Hume Cost Analyses for Imperative Programs

***Abstract*— Cost analysis of conventional imperative/object-oriented languages, such as C or Java, is both undecidable in theory and highly restricted in practice. In contrast, since the novel *Hume* language is based on strong formal foundations, it allows close alignment between implementations and cost analyses, so providing high-quality static cost analysis. In this paper, we present a formal translation from a C subset (*miniC*) to Hume, explore the scope for applying Hume cost analyses to such a translation, and discuss the efficacy of applying Hume worst-case execution time (WCET) analysis to translated *miniC* exemplars.**

I. Introduction

Despite sustained research over three decades, curiously little progress has been made in developing cost analyses of Turing-complete (TC) languages. Of course, TC languages suffer from the classic undecidability results which preclude algorithmic determination of termination, in general, and hence of behavioural costs such as time and space needs. Nonetheless, the gut feeling that heuristic automatic cost analysis should be tractable for all but pathological programs, based on human success in hand analysis of a wide body of algorithms, has simply not been borne out to date. Conversely, languages that are tractable to automatic cost analysis, usually impose significant limitations on the programmer (prohibiting, for example, recursion, exceptions, or complex data structures [30]).

Hume [9] was substantially developed as a response to this apparent impasse. Rather than trying to provide a language that artificially restricts expressiveness to syntactic or skeletal constructs with known properties, the Hume design encompasses a hierarchy of programming levels, where lower levels have less expressive power but stronger analyses: full TC Hume, PR-Hume (primitive recursive), Template-Hume (higher-order functions), FSM-Hume (finite state) and HW-Hume (hardware abstractions). The Hume methodology is to develop a program without consideration of level or analysis. The program is then repeatedly analysed, and, where analysis is problematic or suggests unacceptable costs, offending constructs are reformulated in a lower level.

To support these levels, the Hume design is based around two layers: a *coordination layer*, that abstracts over concurrent finite state *boxes* linked by *wires*. Box transitions are then specified in a pattern matching *expression layer*, that offers at full strength the usual constructs of a contemporary polymorphic functional language, Hume levels are then determined by the types that may be used on wires and in patterns, and the constructs that are permissible on the output (expression) side of box transitions.

Hume has synergistic formally-specified semantics and cost models, and the HUume tool chain closely aligns with this specification. There are now a stable reference interpreter, abstract machine and native code compiler for full Hume, complemented by architecture-specific, robust time- and space-analyses for HW- and FSM-Hume, and highly promising research analyses for PR-Hume(see `http://www.hume-lang.org`).

Given the progress with Hume, the question now arises as to how applicable the approach is to more traditional languages, in particular commonly-used imperative languages. It would certainly be possible to build analogous models and analyses from scratch for an extant language, such as C. However, to do so would depend crucially on the availability of a formal semantics for that language, and on a tool chain with formally known characteristics. Clearly, to pursue such an approach would be very labour-intensive. We have therefore instead been exploring direct translation from imperative source programs to Hume for subsequent costing using the existing Hume cost analyses. Our premise is that, given a formally-defined translation schema, it might then be feasible to relate Hume target analyses back to the original source program constructs. Of course, this approach is fraught with difficulties, not least the semantic-gap between the source language and Hume, potentially necessitating the generation of cost-distorting “glueware” artefact’s in translations.

Our initial source language is “miniC”, a C subset which offers iteration and choice over assignable integer arrays. In this paper, we discuss the translation of miniC to the Hume expression layer and the analysis of the resultant Hume programs. In the following sections we describe miniC and our translation schemes to Hume expressions, discuss the results of analyses of translated miniC exemplars and compare them with instrumented compiled miniC, and reflect on the efficacy and future of this approach.

II. Source language: miniC

Figure 1 shows the abstract syntax of *miniC*, miniC is a proper subset of ANSI-C, where all declarations precede all statements, and all variables must be given an explicit

```
program ::=  prelude main() body
prelude ::=  include_0 ... include_n   n ≥ 0
include ::=  #include <id.h>
body ::=     {decl_1 ··· decl_n stmt_1 ··· stmt_m}
cbody ::=    stmt | {stmt_1 ··· stmt_n}    n ≥ 1
decl ::=     int id; | int id[ int ];
stmt ::=     id = expr; | id[ expr_1 ] = expr_2;
             | if ( expr ) cbody
             | if ( expr ) cbody_1 else cbody_2
             | for ( stmt_1 ; expr ; stmt_2 ) cbody
             | printf("%d",id); | scanf("%d",&id);
expr ::=     int | id | id[ expr ]
             | expr_1 binop expr_2 | ( expr )
binop ::=    == | != | < | <= | + | - | * | / | %
```

Fig. 1. miniC abstract syntax

type (either an integer or an array of integers – sufficient to implement many interesting examples). *miniC* does not provide any form of function declaration, apart from the `main` function, which must contain the entire program. At the expression level, *miniC* supports both standard variable and array assignments, conditionals (`if`) with or without `else` branches, and repetition based on a limited version of a `for`-statement. Finally, miniC allows integers to be read from standard input via `scanf` and written to standard output via `printf`. Overall, miniC constitutes a small, but Turing-complete and representative, imperative language. To illustrate the use of *miniC*, and its translation into Hume in the next section, we will define an algorithm which multiplies two 3×3 matrices (`a` and `b`) and stores the result in a 3×3 matrix (`c`):

```
main (){
  int a[9]; int b[9]; int c[9];
  int i; int j; int k;
  for(i=2; 0 <= i; i=i-1)
    for(j=2; 0 <= j; j=j-1) {
      c[(i*3)+j] = 0;                        /* ex */
      for(k=2; 0 <= k; k=k-1)
        c[(i*3)+j]=c[(i*3)+j]+a[(i*3)+k]*b[(k*3)+j];
  }
}
```

Since multi-dimensional arrays are not supported, a 3×3 matrix is flattened into a one-dimensional array of length 9. Thus an array access to element `a[i][j]` is instead written as `a[(i*3)+j]`, where 3 refers to the number of columns. In the algorithm, `i` is used to enumerate the columns, while `j` enumerates the rows. When multiplying `a` with `b`, each element of the result, `c`, is the sum of the corresponding column of `a` multiplied with the corresponding row of `b`. In the innermost loop, `k` enumerates over these elements.

III. Translation into Hume

Each *miniC* construct is translated into a corresponding set of Hume boxes and expressions. A *miniC* `int` is 16 bits, and is represented as a 16 bit Hume integer, of type `int 16`. A *miniC* array has a fixed size (and type), and is represented as a Hume. So, a *miniC* array of N integers is represented as a Hume vector of N `int 16`s. Since *miniC* arrays are indexed from 0, where Hume vectors are indexed from 1, index `i` of a *miniC* array corresponds to index `i+1` of a Hume vector.

Let $[\![program]\!]$ be the Hume translation of a *miniC program*. A full *miniC program* is represented by one box. Each statement is a function, and sequentiality is ensured by correct function composition. In Hume, streams are handled in the coordination layer, and there are thus two additional requirements that are not present in C: a `scanf` statement cannot be preceded by a non-`scanf` statement (i.e. all `scanf`s appear first in the program); and a `printf` statement cannot be followed by a non-`printf` statement (i.e. all `printf`s appear last in the program). These properties can easily be checked statically, and we will thus assume, in the remainder of this paper, that they hold in all cases. The formal translation rule for *miniC* programs $[\![program]\!]$ is shown in Figure 4. This builds on translation rules for the other *miniC* program structures in Figures 2–3.

A. The state space

In *miniC*, the (implicit) meaning of a variable *id* is the projection of that *id* onto the values held on the *miniC* stack. Assignments to *id* then update the stack value that *id* points to. This is not the case in a functional language. Here, the state-space must be explicitly sent between each "statement function". The state-space is represented by a tuple, which is created using the *miniC* declarations (*decl*). To manipulate and access the state space, each declared variable is given a `store` and `load` function, with the obvious meaning. The rules $[\![s]\!]_{store}$ and $[\![s]\!]_{load}$ creates the two functions. These rules assume that s is a list of variable name and type pairs. This is created by the *sp* function using the *decl* rule:

$$\begin{array}{lcl} sp\ (\texttt{int}\ id) & = & [(id, \texttt{int 16})] \\ sp\ (\texttt{int}\ id\texttt{[}int\texttt{]}) & = & [(id, \texttt{vector}\ int\ \texttt{of int 16})] \\ sp\ (decl_1\texttt{;}decl_2) & = & sp\ (decl_1)\ @\ sp\ (decl_2). \end{array}$$

Note that $=$ is used for auxiliary meta-functions like *sp*, while $\rightsquigarrow$ is used for a translation rule (which generates Hume source code). Moreover, to separate the target Hume code from the source *miniC* code, the Hume code is <u>`underlined`</u>. Standard list notation is used, where @ represents list append. $[\![s]\!]_{load}$ is defined using the auxiliary *loadf* function:

$$\begin{array}{lcl} loadf\ ((id,t):[])\ s & \rightsquigarrow & \underline{\texttt{load}}(CAP\ id)\ \underline{(}patt\ s\underline{)\ =\ }id\ \underline{;} \\ loadf\ ((id,t):tl)\ s & \rightsquigarrow & \underline{\texttt{load}}(CAP\ id)\ \underline{(}patt\ s\underline{)\ =\ }id\ \underline{;} \\ & & loadf\ tl\ s \\ [\![s]\!]_{load} & \rightsquigarrow & loadf\ s\ s \end{array}$$

Here, standard pattern matching is used on the lists where [] is the empty list and : is a list constructor. *CAP* prints the given variable name with capital letters, while *patt* prints the list of the variables in a variable/type pair. This is a simpler version of the *rep* rule used by $[\![s]\!]_{store}$:

$$\begin{array}{lcl} rep\ ((id,T):[])\ x\ e & \rightsquigarrow & IF\ id = x\ THEN\ \underline{e}\ ELSE\ \underline{id} \\ rep\ ((id,t):sp)\ x\ e & \rightsquigarrow & IF\ id = x\ THEN\ \underline{e}\ ELSE\ \underline{id\ ,} \\ & & rep\ sp\ x\ e \end{array}$$

which prints the list of the variables, except for x where e is printed instead. Note that *IF-THEN-ELSE* is part of the meta-language used to define the translation. $[\![s]\!]_{store}$ is then defined using the auxiliary *stf* function:

$$\begin{array}{lcl} stf\ ((id,t):[])\ s & \rightsquigarrow & \underline{\texttt{store}}(CAP\ id)\ \underline{\texttt{e}\ (}patt\ s\underline{)} \\ & & \underline{=(}replace\ s\ id\ \underline{\texttt{e}})\ \underline{;} \\ stf\ ((id,t):tl)\ s & \rightsquigarrow & \underline{\texttt{store}}(CAP\ id)\ \underline{\texttt{e}\ (}patt\ s\underline{)} \\ & & \underline{=(}replace\ s\ id\ \underline{\texttt{e}})\ \underline{;} \\ & & stf\ tl\ s \\ [\![s]\!]_{store} & \rightsquigarrow & stf\ s\ s \end{array}$$

To illustrate the use of $[\![s]\!]_{load}$ and $[\![s]\!]_{store}$,

```
storeC e (a,b,c,i,j,k) = (a,b,e,i,j,k);
loadC (a,b,c,i,j,k) = c;
```

are generated for the `c` variable in the above example, and `(a,b,e,i,j,k)` is the state space tuple. Henceforth, we will always use `st` to refer to the state-space tuple.

$$\begin{array}{lll}
[\![\texttt{==}]\!]_o & \rightsquigarrow & \underline{\texttt{==}} \\
[\![\texttt{!=}]\!]_o & \rightsquigarrow & \underline{\texttt{!=}} \\
[\![\texttt{<}]\!]_o & \rightsquigarrow & \underline{\leq} \\
[\![\texttt{<=}]\!]_o & \rightsquigarrow & \underline{\texttt{<=}} \\
[\![\texttt{+}]\!]_o & \rightsquigarrow & \underline{\texttt{+}} \\
[\![\texttt{-}]\!]_o & \rightsquigarrow & \underline{\texttt{-}} \\
[\![\texttt{*}]\!]_o & \rightsquigarrow & \underline{\texttt{*}} \\
[\![\texttt{/}]\!]_o & \rightsquigarrow & \underline{\texttt{div}} \\
[\![\texttt{\%}]\!]_o & \rightsquigarrow & \underline{\texttt{mod}} \\
 & & \\
[\![\mathit{int}]\!]_e & \rightsquigarrow & \underline{\mathit{int}} \\
[\![\mathit{id}]\!]_e & \rightsquigarrow & \underline{\texttt{load }\mathit{id}\texttt{ st}} \\
[\![\mathit{id}\,\texttt{[}\mathit{expr}\texttt{]}]\!]_e & \rightsquigarrow & \underline{\texttt{(load }\mathit{id}\texttt{ st)@(}}\ [\![\mathit{expr}]\!]_e\ \underline{\texttt{+1)}} \\
[\![\mathit{expr}_1\ \mathit{binop}\ \mathit{expr}_2]\!]_e & \rightsquigarrow & [\![\mathit{expr}_1]\!]_e\ [\![\mathit{binop}]\!]_o\ [\![\mathit{expr}_2]\!]_e \\
[\![\texttt{(}\mathit{expr}\texttt{)}]\!]_e & \rightsquigarrow & \underline{\texttt{(}}[\![\mathit{expr}]\!]_e\underline{\texttt{)}} \\
 & & \\
[\![\mathit{expr}]\!]_{e'} & \rightsquigarrow & \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
 & & \underline{x\texttt{ st =}}\ [\![\mathit{expr}]\!]_{e'}\ \underline{\texttt{;}} \\
 & & \mathit{RETURN}\ x
\end{array}$$

Fig. 2. Translation rules for *miniC* binary operators & expressions

B. miniC *binary operators and expressions*

Figure 2 shows the translation rules for *miniC* binary operators $[\![\mathit{binop}]\!]_o$ and expressions $[\![\mathit{expr}]\!]_e$. The *miniC* binary operators (*binop*) are directly translated into the corresponding Hume operators. In a *miniC* expression (*expr*), an integer is translated directly into its Hume equivalent. A *miniC* variable is translated into a call to the corresponding `load` function: e.g. $[\![\texttt{i}]\!]_e$ becomes `loadI st`. In Hume, vector projection is written using the infix `@` notation, so the array projection `x[e]` is translated into $[\![\texttt{x}]\!]_e$`@(`$[\![\texttt{e}]\!]_e$`+1)`, where `+1` is because Hume vectors are indexed from 1. The remaining $[\![\mathit{expr}]\!]_e$ rules are straightforward. Note that $[\![\mathit{expr}]\!]_{e'}$ represents the translated expression as a function and returns the function identifier. *FRESH()* is part of the meta-language and creates an unused/fresh function/variable name. *LET*/*RETURN* are also part of the meta-language, with the obvious meaning. $[\![\mathit{expr}]\!]_{e'}$ is needed by one of the `for`-statement translation rules.

C. miniC *statements*

Figure 3 defines the translation rules $[\![\mathit{stmt}]\!]_s^\rho$ for *miniC* statements *stmt*. To achieve a correct translation of input and output streams, the rules are augmented by an environment ρ which will be elaborated in due course. Moreover, each statement creates a function (which may require sub-functions), and each rule return this function identifier together with the new environment. This is illustrated in the first rule of the figure, which handles sequences. A *miniC* assignment statement uses the `load` and `store` functions and a *miniC* array assignment $[\![\texttt{x[n] = e}]\!]_s^\rho$ uses the built-in Hume `update` function for vectors. Thus, the expression (body) of the function generated by $[\![\texttt{x[n] = e}]\!]_s^\rho$ becomes `storeX (update (loadX st) (`$[\![\texttt{n}]\!]_e$`+1) (`$[\![\texttt{e}]\!]_e$`)) st`. For example, ($[\![\texttt{c[(i*3)+j] = 0;}]\!]$) (the statement labelled with `/* ex */` above) results in a function `f st`, with the body:

$$\begin{array}{l}
[\![\mathit{stmt}_1\texttt{;}\,\mathit{stmt}_2]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ \langle f, \rho' \rangle = [\![\mathit{stmt}_1]\!]_s^\rho\ \mathit{IN} \\
\quad \mathit{LET}\ \langle g, \rho'' \rangle = [\![\mathit{stmt}_2]\!]_s^{\rho'}\ \mathit{IN} \\
\quad \mathit{RETURN}\ \langle g(f), \rho'' \rangle \\
[\![\mathit{id}\ \texttt{=}\ \mathit{expr}]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
\quad \underline{x\texttt{ st = store}}(\mathit{CAP\ id})\underline{\texttt{ (}[\![\mathit{expr}]\!]_e\texttt{)) st;}} \\
\quad \mathit{RETURN}\ \langle x, \rho \rangle \\
[\![\mathit{id}\,\texttt{[}\mathit{expr}_1\texttt{]=}\mathit{expr}_2]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
\quad \underline{x\texttt{ st = store}}(\mathit{CAP\ id})\underline{\texttt{(update (load}}(\mathit{CAP\ id}) \\
\qquad\qquad \underline{\texttt{st)}}[\![\mathit{expr}_1]\!]_e\ \ [\![\mathit{expr}_2]\!]_e\ \underline{\texttt{) st;}} \\
\quad \mathit{RETURN}\ \langle x, \rho \rangle \\
[\![\texttt{if (}\mathit{expr}\texttt{)}\ \mathit{stmt}]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
\quad \mathit{LET}\ \langle s, \rho \rangle = [\![\mathit{stmt}]\!]_s^\rho\ \mathit{IN} \\
\quad \underline{x\texttt{ st = exIf}}\ [\![\mathit{expr}]\!]_e\ \underline{s\texttt{ st;}} \\
\quad \mathit{RETURN}\ \langle x, \rho \rangle \\
[\![\texttt{if (}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
\quad \mathit{LET}\ \langle s, \rho \rangle = [\![\mathit{stmt}_1]\!]_s^\rho\ \mathit{IN} \\
\quad \mathit{LET}\ \langle t, \rho \rangle = [\![\mathit{stmt}_2]\!]_s^\rho\ \mathit{IN} \\
\quad \underline{x\texttt{ st = exIfElse}}\ [\![\mathit{expr}]\!]_e\ \underline{s\ t\texttt{ st;}} \\
\quad \mathit{RETURN}\ \langle x, \rho \rangle \\
[\![\texttt{for(}\mathit{id}\texttt{=}\mathit{expr}_1\texttt{;}\ \mathit{expr}_2\ \mathit{binop}\ \mathit{id}\texttt{;}\ \mathit{id}\texttt{=}\mathit{id}-\mathit{expr}_3\texttt{)}\ \mathit{stmt}_3]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
\quad \mathit{LET}\ y = \mathit{FRESH()}\ \mathit{IN} \\
\quad \mathit{LET}\ \langle f, \rho \rangle = [\![\mathit{id}\ \texttt{=}\ \mathit{expr}_1]\!]_s^\rho\ \mathit{IN} \\
\quad \mathit{LET}\ \langle s, \rho \rangle = [\![\mathit{stmt}_3\texttt{;}\ \mathit{id}\texttt{=}\mathit{id}-\mathit{expr}_3]\!]_s^\rho\ \mathit{IN} \\
\quad \underline{y\texttt{ st }\mathit{id}\texttt{ = if}}\ [\![\mathit{expr}_2]\!]_e\ [\![\mathit{binop}]\!]_o\ \underline{\mathit{id}} \\
\qquad \underline{\texttt{then }y\texttt{ (}s\texttt{ st) (}\mathit{id}\ -}\ [\![\mathit{expr}_3]\!]_e\underline{\texttt{) else st;}} \\
\quad \underline{x\texttt{ st = }y\texttt{ st}}\ [\![\mathit{expr}_1]\!]_e\ \underline{\texttt{;}} \\
\quad \mathit{RETURN}\ \langle x(f), \rho \rangle \\
[\![\texttt{for (}\ \mathit{stmt}_1\ \texttt{;}\ \mathit{expr}\ \texttt{;}\ \mathit{stmt}_2\ \texttt{)}\ \mathit{stmt}_3]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
\quad \mathit{LET}\ \langle f, \rho \rangle = [\![\mathit{stmt}_1]\!]_s^\rho\ \mathit{IN} \\
\quad \mathit{LET}\ e = [\![\mathit{expr}]\!]_{e'}\ \mathit{IN} \\
\quad \mathit{LET}\ \langle s, \rho \rangle = [\![\mathit{stmt}_3\ \texttt{;}\ \mathit{stmt}_2]\!]\ \mathit{IN} \\
\quad \underline{x\texttt{ st = loop }e\ s\texttt{ st;}} \\
\quad \mathit{RETURN}\ \langle x(f), \rho \rangle \\
[\![\texttt{scanf("\%d",\&}\mathit{id}\texttt{)}]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
\quad \mathit{LET}\ (s, t) = \mathit{head}(\mathit{rev}\ \rho_{\mathit{st}})\ \mathit{IN} \\
\quad \mathit{LET}\ \rho' = \rho[\mathit{st} \mapsto \mathit{rev}(\mathit{tail}(\mathit{rev}\ \rho_{\mathit{st}}))]\ \mathit{IN} \\
\quad \underline{\texttt{stream }s\texttt{ from "std_in";}} \\
\quad \underline{x\texttt{ (}\mathit{patt}\ \rho_{\mathit{st}}\texttt{) = (}\mathit{replace\ id}\ s\ \rho'_{\mathit{st}}\texttt{)}} \\
\quad \mathit{RETURN}\ \langle x, \rho' \rangle \\
[\![\texttt{printf("\%d",}\mathit{id}\texttt{)}]\!]_s^\rho \rightsquigarrow \\
\quad \mathit{LET}\ s = \mathit{FRESH()}\ \mathit{IN} \\
\quad \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
\quad \mathit{LET}\ \rho' = \rho[\mathit{os} \mapsto \rho_{\mathit{os}}@[(s, \texttt{int 16})], \\
\qquad\qquad \mathit{st} \mapsto \rho_{\mathit{st}}@[(s, \texttt{int 16})]]\ \mathit{IN} \\
\quad \underline{\texttt{stream }s\texttt{ to "std_out";}} \\
\quad \underline{x\texttt{ (}\mathit{patt}\ \rho_{\mathit{st}}\texttt{) = (}\mathit{patt}\ \rho_{\mathit{st}}\texttt{ , }\mathit{id}\texttt{);}} \\
\quad \mathit{RETURN}\ \langle x, \rho' \rangle
\end{array}$$

Fig. 3. Translation rules for *miniC* statements

```
storeC (update (loadC st)((((loadI st)*3)
                + (loadJ st))+1)0) st;
```

miniC control structures are represented by a common set of *higher order functions* (HOFs):

```
exIf true s st = (s st);
exIf false _ st = st;
exIfElse true s _ st = (s st);
exIfElse false _ t st = (t st);
loop e s st = if ((e st)
                then (loop e s (s st))
                else st;
```

`exIf` represents an `if`-statement without an `else`-clause, while `exIfElse` represent the version with an `else`-clause. Both functions are defined by pattern-matching the input condition (the Hume translation of the original condition), and are parameterised on the state space (`st`). `exIf` also takes the translation of the body of the `if`-statement, represented as a function from the state-space tuple into a new state-space tuple, while `exIfElse` also takes the translated `else`-clause. The `loop` HOF captures all but the initialisation statement of a `for` loop. Note that because the condition refers to a variable which changes value, this is defined as a predicate on the state-space rather than a boolean value. Unfortunately, `loop` is often too generic to obtain an adequate cost from the Hume analysis tools. Thus, a more costable subset is created using a specific translation rule (the first rule for `for`-statements in Figure 3). We use this rule rather than the second, generic, rule whenever a `for`-statement has the form

`for(`*id=expr*`;` int_1`<` *id*`;` $id = id - int_2$`)` *cbody*
`for(`*id=expr*`;` int_1`<=`*id*`;` $id = id - int_2$`)` *cbody*

and *cbody* does not change *id*, i.e. does not contain a statement of the form *id = expr*. Since all three `for`-statements of the matrix multiplication example are of this form, we can now use this rule to create specific functions in each case rather than using the generic HOF. For example, the innermost `for` statement can be translated as:

```
g st k = (if (0 <= k)
          then (g (fbody st) (k - 1))
          else st);
h st = g (i st) 2;
```

where `fbody` is the function identifier holding the result of translating the loop body (including `k=k-1`), `i` translates the initialisation statement `k = 2`, and `h` initialises the accumulator variable.

The environment ρ is required for the translation of streams. $[\![$`scanf("%d",&`id`)`$]\!]^{\rho}_{s}$ yields an additional input wire for the program box, wired to the Hume standard input stream, while $[\![$`printf("%d",`id`)`$]\!]^{\rho}_{s}$ results in an additional Hume output stream, wired to standard output. Furthermore, $[\![$`scanf("%d",&`id`)`$]\!]^{\rho}_{s}$ returns a tuple of length one less than the input, while $[\![$`printf("%d",`id`)`$]\!]^{\rho}_{s}$ adds one element to the tuple. To achieve this, ρ contains two partial maps *st* and *os*. We use a subscript to access these variables (e.g. ρ_{st}), and define ρ^e to be the empty environment. $\rho[st \mapsto e]$ is ρ with the exception of ρ_{st} which is now e. For input streams, the environment (ρ_{st}) is assumed to already have the state space (from *decl*) and the input streams. Thus, this must be created

$[\![$*prelude* `main()`$\{$ *decls stmts* $\}]\!] \leadsto$
LET $s = sp\ decls$ *IN*
LET $istrs = is\ stmts$ *IN*
LET $\rho = \rho^e[st \mapsto s \ @\ istrs]$ *IN*
$[\![\mathrm{s}]\!]_{load}$ $[\![\mathrm{s}]\!]_{store}$
LET $\langle e, \rho' \rangle = [\![stmts]\!]^{\rho}_{s}$ *IN*
LET $x = FRESH()$ *IN*
x (*patt* ρ'_{st}) = (*ignore* s , *patt* ρ'_{os});
`box program`
  `in(` *head* s "" , *head istrs* "")
  `out(` *head* s "," , *head* ρ'_{os} "")
`match`
  (*patt* s ,*patt istrs*) -> x (e (*patt* s,*patt istrs*)) ;
`wire program`
  (*init_wires* `program` s "," , *patt istrs*)
  (*wires* `program` s "" , *patt* ρ'_{os}) ;

Fig. 4. Translation rules for *miniC* programs

before the translation is performed. The input stream list of variable/type pair is found by the following meta-function:

$$
\begin{array}{lcl}
is\ (stmt_1 \texttt{;} stmt_2) & = & is\ (stmt_1)\ @\ is\ (stmt_2) \\
is\ (\texttt{scanf("\%d",\&}id\texttt{)}) & = & [(FRESH(), \texttt{int 16})] \\
is\ \_ & = & [].
\end{array}
$$

where _ succeeds for any value. In ρ_{st}, the list created by *is* is assumed to be appended to the state-space list (created by *sp*). $[\![$`scanf("%d",&`id`)`$]\!]^{\rho}_{s}$ creates a stream, and removes the last element of ρ_{st}. Note that *rev* reverses a list, while *head* and *tail* return respectively the head and tail of a list. As an example, the function `f` created from a `scanf("%d",&x)`, where the input stream is given the *FRESH()* name s, and the program variables are `x`, `y` and `z`, becomes:

```
f (x,y,z,s) = (s,y,z);
```

For $[\![$`printf("%d",`id`)`$]\!]^{\rho}_{s}$, ρ_{os} is assumed to be initially empty. ρ_{os} is used to link the "program box" and the output stream. The rule adds the output to both partial maps of ρ, creates a new output stream and a function that maps the correct variable to this output. For example, `printf("%d",x)`, with the above assumptions becomes:

```
f (x,y,z) = (x,y,z,x);
```

D. The full program

Figure 4 shows the $[\![program]\!]$ translation rule for a full *miniC* program. The *prelude* contains any required C libraries (to enable support for C compilers), and is thus not required in Hume. First, the "state space list" and "input streams list" are generated, and the initial environment created. Then the `load` and `store` functions are generated, and all the "statement functions" created. Then a function x is created which discards all variables except the output streams, replacing them with the Hume $\star$ (ignore) construct. This stops the program from entering an infinite loop. Next, the `program` box is created. The state-space, together with any optional inputs from the inputs for the box. Similarly, the state-space and outputs are the box outputs. The *head* function creates a syntactically-correct box header, containing the variable names and types. The last argument of this function is a suffix which is added to the end of the name to ensure that all input and output names are unique. Thus, the *miniC* variable, `x`, becomes the input variable, `x`, and the output variable, `x'`. The state-space is wired as a feedback loop. Thus, e.g. `program.x'` is wired back to `program.x`. This is achieved by the *wires* and *init_wires* rules. *wires* is defined as

$$\begin{array}{lll} \mathit{wires}\ \mathit{box}\ ((\mathit{id},T):[])\ l & \rightsquigarrow & \mathit{box}\underline{.}\mathit{id}\ l \\ \mathit{wires}\ \mathit{box}\ ((\mathit{id},T):\mathit{sp})\ l & \rightsquigarrow & \mathit{box}\underline{.}\mathit{id}\ l\ \underline{,}\ \mathit{wires}\ \mathit{box}\ \mathit{sp}\ l \end{array}$$

while *init_wires* is similar but also gives all variables an initial value. This is required since all Hume variables must have a value. Thus, initially all values (including vector elements) are `0`. This may, of course, deviate from *miniC*, which is simply given the old value of the stack cell it is allocated to, without really having any effect when applying the Hume analysis.

IV. Hume Resource Analysis

Accurately determining the cost of executing a program faces two primary difficulties: firstly, it is generally necessary to reduce the complexity of the program in the cost metric, since there is little insight gained if the description of the cost is as complex as the program itself; and secondly, the costs of executing individual machine instructions/source statements may vary significantly, depending on the machine state, but tracking all possible machine states at all program points is infeasible in general. The solution lies in abstraction, trading precision for clarity. However, one has to be very careful how this is done. For example, achieving constant costs by simply assuming the worst-case over all consistent states, as opposed to all states that could be possible at a certain point for a certain input, does not work, since such a naive approach would assign infinite worst-case execution time to well-behaved functions such as sleep(n). Compared with other approaches, our analysis is particularly 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. The analysis then constructs very simple linear constraints describing relative changes to the potential. This ensures both simple understandable bounds and highly efficient solving. Furthermore it allows the analysis to scale very well for increasingly complex and large programs.

It is important to 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 relative effect of each program step on the overall potential and define the *amortised cost* of an instruction $\mathcal{I}$ as a suitable constant such that

$$\begin{aligned} \mathit{amortised\ cost}_{\mathcal{I}} \geq{} & \text{actual cost}_{\mathcal{I}} \\ & - \text{potential before}_{\mathcal{I}} + \text{potential after}_{\mathcal{I}} \end{aligned}$$

holds for all possible states, with equality being preferred. The benefit is that determining the amortised cost for an entire sequence of operations is then very easy, since the amortised cost is constant and no longer depends on the state. The *actual cost* of the entire sequence is then bounded by the sum of the amortised costs plus the potential of the initial state, which is easy to determine. This technique is well known in complexity theory where it is referred to as "Amortised Analysis" [29], and used as a manual analysis technique. A significant challenge of applying this technique automatically is how to design the abstraction of the machine state. This problem has been overcome by Hofmann & Jost's automatic inference system [11], [14], at the expense of restricting the potential so that it depends linearly on the sizes of objects (a restriction which is not inherent to the amortised analysis approach). It follows that only programs whose cost can be linearly bounded by their input can be analysed using the Hofmann & Jost method. However, this still admits many interesting programs.

The automatic analysis first constructs a standard typing derivation for the Hume program. It then associates variables with each different type of input value. The analysis then generates a set of constraints over this set of variables, according to the program's dataflow and the actual costs for each possible path of computation. Each Hume source construct is examined precisely once. Loops in the source program are dealt with by identifying some resource variables from the constraint set. The generated constraint sets are well behaved and can easily be solved using a standard LP-solver, such as [1]. In this way, bounds on resource consumption are associated with each source expression through their types. The potential annotated types then give rise to a simple linear closed-form expression that depends on the input sizes. We have formally proved that these expressions always yield *guaranteed upper bounds* on the resource consumption of the analysed program.

V. Results

Table I summarises the results of analysing and measuring a range of example programs written in *miniC* and compiled to Hume using the translation approach described above. The Hume is then compiled to native code using the `humec` compiler which successively generates Hume Abstract Machine code and C for final compilation with `gcc`. Analysis and instrumentation results are given for a Renesas board incorporating an M32C/85U processor with 24KBytes of RAM. This restricts the maximum possible heap usage to about 4000 4-byte cells.

Despite the memory limitations, we have been able to run and analyse a number of small testbed examples. The `fact` program computes the factorial of 15. The `fibsum` program computes the sum of the first 20 Fibonacci numbers. The `matmult` programs performs a matrix multiplication with input matrices of sizes 2×2 and 3×3, respectively. The `arrayrotate` program rotates the array of length 10. The `mediumarrayrotate` program rotates an array of length 20. Finally, the `smallarraysearch` program searches for an input value in an array of length 25.

HTA/HT in Table I compares analysis results with measured runtime for the Hume code that was generated from the *miniC* source. This is really a "sanity check" on the analysis: it

Program	Hume Time Analysis	Hume Time	HTA/HT	C-code Time	HTA/CT
`fact`	130431	83659	1.56	1751	74
`fibsum`	224298	133193	1.68	1413	159
`matmult2`	464956	229469	2.03	713	652
`matmult3`	1410990	665363	2.12	2126	664
`arrayrotate`	529560	186851	2.83	746	710
`mediumarrayrotate`	1363580	378035	3.61	1737	785
`smallarraysearch`	1235690	477408	2.59	2705	457

TABLE I

ANALYSIS AND MEASUREMENT RESULTS

shows that there is a fair consistency of actual with predicted execution times for Hume code on the Renesas board. The over-estimation is explained by the special structure of the automatically generated Hume code: it consists of many small functions, with heavy use of higher-order functions. The former leads to a high impact of inaccuracies in costing function calls. The latter necessarily leads to defensive approximations, since the analysis must account for all possible instances of the supplied function arguments. However, considering that these results are guaranteed upper bounds on execution time, we find them acceptable.

Of more interest is HTA/CT which shows the ratio of Hume analysis to *miniC* runtime. We would not expect the Hume analysis itself to correspond closely to the *miniC* time: after all, we have compiled *miniC* to Hume to C and so the resulting program is inevitably far less efficient than the original. Nonetheless, the ratios show considerable consistency for four of the test programs: the `matmults` and the `arrayrotates`. In the cases of `fact` and `fibsum` the bounds inferred by the analysis are significantly closer, because these programs use only a small number of variables and no compound data-structures. Therefore, the state space is small and updates are fairly cheap.

VI. RELEVANT WORK

The translation of imperative to functional languages has been known since the first use of functional notations in denotational semantics [25]. In particular, functional meta-languages are commonly deployed in both semantics-directed compiler-compilers[21] and theorem provers. For example, [22] formalises a Java-like language in Isabelle/HOL, while [24] mechanises a more C-like language using the same theorem prover. 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 language LFPL [10], in the Java-like language RAJA [12], [13], in Camelot [20], and for several languages in the AHA project [27]. A stack-space analysis using this approach is given in [2]. A system that combines type-based resource inference with the automatic generation of certificates for such bounded resource consumption is described in [4]. An alternative type-based approach used to infer size bounds is that of *sized types* [17]. Several heap- or stack-space consumption analyses build on this work [17], [16], [23], [3], [32], [7]. While Vasconcelos states in his PhD thesis [31] that this is equally within reach of these methods, none of these currently considers worst-case execution time. Other functional notations with ad-hoc techniques for analysing resource consumption include GeHB [28], with a two-level *staged* notation that also builds on LFPL, and RT-FRP [33], which, like Hume, targets embedded systems.

Finally, a variety of academic and commercial tools exist for calculating guaranteed bounds on worst-case execution time [34], including aiT[5], bound-T[15], SWEET[26], [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 cases, closely conform to the actual execution time. In constructing solutions for concrete programs, however, the programmer must usually provide additional detailed information, and this may require significant effort. 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.

VII. CONCLUSION

We have formalised a translation from a canonical core imperative language *miniC* to Hume, to explore direct use of the Hume WCET analysis to characterise *miniC* programs. Our results suggests that:

- the Hume WCET analysis is in itself fairly robust;
- naive translation from *miniC* to Hume captures salient components of the complexity of the source programs;
- hence, the WCET analyses of translated *miniC* programs may be used to compare at least their *relative* time complexities.

Of course, our translation is naive and we have only conducted experiments on a small cohort of very simple test programs. Nonetheless, our results suggest that this approach would repay further study and that analytic techniques for one language may fruitfully be used directly with another.

We next plan to study the use of the Hume space analyses to characterise space complexity of *miniC*. We have also developed a translation from *miniC* to the coordination layer, and of *miniC* extended with pointers and dynamic memory allocation to the expression layer. We intend to investigate properties of the *miniC* translation to the Hume coordination language and of extended *miniC*.

While our work is *formally motivated* it is by no means yet fully *formalised*. In particular, while we do not anticipate significant technical problems, we would like to prove that the translation from *miniC* to Hume is *sound*, i.e. that it preserves the meanings of source *miniC* programs in the translated Hume code. We have already produced full semantic definitions of Hume [8] and of *miniC*; establishing soundness would require inductive proofs of equivalence of meanings of source and translated target for each *miniC* abstract syntax construct.

Finally, a longer term challenge of substance would be to formally "reverse engineer" the Hume cost model in order to develop a direct WCET model for *miniC*, or even a richer subset of ANSI C.

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] K. Hammond. The Dynamic Semantics of Hume: a Functionally-Based Concurrent Language with Bounded Time and Space Behaviour. In *Proc. 2000 Intl. Workshop on Implementation of Functional Languages*, number 2011 in Lecture Notes in Computer Science, pages 122–139, Aachen, Germany, Sept. 2000. Springer-Verlag.

[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. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proc. POPL '03: ACM Symp. on Principles of Prog. Langs.*, pages 185–197, Jan. 2003.

[13] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In P. Sestoft, editor, *Programming Languages and Systems : 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer-Verlag, 2006.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[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] T. Nipkow, D. v. Oheimb, and C. Pusch. μJava: Embedding a Programming Language in a Theorem Prover. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[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] N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2005.

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

[26] 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.

[27] 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.

[28] 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.

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

[30] D. Turner. Elementary Strong Functional Programming. In *Proc. 1995 Symp. on Functl. Prog. Langs. in Education — FPLE '95*, LNCS. Springer-Verlag, Dec. 1995.

[31] P. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, 2008. submitted.

[32] 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.

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

[34] 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.