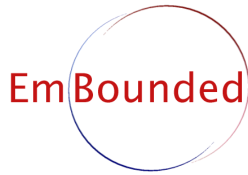

IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D14 (WP3a): Report on WCET analysis

Due date of deliverable: 30th November 2006
Actual submission date: 28th February 2007

Start date of project: 1st March 2005 Duration: 36 months

Lead contractor: St Andrews University Revision: 1.5

Purpose: To construct an effective Worst-Case Execution Time analysis based on a formal semantics for Hume incorporating time cost information (Deliverable D12a/D12b).

Results: We have constructed a resource-aware type system for a slightly simplified version of HUME, whose annotated typings yield provable upper bounds on WCET of the typed terms.

Conclusion: The principal goal of this piece of work has been successfully achieved. We have constructed a formal analysis that is capable of yielding upper-bound costs on WCET.

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

Report on WCET analysis

Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond

Abstract

We describe a formally-based static analysis for determining Worst-Case Excecution Time (WCET) for HUME, based on the Time costs as defined in the formal HUME Semantics (Deliverable D12 [WP7a]). Our analysis consists of an elaborate type system, whose typings yield strict upper bounds on the Worst-Case Execution Time of the typed terms. Our approach is as follows:

a) We enhance the standard type system for the HUME language defined in Deliverable D11, which forms a common basis for constructing analyses, for Worst Case Execution Time (this deliverable), Stack-space usage (Deliverable D5) and Heap space usage (Deliverable D11). This type system is enhanced with an amortised cost analysis for Worst-Case Execution Time. in accordance to the Time costs defined in the formal HUME Semantics (Deliverable D12). The enhancement consists of added annotations to each type, referred to as *"Potential"*, and a number of side-conditions to each type-rule governing the use of the *potential* Worst-Case Execution Time.

b) Abstraction of the *potential* WCET then leads to a standard linear program, which can be solved easily by standard techniques. Any solution to the derived linear program that is found by the solver can then be used to assign a concrete potential to the input of a program; this potential then furnishes an upper bound on the WCET for the computation of the analysed program for that input.

Through the nature of a type-based approach to generating constraints, and since we are able to solve the generated linear program at compile time using standard, efficient solver technology this directly yields an *efficient* static WCET analysis for HUME. Since this report is similar to that describing our Heap-space analysis (Deliverable D11), we will focus on the specific requirements for analysing the Worst-Case Execution Time as opposed to Heap-space, omitting those aspects that are common to the two metrics.

Contents

1 Notational Preliminaries **3**

2 A Generic Formal Foundation for the HUME Resource Analyses **4**

3 Determining Worst-Case Execution Time **4**
- 3.1 Annotated Type Rules for Pattern Matches 8
- 3.2 Annotated Type Rules for Boxes and Declarations 9
- 3.3 Timing Complete Hume Programs 10

4 Concrete Time Costs for the Renesas M32C/85 Processor **10**
- 4.1 Determining WCET using the aiT tool 10
- 4.2 HAM Instruction WCET Costs for Renesas M32C/85 12
- 4.3 Timing Results 13
- 4.4 Quality of the Static Analysis using the aiT Tool 13

5 Simple Analysis Examples **14**
- 5.1 Example: factorial function 14
- 5.2 Example: sum-over-list 16
- 5.3 Example: multiplication (box- vs expression-level) 17
- 5.4 Example: vending machine 19
- 5.5 Example: core of Canny edge detection 23

6 Summary **24**

1 Notational Preliminaries

We denote the non-negative rational numbers by $\mathbb{Q}^+$. We denote $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$, i.e., the set of non-negative real numbers together with an element ∞. Ordering and addition on $\mathbb{R}^+$ extend to $\mathbb{D}$ by $\infty + x = x + \infty = \infty$ and $x \leq \infty$. If U is a subset of $\mathbb{D}$ we write $\sum U$ for the (possibly infinite) sum over all its elements. Since $\mathbb{D}$ contains no negative numbers, questions of ordering and non-absolute convergence do not play a role; it is also the case that any subset of $\mathbb{D}$ has a sum, perhaps ∞. We write $\sum_{i \in I} x_i$ for $\sum\{x_i \mid i{\in}I\}$ and use other similar standard notations.

For index numbers ranging over the natural number $\mathbb{N}$, we commonly omit the limits if they are clear from the context, i.e. if $A_1, \ldots, A_k$ are defined in the surrounding context, then stating that formula Ψ_i is true for all i between 1 and k can be abbreviated as $\forall i \,.\, \Psi_i$. Furthermore, writing $\forall i \,.\, \mathcal{A}_i = \{1, \ldots, (i-1)\}$ states that $\mathcal{A}_1$ is empty, $\mathcal{A}_2$ is the singleton set containing the number one, and so on, eventually stating that $\mathcal{A}_k$ contains all natural numbers from 1 up to and excluding k.

The disjoint union of sets is denoted by $\dot\cup$. Since we are often dealing with multisets, we write $\uplus$ for the multiset union in order to distinguish it from the ordinary union of sets $\cup$.

For partial maps, we use the following notations when used to model stacks, heaps, or typing contexts: Let f be a partial map. The domain, co-domain and range (i.e. the image of the domain) of f are denoted by $\mathrm{dom}(f)$, $\mathrm{codom}(f)$ and $\mathrm{ran}\,(f)$ respectively. We may abbreviate $x \in \mathrm{dom}(f)$ by $x \in f$ and sometimes $f(x)$ by writing f_x. We denote by $f[x \mapsto y]$ the partial map that sends x to y and acts like f otherwise. Conversely, $f \setminus x$ denotes the map which is undefined for x and acts like f otherwise. The restriction of f on $\mathcal{X}$ is written $f{\upharpoonright}\mathcal{X}$, i.e. the map that acts like f for all $x \in \mathcal{X}$ and that is undefined otherwise. The empty map $\varnothing$ is often omitted, i.e. $[x \mapsto y]$ denotes the singleton map sending x to y. We write f, g for the disjoint union of the partial maps f and g. The expression f, g is undefined if the domains of f and g are not disjoint. In the special case of typing contexts, we allow ourselves to write simply $x{:}A$ for the partial maps otherwise denoted by $[x \mapsto A]$.

We write $[\,]$ to denote the empty list and $[a, b, c]$ for the list containing the elements a, b and c. The concatenation of two lists is written $l_1 +\!\!+\, l_2$. If h is a suitable element and t is a (possibly empty) list, then $h :: t$ is the list obtained by prepending h to t; while $t +\!\!+ [h]$ denotes the list obtained by attaching element h at the end of list t. The cardinality of a list l is denoted by $|l|$, e.g. $|[a, b, c]| = 3$. We identify each list with its own index map, i.e. if $l = [a, b, c]$ then $l_2 = b$ according to our previously introduced abbreviations. The expression l_5 is undefined in this example. We may sometimes write $\vec{l}$ for a list l to enhance readability.

Finally, for readability within program examples, we allow ourselves to replace unimportant free variables by the underscore symbol '_', as in Hume. Multiple occurrences of the underscore symbol '_' stand for different unnamed variables as usual, hence they are not connected with each other in any way.

2 A Generic Formal Foundation for the HUME Resource Analyses

Section 2 of Deliverable D11 describes a generic foundation for constructing static analyses for determining time and space costs for HUME programs, based on the formal cost semantics of Deliverable D12. Since this common foundation will be exploited throughout this report, we will therefore assume familiarity with Deliverable D11. We refer the reader to that deliverable for further details on notation, and explanations of the approach etc.

3 Determining Worst-Case Execution Time

The basic definitions of annotated types given in Section 3 of Deliverable D11 also hold for Worst-Case Execution Time (WCET). The constructor of each variant-type is annotated with a resource variable $\in \mathsf{CV}$, representing a factor determining the potential associated with all values of this type. Note that the notion of potential is entirely independent of the resource that is being analysed. Therefore, any valid annotated type derivation will require a different *valuation*, i.e. that different values $\in \mathbb{Q}^+$ are assigned to the resource variables for different types of resource.

The type rules for expressions retain the form

$$\Sigma; \Gamma \mathrel{\vdash^{t}_{t'}} e : A \mid \psi$$

where Γ is a context mapping variables belonging to the set Var to enriched HUME types $m, m' \in \mathsf{CV}$, e is the HUME expression, A is an enriched HUME type, and ψ is a set of constraints involving resource variables $\in \mathsf{CV}$ and constant values $\in \mathbb{Q}^+$. However, the meaning of this statement, when derived using the type rules detailed below, is now as follows: For all valuations $\boldsymbol{v}$ mapping all resource variables to $\mathbb{Q}^+$ such that the constraint set $\boldsymbol{v}(\psi)$ is satisfied, the HUME expression e has type $\boldsymbol{v}(A)$ in the context $\boldsymbol{v}(\Gamma)$. Furthermore, for all memory configurations consisting of environment $\mathcal{V}$ and heap η fitting the context Γ, executing the expression e will require at most $\boldsymbol{v}(m) + \Phi_\eta\big(\mathcal{V} : \boldsymbol{v}(\Gamma)\big)$ time units (where a time unit is usually defined as a single clock cycle of the processor, but other units such as nanoseconds are also possible if desired).

Furthermore, if the computation finishes with heap η' and result value v, then at least $\boldsymbol{v}(m') + \Phi_{\eta'}\big(v : \boldsymbol{v}(\Gamma)\big)$ time units remain unused after the computation has finished. This notion of unused time units is required for compositionality. This can be formulated in the following theorem:

Theorem 1 (Correctness)**.** *Fix a HUME program.*

$$\Gamma \mathrel{\vdash^{q}_{q'}} e{:}A \mid \phi \tag{1.I}$$

$$\mathcal{V}, \eta \vdash e \rightsquigarrow \ell, \eta' \tag{1.II}$$

$$\eta \models \mathcal{V} : \Gamma \tag{1.III}$$

$$\boldsymbol{v} : \mathsf{CV} \to \mathbb{Q}^+, \text{ satisfying } \phi \tag{1.IV}$$

If the four statements above are all satisfied, then it follows that for all $r \in \mathbb{Q}^+$ and $p \in \mathbb{N}$ such that $p \geq \boldsymbol{v}(q) + \Phi_\eta\big(\mathcal{V} : \boldsymbol{v}(\Gamma)\big) + r$ holds, there exists $p' \in \mathbb{N}$ satisfying $p' \geq \boldsymbol{v}(q') + \phi_{\eta'}\big(v{:}\boldsymbol{v}(A)\big) + r$ and also $\eta, \mathcal{V} \mathrel{\vdash^{t}_{t'}} e \rightsquigarrow_\diamond v, \eta'$ for some $p, p' \in \mathbb{Q}^+$.

The statement $\eta, \mathcal{V} \mathrel{\vdash^{t}_{t'}} e \rightsquigarrow_\diamond v, \eta'$ refers to the formal HUME operational semantics (Deliverable D12), as defined for Worst-Case Execution Time. This specifies that e evaluates successfully in the given memory configuration with at least t time-units available and that at least t' time-units remain unused after the evaluation is finished. It therefore *measures* the exact cost of an evaluation given some input, whereas the annotated type rules which we will now define *bound* the Worst-Case Execution Time for

evaluating an expression for *all* well-typed inputs. We now define the annotated type rules for HUME expressions for *bounding* Worst-Case Execution Time:

$$\frac{}{\varnothing \mathrel{\vdash^{\texttt{Tmkunit}}_{0}} \texttt{()} : \mathsf{unit} \mid \emptyset} \quad \text{(UNIT)}$$

$$\frac{b \in \mathbb{B}}{\varnothing \mathrel{\vdash^{\texttt{Tmkbool}}_{0}} b : \mathsf{bool} \mid \emptyset} \quad \text{(BOOL)}$$

$$\frac{n \in \mathbb{Z}}{\varnothing \mathrel{\vdash^{\texttt{Tmkint}}_{0}} n : \mathsf{int} \mid \emptyset} \quad \text{(INT)}$$

$$\frac{r \in \mathbb{R}}{\varnothing \mathrel{\vdash^{\texttt{Tmkfloat}}_{0}} r : \mathsf{float} \mid \emptyset} \quad \text{(FLOAT)}$$

$$\frac{c \texttt{ is a character}}{\varnothing \mathrel{\vdash^{\texttt{Tmkchar}}_{0}} c : \mathsf{char} \mid \emptyset} \quad \text{(CHAR)}$$

$$\frac{s \texttt{ is a string}}{\varnothing \mathrel{\vdash^{\texttt{Tmkstring}(|\mathsf{s}|)}_{0}} s : \mathsf{string} \mid \emptyset} \quad \text{(STRING)}$$

$$\frac{}{x{:}A \mathrel{\vdash^{\texttt{Tpushvar}}_{0}} x : A \mid \emptyset} \quad \text{(VAR)}$$

$$\frac{\mathrm{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{x{:}\mathsf{int}, y{:}\mathsf{int} \mathrel{\vdash^{\texttt{Tcallprim(op)}}_{0}} x \ \mathrm{op}\ y : \mathsf{int} \mid \emptyset} \quad \text{(PRIMBOP INT)}$$

$$\frac{\mathrm{op} \in \{\texttt{-}\}}{x{:}\mathsf{int} \mathrel{\vdash^{\texttt{Tcallprim(op)}}_{0}} \mathrm{op}\ y : \mathsf{int} \mid \emptyset} \quad \text{(PRIMUOP INT)}$$

$$\frac{\mathrm{op} \in \{\texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}\}}{x{:}\mathsf{float}, y{:}\mathsf{float} \mathrel{\vdash^{\texttt{Tcallprim(op)}}_{0}} x \ \mathrm{op}\ y : \mathsf{float} \mid \emptyset} \quad \text{(PRIMBOP FLOAT)}$$

$$\frac{\mathrm{op} \in \{\texttt{-.}\}}{x{:}\mathsf{float} \mathrel{\vdash^{\texttt{Tcallprim(op)}}_{0}} \mathrm{op}\ y : \mathsf{float} \mid \emptyset} \quad \text{(PRIMUOP FLOAT)}$$

$$\frac{\mathrm{op} \in \{\texttt{==}, \texttt{>=}, \texttt{<=}\}}{x{:}A, y{:}A \mathrel{\vdash^{\texttt{Tcallprim(op)}}_{0}} x \ \mathrm{op}\ y : \mathsf{bool} \mid \emptyset} \quad \text{(PRIMBOP EQ)}$$

$$\frac{\mathrm{op} \in \{\texttt{and}, \texttt{or}\}}{x{:}\mathsf{bool}, y{:}\mathsf{bool} \mathrel{\vdash^{\texttt{Tcallprim(op)}}_{0}} x \ \mathrm{op}\ y : \mathsf{bool} \mid \emptyset} \quad \text{(PRIMBOP BOOL)}$$

$$\frac{\mathrm{op} \in \{\texttt{not}\}}{x{:}\mathsf{bool} \mathrel{\vdash^{\texttt{Tcallprim(op)}}_{0}} \mathrm{op}\ y : \mathsf{bool} \mid \emptyset} \quad \text{(PRIMUOP BOOL)}$$

$$\frac{\Sigma^{\mathcal{F}}(\mathit{fid}) = \big(_;_;\ \xrightarrow[t']{t} C\big)}{\varnothing \vdash^{t}_{t'} \mathit{fid} : C \mid \emptyset} \quad \text{(App0)}$$

$$\frac{\Sigma(\mathit{fid}) = \big(_;_;\forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[t']{t} C\big) \qquad k \geq 0 \qquad k = a}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{t}_{t'} \mathit{fid}\ y_1\cdots y_k : C \mid \psi} \quad \text{(App)}$$

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \big(_;_;\forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[t']{t} C\big) \qquad k \geq 1 \qquad k < a\\ \forall i \leq k\,.\,\big(\phi_i = \curlyvee(A_i\,|A_i,A_i)\big) \qquad \beta = \alpha \setminus \mathrm{FV}\big(A_1,\ldots,A_k\big)\end{array}}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{\texttt{Tmkfun(k)}}_{0} \mathit{fid}\ y_1\cdots y_k : \forall\beta\in\psi.A_{k+1},\ldots,A_a \xrightarrow[t'+\texttt{Treturn}+\texttt{Tslide}]{t+\texttt{Tcall}} C \mid \bigcup_i \phi_i} \quad \text{(Under App)}$$

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \big(_;_;\forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[t']{t} C\big) \qquad a \geq 1 \qquad k > a\\ y_1{:}A_1,\ldots,y_a{:}A_a \vdash^{t}_{t'} \mathit{fid}\ y_1\cdots y_a : C \mid \phi \qquad x \text{ is fresh}\\ x{:}C, y_{a+1}{:}A_{a+1},\ldots,y_k{:}A_k \vdash^{t'-\texttt{Tap}}_{t''} x\ y_{a+1}\cdots y_k : E \mid \chi\end{array}}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{t}_{t''-\texttt{Tslide}} \mathit{fid}\ y_1\cdots y_k : E \mid \phi\cup\psi\cup\chi} \quad \text{(Over App)}$$

$$\frac{D = \forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[t']{t} C \qquad k \geq 1 \qquad k = a}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{t}_{t'} z\ y_1\cdots y_k : C \mid \psi} \quad \text{(App Var)}$$

$$\frac{\begin{array}{c}D = \forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[t']{t} C \qquad k \geq 1 \qquad k < a\\ \forall i \leq k\,.\,\big(\phi_i = \curlyvee(A_i\,|A_i,A_i)\big) \qquad \beta = \alpha \setminus \mathrm{FV}\big(A_1,\ldots,A_k\big)\\ B = \forall\beta\in\psi.A_{k+1},\ldots,A_a \xrightarrow[t'+\texttt{Treturn}+\texttt{Tslide}]{t+\texttt{Tcall}} C\end{array}}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{\texttt{Tmkfun(k)}}_{0} z\ y_1\cdots y_k : B \mid \bigcup_i \phi_i} \quad \text{(Under App Var)}$$

$$\frac{\begin{array}{c}D = \forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[t']{t} C \qquad a \geq 1 \qquad k > a\\ y_1{:}A_1,\ldots,y_a{:}A_a \vdash^{t}_{t'} z\ y_1\cdots y_a : C \mid \phi \qquad x \text{ is fresh}\\ x{:}C, y_{a+1}{:}A_{a+1},\ldots,y_k{:}A_k \vdash^{t'-\texttt{Tap}}_{t''} x\ y_{a+1}\cdots y_k : E \mid \chi\end{array}}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{t}_{t''-\texttt{Tslide}} z\ y_1\cdots y_k : E \mid \phi\cup\psi\cup\chi} \quad \text{(Over App Var)}$$

$$\frac{\begin{array}{c}c \in \mathsf{Con} \qquad C = \mu X.\{\cdots \mid c : q; B_1,\ldots,B_k \mid \cdots\}\\ A_i = B_i \vee (A_i = C \wedge B_i = X)\ (\text{for } i = 1,\ldots,k)\end{array}}{x_1{:}A_1,\ldots,x_k{:}A_k \vdash^{q+\textsc{Size}(c)}_{0} c\ x_1\ldots x_k : C \mid \emptyset} \quad \text{(Constr)}$$

$$\frac{\Gamma \vdash^{t-\texttt{Tiftrue}}_{t'} e_t : A \mid \phi \qquad \Gamma \vdash^{t-\texttt{Tiffalse}}_{t'+\texttt{Tgoto}} e_f : A \mid \psi}{\Gamma, x{:}\mathsf{bool} \vdash^{t}_{t'} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A \mid \phi\cup\psi} \quad \text{(Conditional)}$$

$$\frac{\text{ck} \in \{\texttt{exprcase}, \texttt{funcase}\} \qquad \forall i\,.\ \begin{cases} A \vdash^{t_{(i-1)} - \texttt{Tmatchrule}}_{t_i} pat_i \mathrel{?\triangleright} \Delta_i; \pi_i; \psi_i \\ \Gamma, \Delta_i \vdash^{t'_i - \texttt{Tmatchedrule}}_{t''} e_i : B \mid \phi_i \\ \chi_i = \{\pi_i + t_i \geq t'_i\} \end{cases}}{\Gamma, x{:}A \vdash^{t_0 + \texttt{Tcall} + \texttt{Tcreateframe}}_{t'' - \texttt{Tslide} - \texttt{Treturn} - \texttt{Tgoto}} \texttt{case}\ \text{ck}\ x\ \texttt{of}\ pat_1\ \texttt{->}\ e_1 \,|\, \cdots \,|\, pat_k\ \texttt{->}\ e_k : B \mid \bigcup_i \psi_i \cup \phi_i \cup \chi_i} \text{(Case Expr/Fun)}$$

$$\frac{\begin{array}{c}\text{ck} = \texttt{boxcase} \\ \chi_0 = \{t \geq t_0 + \texttt{Tcall} + \texttt{Tcreateframe}\}\end{array} \qquad \forall i\,.\ \begin{cases} A \vdash^{t_{(i-1)} - \texttt{Tmatchrule}}_{t_i} pat_i \mathrel{?\triangleright} \Delta_i; \pi_i; \psi_i \\ \Gamma, \Delta_i \vdash^{t'_i - \texttt{Tmatchedrule}}_{t''} e_i : B \mid \phi_i \\ \chi_i = \{\pi_i + t_i \geq t'_i + t_0\} \end{cases}}{\Gamma, x{:}A \vdash^{t_0}_{t'' - \texttt{Tslide} - \texttt{Treturn} - \texttt{Tgoto}} \texttt{case}\ \text{ck}\ x\ \texttt{of}\ pat_1\ \texttt{->}\ e_1 \,|\, \cdots \,|\, pat_k\ \texttt{->}\ e_k : B \mid \bigcup_i \psi_i \cup \phi_i \cup \chi_i} \text{(Case Box)}$$

Note the different definition of χ_i here compared with that used for expressions: A coordination-layer case-expression has no initial potential for the cost of the pattern match itself. Hence, we use a new variable t to pay for this cost, which must then be justified from the potential that is gained during the pattern match, which we know must have succeeded for this rule to be used.

$$\frac{\forall i\,.\ \begin{cases} \Delta_i = \{x_1{:}A^i_1, \ldots, x_{(i-1)}{:}A^i_{(i-1)}\} \restriction \mathrm{FV}(e_i) \\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\}) \\ \psi_i = \curlyvee(A_i \,|\, \mathcal{A}_i) \\ \Delta_i, \Gamma_i \vdash^{t_{(i-1)} - \texttt{Tmakevar}}_{t_i} e_i : A_i \mid \phi_i \end{cases}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash^{t_0}_{t_{k+1}} \texttt{LET}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{IN}\ e : A \mid \bigcup_i \phi_1 \cup \psi_i} \text{(Ghost let)}$$

$$\frac{\forall i\,.\ \begin{cases} \Delta_i = \{x_1{:}A^i_1, \ldots, x_{(i-1)}{:}A^i_{(i-1)}\} \restriction \mathrm{FV}(e_i) \\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\}) \\ \psi_i = \curlyvee(A_i \,|\, \mathcal{A}_i) \\ i \neq (k+1) \implies \Delta_i, \Gamma_i \vdash^{t_{(i-1)} - \texttt{Tmakevar}}_{t_i} e_i : A_i \mid \phi_i \end{cases} \qquad \Delta_{(k+1)}, \Gamma_{(k+1)} \vdash^{t_k}_{t_{(k+1)}} e_{k+1} : A_{k+1} \mid \phi_{(k+1)}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash^{t_0 + \texttt{Tcall} + \texttt{Tcreateframe}}_{t_{(k+1)} - \texttt{Tgoto} - \texttt{Treturn}} \texttt{let}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{in}\ e_{k+1} : A_{k+1} \mid \bigcup_i \phi_i \cup \psi_i} \text{(Let)}$$

Exceptions

$$\frac{\Gamma \vdash^{t}_{t'} e : Err \mid \psi}{\Gamma \vdash^{t + \texttt{Traise}}_{t'} \texttt{raise}\ exn\ e : A \mid \psi} \text{(Raise)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{t}_{t' + \texttt{Tgoto} + \texttt{Tdonewithin}} e : A \mid \phi \\ \Delta \vdash^{t' + \texttt{Traisewithin}}_{t''} \texttt{raise}\ exn\ e_x : A \mid \psi\end{array}}{\Gamma, \Delta \vdash^{t + \texttt{Twithin}}_{t''} e\ \texttt{within}\ q\ \texttt{time}\ \texttt{raise}\ exn\ e_x : A \mid \phi \cup \psi} \text{(Within Time)}$$

$$\frac{\Gamma \vdash^{t}_{t' + \texttt{Tgoto} + \texttt{Tdonewithin}} e : A \mid \phi \qquad \Delta \vdash^{t' + \texttt{Traisewithin}}_{t''} \texttt{raise}\ exn\ e_x : A \mid \psi}{\Gamma, \Delta \vdash^{t + \texttt{Twithin}}_{t''} e\ \texttt{within}\ q\ \texttt{stack}\ \texttt{raise}\ exn\ e_x : A \mid \phi \cup \psi} \quad \text{(Within Stack)}$$

$$\frac{\Gamma \vdash^{t}_{t' + \texttt{Tgoto} + \texttt{Tdonewithin}} e : A \mid \phi \qquad \Delta \vdash^{t' + \texttt{Traisewithin}}_{t''} \texttt{raise}\ exn\ e_x : A \mid \psi}{\Gamma, \Delta \vdash^{t + \texttt{Twithin}}_{t''} e\ \texttt{within}\ q\ \texttt{heap}\ \texttt{raise}\ exn\ e_x : A \mid \phi \cup \psi} \quad \text{(Within Heap)}$$

Substructural rules

$$\frac{\Gamma, x{:}B \vdash^{t}_{t'} e : C \mid \psi}{\Gamma, x{:}A \vdash^{t}_{t'} e : C \mid \psi \cup A <: B} \quad \text{(Supertype)}$$

$$\frac{\Gamma \vdash^{t}_{t'} e : C \mid \psi}{\Gamma \vdash^{t}_{t'} e : D \mid \psi \cup C <: D} \quad \text{(Subtype)}$$

$$\frac{\Gamma \vdash^{r}_{r'} e : A \mid \psi}{\Gamma \vdash^{t}_{t'} e : A \mid \psi \cup \{t \geq r, t - r \geq t' - r'\}} \quad \text{(Relax Time)}$$

$$\frac{\Gamma \vdash^{t}_{t'} e : C \mid \psi}{\Gamma, x{:}A \vdash^{t}_{t'} e : C \mid \psi} \quad \text{(Weak)}$$

$$\frac{\Gamma, x{:}A_1, y{:}A_2 \vdash^{?}_{?} e : C \mid \phi}{\Gamma, z{:}A \vdash^{t}_{t'} e[z/x, z/y] : C \mid \phi \cup \curlyvee(A \mid A_1, A_2)} \quad \text{(Share)}$$

The definitions for sharing $\curlyvee(A \mid A_1, A_2, \ldots)$ and subtyping $A <: B$ of Deliverable D11 remain unaltered since they deal with an abstract notion of potential obtained by manipulating resource variables in a general, resource-independent manner

3.1 Annotated Type Rules for Pattern Matches

The annotated type rules dealing with pattern matches have the form

$$A \vdash^{t}_{t'} pat \mathbin{?\triangleright} \Gamma; \pi; \psi$$

where *pat* is a HUME pattern to be tested against an object of enriched HUME type A. If the pattern matches successfully, then the resulting bindings are given by the context Γ; the released potential is given in π as a linear combination of resource variables; and any constraints that arise are collected in ψ. The time required to test this pattern is at most t time-units, of which at least t' remain after the pattern match is finished, regardless of the outcome of the test.

$$\frac{}{\mathsf{unit} \vdash^{\texttt{Tmatchunit}}_{0} \texttt{()} \mathbin{?\triangleright} \varnothing; 0; \emptyset} \quad \text{(Pattern Unit)}$$

$$\frac{b \in \mathbb{B}}{\mathsf{bool} \vdash^{\texttt{Tmatchbool}}_{0} b \mathbin{?\triangleright} \varnothing; 0; \emptyset} \quad \text{(Pattern Bool)}$$

$$\frac{n \in \mathbb{Z}}{\mathsf{int} \vdash^{\texttt{Tmatchint}}_{0} n \mathrel{?\triangleright} \varnothing; 0; \emptyset} \quad \text{(Pattern Int)}$$

$$\frac{r \in \mathbb{R}}{\mathsf{float} \vdash^{\texttt{Tmatchfloat}}_{0} r \mathrel{?\triangleright} \varnothing; 0; \emptyset} \quad \text{(Pattern Float)}$$

$$\frac{c \texttt{ is a character}}{\mathsf{char} \vdash^{\texttt{Tmatchchar}}_{0} c \mathrel{?\triangleright} \varnothing; 0; \emptyset} \quad \text{(Pattern Char)}$$

$$\frac{s \texttt{ is a string}}{\mathsf{string} \vdash^{\texttt{Tmatchstring}(|s|)}_{0} s \mathrel{?\triangleright} \varnothing; 0; \emptyset} \quad \text{(Pattern String)}$$

$$\frac{}{A \vdash^{\texttt{Tmatchvar}}_{0} x \mathrel{?\triangleright} x{:}A; 0; \emptyset} \quad \text{(Pattern Var)}$$

$$\frac{}{A \vdash^{\texttt{Tmatchany}}_{0} _ \mathrel{?\triangleright} \varnothing; 0; \emptyset} \quad \text{(Pattern Wild)}$$

$$\frac{\forall i \,.\, \left(B_i \vdash^{t_{i-1}}_{t_i} pat_i \mathrel{?\triangleright} \Gamma_i; \pi_i; \phi_i\right)}{\mu X.\{\cdots \mid c : q; B_1, \ldots, B_k \mid \cdots\} \vdash^{t_0 + \texttt{Tcopy} + \texttt{Tunpack} + \text{PatSize}(c)}_{t_k - \texttt{Tpop}} c\ pat_1 \cdots pat_k \mathrel{?\triangleright} \Gamma_1, \cdots, \Gamma_k; q + \Sigma_i \pi_i; \bigcup_i \phi_i} \quad \text{(Pattern Constr)}$$

Substructural Rules

$$\frac{A \vdash^{r}_{r'} pat \mathrel{?\triangleright} \Gamma; \pi; \phi}{A \vdash^{t}_{t'} pat \mathrel{?\triangleright} \Gamma; \pi; \phi \cup \{t \geq r, t - r \geq t' - r'\}} \quad \text{(Pattern Relax Time)}$$

For WCET the rules for pattern matches require the auxiliary definition of $\text{PatSize} : \mathsf{Con} \to \mathbb{Q}^+$, which maps a constructors c to the number of time-units required to match any value to this constructor, which is in general equal to `Tmatchcon`. However, in the case of Tuples or Vectors, the WCET for a pattern match is significantly lower. This auxiliary definition is not required for the space analyses.

3.2 Annotated Type Rules for Boxes and Declarations

Given a set of identifiers Id of a certain HUME program, this program is well-typed if and only if

a) for all functions $fid \in \mathsf{Id}$ with

$$\Sigma(fid) = \left(e_f; [y_1, \ldots, y_k]; A_1, \ldots, A_k \xrightarrow[t']{t} C\right)$$

there exists a finite type derivation such that $y_1{:}A_1, \ldots, y_k{:}A_k \vdash^{t}_{t'} e_f : C$ holds.

b) For all boxes $box \in \mathsf{Id}$ with

$$\Sigma(box) = e_b; y; A \xrightarrow[t']{t} C; \text{fairness}; B_x; e_x \mid \phi$$

there exists a finite type derivation such that $y{:}A \vdash^{t_x}_{t'} e_b : C \mid \phi$ and $err{:}Err \vdash^{t}_{t_x} e_x : C \mid \psi$

c) For all pairs of boxes sharing a wire, type assigned to each wire is identical, including the resource variable denoting the potential communicated through the wire.

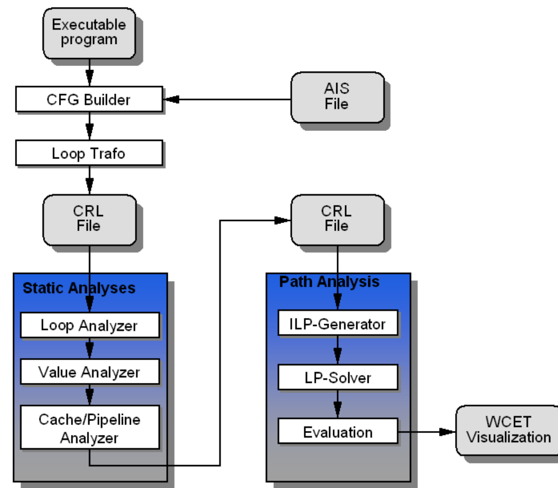

Figure 1: Phases of WCET computation

3.3 Timing Complete Hume Programs

The model we have provided is capable of determining all time costs within a HUME box. Clearly, it is not sensible, in general, to time a complete Hume program, since Hume programs are not usually expected to terminate. However, it is sensible to determine reactivity times to specific inputs. By using the models derived here, we are able to determine this for individual HUME boxes, since we can determine upper bound times on all costs from matching inputs on wires to producing wire outputs. In this way, we are, in principle, able to verify timing constraints provided by HUME programmers within a single box, for programs that include data structures and recursion.

Extending these costs to cover sequences of multiple boxes involves obtaining measurements of inter-box scheduling time, and then combining these using models of multiple box behaviours. Since boxes are combined using a static process network, and the range of inputs and outputs accepted by/produced by each rule can also be determined statically, it follows that we would then be able to construct a complete model of input-output WCET for any input-output combination. We intend to study this extension in the longer term.

4 Concrete Time Costs for the Renesas M32C/85 Processor

In the WCET analysis described in the previous sections, we have used a set of manifest constants rather than actual costs. This allows our static amortised WCET analysis to be generalised to different types of processor. In this section, we describe how we have obtained preliminary costs for the Renesas M32C/85 microprocessor using the **aiT** tool, and verified these against concrete measurements. Further details can be found in a forthcoming paper [BFHH07].

4.1 Determining WCET using the aiT tool

The **aiT** tool determines the worst-case execution time of a program task in several phases, as shown in Figure 1. These phases are:

- **CFG Building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from an executable binary program;
- **Value Analysis** computes address ranges for instructions accessing memory;

- **Cache Analysis** classifies memory references as cache misses or hits [Fer97];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [LTH02];
- **Path Analysis** determines a worst-case execution path of the program [TF98].

The cache analysis phase uses the results of the value analysis phase to predict the behavior of the (data) cache based on the range of values that can occur in the program. The results of the cache analysis are then used within the pipeline analysis to allow prediction of those pipeline stalls that may be due to cache misses. The combined results of the cache and pipeline analyses are used to compute the execution times of specific program paths. By separating the WCET determination into several phases, it becomes possible to use different analysis methods that are tailored to the specific subtasks. Value analysis, cache analysis, and pipeline analysis are all implemented using abstract interpretation [CC77], a semantics-based method for static program analysis. Integer linear programming is then used for the final path analysis phase.

Whilst the analysis works at a level that is more abstract than simple basic blocks, it is not capable of managing the complex high-level constructs that we require. It can, however, provide useful and accurate worst-case time information about lower level constructs. We are thus motivated to link the two levels of analysis, combining information on recursion bounds and other high-level constructs that we will obtain from the Hume source analysis we are constructing, with the low-level worst-case execution time analysis that can be obtained from the AbsInt analysis. In order to achieve this, we will eventually require two-way information flow between the analyses. In the short-term, it is sufficient to provide one-way flow from the language-level analysis to the lower-level analysis. The use of an abstract machine as the analysis target represents a new challenge for the **aiT** tool, since the structure of instructions that need to be analysed can be significantly different from those that are hand-produced, and the associated technical problems in producing cost information can therefore be more complex.

Instruction	gcc	IAR	Ratio
Call	73	70	1.04
Copy	43		
CopyArg	40	35	1.14
CreateFrame	76	72	1.06
Goto	5	3	1.67
If (true)	41	32	1.28
If (false)	41	32	1.28
MakeVar	43	36	1.19
MatchExn	808		
MatchedRule	11	11	1.00
MatchInt	811	137	5.92
MatchRule	22	22	1.00
MatchVar	46	36	1.28
MkBool	136		

Instruction	gcc	IAR	Ratio
MkChar	136		
MkCon 2	348	242	1.44
MkFun 0	198	165	1.20
MkInt	136	91	1.49
MkNone	26	21	1.24
MkVector 3	392	205	1.91
Pop	13		
Push	12	11	1.09
PushVar	40	35	1.14
Return	1756		
Schedule	410	602	0.68
Slide	62	53	1.17
SlideVar	94		
TailCall	91	178	0.51

Figure 2: aiT HAM analysis: gcc and IAR compiled code

Instructions	AVG	WCET	Ratio
Ap	760	761	1.00
Call	61	62	1.02
Callprim			
== Bool	246	251	1.02
* Float	240	242	1.01
+ Float	262	267	1.02
== Float	255	260	1.02
− Int	114	119	1.04
* Int	130	132	1.02
/ Int	168	177	1.05
+ Int	114	119	1.04
< Int	215	220	1.02
== Int	216	221	1.02
> Int	217	223	1.03
Consume	27	31	1.24
Copy	27	31	1.15
CopyArg	27	30	1.11
CreateFrame	51	57	1.12
Goto	1	2	2.00
If (true)	24	29	1.21
If (false)	24	26	1.08
MakeVar	26	31	1.19
MatchAny	6	10	1.67
MatchAvailable	7	10	1.43
MatchBool	24	29	1.21
MatchCon	22	26	1.18
MatchedRule	8	12	1.50
MatchExn	22	28	1.27
MatchFloat	24	29	1.21
MatchInt	23	29	1.26

Instructions	AVG	WCET	Ratio
MatchNone	6	10	1.67
MatchRule	18	23	1.28
MatchString n	$3 \times n + 45$	$3 \times n + 47$	
MatchTuple	6	10	1.67
MatchVar	26	31	1.19
MaybeConsume	20	28	1.40
MkBool	63	70	1.11
MkChar	63	70	1.11
MkCon n	$41 \times n + 84$	$41 \times n + 89$	
MkFun n	$42 \times n + 108$	$42 \times n + 113$	
MkInt	64	65	1.02
MkNone	15	21	1.40
MkString n	$13 \times n + 133$	$13 \times n + 140$	
MkTuple n	$41 \times n + 63$	$41 \times n + 66$	
MkVector n	$41 \times n + 63$	$41 \times n + 65$	
Pop	6	9	1.50
Push	6	9	1.50
PushVar	27	30	1.11
PushVarF	37	40	1.08
Raise	374	377	1.01
Return	112	116	1.04
Slide	41	44	1.07
SlideVar	58	63	1.09
Unpack	114	118	1.04

Figure 3: Experimental average and worst-case timings for HAM instructions

4.2 HAM Instruction WCET Costs for Renesas M32C/85

Figure 2 lists guaranteed worst-case execution time results, in clock cycles, for a subset of Hume Abstract Machine instructions, ordered alphabetically. These timings were obtained using the **aiT** tool from code generated using the **ham2c** Hume to C compiler, cross-compiling through either gcc Version 3.4 or the IAR C compiler [Sys06] to the Renesas M32C. As expected from a commercial compiler targeting a few architectures, the IAR compiler generally produces more efficient code than gcc, with our results being 42% lower on average, and up to 5.92 times more efficient in the case of **MatchInt**. In a few cases, the **aiT** tool was unable to provide timing information directly, requiring additional information such as loop bounds to be provided in order to produce timing results. In the long term, we anticipate that we will be able to provide this information by analysis of Hume source constructs, and by modifying the HAM code to include type information and other information that can be exploited by the **aiT** tool. In the short term, we have calculated the information by hand, as far as possible. For some instructions, however, we were unable to provide this information for the IAR-compiled code, and results for these instructions are therefore given only for gcc-produced code.

Instructions	aiT bound	Measured WCET	Ratio
Call	70	62	1.13
CopyArg	35	30	1.17
CreateFrame	72	57	1.26
Goto	3	2	1.50
If (true)	32	29	1.10
If (false)	32	26	1.23
MakeVar	36	31	1.16
MatchRule	**22**	**23**	**0.96**

Instructions	aiT bound	Measured WCET	Ratio
MatchVar	36	31	1.16
MkCon 2	242	170	1.42
MkFun 0	165	113	1.46
MkInt	91	65	1.40
MkNone	21	21	1.00
Push	11	9	1.22
PushVar	35	30	1.17
Slide	53	44	1.20

Figure 4: Quality of the aiT Analysis

4.3 Timing Results

Figure 3 shows average execution and worst-case execution times obtained using the timing approach described above, for HAM instructions compiled using the IAR compiler. Each average and worst-case entry has been obtained from 10000 individual timings. We can see from the table that the worst-case times and average-case times are very similar for most instructions, indicating that the instruction timings are highly consistent in practice. Since certain instructions are parameterised on some argument (for example, **MkVector** is parameterised on the vector size), in these cases, we have measured several points and applied linear interpolation to obtain a cost formula. It is interesting to note that in these case, the linear factor is identical for both WCET and average times and the constants are also very close. In each case, we have subtracted the least time obtained from timing the empty sequence of instructions (39 clock cycles), in order to give a conservative worst-case time. Since the worst-case time for the empty sequence was 42 cycles, this means that the worst-case may, in fact, be up to three cycles less than the numbers reported here.

Since we must save and restore system state, we needed to develop code that does this correctly. A few abstract machine instructions have therefore not been costed, mainly because they perform more complex state changes that may require additional intervention. It is worth noting that the values included in this table give a good timing predictor, but one that could only be used to provide absolute worst-case guarantees under some statistical probability.

4.4 Quality of the Static Analysis using the aiT Tool

Figure 4 compares the upper bounds on worst-case execution timing obtained using the aiT tool from Figure 2 with the corresponding measured worst cases from Figure 3. We can see that in all cases apart from **MatchRule**, the static analyis gives an upper bound that is greater than or equal to the measured execution time. For **MatchRule**, the static analysis yields an upper bound that is one cycle smaller than our measured worst-case. Since our worst case timings are conservative, and may have an experimental error of up to three clock cycles, as described above, we conclude that the static analysis correctly yields upper bounds on execution costs for these HAM instructions. For the instructions we have compared, the bound given by the static analyis is at most 50% greater than the measured worst-case (for **Goto**, representing a difference of only one clock cycle); the mean difference is 22%, with a standard deviation of 16%. We conclude that the static analysis provides an accurate upper bound on execution time.

5 Simple Analysis Examples

In this section we present several examples of running our analysis on some simple Hume programs. We first consider the costs of simple expressions and functions, then extend this to programs including boxes. We have used identical examples in this deliverable, the report on heap space analysis (Deliverable D11), and on stack space analysis (Deliverable D05). In order to make this section self-contained, we have repeated the description and source code for each example here.

For illustrative purposes, we have chosen to use the *measured* worst-case execution times for the Renesas M32C/85 in this section rather than those obtained using the **aiT** tool. Since we have only obtained analytical **aiT** timings for a relatively small subset of the HAM instructions, but have a near-complete set of measured WCET timings, this approach allows us to cover a wider range of applications, and to demonstrate more features of our analytical framework and implementation. We would emphasise that the figures given here are consequently only worst-case, and not formally guaranteed bounds on execution time. However, we will easily be able to replace these timings with the analytical timings, once they become available.

5.1 Example: factorial function

Our first example is the factorial function, implemented over floating point numbers rather than natural numbers. We chose to use a floating point representation because such values are commonly used in the computer vision domain that is one of our targets, and also because floating-point values tend to be more difficult to handle in analyses than, for example, natural numbers. Additionally to annotating all user-defined datatypes with potentials, as defined in the previous section, we employ a datatype-independent interval analysis, which aims to statically deduce ranges for the possible value of a variable at runtime, and propagate this information to the recursive call in the function body. Thus, we are able to deal with recursion not restricted to special types such as natural numbers or linearly structured datatypes such as lists, and in particular we can handle floats. A current limitation of the interval analysis is its intra-procedural (or -functional) nature. We plan to extend it to cope with more complex recursion patterns in the near future. The Hume code for the obvious factorial function is:

```
program

type _float = float 32;

fac n = if (n==0.0)
          then 1.0
          else n * (fac (n - 1.0));

expression (fac);
```

In the first stage of the analysis this code is translated into an intermediate format:

```
program

-- type of main
val main :: float, ->float
-- Functions
{fac :: float, ->float (n :: float) =
 glet ?z_1    = 0.0
 in glet ?z_2    = n==?z_1
    in if ?z_2
         then 1.0
```

```
         else glet ?z_3    = 1.0
              in glet ?z_4    = n-.?z_3
                 in glet ?z_5    = (fac <> ?z_4)
                    in n*.?z_5}
-- Boxes
-- Expression:
(fac)
```

Desugaring the program into our core syntax produces a new function body for `fac`, where (possibly nested) case expressions are used to match against given patterns (if any). Note that in this format all functions are in let-normal-form, with internally created variables always starting with an `?`.. Furthermore, all overloaded binary operators have been instantiated with their monomorphic counterparts (where `*.` stands for multiplication on floating-point numbers etc). Function calls are specially annotated to indicated whether they have the specified number of arguments or are over- or under-saturated. In this example, we see that we have a call to `fac` with the specified number of arguments (1), which is indicated by the `<>` operator. The worst-case time consumption is given in terms of the following (rich) type of the main function:

```
ARTHUR3 typing for resource "Time":
  30, (float<1043>) -501/0-> float<59> ,0
```

This type indicates that, for an input value of n, the execution of the main function will require $1043n + 501$ machine cycles, plus 30 cycles to set-up the main expression (for the other examples we will ignore this set-up time). An interval analysis, which aims to statically deduce ranges for the possible value of a variable at runtime, is performed on the floating-point variables, to generate linear (in-)equality constraints. These constraints are then solved by a separate LP-solver.

We now analyse a variant of the factorial function, for which our heap inference was able to give tighter bounds.

```
program

type _float = float 32;

fac :: _float -> _float;
fac 0.0 = 1.0 ;
fac n = n * (fac (n - 1.0));

expression (fac);
```

The only difference to the previous version is the use of top-level pattern matching rather than an explicit conditional in the body. When translated to intermediate code this gives:

```
program

-- type of main
val main :: float, ->float
-- Functions
{fac :: float, ->float (?arg_11 :: float) =
 case ?arg_11 of
   (0.0) -> 1.0|
   (n) -> glet ?z_1    = 1.0
          in glet ?z_2    = n-.?z_1
```

```
              in glet ?z_3    = (fac <> ?z_2)
                 in n*.?z_3
 esac}
-- Boxes
-- Expression:
(fac)
```

The key difference for the analysis is that now the variable `n` is used in only one branch, whereas before it was used outside the recursion case, namely in the head of the conditional, as well. Such usage requires a sharing of the associated potential and causes weaker bounds to be inferred.

For this version the rich type of the main expression is:

```
ARTHUR3 typing for resource "Time":
  30, (float<785>) -245/0-> float<116> ,0
```

i.e. for an input n, in total $785n + 245$ cycles are needed. So, we see a considerable improvement of runtime (as well as heap usage) for this version, wheras only slightly more stack space is required. It seems that this version is the clear winner in this head-to-head comparison.

5.2 Example: sum-over-list

The next example infers the costs for a list-traversing function, computing the sum over a list of float values.

```
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;
```

The intermediate code for this example shows how a function with pattern matching is translated into (possibly nested) case statements. The overloaded multiplication operation on Hume-level is instantiated to a monomorphic `*.` over floats.

```
program

type flist = Cons {-2-} float  flist | Nil {-0-}

-- type of main
val main :: flist, ->float

-- Functions
{sum11 :: flist, ->float (?arg_11 :: flist) =
 case ?arg_11 of
   ((Nil)) -> 0.0|
   (Cons f  fs) -> glet ?z_1    = (sum11 <> fs)
                   in f+.?z_1
 esac}
-- Boxes
```

```
-- Expression:
sum11
```

Unsurprisingly the time consumption of the main expression, namely the function `sum11`, is linear in the length of the input list, as shown by the following (rich) type:

```
ARTHUR3 typing for resource "Time":
  30, (flist[934;float<0>,#|0]) -476/126-> float<0> ,0
```

For every `Cons` node of the list, represented as `float<0>,#` in the type, 934 cycles are needed to perform the computation. In total, this gives a worst-case time consumption of $934n + 476$ for an input list of length n.

5.3 Example: multiplication (box- vs expression-level)

One powerful technique to infer costs for complex (recursive) functions is to transform the program by "lifting" functions from the expression-level to the box-level. This has been frequently used in the past to cost Hume programs. The following example implements multiplication over floating-point numbers in terms of repeated addition operations. By lifting the recursion to box level, and encoding the state of the recursion in 3 wires, we obtain an iterative version with 2 boxes. The `mult2` box drives the computation by feeding the two input values that should be multiplied along the output wires `iter1`, `iter2` and `iter3` to the worker box `itermult`. The latter takes input from wires `i1`, `i2` and `i3`, and uses the output wires `iter1'`, `iter2'` and `iter3'` to hold the state of the computation by feeding them back to its own input wires `iter1`, `iter2` and `iter3`.

```
program

type _float = float 64;

stream stdin1 to "std_in";
stream stdin2 to "std_in";
stream stdout to "std_out";

-- takes 2 floats as input and initialises the inputs
-- for the itermult box, which does the main work
box mult2
in (i::_float, j::_float)
out (iter1 ::_float, iter2 ::_float, iter3 ::_float)
match
(x,y) -> (0.0,x,y);

stream output to "std_out";

wire mult2 (stdin1, stdin2)
           (itermult.i1,itermult.i2,itermult.i3);

-- computing (_,x,y,_,_,_,_) -> x*y, using the other last 3 inputs
-- and first 3 outputs as state via feedback wires
box itermult
in  (i1::_float, i2::_float, i3::_float, iter1::_float, iter2::_float, iter3::_float)
out (iter1'::_float, iter2'::_float, iter3'::_float, r::_float)
match
 (r,x,y,*,*,*) -> (r,x,y,*) |
 (*,*,*,r,x,y) -> if y==0.0
```

```
                   then (  *, *,       *, r)
                   else (r+x, x, y - 1.0, *);

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

It is easy to see that the computation in both boxes represents code on FSM Hume level, since no recursion is used. Inferring the worst-case execution time we get the following rich type:

```
ARTHUR3 typing for resource "Time":

Box: mult2
 ?v_8: wire1float[1073;float<0>|*], ?v_9: wire1float[0;float<1659.5>|*]
---621/0--->
 ?v_1: wire1float[0;float<0>|*], ?v_2: wire1float[0;float<0>|*], ?v_3: wire1float[0;float<1659.5>|*]

Box: itermult
 ?v_1: wire1float[0;float<0>|*], ?v_2: wire1float[0;float<0>|*], ?v_3: wire1float[0;float<1659.5>|*],
 ?v_4: wire1float[0;float<0>|*], ?v_5: wire1float[0;float<0>|*], ?v_6: wire1float[796.5;float<1659.5>|*]
---3588.5/0--->
 ?v_4: wire1float[0;float<0>|*], ?v_5: wire1float[0;float<0>|*], ?v_6: wire1float[796.5;float<1659.5>|*],
 ?v_7: wire1float[0;float<0>|*]
```

In this case the time consumption of the `itermult` box is linear in the input wires `?v_3` and `?v_6`. Altogether for input `y` (along wire `?v_3`) of size n and input `y` (along wire `?v_6`) of size m $1659.5n + 1659.5m + 796.5$ cycles are needed by the `itermult` box. This bound is in fact significantly weaker than we had hoped for, and we are currently examining the exact reason.

To demonstrate the usefulness of our resource inference, we now look at a function, directly implementing the computation as recursive program on the Hume expression level.

```
program

stream stdin1 to "std_in";
stream stdin2 to "std_in";
stream stdout1 to "std_out";
stream stdout2 to "std_out";

type _float = float 32;

dec :: _float -> _float;
dec x = x - 1.0;

mult :: _float -> _float -> _float -> _float;
mult r x y = if y==0.0
               then r
               else mult (r+x) x (dec y);


box mult13
in  (i  :: _float, j  :: _float)
out (i' :: _float, o :: _float)
```

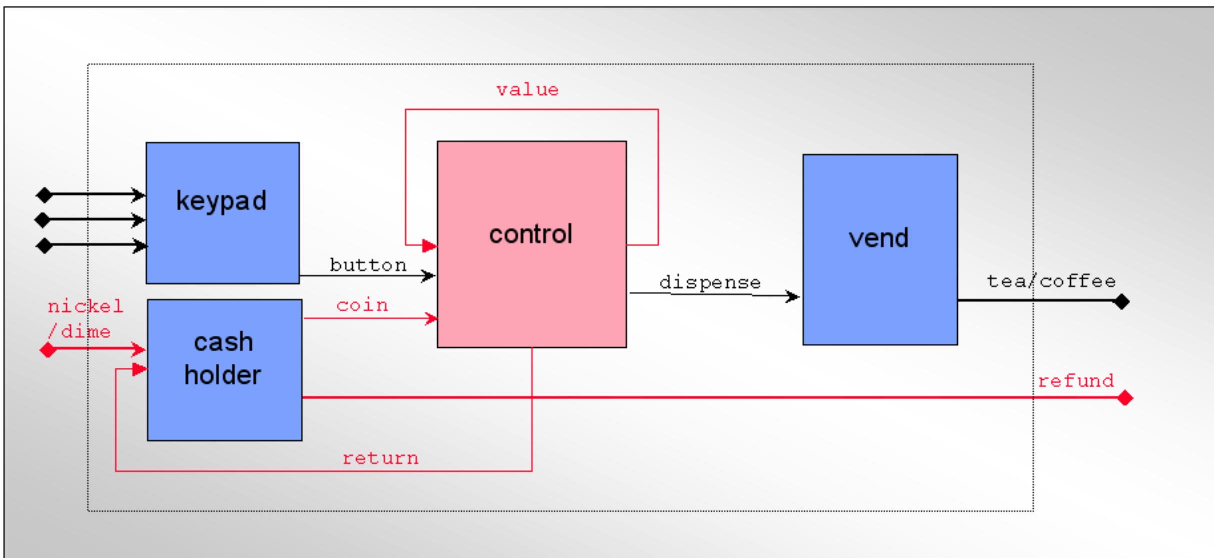

Figure 5: Hume example: vending machine box diagram

```
match
 (x, y) -> (1.0, mult 0.0 x y);

wire mult13 (stdin1, stdin2) (stdout1, stdout2);
```

Now `mult` is a recursive function and the overall program is on PR Hume level. Still, we can derive an upper bound for the box `mult13`, which directly calls this recursive function:

```
Solution has been written to file constraints.solved
ARTHUR3 typing for resource "Time":

Box: mult13
    ?v_1: wire1float[1428;float<0>|*], ?v_3: wire1float[0;float<1068>|*]
   ---621/0--->
    ?v_2: wire1float[0;float<0>|*], ?v_4: wire1float[0;float<0>|*]
```

The costs are linear in the second input and constant in the first input. For an input `j` (along wire `?v_3`) of size n the box needs $1068n + 1468 + 621$ cycles. We see that with this inference, the programmer can write much simpler code to implement the program, and still obtain upper bounds for the resource consumption.

5.4 Example: vending machine

The next example simulates the behaviour of a simple drinks vending machine [Ham05]. A system diagram is shown in Figure 5. We will show Hume code only for the most important 3 boxes of the system. The *control* box (`coffee`) responds to inputs from the keypad box (`inp`) 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. 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 (`outp`) is instructed to produce the requested drink. Otherwise, the button press is ignored.

```
-- Vending machine example, (c) K.Hammond, University of St Andrews
-- after specification by Pieter Koopman, CEFP July 2005

type _float = float 32;

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

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

-- coffee vending box

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

box coffee
in  ( coin :: coins, button :: buttons, value :: _float )
out ( drink :: drinks, value' :: _float, return :: _float )
match
    ( Nickel, *,       v )  ->  ( *, v + 5.0, * )
  | ( Dime,   *,       v )  ->  ( *, v + 10.0, * )
  | ( *,      BCoffee, v )  ->  vend Coffee 10.0 v
  | ( *,      BTea,    v )  ->  vend Tea 5.0 v
  | ( *,      BCancel, v )  ->  ( *, 0.0, v )  ;

showdrink :: drinks->(int 2);
showdrink Coffee = 0;
showdrink Tea = 1;

-- output handling box
box outp
in ( drink :: drinks, return :: _float )
out (d :: int 2, r :: _float)
match
  ( d, * ) -> (showdrink d, *)
| ( *, r ) -> (* , r)
;

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

wire inp    ( stdin )
            ( coffee.coin, coffee.button );
```

```
wire coffee ( inp.coin, inp.button, coffee.value' initially 0.0 )
            ( outp.drink,  coffee.value, outp.return );
wire outp   ( coffee.drink, coffee.return )
            ( stdout, stderr );
```

Below we show only the interesting part of the generated intermediate code:

```
program

type wire1float = W1float {-1-} float| NOVAL1float {-0-}
type wire1int = W1int {-1-} int| NOVAL1int {-0-}
type wire1drinks = W1drinks {-1-} drinks | NOVAL1drinks {-0-}
type wire1buttons = W1buttons {-1-} buttons | NOVAL1buttons {-0-}
type wire1coins = W1coins {-1-} coins | NOVAL1coins {-0-}
type wire1char = W1char {-1-} char| NOVAL1char {-0-}
...
type bundle2coinsbuttons = B2coinsbuttons {-2-} wire1coins  wire1buttons
type bundle3drinksfloatfloat = B3drinksfloatfloat {-3-} wire1drinks  wire1float  wire1float
...
type tuple2coinsbuttons = T2coinsbuttons {-2-} coins   buttons
type tuple3drinksfloatfloat = T3drinksfloatfloat {-3-} drinks   float  float
...

type coins = Nickel {-0-}| Dime {-0-}
type drinks = Coffee {-0-}| Tea {-0-}
type buttons = BCoffee {-0-}| BTea {-0-}| BCancel {-0-}

-- Functions
{vend :: drinks,float,float, ->bundle3drinksfloatfloat (drink :: drinks)  (cost :: float)  (v :: float) =
 glet ?z_21    = v>=cost
 in if ?z_21
      then glet ?z_22    = v-.cost
           in glet ?q_23    = NOVAL1float
              in glet ?w_58    = W1drinks drink
                 in glet ?w_57    = W1float ?z_22
                    in B3drinksfloatfloat ?w_58  ?w_57  ?q_23
      else glet ?q_24    = NOVAL1drinks
           in glet ?q_25    = NOVAL1float
              in glet ?w_56    = W1float v
                 in B3drinksfloatfloat ?q_24  ?w_56  ?q_25;
... }

-- Boxes
val ?arg_00 :: bundle1char
box inp
in  (?v_6 :: wire1char) -- bundle1char
out (?v_2 :: wire1coins)  (?v_1 :: wire1buttons) -- bundle2coinsbuttons
match
 case ?arg_00 of
   (B1char (W1char ('N'))) -> glet ?z_9    = (Nickel)
                              in glet ?q_10    = NOVAL1buttons
                                 in glet ?zz_0    = W1coins ?z_9
                                    in B2coinsbuttons ?zz_0  ?q_10|
   (B1char (W1char ('D'))) -> glet ?z_11    = (Dime)
                              in glet ?q_12    = NOVAL1buttons
                                 in glet ?zz_0    = W1coins ?z_11
                                    in B2coinsbuttons ?zz_0  ?q_12|
   (B1char (W1char ('C'))) -> glet ?q_13    = NOVAL1coins
                              in glet ?z_14    = (BCoffee)
                                 in glet ?zz_1    = W1buttons ?z_14
                                    in B2coinsbuttons ?q_13  ?zz_1|
```

```
   (B1char (W1char ('T'))) -> glet ?q_15    = NOVAL1coins
                              in glet ?z_16    = (BTea)
                                 in glet ?zz_1    = W1buttons ?z_16
                                    in B2coinsbuttons ?q_15  ?zz_1|
   (B1char (W1char ('X'))) -> glet ?q_17    = NOVAL1coins
                              in glet ?z_18    = (BCancel)
                                 in glet ?zz_1    = W1buttons ?z_18
                                    in B2coinsbuttons ?q_17  ?zz_1|
   (B1char ?wild_55) -> glet ?q_19    = NOVAL1coins
                        in glet ?q_20    = NOVAL1buttons
                           in B2coinsbuttons ?q_19  ?q_20
 esac
;
...
```

Note that in the intermediate code we do not have special data structures for tuples, vectors etc. All these data structures are mapped into user-defined data structures, which are automatically inserted at the begin of the program. To deal with values passed along wires between boxes, we automatically generate 2 forms of datatypes in the intermediate code. A unary *wire datatype*, with a constructor of the form `W1<type>` to represent the wire itself, and nullary constructor `NOVAL1<type>` representing the absence of input. A *bundle datatype* collects all input- or output-wires of a box into one datatype. Both forms need to be monomorphic in the current system and are therefore instantiated for all (combinations of) types needed in the particular program. These data-structures are handled specially by the costing module of the inference to avoid over-estimates due to the inserted constructors. With these in-place, we can use the same inference machinery for boxes as is used for functions, in particular we can attach potentials to the constructors of wire data-types. Also note, that in this intermediate code the names of the wires between boxes match up, thus implicitly representing the connections between the boxes. For the above example the inferred costs are:

```
ARTHUR3 typing for resource "Time":

Box: inp
    ?v_6: wire1char[1809;char|*]
   ---2603/0--->
    ?v_2: wire1coins[0;coins[0|0]|*], ?v_1: wire1buttons[1741;buttons[0|402|0]|*]

Box: coffee
    ?v_2: wire1coins[0;coins[0|0]|*], ?v_1: wire1buttons[1741;buttons[0|402|0]|*],
    ?v_5: wire1float[0;float<0>|*]
   ---4790/0--->
    ?v_3: wire1drinks[1104;drinks[0|207]|*], ?v_5: wire1float[0;float<0>|*],
    ?v_4: wire1float[626;float<0>|*]

Box: outp
    ?v_3: wire1drinks[1104;drinks[0|207]|*], ?v_4: wire1float[626;float<0>|*]
   ---1180/0--->
    ?v_7: wire1int[687;int|*], ?v_8: wire1float[0;float<0>|*]
```

In contrast to the situations for heap and stack, the time consumption is not constant over the inputs (but it is independent of the size of the float value, as shown by the 0 potential in `float<0>`). On the wire `?v_1` we see a cost of 402 cycles attached to the `BTea` case, whose processing expands to a call of the function `vend`. For this bigger example it is unclear from which constraints this result is coming. However, it should be pointed out that after testing our prototype analysis on a set of test programs, the next step will be the validation of the data. This will be covered in Deliverable 28 and 30, both due in the third year.

5.5 Example: core of Canny edge detection

As final example we take the computational core of the Canny edge detection algorithm [Can86], implemented by the LASMEA partner group (see also Deliverable 07 [SS06] on real-time testbed applications). The full algorithm detects edges in a 2-dimensional image, by comparing the contrast values of neighbouring points in a window that is traversing the image. The core of this algorithm is a function `mapapply job limit next sr si img` of type
`(img->img) -> (img->img) -> (img->img) -> float -> float -> img -> img`.
Here `job` describes the function that should be applied to each window, `limit` selects the window based on the current position, `next` moves to the next position, `sr` and `si` are boundary values to prevent the window from overruning the end of the image, and `img` is the image to process. This function is instantiated several times in the Canny code with convolution operations using the Gaussian filters as worker functions. The original Hume code for this function is:

```
mapapply job limit next sr si img =
  if (sr<=si)
  then (job (limit img)):(mapapply job limit next sr (si-1) (next img))
  else [];
```

Several versions have been implemented, based on lists and vectors. As test case for the analysis we chose a list-based version. We instantiate the function parameters with concrete functions to get a first order version of the code (it should be noted, however, that the analysis itself can deal with higher-order functions, but due to limitations of the translation into intermediate code, this hasn't been used so far). Also, to simplify the program we use a list of floats rather than a list of lists of floats, i.e. we consider only a 1-dimensional image. The program we are analysing is this:

```
type _float = float 32;
data flist = NNil | CCons _float flist;

hd_flist :: flist -> _float;
hd_flist l =
   case l of
     (CCons h t) -> h;

tl_flist :: flist -> flist;
tl_flist l =
   case l of
     (CCons h t) -> t;

inc :: _float -> _float;
inc x = x + 1.0;

max3 :: flist -> _float;
max3 xs = case xs of (CCons x1 xs1) ->
          case xs1 of (CCons x2 xs2) ->
          case xs2 of (CCons x3 xs3) ->
           if (x1<x2)
            then if (x2<x3) then x3 else x2
            else if (x1<x3) then x3 else x1;


take3 :: flist -> flist;
take3 xs = case xs of (CCons x1 xs1) ->
           case xs1 of (CCons x2 xs2) ->
```

```
          case xs2 of (CCons x3 xs3) ->
          (CCons x1 (CCons x2 (CCons x3 NNil)));

next :: flist -> flist;
next xs = tl_flist xs;

mapit :: _float -> _float -> flist -> flist;
mapit sr si l =
  if (sr<=si)
   then CCons (max3 (take3 l)) (mapit sr (si - 1.0) (next l))
   else NNil;

expression (mapit);
```

Note that this program has a significantly more complex computational structure compared to the vending machine in the previous section. By performing only one pass over the data structure we still can hope to derive an upper bound using our approach of linear programming for solving the generated constraints. Indeed, when we analyse the code we get the following upper bound:

```
ARTHUR3 typing for resource "Time":
  30, (float<0>,float<567>,flist[0|4605;float<0>,#]) -4236/0-> flist[3891|0;float<0>,#] ,0
```

In total the `mapit` function takes $567m + 4605n + 4236$ cycles for an input list of length n and a float value `si` of size m. Again, the inference can capture linear bounds in several input arguments.

6 Summary

This report presents the formal definition of a fully automatic static WCET analysis for HUME, built on a generic framework that is also used for the analyses of bounded stack and heap costs (Deliverables D5 and D11). The prototype implementation of the WCET analysis is described in Deliverable D15 (WP3). The report has been revised to incorporate experience arising from this implementation. Section 5 describes the results we have obtained by analysing a series of simple HUME examples. As early results, these are extremely promising, and we will soon examine the more complex testbed applications that have been produced as part of Deliverable D7 (WP8).

The formalisation and implementation of this analysis have also exposed some ideas for ways to improve our analysis, which we would like to explore further in order to judge their impact on both the accuracy of the results obtained and the range of programs that can be analysed. Two of these ideas, namely potential-carrying numeric types and linearity parameters for function closures have already been partly incorporated as experimental features in the prototype implementation. However, further investigation is needed to determine whether these features should be incorporated as standard in future versions of the analysis.

References

[BFHH07] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. Worst-Case Execution Times for a Purely Functional Language. In *Proc. 2006Intl. Symp. on Impl. and Appl. of Functional Langs. (IFL 2006), to appear*. Springer-Verlag, 2007.

[Can86] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8:679–698, 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[Fer97] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems, Saarland University, Saarbrücken, Germany.* PhD thesis, 1997.

[Ham05] Kevin Hammond. Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach. In *Central European Functional Programming School (CEFP05)*, number 4164 in LNCS, pages 100–134, Budapest, Hungary, July 4–15,, 2005. Springer-Verlag.

[LTH02] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proc. 9th International Static Analysis Symposium SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.

[SS06] Jocelyn Serot and Norman Scaife. Real-time testbed applications. Deliverable 07 of project EmBounded (IST-510255), September 2006. Laboratoire LASMEA, Blaise Pascal University.

[Sys06] IAR Systems. http://www.iar.com/. *Home Page*, 2006.

[TF98] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. RTSS '98: IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.