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

Specific Targeted Research Project (STReP)
FET Open

D05 (WP4a): Report on Stack-space analysis (Revised)

Due date of deliverable: 30th April 2006
Actual submission date: 20th February 2007

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

Lead contractor: St Andrews University Revision: 1.7

Purpose: To construct an effective stack-space usage analysis based on a formal semantics for Hume incorporating space 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 Stack-space consumption 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 stack usage information.

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 Stack-space analysis (Revised)

Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond

Abstract

We describe a formally-based static analysis for determining Stack-space usage for HUME, building on the Stack-space 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 Stack-space consumption 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 (Deliverable D14), Stack-space usage (this deliverable) and Heap space usage (Deliverable D11). This type system is enhanced with an amortised cost analysis for Stack-space usage, in accordance to the Stack-space 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* Stack-space cost.

b) Abstraction of the *potential* Stack-space costs 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 Stack-space consumption of 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 Stack-space 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 Stack-space 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 Costing Stack-space **4**
3.1 Annotated Type Rules for Pattern Matches 8
3.2 Annotated Type Rules for Boxes and Declarations 9

4 Simple Analysis Examples **9**
4.1 Example 1: factorial function 9
4.2 Example: sum-over-list 11
4.3 Example 3: multiplication (box- vs expression-level) 12
4.4 Example 4: drinks vending machine 14
4.5 Example 5: core of Canny edge detection 18

5 Summary **20**

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, $f \backslash x$ denotes the map which is undefined for x and acts like f otherwise. The restriction of f on $\mathcal{X}$ is written $f \restriction \mathcal{X}$, 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. This common foundation will be exploited throughout this report, and we will assume familiarity with the notations and explanations contained therein.

3 Costing Stack-space

The basic definitions of annotated types given in Section 3 of Deliverable D11 also hold for stack space usage. 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

$$\Sigma; \Gamma \vdash^{p}_{p'} e : A \mid \psi$$

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 $\boldsymbol{v}(m) + \Phi_\eta\big(\mathcal{V} : \boldsymbol{v}(\Gamma)\big)$ stack units. Furthermore, if the computation finishes with heap η' and result value v, then at least $\boldsymbol{v}(m') + \Phi_{\eta'}\big(v : \boldsymbol{v}(\Gamma)\big)$ stack units remain unused once the computation has finished. This can be formulated in the following theorem:

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

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

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

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

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

If all four statements above are 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 $\eta, \mathcal{V} \vdash^{p}_{p'} \vdash^{m}_{m'} e \rightsquigarrow_\diamond v, \eta'$ for some $p, p' \in \mathbb{Q}^+$.

The statement $\eta, \mathcal{V} \vdash^{p}_{p'} \vdash^{m}_{m'} e \rightsquigarrow_\diamond v, \eta'$ was defined in the formal HUME operational semantics (Deliverable D12). It specifies that e evaluates successfully in the given memory configuration with at least m heap-units and p stack-units available and that at least m' heap-units and p' stack-units remain unused after the evaluation is finished. It therefore measures the cost of an evaluation, whereas the annotated type rules which we will now define *bound* the Stack-space usage for evaluating an expression for *all* well-typed inputs. We now define the annotated type rules for HUME expressions for *bounding* Stack-space consumption:

$$\frac{}{\varnothing \vdash^{1}_{0} \ \texttt{()} : \mathsf{unit} \mid \emptyset} \tag{Unit}$$

$$\frac{b \in \mathbb{B}}{\varnothing \vdash^{1}_{0} b : \mathsf{bool} \mid \emptyset} \quad (\textsc{Bool})$$

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash^{1}_{0} n : \mathsf{int} \mid \emptyset} \quad (\textsc{Int})$$

$$\frac{r \in \mathbb{R}}{\varnothing \vdash^{1}_{0} r : \mathsf{float} \mid \emptyset} \quad (\textsc{Float})$$

$$\frac{c \texttt{ is a character}}{\varnothing \vdash^{1}_{0} c : \mathsf{char} \mid \emptyset} \quad (\textsc{Char})$$

$$\frac{s \texttt{ is a string}}{\varnothing \vdash^{1}_{0} s : \mathsf{string} \mid \emptyset} \quad (\textsc{String})$$

$$\frac{}{x{:}A \vdash^{1}_{0} x : A \mid \emptyset} \quad (\textsc{Var})$$

$$\frac{\mathrm{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash^{0}_{1} x \ \mathrm{op}\ y : \mathsf{int} \mid \emptyset} \quad (\textsc{PrimBOp Int})$$

$$\frac{\mathrm{op} \in \{\texttt{-}\}}{x{:}\mathsf{int} \vdash^{0}_{0} \ \mathrm{op}\ y : \mathsf{int} \mid \emptyset} \quad (\textsc{PrimUOp Int})$$

$$\frac{\mathrm{op} \in \{\texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}\}}{x{:}\mathsf{float}, y{:}\mathsf{float} \vdash^{0}_{1} x \ \mathrm{op}\ y : \mathsf{float} \mid \emptyset} \quad (\textsc{PrimBOp Float})$$

$$\frac{\mathrm{op} \in \{\texttt{-.}\}}{x{:}\mathsf{float} \vdash^{0}_{0} \ \mathrm{op}\ y : \mathsf{float} \mid \emptyset} \quad (\textsc{PrimUOp Float})$$

$$\frac{\mathrm{op} \in \{\texttt{==}, \texttt{>=}, \texttt{<=}\}}{x{:}A, y{:}A \vdash^{0}_{1} x \ \mathrm{op}\ y : \mathsf{bool} \mid \emptyset} \quad (\textsc{PrimBOp Eq})$$

$$\frac{\mathrm{op} \in \{\texttt{and}, \texttt{or}\}}{x{:}\mathsf{bool}, y{:}\mathsf{bool} \vdash^{0}_{1} x \ \mathrm{op}\ y : \mathsf{bool} \mid \emptyset} \quad (\textsc{PrimBOp Bool})$$

$$\frac{\mathrm{op} \in \{\texttt{not}\}}{x{:}\mathsf{bool} \vdash^{0}_{0} \ \mathrm{op}\ y : \mathsf{bool} \mid \emptyset} \quad (\textsc{PrimUOp Bool})$$

$$\frac{\Sigma(\mathit{fid}) = \big(_; _; \forall \alpha \in \psi.A_1, \ldots, A_a \xrightarrow[p']{p} C\big) \qquad k \geq 0 \qquad k = a}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash^{p+4}_{p'+4+k} \mathit{fid}\ y_1 \cdots y_k : C \mid \psi} \quad (\textsc{App})$$

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \big(_; _; \forall \alpha \in \psi.A_1, \ldots, A_a \xrightarrow[p']{p} C\big) \qquad k \geq 1 \qquad k < a \\ \forall i \leq k \,.\, \big(\phi_i = \curlyvee(A_i \,|A_i, A_i)\big) \qquad \beta = \alpha \setminus \mathrm{FV}\big(A_1, \ldots, A_k\big)\end{array}}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash^{0}_{k-1} \mathit{fid}\ y_1 \cdots y_k : \forall \beta \in \psi.A_{k+1}, \ldots, A_a \xrightarrow[p'+k]{p+k} C \mid \bigcup_i \phi_i} \quad (\textsc{Under App})$$

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \big(_;_;\forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[p']{p} C\big) \qquad a\geq 1 \qquad k>a \\ y_1{:}A_1,\ldots,y_a{:}A_a \vdash^{p}_{p'} \mathit{fid}\ y_1\cdots y_a : C \mid \phi \qquad x \text{ is fresh} \\ x{:}C, y_{a+1}{:}A_{a+1},\ldots,y_k{:}A_k \vdash^{p'}_{p''} x\ y_{a+1}\cdots y_k : E \mid \chi\end{array}}{y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{p}_{p''} \mathit{fid}\ y_1\cdots y_k : E \mid \phi\cup\psi\cup\chi} \quad (\textsc{Over App})$$

$$\frac{D = \forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[p']{p} C \qquad k\geq 1 \qquad k=a}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{p+4}_{p'+4+k} z\ y_1\cdots y_k : C \mid \psi} \quad (\textsc{App Var})$$

$$\frac{\begin{array}{c}D = \forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[p']{p} C \qquad k\geq 1 \qquad k<a \\ \forall i\leq k\,.\,\big(\phi_i = \curlyvee(A_i\,|A_i,A_i)\big) \qquad \beta = \alpha\setminus \mathrm{FV}\big(A_1,\ldots,A_k\big) \qquad B = \forall\beta\in\psi.A_{k+1},\ldots,A_a \xrightarrow[p'+k]{p+k} C\end{array}}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{0}_{k-1} z\ y_1\cdots y_k : B \mid \bigcup_i \phi_i} \quad (\textsc{Under App Var})$$

$$\frac{\begin{array}{c}D = \forall\alpha\in\psi.A_1,\ldots,A_a \xrightarrow[p']{p} C \qquad a\geq 1 \qquad k>a \\ y_1{:}A_1,\ldots,y_a{:}A_a \vdash^{p}_{p'} z\ y_1\cdots y_a : C \mid \phi \qquad x \text{ is fresh} \\ x{:}C, y_{a+1}{:}A_{a+1},\ldots,y_k{:}A_k \vdash^{p'}_{p''} x\ y_{a+1}\cdots y_k : E \mid \chi\end{array}}{z{:}D, y_1{:}A_1,\ldots,y_k{:}A_k \vdash^{p}_{p''} z\ y_1\cdots y_k : E \mid \phi\cup\psi\cup\chi} \quad (\textsc{Over App Var})$$

$$\frac{c\in\mathsf{Con} \qquad C = \mu X.\{\cdots \mid c : q \mid \cdots\}}{\varnothing \vdash^{1+q}_{0} c : C \mid \emptyset} \quad (\textsc{Constr Zero})$$

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

$$\frac{\Gamma \vdash^{p+1}_{p'} e_t : A \mid \phi \qquad \Gamma \vdash^{p+1}_{p'} e_f : A \mid \psi}{\Gamma, x{:}\mathsf{bool} \vdash^{p}_{p'} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A \mid \phi\cup\psi} \quad (\textsc{Conditional})$$

$$\frac{\mathrm{ck}\in\{\texttt{exprcase},\texttt{funcase}\} \qquad q = \mathrm{maxVars}\,(\mathit{pat}_1,\ldots,\mathit{pat}_k) \qquad \forall i\,.\begin{cases} A \vdash^{p_{(i-1)}}_{p_i} \mathit{pat}_i \ ?\!\triangleright\ \Delta_i;\pi_i;\psi_i \\ \Gamma,\Delta_i \vdash^{p'_i}_{p''} e_i : B \mid \phi_i \\ \chi_i = \{\pi_i + p_i \geq p'_i\}\end{cases}}{\Gamma, x{:}A \vdash^{p_0+4+q}_{p''+5+q} \texttt{case}\ \mathrm{ck}\ x\ \texttt{of}\ \mathit{pat}_1\ \texttt{->}\ e_1 \mid\cdots\mid \mathit{pat}_k\ \texttt{->}\ e_k : B \mid \bigcup_i \psi_i\cup\phi_i\cup\chi_i} \quad (\textsc{Case Expr/Fun})$$

$$\frac{\begin{array}{c}\mathrm{ck} = \texttt{boxcase} \\ q = \mathrm{maxVars}\,(\mathit{pat}_1,\ldots,\mathit{pat}_k) \qquad \forall i\,.\begin{cases} A \vdash^{p_{(i-1)}}_{p_i} \mathit{pat}_i \ ?\!\triangleright\ \Delta_i;\pi_i;\psi_i \\ \Gamma,\Delta_i \vdash^{p'_i}_{p''} e_i : B \mid \phi_i \\ \chi_i = \{\pi_i + p_i \geq p'_i + p_0\}\end{cases} \\ \chi_0 = \{p\geq p_0+4+q\}\end{array}}{\Gamma, x{:}A \vdash^{p_0}_{p''+5+q} \texttt{case}\ \mathrm{ck}\ x\ \texttt{of}\ \mathit{pat}_1\ \texttt{->}\ e_1 \mid\cdots\mid \mathit{pat}_k\ \texttt{->}\ e_k : B \mid \bigcup_i \psi_i\cup\phi_i\cup\chi_i} \quad (\textsc{Case Box})$$

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 p 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.

$$\frac{\forall i\,.\begin{cases}\Delta_i = \{x_1{:}A^i_1, \ldots, x_{(i-1)}{:}A^i_{(i-1)}\}\restriction \mathrm{FV}(e_i)\\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\})\\ \psi_i = \curlyvee(A_i \,|\mathcal{A}_i)\\ \Delta_i, \Gamma_i \vdash^{p_{(i-1)}}_{p_i} e_i : A_i \mid \phi_i\end{cases}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash^{p_0}_{p_{k+1}} \texttt{LET}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{IN}\ e : A \mid \bigcup_i \phi_1 \cup \psi_i} \quad \text{(Ghost let)}$$

$$\frac{\forall i\,.\begin{cases}\Delta_i = \{x_1{:}A^i_1, \ldots, x_{(i-1)}{:}A^i_{(i-1)}\}\restriction \mathrm{FV}(e_i)\\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\})\\ \psi_i = \curlyvee(A_i \,|\mathcal{A}_i)\\ \Delta_i, \Gamma_i \vdash^{p_{(i-1)}}_{p_i} e_i : A_i \mid \phi_i\end{cases}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash^{p_0+4+k}_{p_{(k+1)}+4+k} \texttt{let}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{in}\ e_{k+1} : A_{k+1} \mid \bigcup_i \phi_i \cup \psi_i} \quad \text{(Let)}$$

Exceptions

$$\frac{\Gamma \vdash^{p}_{p'+1} e : \mathit{Err} \mid \psi}{\Gamma \vdash^{p}_{p'} \texttt{raise}\ \mathit{exn}\ e : A \mid \psi} \quad \text{(Raise)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{p}_{p'} e : A \mid \phi\\ \Delta \vdash^{p'}_{p''} \texttt{raise}\ \mathit{exn}\ e_x : A \mid \psi\end{array}}{\Gamma, \Delta \vdash^{p}_{p''} e\ \texttt{within}\ q\ \texttt{time raise}\ \mathit{exn}\ e_x : A \mid \phi \cup \psi} \quad \text{(Within Time)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{p}_{p'} e : A \mid \phi\\ \Delta \vdash^{p'}_{p''} \texttt{raise}\ \mathit{exn}\ e_x : A \mid \psi\end{array}}{\Gamma, \Delta \vdash^{p}_{p''} e\ \texttt{within}\ q\ \texttt{stack raise}\ \mathit{exn}\ e_x : A \mid \phi \cup \psi} \quad \text{(Within Stack)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{p}_{p'} e : A \mid \phi\\ \Delta \vdash^{p'}_{p''} \texttt{raise}\ \mathit{exn}\ e_x : A \mid \psi\end{array}}{\Gamma, \Delta \vdash^{p}_{p''} e\ \texttt{within}\ q\ \texttt{heap raise}\ \mathit{exn}\ e_x : A \mid \phi \cup \psi} \quad \text{(Within Heap)}$$

Substructural rules

$$\frac{\Gamma, x{:}B \vdash^{p}_{p'} e : C \mid \psi}{\Gamma, x{:}A \vdash^{p}_{p'} e : C \mid \psi \cup A <: B} \quad \text{(Supertype)}$$

$$\frac{\Gamma \vdash^{p}_{p'} e : C \mid \psi}{\Gamma \vdash^{p}_{p'} e : D \mid \psi \cup C <: D} \quad \text{(Subtype)}$$

$$\frac{\Gamma \vdash^{q}_{q'} e : A \mid \psi}{\Gamma \vdash^{p}_{p'} e : A \mid \psi \cup \{p \geq q, p - q \geq p' - q'\}} \quad \text{(Relax Stack)}$$

$$\frac{\Gamma \vdash^{p}_{p'} e : C \mid \psi}{\Gamma, x{:}A \vdash^{p}_{p'} e : C \mid \psi} \quad \text{(Weak)}$$

$$\frac{\Gamma, x{:}A_1, y{:}A_2 \vdash^{p}_{p'} e : C \mid \phi}{\Gamma, z{:}A \vdash^{p}_{p'} e[z/x, z/y] : C \mid \phi \cup \curlyvee(A \,|\, A_1, A_2)} \quad \text{(Share)}$$

The definitions for sharing $\curlyvee(A \,|\, A_1, A_2, \ldots)$ and subtyping $A <: B$ remain unaltered as defined in Deliverable D11, since they only deal with abstract potential through manipulating resource variables in a general, resource-independent manner The auxiliary definition of $\textsc{Size} : \mathsf{Con} \to \mathbb{Q}^+$ is not required for bounding Stack-space.

3.1 Annotated Type Rules for Pattern Matches

The annotated type rules dealing with pattern matches have the form

$$A \vdash^{p}_{p'} pat \;?\!\triangleright\; \Gamma; \pi; \psi$$

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.

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

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

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

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

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

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

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

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

$$\frac{\forall i \,.\, \left(B_i \vdash^{p_{i-1}}_{p_i} pat_i \;?\!\triangleright\; \Gamma_i; \pi_i; \phi_i\right)}{\mu X.\{\cdots \mid c : q; B_1, \ldots, B_k \mid \cdots\} \vdash^{p_0+4+k}_{p_k+4+k} c\; pat_1 \cdots pat_k \;?\!\triangleright\; \Gamma_1, \cdots, \Gamma_k; q + \Sigma_i \pi_i; \bigcup_i \phi_i} \quad \text{(Pattern Constr)}$$

Substructural Rules

$$\frac{A \vdash^{q}_{q'} \ pat \ ?\triangleright \ \Gamma; \pi; \phi}{A \vdash^{p}_{p'} \ pat \ ?\triangleright \ \Gamma; \pi; \phi \cup \{p \geq q, p - q \geq p' - q'\}} \ (\textsc{Pattern Relax Stack})$$

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 $\mathit{fid} \in \mathsf{Id}$ with

$$\Sigma(\mathit{fid}) = \big(e_f; [y_1, \ldots, y_k]; A_1, \ldots, A_k \xrightarrow[p']{p} C\big)$$

there exists a finite type derivation such that $y_1{:}A_1, \ldots, y_k{:}A_k \vdash^{p}_{p'} e_f : C$ holds.

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

$$\Sigma(\mathit{box}) = e_b; y; A \xrightarrow[p']{p} C; \text{fairness}; B_x; e_x \mid \phi$$

there exists a finite type derivation such that $y{:}A \vdash^{p_x}_{p'} e_b : C \mid \phi$ and $\mathit{err}{:}\mathit{Err} \vdash^{p}_{p_x} e_x : C \mid \psi$

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.

4 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 time analysis (Deliverable D14). In order to make this section self-contained, we have repeated the description and source code for each example here.

4.1 Example 1: 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. In order to allow this in a natural manner, we have extended our rules to handle rational costs for floating-point values in addition to the fixed integer costs for general values described above. This has been incorporated into our prototype implementation as an extension to the potential that is recorded for floating-point values. The potential of a floating-point value is defined to be the value of the annotation multiplied by the absolute value of the float. We employ a datatype-independent interval analysis, building on [Vas07], whose purpose is to statically deduce ranges for the possible value of a variable at runtime, and to propagate this information to the recursive call in the function body. Thus, we are able to deal with forms of recursion that are not restricted to special types such as natural numbers or linearly structured datatypes such as lists. In particular we can handle the floats required here, knowing enough about the sign of certain floats to allow their type to safely carry potential. 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 a `?`. 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 stack consumption is given in terms of the following (rich) type of the main function:

```
ARTHUR3 typing for resource "Stack":
  1, (float<3>) -1/0-> float<1> ,0
```

This type indicates that, for an input value of n, the execution of the main function will require $3n + 1$ stack units, plus 1 stack unit to set-up the main expression (as is the case for all programs here). 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.

Let us now examine the costs of this program in more detail. The 3 stack units required in each recursive call correspond to the stack configuration immediately prior to executing the recursive call: a pointer to the function, the argument `?z_4` and the original value `n` is needed, to perform the final multiplication.

We now analyse a variant of the factorial function, for which our heap inference in Deliverable D11 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 in the source program compared with 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.

The rich type computed by the inference reflects the solution above:

```
ARTHUR3 typing for resource "Stack":
  1, (float<4>) -1/0-> float<0> ,0
```

i.e. for an input n, in total $4n + 1$ stack units are needed, one more, per iteration, than in the first version. We can see that this version of the code trades one unit of stack, for four units of heap. In general, such generic analyses, that examine not only one resource, but several related resources (such as heap and stack) enable the programmer to make such tradeoffs, depending on the underlying hardware characteristcs.

4.2 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
```

The stack 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 "Stack":
  1, (flist[4;float<0>,#|0]) -2/1-> float<0> ,0
```

For every `Cons` node of the list, represented as `float<0>,#` in the type, 4 stack units are needed to perform the computation. Again the highest stack size is met just before the recursive call, where `f`, `?z_1`, `sum11` and `fs` are still needed. A constant of 2 units is needed for the match (which potentially has to unpack the 2 components of `Cons` onto the stack. So, the stack consumption is $4n + 2$ for an input list of length n.

4.3 Example 3: multiplication (box- vs expression-level)

One powerful technique that can be used 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 is FSM-Hume, since no recursion is used. Inferring the upper bounds on heap consumption, gives the following rich type

```
ARTHUR3 typing for resource "Stack":
Box: mult2
 ?v_8: wire1float[0;float<0>|*], ?v_9: wire1float[0;float<0>|*]
---3/2--->
 ?v_1: wire1float[0;float<0>|*], ?v_2: wire1float[0;float<0>|*], ?v_3: wire1float[2;float<0>|*]

Box: itermult
 ?v_1: wire1float[0;float<0>|*], ?v_2: wire1float[0;float<0>|*], ?v_3: wire1float[2;float<0>|*],
 ?v_4: wire1float[8;float<0>|*], ?v_5: wire1float[0;float<0>|*], ?v_6: wire1float[0;float<0>|*]
---7/3--->
 ?v_4: wire1float[8;float<0>|*], ?v_5: wire1float[0;float<0>|*], ?v_6: wire1float[0;float<0>|*],
 ?v_7: wire1float[13;float<0>|*]
```

In this case the stack consumption of the `itermult` box is constant and only non-zero for the wires `?v_3` and `?v_4`. Altogether $2+8+7=17$ stack units are needed by the `itermult` box. To demonstrate the usefulness of our resource inference, we now consider a function that directly implements the computation as a recursive program.

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

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

Here, `mult` is a recursive function and the overall program is PR-Hume. We are still able to derive an upper bound for the box `mult13`, which directly calls this recursive function:

```
ARTHUR3 typing for resource "Stack":
Box: mult13
    ?v_1: wire1float[6;float<0>|*], ?v_3: wire1float[0;float<1>|*]
   ---3/1--->
    ?v_2: wire1float[10;float<0>|*], ?v_4: wire1float[0;float<0>|*]
```

As expected, the costs are now linear in the second input to the box, `j`, over which the recursion is performed: for an input of size n the box needs $n + 9$ units of stack space. This example shows that the programmer has considerable flexibility over choice of appropriate abstractions, whilst still being able to obtain upper bounds on resource consumption.

4.4 Example 4: drinks vending machine

The next example simulates the behaviour of a simple drinks vending machine [Ham05]. A system diagram is shown in Figure 1. We will show the Hume code only for the most important three boxes of the system. The *control* box (`coffee`) responds to inputs from the keypad box (`inp`) and the cash holder box records 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.

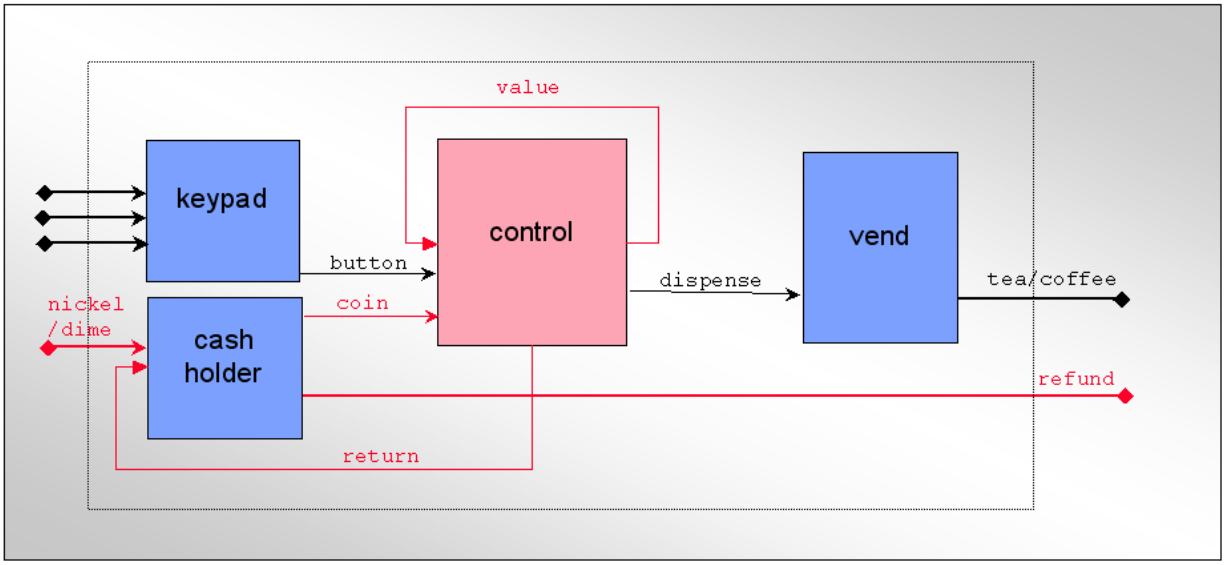

Figure 1: Hume example: vending machine box diagram

```
-- 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 start of the program. To deal with values passed along wires between boxes, we automatically generate two 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 a nullary constructor `NOVAL1<type>` representing the absence of input. A *bundle datatype* collects all input- or output-wires of a box into a single datatype. Both these 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 are then able to use the same inference machinery for boxes that is also used for functions. In particular, we are able attach potentials to the constructors of wire data-types. Note also, 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 "Stack":

Box: inp
```

```
    ?v_6: wire1char[2;char|*]
   ---2/1--->
    ?v_2: wire1coins[0;coins[0|0]|*], ?v_1: wire1buttons[2;buttons[0|0|0]|*]

Box: coffee
    ?v_2: wire1coins[0;coins[0|0]|*], ?v_1: wire1buttons[2;buttons[0|0|0]|*],
    ?v_5: wire1float[0;float<0>|*]
   ---4/2--->
    ?v_3: wire1drinks[2;drinks[0|0]|*], ?v_5: wire1float[0;float<0>|*],
    ?v_4: wire1float[4;float<0>|*]

Box: outp
    ?v_3: wire1drinks[2;drinks[0|0]|*], ?v_4: wire1float[4;float<0>|*]
   ---3/1--->
    ?v_7: wire1int[4;int|*], ?v_8: wire1float[6;float<0>|*]
```

We observe that all boxes can operate in constant heap space, since all potentials attached to the data structures used here have a factor of 0. Of course, in this program this is not surprising, since the bodies of the boxes contain fairly short code segments. However, we have combined here distinct two functions, whose resource consumption needs to be propagated, thereby demonstrating the inter-procedural character of our analysis. Furthermore, in typical embedded system code we would anticipate that there will be many small functions (and boxes) that need to be analysed, and we therefore expect that this example will be typical of our intended application domain.

4.5 Example 5: core of Canny edge detection

As our final example, we will take the computational core of the Canny edge detection algorithm [Can86], implemented by LASMEA as part of deliverable D7.The full algorithm detects edges in a 2-dimensional image, by comparing contrasting 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 overrunning 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 [];
```

For experimental purposes, we have implemented several versions of this algorithm, based on different representations of lists and vectors. As the test case for the analysis we have chosen to use a list-based version. We instantiate the function parameters with concrete functions to obtain 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 analyse is:

```
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 with that of the drinks vending machine example above. By performing only a single pass over the data structure we can still hope to derive an upper bound using our linear programming approach. Indeed, when we analyse the code we get the following upper bound (110 inequalities in 127 variables; solved in 0.17s on a 1.7GHz Pentium laptop, with 1GB main memory and 2MB cache):

```
ARTHUR3 typing for resource "Stack":
  1, (float<0>,float<0>,flist[0|2;float<0>,#]) -4/5-> flist[0|2;float<0>,#] ,0
```

It follows that the maximum stack depth for `mapit` is $2n + 4$ for an input list of length n. We also see that $2m + 5$ stack units become free at the end of the evaluation (where m is the length of the result list). While this is less relevant on the box-level, where the box reset frees all units after an iteration, this information is very valuable on the expression-level, in particular for stack size, which grows and shrinks within one iteration, possibly involving several functions. Such an output value enables the

use of the resources by the succeeding function and therefore makes the inference compositional on the function level.

5 Summary

This report presents the formal definition of a fully automatic static bounded stack usage analysis for HUME, built on a generic framework that is also used for the analyses of bounded heap and time costs (Deliverables D11 and D14). The implementation of the stack and heap space analysis is described in Deliverable D13 (WP4). The report has been revised to incorporate experience arising from this implementation. Section 4 describes the results we have obtained by analysing a series of simple HUME examples. As early results, these are extremely promising. In order to achieve the proper evaluation promised as part of deliverable D30 (WP4), we must now turn our attention to the more complex testbed applications that have been produced as part of Deliverable D7 (WP8). In addition to this practical work, we also intend to establish a formal correctness proof for the theorem stated in the beginning of section 3. Based on our previous experience with such proofs, it is reasonable to assume that this is achievable, but further generalisation may be necessary in order to allow the necessary induction steps to be performed. This will also be reported in Deliverable D30 (WP4).

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

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

[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.

[Vas07] P.B. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, 2007. in preparation.