have undecidable termination for arbitrary programs. If it is not possible to determine whether or not a program terminates then it it is not possible to determine how long it runs for and hence how much memory it consumes or how much power it requires. Sub-TC languages like those based on primitive recursion may guarantee program termination and hence enable the establishment of strong bounds on resource use, but at the price of compromised expressivity.

The Hume approach has been grounded in a separation of an impoverished coordination layer based on concurrent finite state automata controlled by a variable expressivity expression layer based on modern functional language concepts. Thus, the coordination layer populated with a minimally expressive expression layer of tuples of bits offers precise cost analysis, and varying the types and control structures deployed in the expression layer increases expressivity while reducing costability, but in a principled manner.

Hume now enjoys a stable tool set grounded in strong semantic specifications including a reference interpreter, abstract machine compiler and interpreter, native code compiler, and WCET and space analysers. The base language implementations are highly portable, and there is a well enunciated methodology for instantiating the cost analysers for new CPUs. To date, Hume offers highly accurate analyses for the Renesas M32C CPU and substantial progress has been made towards PowerPC 3 analyses.

Recently, we have been exploring the applicability of the Hume models and analyses to more traditional imperative language settings. Rather than constructing new models and analyses, we have built translators to Hume and then looked at how well costs from Hume static analyses correspond to the instrumented behaviours of the original programs.

Our first experiments[8] were with a very simple imperative language termed miniC, with integers, one-dimensional arrays, assignment, choice, iteration and input/output. We elaborated a set of formal rules mapping miniC constructs to pure Hume expressions, with explicit state passing, which formed the basis of a translator implementation. We then applied the Hume worst case execution time (WCET) analysis instantiated for the Renesas board to translated miniC, and also timed the miniC compiled directly to M32C machine code. Results from a small set of test programs suggest that there is a discernible correspondence between observed miniC behaviour and translation analysis prediction. This gives us confidence that translation from miniC to Hume is preserving key semantic and complexity features of miniC, despite the inevitable mismatch of expressibility.

We now turn our attention to a somewhat more elaborate language, miniC*, which is miniC augmented with explicit dynamic memory allocation and pointers. The following sections discuss amortised cost models, the mapping from miniC* to Hume, compare analytic and empirical results, and consider the efficacy of reusing models and analyses for one language with another through translation.

1.2 HUME RESOURCE ANALYSIS

Determining the cost of executing a sequence of instructions can be easy for a given machine model, if each instruction has a constant cost. It is also possible to determine a useful upper bound on the execution costs if the variation in cost is quite small for each instruction, such as bounding the cost of adding two arbitrary integer. However, in practise the cost of many instructions varies significantly, depending on arguments and/or the overall state of the machine. As an extreme example, a function call might return after an arbitrarily large amount of time, which makes it useless to assume the worst case (never). Therefore one needs to consider all possible machine states at the start of the computation and track all possible state transformations during execution, which is a tedious and generally infeasible approach.

The only solution to this problem is to abstract all possible states into a smaller, more manageable classes of equivalent states. Our approach is especially radical, since we will abstract the entire state and represent it by a single, non-negative rational number, referred to as the *potential* of the machine state. Note that we will never actually compute this number, the potential, for any actual machine state other than the initial state. Instead, we examine the effect of each operation on the overall potential and define the *amortised cost* of an instruction $\mathscr{I}$ as a suitable constant such that

$$\textit{amortised cost}_{\mathscr{I}} \geq \text{actual cost}_{\mathscr{I}} - \text{potential before}_{\mathscr{I}} + \text{potential after}_{\mathscr{I}}$$

holds for all possible states, with equality being preferred. The benefit is that determining the amortised cost for a sequence of operation is very easy, since the amortised cost is constant and does not depend on the machine. The actual cost of the entire sequence is then bounded by the sum of the amortised costs plus the potential of the initial state.

This technique is well known in complexity theory and referred to as "Amortised Analysis", pioneered by R. E. Tarjan [28]. However, a significant challenge of this technique is how to design the abstraction of the machine state. This problem was overcome by an automatic inference presented by Hofmann & Jost [11, 12], at the expense of restricting the potential to depend linearly on different sizes of objects within a machine's memory. Hence only programs whose cost can be linearly bounded by their input can be analysed. Nevertheless, many interesting programs can be successfully analysed by this technique as already shown Hofmann & Jost.

Another interesting improvement in the way we apply the amortised analysis technique is that the potential contributed by each memory object is assigned on a per reference basis. Note that we do *not* need to count the references to a particular object, all we need to take care of is the point where aliasing is introduced. This allows us to assign differing potential to states which differ only by some pointers, but otherwise contain identical objects in the memory. The assignment of potential on a per reference basis is a major improvement over previous work, such as the important work of C. Okasaki [22], who resorted to lazy evaluation in

order to circumvent this problem.

For example, deep-copying a list takes time proportional to the length of the list, say 123 cycles per element. The deep-copy operation can nevertheless be assigned an amortised cost of zero, if we require that it receives a reference which contributes 123 potential credits per element. So a list of length 5 would contribute a total 615 credits towards the overall potential. Once the first element has been copied, the pointer moves to the next list element, thereby decreasing the potential contributed by this pointer down to 492. The difference of 123 credits thus "pays" for the cost of copying the first element, and so on.

Automatically performing the analysis means that we first construct a standard typing derivation. Next, each constructor is then assigned a resource variable (ranging over non-negative rational numbers), representing the potential credited by each node of that constructor of that particular type (note that types may differ by their potential only). The analysis generates a set of constraints over those variables, according to the dataflow and the actual cost occurring in each possible path of computation.

Each instruction of the source program is examined precisely once. Loops in the source program are dealt with by identifying some resource variables contained in the constraint set. The generated constraint sets are well behaved and can easily be solved by using a standard LP-solver, such as [1]. In this way, bounds on resource consumption are associated with each expression in the program through their types. The potential annotated types then give rise to a linear closed form expression, depending on the input sizes, which represents an upper bound on the execution costs. Because we have used a type-based approach, we have already formally proven that our analysis will always give a *guaranteed upper bound* on resource consumption.

1.3 SOURCE LANGUAGE: MINIC*

Figure 1.1 shows the abstract syntax of miniC*, where square brackets $[\cdots]$ denotes optional features. It does not allow any function declaration, except the `main` function, which contains the whole program. Since it is a proper ANSI-C subset, all declarations must precede the statements. It supports both integers and pointer to integers with multiple indention.

Both input streams and output streams are supported via `scanf` and `printf`, however, to easy the translation into Hume, a `scanf` must come first, while all `printf` statement are at the end of the code. Both standard assignment, and assignment using the `*` pointer operator is supported, as are conditionals (`if`) with and without `else` branching is supported. Repetition is based on a limited version of a `for`-statement.

Dynamic memory management is supported by `malloc` and `free` [17]. To ensure portability, the `malloc` argument can be multiplied with `sizeof(int)`, and the translation rules assumes that this is included for pointers directly to integers – unless each heap cell is 16 Bytes, which is the size of a miniC* `int`. Finally, normal arithmetic and comparison expression are supported, as are the

program ::= *prelude* `main()` *body*

prelude ::= $include_0 \ldots include_n$ $n \geq 0$
include ::= `# include <`*id*`.h> | # include "`*id*`.h"`

body ::= $\{decl_1 \cdots decl_n\ [scan]\ stmt_1 \cdots stmt_m\ print_0 \cdots print_l\}$ $n,m \geq 1, l \geq 0$
cbody ::= *stmt* | $\{stmt_1 \cdots stmt_n\}$ $n \geq 1$
star ::= *id* | $\star$*star*
decl ::= `int` *star;*
scan ::= `scanf ( "%d" ,` *expr* `);`
print ::= `printf ( "%d" ,` *expr* `);`
caststar ::= `int` | *caststar*$\star$
cast ::= `(` *caststar* `)`
stmt ::= *id* = *expr;* | `if (` *expr* `)` *cbody* | `if (` *expr* `)` $cbody_1$ `else` $cbody_2$
| `for (` $stmt_1$ `;` *expr* `;` $stmt_2$ `)` *cbody* | $\star expr_1 = expr_2$`;`
| $expr_1$ = [*cast*] `malloc(` $expr_2$ [`* sizeof(int)`] `)` | `free(` *expr* `);`
expr ::= *int* | *id* | $expr_1$ *binop* $expr_2$ | `(` *expr* `)` | *cast expr* | $\star$*expr* | `&`*id*
binop ::= `==` | `!=` | `<` | `<=` | `+` | `-` | `*` | `/` | `%`

FIGURE 1.1. miniC* abstract syntax

de-reference $\star$ and reference `&` operators. Overall, miniC* constitutes a Turing-complete pointer language.

As illustration of miniC*, a program which multiplies two 2×2 matrices is implemented:

```
#include <stdio.h>
#include <stdlib.h>
main (){
int **a; int **b; int **c;
int i; int j; int k;

a = (int**) malloc(2 * sizeof(int));
b = (int**) malloc(2 * sizeof(int));
c = (int**) malloc(2 * sizeof(int));

for(i = 2; 0 < i; i = i - 1){
   *(a+i-1) = (int*) malloc(2 * sizeof(int));
   *(b+i-1) = (int*) malloc(2 * sizeof(int));
   *(c+i-1) = (int*) malloc(2 * sizeof(int)); }

for(i = 2; 0 < i; i = i - 1){
  for(j= 2 ; 0 < j; j = j - 1){
    *(*(c+j-1)+i-1) = 0;
    for(k = 2; 0 < k; k = k - 1)
     *(*(c+j-1)+i-1) = *(*(c+j-1)+i-1)
```

```
+ *(*(a+k-1)+i-1) * *(*(b+j-1)+k-1); }}}
```

Here `a` is multiplied with `b`, and the result is stored in `c`. A matrix has two dimensions, thus it is represented as an array of pointers, i.e. a pointer to a pointer, in miniC*. Before the multiplication, all elements are allocated space in the heap. Matrix multiplication requires three nested loops and the `i`,`j` and `k` variables are used for this iteration. Here, `i` enumerates over the columns, while `j` enumerates of the rows. When multiplying the `a` matrix with the `b` matrix, each element of the resulting `c` matrix is the sum of each each corresponding column of `a` multiplied with the corresponding row of `b`. In the inner-most loop, `k` enumerates over these elements. Note, that two dimensional array projection `a[i][j]` is written `*(*(a+j)+i)` in miniC*. Further, note that the "strange iteration like `for(i = 2; 0 < i; i = i - 1)`, with `-1` in the array projection, instead of `for(i = 1; 0 <= i; i = i - 1)` and direct projection, is to obtain more accurate costing.

1.4 TRANSLATION RULES

1.4.1 The state space

In the translation, the state space is given a "deep" embedding in Hume: it is represented as a (large) Hume vector; variables are mappings from an environment to a vector index; and the value is return by projecting the index of the state space vector. To simplify the memory management the following assumptions are added with respect to the use of `free`: it has to be applied in the opposite order as `malloc`. Thus, `free` can only be applied to the last allocated (and non de-allocated) variable. With this assumption, the state-space vector can be divided into 3 parts:

$$<<\underbrace{v_1,\cdots,v_{stack}}_{\textbf{1.stack}},\underbrace{\overbrace{v_{stack+1},\cdots,v_l-1}^{\textbf{2.allocated}},\overbrace{v_l,\cdots v_{maxsize}}^{\textbf{3.unallocated}}}_{\textbf{heap}}>>$$

The first element represents the *stack* which holds all the variables: this means the value for an integer variable, and a vector index for a pointer variable. Hence, and index (location) and a value has the same type.

The remaining part of the state-space vector is the *heap*. Due to the assumption with the usage of `free`, the heap part of the state vector can be split into the allocated and un-allocated heap, by using a variable represented by `l` above. Here, `l` points to the first un-allocated location. This is in fact the `alloc` and `afree` memory management model, as described in [17]. However, it is not supported in (newer versions of) `gcc`, thus the use of `alloc`/`free`. Now, allocation and de-allocation can thus be represented by the following functions in Hume:

```
afree n l = if n > stack && n < l then n else 0;
alloc n l = if n <= maxsize - l then l+n else 0;
```

$$
\begin{array}{lcl}
\mathscr{T}'_e\ (\ int\ ) & \rightsquigarrow & \underline{int} \\
\mathscr{T}'_e\ (\ id\ ) & \rightsquigarrow & \underline{\texttt{state@(env}\ \mathscr{T}_{var}\ (id)\ \texttt{)}} \\
\mathscr{T}'_e\ (\ expr_1\ binop\ expr_2\ ) & \rightsquigarrow & \mathscr{T}'_e\ (expr_1)\ \ \mathscr{T}_o\ (\ binop\ )\ \ \mathscr{T}'_e\ (expr_2) \\
\mathscr{T}'_e\ (\ (expr)\ ) & \rightsquigarrow & \underline{\texttt{(}}\ \mathscr{T}'_e\ (expr)\ \underline{\texttt{)}} \\
\mathscr{T}'_e\ (\ cast\ expr\ ) & \rightsquigarrow & \mathscr{T}'_e\ (expr) \\
\mathscr{T}'_e\ (\ \&id\ ) & \rightsquigarrow & \underline{\texttt{env}}\ \ \mathscr{T}_{var}\ (id) \\
\mathscr{T}'_e\ (\ \star expr\ ) & \rightsquigarrow & \underline{\texttt{state@(}}\ \mathscr{T}'_e\ (\ expr\ )\underline{\texttt{)}} \\
 & & \\
\mathscr{T}_e\ (\ expr\ ) & \rightsquigarrow & LET\ x = FRESH()\ IN \\
 & & \underline{x\ \texttt{state =}}\ \mathscr{T}'_e\ (\ expr\ )\ \underline{\texttt{;}} \\
 & & RETURN\ x \\
 & & \\
\mathscr{T}''_e\ (\ int\ ) & \rightsquigarrow & \underline{int} \\
\mathscr{T}''_e\ (\ id\ ) & \rightsquigarrow & \underline{\texttt{env}}\ \ \mathscr{T}_{var}\ (id) \\
\mathscr{T}''_e\ (\ expr_1\ binop\ expr_2\ ) & \rightsquigarrow & \mathscr{T}''_e\ (expr_1)\ \ \mathscr{T}_o\ (\ binop\ )\ \ \mathscr{T}''_e\ (expr_2) \\
\mathscr{T}''_e\ (\ (expr)\ ) & \rightsquigarrow & \underline{\texttt{(}}\ \mathscr{T}''_e\ (expr)\ \underline{\texttt{)}} \\
\mathscr{T}''_e\ (\ \&id\ ) & \rightsquigarrow & \underline{\texttt{env}}\ \ \mathscr{T}_{var}\ (id) \\
\mathscr{T}''_e\ (\ \star expr\ ) & \rightsquigarrow & \mathscr{T}'_e\ (\ expr\ )
\end{array}
$$

FIGURE 1.2. $\mathscr{T}_e$ **,**$\mathscr{T}'_e$ **and** $\mathscr{T}''_e$ **: translation rules for miniC* expression**

The environment is a map, i.e. a Hume function, from variable names into indexes of the stack. Since there are no local declarations, and this is a proper ANSI-C subset, the variables are fixed by the declarations. In Hume variables are represented by an enumeration type `var`. A miniC* integer is 16 Bytes, which is a Hume `int 16` type, meaning the environment has type `var -> int 16`. The translation of the matrix multiplication example above, results the following enumeration type:

```
data var = A  |  B  |  C  |  I  |  J  |  K;
```

with the following environment:

```
env A = 1;  env B = 2;  env C = 3;
env I = 4;  env J = 5;  env K = 6;
```

1.4.2 Expressions

The translation rules for miniC* expressions are given in Figure 1.2. Note that the translation of miniC* binary operators $\mathscr{T}_o$ is trivial, and miniC* and Hume operators are assumed to have the same precedence. Now, $\mathscr{T}'_e$ gives the translation rule for miniC* expressions. Here, `@` is the built-in infix Hume vector projection functions. In the rules, a variable preceded by the `&`, is not projected, meaning the

location is returned. Moreover, an expression preceded by `*` is projected in the state vector, thus returning the value rather than the location.

$\mathscr{T}_e$ represents the expression as a function on a given state, required for when translation a subset of the `for`-statements. Here, *FRESH()*, *LET − IN* and *RETURN* is part of the meta-language describing the translation. *FRESH()* returns an unused identifier while the other two meta-expressions have obvious meaning. In the translation the resulting Hume code is `underlined` to separate it from the miniC* code. $\mathscr{T}_e''$ is required to translate `malloc` statements and assignments, discussed below. Finally, miniC* does not have a boolean type: *False* is `0` and *True* is not `0`. $\mathscr{T}_c'$ turns an arithmetic expression into a Hume boolean, ($\mathscr{T}_c'(e)$ equals $\mathscr{T}_e'(e)$ `!= 0`), and $\mathscr{T}_c$ is the function version of this rule.

1.4.3 Statements

A miniC* statement translated into a Hume function. A miniC* statement sequence is represented as a function composition. A statement may change the state space vector `state`, and the "heap allocation variable" `l`. Thus, a statement has the type

```
(vector maxsize of int 16,int 16)
     -> (vector maxsize of int 16,int 16)
```

where `maxsize` is the size of state-space vector, provided by the user to the translation rule. With the exception of memory management, most statements uses a set of pre-defined *higher order functions* (HOFs):

```
assign loc v (state,l) = (update state loc v,l);
exIf True  s (state,l) = s (state,l);
exIf False _ (state,l) = state,l);
exIfElse True  s _ (state,l) = s (state,l);
exIfElse False _ t (state,l) = t (state,l);
loop e s (state,l) =
   if (e state) then (loop e s (s (state,l)))
                else (state,l);
```

`assign` replaces the value of index `loc` with the value `v`, using the built-in Hume vector update function `update`; `exIf` represents an `if`-statement without an `else`-clause, while `exIfElse` represent the version with an `else`-clause. Both function are defined by pattern matching the input condition. Together with condition, `exIf` should receive the translated body of the `if` statement, of type described above. `exIfElse` additionally receives the `else` clause body, of the same type.

The `loop` HOF captures all but the initialisation statement of a `for` loop. Due to the recursion the condition there is a predicate on the state-space and not a boolean. Thus, $\mathscr{T}_e$ should be used instead of $\mathscr{T}_e'$ in this translation, since the value of the condition may change between each call of `loop`.

Figure 1.3 shows the key translation rules for miniC* statements: in the translation of `malloc`, the function argument is assumed to be either the number of

$\mathscr{T}_s$ (*expr*$_1$ = [*cast*] `malloc(` *expr*$_2$ [`* sizeof(int)`] `)`)
$\rightsquigarrow$ *LET x = FRESH() IN*
`x (state,l) =`
`(update state (` $\mathscr{T}''_e$ (*expr*$_1$) `) l,`
`alloc` $\mathscr{T}'_e$ (*expr*$_2$) `l);`

$\mathscr{T}_s$ (`free(` *expr* `)`) $\rightsquigarrow$ *LET x = FRESH() IN*
`x (state,l) =`
`let v =` $\mathscr{T}'_e$ (*expr*)
`in (update state v 0, afree v l);`

$\mathscr{T}_s$ (`*`*expr*$_1$ = *expr*$_2$) $\rightsquigarrow$ *LET x = FRESH() IN*
`x (state,l) =`
`assign (` $\mathscr{T}'_e$ (*expr*$_1$) `) (`$\mathscr{T}'_e$ (*expr*$_2$) `)`
`(state,l);`
RETURN x

$\mathscr{T}_s$ (*id* = *expr*) $\rightsquigarrow$ *LET x = FRESH() IN*
`x (state,l) =`
`assign` $\mathscr{T}''_e$ (*id*) `(` $\mathscr{T}'_e$ (*expr*) `)`
`(state,l);`
RETURN x

$\mathscr{T}_s$ (`if (` *expr* `)` *stmt*) $\rightsquigarrow$ *LET x = FRESH() IN*
LET s = $\mathscr{T}_s$ (*stmt*) *IN*
`x (state,l) = exIf (` $\mathscr{T}'_c$ (*expr*) `)`
s `(state,l);`
RETURN x

FIGURE 1.3. $\mathscr{T}_s$ **: translation rules for miniC* statements**

heap cells, or the number of heap cells times `sizeof(int)`. Thus, the latter is ignored. The statement allocates *expr*$_2$ locations, and location *expr*$_1$ points to the first of these allocated locations. The previously defined `alloc` function is used to update `l`. *expr*$_1$ should contain the location to the first newly allocated cell, which is `l`. Due to the use of `update`, $\mathscr{T}''_e$ is used to translate *expr*$_1$. This is to ensure that the location, and not the value in the location is returned. As an illustration, the first (allocation) statement of the matrix multiplication example is shown. It results in the Hume function `f1`:

```
f1 (state,l) = ((update state (env A) l),(alloc 2 l));
```

`free` works by de-allocating the last allocated variable, and set the variable to `0` (the `NULL` pointer in C). The translated Hume achieves this by application to the previously defined `afree` function; *assignment* is handled directly by the `assign` HOF. Here, $\mathscr{T}''_e$ is used to find the correct location, while $\mathscr{T}'_e$ translates the new value of that location; the other form of assignment is to prefix an expression with `*`, which is also translated into a call to the `assign` HOF. Due to the

semantics of $\star$, $\mathscr{T}_e'$, and not $\mathscr{T}_e''$, is used to translated the location ($\mathscr{T}_e''\ (\star expr_1)$ equals $\mathscr{T}_e'\ (expr_1)$). To illustrate, the following Hume code, is the result of applying $\mathscr{T}_s$ to the first statement of the body of the middle `for` loop of the actual multiplication algorithm for the two matrices above:

```
f14 (state,l) = (assign ((state@((state@(env C)) +
  ((state@(env J)) - 1))) + ((state@(env I)) - 1))
     0 (state,l));
```

finally, the rule for conditional without an `else` branch uses the `exIf` HOF. Note that $\mathscr{T}_c'$ is used for the conditional expression, which returns a boolean type. The translation for a miniC* `if` statement with an `else` clause is similar.

The `for`-statement can be translated using the `loop` HOF. However, this HOF is too generic in order to obtain an adequate cost from the Hume analyser tools. Thus, a costable subset is created, which has a specific translation rule. In this subset, statements must be on the form

$$\texttt{for (}id\texttt{=}expr\texttt{;}\ int_1 \texttt{<}\ id\texttt{;}\ id = id - int_2\texttt{)}\ cbody$$
$$\texttt{for (}id\texttt{=}expr\texttt{;}\ int_1 \texttt{<=}id\texttt{;}\ id = id - int_2\texttt{)}\ cbody$$

and *cbody* should not change *id*. Due to pointers, the latter cannot be checked by translator program. This requires more intricate techniques, and is thus assumed here. Now, the translation rule for a `for` statement that is NOT in this subset uses the `loop` HOF:

$$\begin{array}{lll}
\mathscr{T}_s\,(\texttt{for (} stmt_1\ \texttt{;}\ expr\ \texttt{;}\ stmt_2\ \texttt{)}\ stmt_3) & \rightsquigarrow & LET\ x = FRESH()\ IN \\
 & & LET\ f = \mathscr{T}_s\,(stmt_1)\ IN \\
 & & LET\ e = \mathscr{T}_c\,(expr)\ IN \\
 & & LET\ s = \mathscr{T}_s\,(stmt_3\ \texttt{;}\ stmt_2)\ IN \\
 & & \underline{x\ \texttt{(state,l) = loop}\ e\ s\ \texttt{(state,l);}} \\
 & & RETURN\ x(f)
\end{array}$$

Here, $\mathscr{T}_c$ ensures that the condition is a function on the state-space. The translation rule for a `for`-statement that belongs to this costable subset is defined as follows:

$$\begin{array}{l}
\mathscr{T}_s\,(\texttt{for (} id = expr_1\texttt{;}\ expr_2\ binop\ id\ \texttt{;}\ id = id - expr_3\texttt{)}\ stmt_3) \rightsquigarrow \\
\quad LET\ x = FRESH()\ IN \\
\quad LET\ y = FRESH()\ IN \\
\quad LET\ f = \mathscr{T}_s\,(id = expr_1)\ IN \\
\quad LET\ s = \mathscr{T}_s\,(stmt_3\ \texttt{;}\ id = id - expr_3)\ IN \\
\quad \underline{y\ \texttt{(state,l)}\ id = \texttt{if}}\ \underline{\mathscr{T}_e'\,(expr_2)\ \mathscr{T}_o\,(binop)\ \underline{id}} \\
\qquad\qquad\qquad\qquad \underline{\texttt{then}\ y\ \texttt{(}\ s\ \texttt{(state,l))}\ \ (id -}\ \mathscr{T}_e'\,(expr_3)\ \underline{\texttt{)}} \\
\qquad\qquad\qquad\qquad \underline{\texttt{else (state,l);}} \\
\quad \underline{x\ \texttt{(state,l)} = y\ \texttt{(state,l)}}\ \mathscr{T}_e'\,(expr_1)\ \underline{\texttt{;}} \\
\quad RETURN\ x(f)
\end{array}$$

All the `for` statements of the matrix multiplication example belong to the costable subset. As the rule above shows, here an accumulator variable is used for the repetition, which is explored by the Hume analyser tools. For example, the innermost `for` loop of the main multiplication algorithm becomes:

```
f18 (state,l) k = (if (0<k) then (f18(f19(state,l))(k - 1))
                             else (state,l));
f20 (state,l) = (f18 (state,l) 2);
```

Now, `k` is the accumulator variable, `f20` initialises the loop, while `f19` is the body of the statement, including the last statement of the header.

1.4.4 The full program

Sequences are represented as function composition: in a sequence $stmt_1$ `;` $stmt_2$, if $stmt_1$ is translated into function `f`, and $stmt_2$ is translated into function `g`, then the sequence is translated into the composition `g(f...)`, where `...` is the translation of the statements preceding $stmt_1$.

The strict stream ordering in miniC* is a result of the stream handling in Hume. In the translation of streams, an input stream is handled in the same way as an assignment, with an additional input to the function, which is the stream. Similarly, an output stream adds another output to the box, which will have the value given by the expression in the `printf` statement.

The full miniC* program is represented as one Hume box. The input is a dummy boolean value, wired to its output, used to initialise the program, in additional to the optional input stream. The output is the dummy value, although nothing ($\star$) is sent to it, since the program should only be execute once, together with optional outputs.

The translation rules have been implemented in a Haskell program. In addition to the miniC* source file, it requires a value holding the size of the state space vector. In the example programs in Section 1.5, the size of this vector is indicated.

1.5 RESULTS

To estimate the quality of our analysis, we compare the values it delivers with measurements on a real embedded systems processor, the Renesas M32C. The actual worst-case execution costs for each primitive Hume instruction on the M32C/85U processor have been determined by the aiT WCET tool [5], building the basic cost-model of the analysis. While the analysis results are necessarily over-estimations, the measurements are significant under-estimations of the worst case, because they deal with particular input data.

Table 1.1 shows the results for some miniC* programs. The first column gives the program name, the second the size of the vector modelling the miniC* state space. The third column gives the result from the WCET analysis on the generated Hume program. The fourth column gives the measured time for running the Hume program. The HTA/HT column gives the ratio of analysis result over measured

Program	Vector Size	Hume Time Analysis	Hume Time	HTA/ HT	C-code Time	HTA/ CT	HTA (unsafe)	HTAu/ CT
`arraycopy`	23	494549	732158	0.67	1094	452.06	419523	383.48
`arrayrotate`	13	616817	303949	2.03	1798	343.06	557019	309.80
`fact`	13	451787	301328	1.50	3513	128.60	405392	115.40
`matmult2`	27	771359	500271	1.54	1861	414.49	687164	369.24
`mediumarrayrotate`	16	824102	376552	2.19	2347	351.13	732066	311.92
`tinyarraysearch`	15	569359	302224	1.89	2596	219.32	521017	200.70

TABLE 1.1. miniC* analysis and measurement results

costs, and assesses the quality of the bound itself. The sixth column gives the measured time for running the miniC* program. The HTA/CT column is the most interesting one, giving the ratio of analysis result over measured C costs.

First of all we observe huge factors in this column. This is not surprising, since we first translated a low-level miniC* program into a high-level Hume program, which makes heavy use of higher-order functions and non-destructive vector updates. The analysis not only reflects the high costs of these language constructs, but also has to be conservative in finding an upper bound on time consumption. However, it is not our goal to give tight upper bounds through this translation process, which would be a hopeless task. We are rather interested in the complexity profile, i.e. how far does the factor between the programs vary? Clearly for such an analysis a large set of programs would be needed to come up with a realistic standard deviation to draw strong conclusions. However, already in this small set of programs we do observe significant variations, reflecting a poor match between bounds obtained from the Hume analysis with the actual miniC* runtimes. There are several possible sources for these inaccuracies. Firstly, the translation could add costs that don't necessarily have to be present in the Hume code (for example using higher-order functions in the translation could be such a source). Secondly, it could be due to the Hume execution model (for example mandating that all vector updates have to be non-destructive). Finally, it could be due to limitations of the WCET analysis (which has to be limited in being a static analysis). To rule out the last reason we have already performed validations of the analysis on several Hume applications and these show reasonable accuracy [16]. In order to focus on the second reason, we have produced an alternative translation route, which generates vector update code that is destructive (this is an optional feature that we have added to Hume in order to make specific experiments but is not part of the official language definition). The HTA (unsafe) column gives the WCET analysis bounds for this unsafe Hume code, and the HTAu/CT column measures the ratio between these analysis results and the miniC* execution time. Notably, the variation in the factors shown in this column is much narrower. Most programs are between a factor 309 and 383, and all of these programs manipulate array data structures of a similar size. Only two programs fall outside this range: `fact` which does not use arrays at all; and `tinyarraysearch` which uses only small arrays in a read-only fashion. Based on this preliminary data, we conjecture that the main

reason for the large variation in the HTA/CT factors is due to Hume's execution model. But this can be circumvented by using destructive updates on vectors to model (destructive) assignment in miniC*.

From the HTAu/CT column we conjecture that, if two miniC* programs have "comparable" dynamic memory consumption, which we can analyse through a related heap analysis, then Hume's WCET analysis, applied on (generated) Hume code that uses destructive updates, does indeed produce a realistic complexity profile for this set of miniC* programs. In particular, if we study two related implementations of the same algorithm, which do not drastically differ in the amount of dynamic heap, then we can give guidance on which version is likely to perform better.

1.6 RELATED WORK

As noted in [8], the translation of imperative to functional languages is very well known since the first use of functional notations in denotational semantics [24]. In particular, functional meta-languages are commonly deployed in both semantics-directed compiler-compilers[21] and theorem provers. The most relevant mechanisation of an imperative language with pointers and dynamic memory management we are familiar with is Tuch's PhD thesis [29]. Here a C language was formalised in the Isabelle/HOL theorem prover. However, his motivation was correctness verification and not resource analysis.

Resource analysis also has a long pedigree. Most closely related to our approach are the *amortised cost based analyses* of heap space in the linearly-typed functional programming languages LFPL [10, 11] and Camelot [20], in the comprehensive Java-subset RAJA [12], and for several languages in the AHA project [26]. A stack space analysis using this approach is given in [2]. However, none of these approaches deals with worst-case execution time.

A system that combines a type-based approach with the automatic generation of checkable certificates for time bounded computation is described in [4], but it lacks the inference of the bounds in the first place.

Other functional notations with ad-hoc techniques for analysing resource consumption are GeHB [27], with a two-level *staged* notation that also builds on LFPL, and RT-FRP [31], which, like Hume, targets embedded systems.

Another type-based approach to infer size bounds are *sized types* [15]. Several analyses on heap or stack space usage build on this concept: [14, 23, 3, 30, 7], but again none considers worst-case execution time, although P. Vasconcelos states in his PhD thesis that this is equally within reach of these methods.

A variety of academic and commercial tools exist for calculating guaranteed bounds on *worst-case execution time (WCET)* [32], including aiT[5], bound-T[13], SWEET[25, 19], The state-of-the-art is epitomised by AbsInt's aiT tool, which uses abstract interpretation to provide a guaranteed, and tight, upper bound on actual run-times for C code fragments with known data inputs. The aiT tool includes precise models of cache [6] and pipeline behaviours [18]. Such tools typically work on machine-code or C fragments, yielding analyses for specific input

cases that, in the best situations, closely conform to the actual execution time. In constructing solutions for concrete programs, however, it is usually necessary for the programmer to provide additional detailed information, and this may require significant effort in some cases. For example, it may be necessary to indicate the range of values that a loop variable may take if the associated iteration is not bounded by a literal value.

1.7 CONCLUSION

We have explored the use of the Hume WCET analysis directly with miniC*, an imperative language with explicit memory allocation and pointers, through translation. Because of the presence of pointers and a dynamic memory model in miniC*, the state-space is modelled as a vector, and the basic read and write operations in miniC* are translated to vector-lookup and -update.

First results from analysing code using the rules presented in Section 1.4 showed a wide variation of analysed (Hume) code over measured (miniC*) code, far wider than our earlier results based on translation from miniC [8], which lacks pointers and memory allocation. In tracking down the source of this mis-match we modified the translation of miniC* to more accurately reflect the costs of the original C program. In particular we use a destructive version of vector update, to model miniC*-level assignment in the generated Hume code. In this version of the code the variation of analysed code over measured code becomes much narrower. We believe that in this version of the translation the analysis bounds can be used as a complexity profile for miniC* programs with comparable memory consumption.

The results presented here are preliminary results gained from the presented translation. To obtain a more solid basis for our conclusions we would like to use more miniC* programs, in particular programs with differing dynamic memory consumption. Since we also have an analysis for the heap consumption of Hume program, its results could be taken as input to classify the analysed programs, and to test our main conjecture on.

ACKNOWLEDGEMENTS

This research is supported by the EU FP6 EmBounded project (IST-510255). We would like to thank all our project collaborators, in particular Robert Pointon (formerly Heriot-Watt) and Reinhold Heckmann (AbsInt).

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.

[2] B. Campbell. *Stack Usage Analysis*. PhD thesis, Edinburgh University, 2008.

[3] W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computing*, 14(2,3):261–300, 2001.

[4] K. Crary and S. Weirich. Resource Bound Certification. In *Proc. ACM Symp. on Principles of Prog. Langs.*, pages 184–198, 2000.

[5] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, pages 469–485. Springer-Verlag LNCS 2211, 2001.

[6] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163–189, 1999.

[7] B. Grobauer. Cost recurrences for DML programs. In *Proc. ICFP '01: Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 253–264, New York, NY, USA, 2001. ACM Press.

[8] G. Grov, G. Michaelson, H.-W. Loidl, S. Jost, and C. Herrmann. An application of Hume analysis to imperative programs. In *submitted to ACM Symposium on Applied Computing (SAC'09) Programming Languages Track*, August 2008.

[9] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, – GPCE 2003, Erfurt, Germany*, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.

[10] M. Hofmann. A Type System for Bounded Space and Functional In-Place Update. *Nordic Journal of Computing*, 7(4), Winter 2000.

[11] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.

[12] M. Hofmann and S. Jost. Type-based amortised heap-space analysis (for an object-oriented language). In P. Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[13] N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proc. WCET '02: Int'l Workshop on Worst-Case Execution Time Analysis*, June 19-21 2002.

[14] R. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99)*, pages 70–81, 1999.

[15] R. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL'96 — Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM.

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

[17] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd edition*. Prentice-Hall, 1988.

[18] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *SAS*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.

[19] B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proc. WCET '03: Int'l Workshop on Worst-Case Execution Time Analysis*, pages 99–102, 2003.

[20] K. Mackenzie and N. Wolverson. Camelot and Grail: Compiling a Resource-Aware Functional Language for the Java Virtual Machine. In *Trends in Functional Prog., Volume 4*, pages 29–46. Intellect, 2004.

[21] G. Michaelson. Interpreters from functions and grammars. *Computer Languages*, 11(2):85–104, 1986.

[22] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[23] R. Pena and C. Segura. A First-Order Functl. Lang. for Reasoning about Heap Consumption. In *Draft Proc. Intl. Workshop on Impl. and Appl. of Functl. Langs. (IFL '04)*, pages 64–80, 2004.

[24] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, 1986.

[25] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems. In *Proc. ISOLA '06: Int'l Symp. on Leveraging Applications of Formal Methods*, Paphos, Cyprus, November 2006.

[26] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial Size Analysis of First-Order Functions. In *TLCA 2007: Typed Lambda Calculi and Applications*, Paris, France, June 26–28, 2007.

[27] W. Taha, S. Ellner, and H. Xi. Generating Heap-Bounded Programs in a Functional Setting. In *Proc. EMSOFT '03: 2003 Intl. Conf. on Embedded Software*, pages 340–355. Springer LNCS 2855, 2003.

[28] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[29] H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, The University of New South Wales, 2008.

[30] P. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. IFL '03: International Workshop on Implementation of Functional Languages*, pages 86–101. Springer-Verlag LNCS, 2004.

[31] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Intl. Conf. on Functional Programming (ICFP '01)*. ACM, Sept. 2001.

[32] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. Accepted for TECS, 2008.