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

Specific Targeted Research Project (STReP)
FET Open

D11 (WP4b): Report on Heap-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: 1.30

Purpose: to describe the formal techniques used in the HUME Heap-space analysis and to explain how the results of the HUME Heap-space analysis are to be interpreted.

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

Conclusion: The principal goal of this piece of work has been successfully achieved, providing a critical foundation for the success of the project as a whole. Moreover, in undertaking the work, we have identified a number of new ideas for further research that could be performed in order to improve the precision of the bounds we obtain and to increase the range of programs whose costs can be accurately determined.

Project co-funded by the European Commission within the 6^{th} 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 Heap-space analysis (Revised)

Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond

Abstract

We construct a static analysis for determining Heap-space usage for HUME, based on the Heap-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 Heap-space consumption of the typed terms. Our approach is as follows:

a) We first formulate a standard type system for the HUME language. This forms a common basis for constructing all of our HUME analyses, for Worst Case Execution Time (D14), Stack-space usage (D5) and Heap space usage (this deliverable).

b) This type system is then enhanced with an amortised cost analysis for Heap-space usage, in accordance to the Heap-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* Heap-space cost.

c) Abstraction of the *potential* Heap-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 Heap-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 Heap-space analysis for HUME. In this report we will explain the underlying techniques that we have used, describe how the annotated types can be derived in a completely automatic manner, introduce some short program examples, and, finally, explain the meaning of the results achieved by the analysis.

Contents

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 $f \restriction \mathcal{X}$, 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

We now describe a generic foundation for constructing static analyses for determining time and space costs for HUME programs, as formally defined in Deliverable D12. Although, as outlined in the work-plan, we had originally constructed an analysis that was specific to one of our space metrics, which we then intended to adapt to the different problems arising in the other situations, we subsequently determined that all three metrics could be incorporated into the same generic formal rules, with significant advantages for consistency and future reuse. This deliverable has therefore been enhanced to follow this new generic approach. It follows that the observations made in this section of the report are completely generic and not specific to costing Heap allocations. They will be reused directly in deliverables D5 and D14, which describe our analyses for Stack and Worst-Case Execution Time (WCET), respectively.

We have chosen to use a type-based approach in order to produce a compile-time static analysis. We improve upon earlier published work (e.g. [HJ03, HJ06]) by allowing higher-order functions, arbitrary recursive datatypes and resource polymorphism, as well as expanding from a fairly limited toy language to the application-oriented HUME language. Our approach is as follows:

a) We first refine the HUME language to a theoretically more manageable core version (Section 2.1). These changes are all non-essential, in the sense of merely removing syntactic sugar. For example, we replace the special syntax for Tuples and Vectors with that for general datatypes. In all cases, we take care to ensure that the correct cost for the specialised syntactic forms is still preserved. For example, in the case of Tuples and Vectors, we introduce general cost parameters for datatypes that can correctly capture the costs for Tuples, Vectors and user-defined datatypes.

b) We then define a standard type system for this core syntax (Section 2.2). This forms a general basis for our analyses, which we will then augment by the proper Heap-space analysis, allowing a clear separation between the underlying generic basis and the specifics that are required for the Heap-space analysis.

c) We then define an augmented type system (Section 3), which includes an *amortised* Heap-space analysis. The basic principle of an amortised analysis is described in more detail in Section 2.3. The augmented typing introduces a number of *linear* numeric constraints. It follows that the process of deriving such an annotated typing can be simplified by abstracting over all these numeric values, and then using a standard linear program solver to instantiate these values. This allows the Heap-space analysis to be automated. Since the linear programs that we generate do not present a challenge for current state of the art solvers for linear programs, it follows that they can be solved very quickly and efficiently, allowing our Heap-space analysis to be sensibly automated. We have previous shown [HJ03] that the generated linear programs have good formal properties, such as being solvable by integers if they are solvable at all (this is known to be an NP-hard problem in general). Although we do not require such a property for the EmBounded project, this property will still carry over to the work described here.

d) Finally, the augmented typings allow us to compute strict upper bounds on the Heap-space consumption of the typed terms, so completing our Heap-space analysis. We describe the interpretation of the obtained results by exploring a number of program examples in Section 4.

Since the first two steps of our approach are not specific to Heap-space usage, we intend to reuse this work when constructing both the Stack-usage analysis and the Worst-Case Execution Time analysis.

2.1 Core HUME Syntax

Our core syntax for HUME is shown in Figure 1. We assume four disjoint sets of identifiers: Id for names of functions, boxes and wires (which we refer to as *identifiers*), usually ranged over by *id*; Var for

program ::=	$decl_1$ `;` ... `;` $decl_n$	$n \geq 1$
decl ::=	*box* | id *varlist* `=` *expr*	
varlist ::=	$\langle$ var_1 `,` ... `,` var_n $\rangle$	$n \geq 0$
box ::=	`box` id *ins outs boxflag expr* [`handle` *exnexpr*]	
ins/outs ::=	$\langle$ $\text{id}_1, \ldots, \text{id}_n$ $\rangle$	$n \geq 0$
boxflag ::=	`fair` | `unfair`	
caseflag ::=	`boxcase` | `funcase` | `exprcase`	
match ::=	*pat* `->` *expr*	
exnmatch ::=	*pat* `->` *exnexpr*	
expr ::=	unit | bool | int | float | char | string	
	| var *varlist*	
	| id *varlist*	
	| con *varlist*	
	| var *bop* var | *uop* var	
	| `if` var `then` $expr_1$ `else` $expr_2$	
	| `case` *caseflag* var `of` $\langle$ $match_1$ `|` $\cdots$ `|` $match_n$ $\rangle$	$n \geq 1$
	| `let` $\langle$ $vdecl_1$ `,` ... `,` $vdecl_n$ $\rangle$ `in` *expr*	$n \geq 1$
	| `LET` $\langle$ $vdecl_1$ `,` ... `,` $vdecl_n$ $\rangle$ `IN` *expr*	$n \geq 1$
	| *expr* `within` int `time` *raiseexpr*	
	| *expr* `within` int `stack` *raiseexpr*	
	| *expr* `within` int `heap` *raiseexpr*	
	| *raiseexpr*	
raiseexpr ::=	`raise` exn *exnexpr*	
exnexpr ::=	unitbool | | int | float | char | string | var	
	| con *varlist*	
	| var *bop* var | *uop* var	
	| `if` var `then` $exnexpr_1$ `else` $exnexpr_2$	
	| `case` *caseflag* var `of` $\langle$ $exnmatch_1$ `|` $\cdots$ `|` $exnmatch_n$ $\rangle$	$n \geq 1$
	| `let` $\langle$ $exnvdecl_1$ `,` ... `,` $exnvdecl_n$ $\rangle$ `in` *exnexpr*	$n \geq 1$
	| `LET` $\langle$ $exnvdecl_1$ `,` ... `,` $exnvdecl_n$ $\rangle$ `IN` *exnexpr*	$n \geq 1$
vdecl ::=	var = *expr*	
exnvdecl ::=	var = *exnexpr*	
pat ::=	unit | bool | int | float | char | string	
	| var | _	
	| con $\langle$ pat_1 $\cdots$ pat_n $\rangle$	$n \geq 0$
bop ::=	`+` | `-` | `*` | `/` | `+.` | `-.` | `*.` | `/.` | `==` | `>=` | `<=` | `and` | `or`	
uop ::=	`-` | `-.` | `not`	

Figure 1: HUME Abstract Syntax

variables, usually ranged over by the letter x,y and z; Con for *constructors*, usually ranged over by the letter c and Exn for *exceptions*, usually ranged over by *exn*. For simplicity we sometimes conveniently assume a further subdivision of Id into three distinct subsets for functions, boxes and wires, omitting preconditions such as "*id is a wire identifier*", since this is usually clear from the context anyway.

There are two main differences from standard HUME syntax: i) elimination of syntactic sugar for specific datatypes; and ii) conversion to let-normal form. These are described in detail in the following subsections. Both changes are largely superficial, but remove a number of redundancies, which would otherwise obfuscate the underlying mechanics of the analysis. The Prototype HUME Abstract Machine Compiler (phamc) used in the EmBounded project has already been augmented to automatically transform normal HUME code into the desired format. Work is also underway to allow automatic tracing of error messages in the phase to the original HUME source code, so that the translation to the core syntax becomes transparent to the HUME programmer. More information about the implementation can be found in Deliverable D13 (WP4).

It is important to note that the translation to the core syntax is designed to be completely *cost neutral*, so that it never changes either the actual execution costs of HUME program or those reported by the analysis. The compilation process to abstract machine code (or concrete machine code) is still based on the original HUME source code, since the original HUME syntax is much more convenient to a programmer, and the compiler can take advantage of information about source level constructs.

2.1.1 Arbitrary recursive datatypes

In principle, the treatment of tuples, vectors, lists, trees, etc. should all be identical from the viewpoint of the analysis. Hence we have decided to eliminate the syntactic sugar provided for certain datatypes such as vectors and tuples and the special "none" value (`*`). Instead, we use the normal mechanisms for arbitrary recursive datatypes, i.e. instead of simply writing `(42,true)` for a pair of an integer and a bool, it is necessary to define a specific datatype and associated constructors and to use these instead, e.g. `T2intbool(42,true)`. The special costs associated with vectors and tuples are preserved by parameterising the cost that is assigned to *all* user-defined, possibly recursive datatypes. This parameterisation also easily allows for future optimisations of some datatypes by simply adjusting the cost parameters for those types. As a side effect of this transformation, the syntax of top-level pattern matches become much more simple, since instead of a list of patterns, only a single pattern of tuple type needs to be matched. Hence all top-level expressions become a single case-instruction. Again, the differences in cost are handled by a simple parameter, called case-kind, which signals whether we deal with a box entry-level, function entry-level or an ordinary expression-level matching. It is interesting to note that this mirrors exactly the treatment within the prototype implementation of the space-analysis.

2.1.2 Let-normal form

Nested expressions quickly become unwieldy in theoretical proof-work. We avoid this problem by using a program transformation into a "let-normal form", i.e. an expression such as "$f(e_1)(e_2)$" is transformed into "`let` x_1 `=` e_1 `in let` x_2 `=` e_2 `in` $f(x_1, x_2)$". While this form has the advantage of conveying the order of evaluation more explicitly, its main advantage is that the `let` construct becomes the only command that models sequential evaluation. For example, if we want to prove something about the term "`if` e_1 `then` e_2 `else` e_3" then this proof will contain similar steps to "`let` x `=` e_1 `in if` x `then` e_2 `else` e_3", which are missing if we only need to deal with conditionals of the form "`if` x `then` e_2 `else` e_3", i.e. where the guard expression is always known to be a simple variable. This transformation therefore allows us to remove many uninteresting repetitions in our theoretical work, allowing us to expose the important parts in each case that we consider. A third benefit is that the transformation also heavily reduces the treatment of exceptions, since evaluating a variable cannot throw an exception, e.g. in the above example, the transformed conditional does not need a specific rule for the case that an exception was

raised during the evaluation of the guarding expression. The further reduces redundancies in the rules. However, if performed naively, the transformation above would clearly alter the consumption of time, stack and perhaps other resources, something that would be unacceptable if we are to obtain correct upper bounds on execution costs. We circumvent this problem by introducing a pseudo-expression that is not available to HUME programmers, the *ghost-let*, written "`LET` $\cdots$ `IN` ". The idea is that the execution of a ghost-let has no operational cost, but behaves otherwise similarly to a normal `let` construct. This means that we can achieve a pseudo-transformation into the let-normal form without altering the actual resource cost of an execution. We have revised deliverable D12 to incorporate this modification. The simple program transformation $\varrho(\cdot)$ into ghost-let-normal form then works as follows (any syntactic constructs not mentioned here are unaltered by the transformation):

Prim Op	$op\ e_1 \cdots e_k$	$\rightsquigarrow$	`LET` x_1 `=` $\varrho(e_1), \ldots, x_k$ `=` $\varrho(e_k)$ `IN` $op\ x_1 \cdots x_k$
Call	$fid\ e_1 \cdots e_k$	$\rightsquigarrow$	`LET` x_1 `=` $\varrho(e_1), \ldots, x_k$ `=` $\varrho(e_k)$ `IN` $fid\ x_1, \ldots, x_k$
Constructor	$c\ e_1 \cdots e_k$	$\rightsquigarrow$	`LET` x_1 `=` $\varrho(e_1), \ldots, x_k$ `=` $\varrho(e_k)$ `IN` $c\ x_1 \cdots x_k$
Tuple	$(e_1, \ldots, e_k)$	$\rightsquigarrow$	`LET` x_1 `=` $\varrho(e_1), \ldots, x_k$ `=` $\varrho(e_k)$ `IN` $(x_1, \ldots, x_k)$
Conditional	`if` e_1 `then` e_2 `else` e_3	$\rightsquigarrow$	`LET` x `=` $\varrho(e_1)$ `IN if` x `then` $\varrho(e_2)$ `else` $\varrho(e_3)$
Case	`case` e `of` pat_1 `->` e_1 `|` $\cdots$ `|` pat_k `->` e_k	$\rightsquigarrow$	`LET` x `=` $\varrho(e)$ `IN` `case` x `of` pat_1 `->` $\varrho(e_1)$ `|` $\cdots$ `|` pat_b `->` $\varrho(e_b)$

where the introduced variables $x, y, x_i, \ldots$ are always assumed to be fresh. One might wonder why we did not add the following transformation for the ordinary let as well:

$$\texttt{let}\ x_1 = e_1, \ldots, x_k = e_k\ \texttt{in}\ e \rightsquigarrow \begin{array}{l}\texttt{LET}\ x_1 = \varrho(e_1), \ldots, x_1 = \varrho(e_k)\ \texttt{IN} \\ \quad \texttt{let}\ x_1 = x_1, \ldots, x_k = x_k\ \texttt{in}\ \varrho(e)\end{array}$$

The reason is that the expression `let` x_1 `=` $e_1, \ldots, x_k$ `=` e_k `in` e is really just an abbreviation for `let` x_1 `=` e_1 `in let` x_2 `=` e_2 `in` $\cdots$`let` x_k `=` e_k `in` e. Hence the subsequent expressions may depend on the previously-introduced variables. This leads to an unusual resource behaviour, especially since the HUME compiler optimises the evaluation of `let` constructs by preallocating a stack frame for the defined variables similar to a call frame, which in turn allows the result of a preceding evaluation of subexpression to be popped from the stack before computing the next. This peculiarity requires us to treat `let` constructs individually, thereby excluding the `let` construct from the transformation into let-normal form.

2.2 Basic HUME Type System

HUME *types* are defined by the following grammar:

$$\begin{array}{lll} A & ::= & B \mid C \mid X \mid \mu X.\{c_1{:}A_{(1,1)}, \ldots, A_{(1,j_1)} \mid \ldots \mid c_k{:}A_{(1,k)}, \ldots, A_{(k,j_k)}\} \\ B & ::= & \mathsf{unit} \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{char} \mid \mathsf{bool} \mid \mathsf{string} \\ C & ::= & A_1, \ldots, A_k \rightarrow A \end{array}$$

where $c_i \in \mathsf{Con}$ are constructor labels. We have the usual set of base types, higher-order types, type variables and type abstractions that can be used to build arbitrary recursive datatypes.

A recursive datatype consists of a type-variable abstraction and a partial-mapping from the set of constructor labels Con to an ordered, possibly empty list of type-terms. For example, standard binary trees with integer-labelled leaves and nodes would be written

$$\mu Y.\{\texttt{Leaf}{:}\mathsf{int} \mid \texttt{Node}{:}\mathsf{int}, Y, Y\}$$

Note that as for all partial mappings, the order in which the constructors are listed is irrelevant, hence the type $\mu Y.\{\texttt{Node}{:}\mathsf{int}, Y, Y \mid \texttt{Leaf}{:}\mathsf{int}\}$ is identical to that given above. Furthermore we treat all HUME

types module α-equivalence on the bound type variables, e.g. $\mu Y.\{\texttt{C}{:}Y\} = \mu X.\{\texttt{C}{:}X\}$. We should point out that non-recursive datatypes are already subsumed, since the abstracted variable is not required to occur within the range of the partial map. Although we may sometimes omit the leading μ-operator and the superfluous abstracted variable for a non-recursive types for the sake of brevity, we do not wish to treat non-recursive types any different from recursive ones, thereby avoiding unnecessary redundancies in our type rules. Note moreover, that when referring to types, we only refer to closed type-terms. There is no explicit folding and unfolding of recursive datatypes. Folding and unfolding is implicitly paired with the construction and the destruction of each datatype in the usual way. Finally, note that we allow higher-order types in an uncurried style. This is to allow for possibly different costs in the case of partial applications, so allowing us to explicitly distinguish between $A_1 \rightarrow A_2 \rightarrow C$ and $A_1, A_2 \rightarrow C$, and so achieving a more accurate costing for higher-order functions. In this document, we will use the term "partial application" to refer only to applications of functions of the latter type.

2.2.1 Type Rules for Expressions

We first show how to type a single HUME expression. Our type rules for HUME expressions have the form

$$\Sigma; \Gamma \vdash e : A$$

meaning that the HUME expression e is of type A within context Γ and signature Σ. A context is a partial map from the set of variables Var to types. A signature Σ maps function identifiers belonging to the set Id to a triple consisting of a term defining the function's body, an ordered list of argument variables and a type. Since the signature Σ remains fixed with the program to be analysed, we refrain from mentioning it within the premises of each expression-level rule in favour of a leaner notation. We will describe the requirements on the signature in the later section 2.2.3, which describes how to treat whole HUME programs.

$$\frac{}{\varnothing \vdash \texttt{()} : \mathsf{unit}} \quad \text{(Unit)}$$

$$\frac{b \in \mathbb{B}}{\varnothing \vdash b : \mathsf{bool}} \quad \text{(Bool)}$$

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash n : \mathsf{int}} \quad \text{(Int)}$$

$$\frac{r \in \mathbb{R}}{\varnothing \vdash r : \mathsf{float}} \quad \text{(Float)}$$

$$\frac{c \texttt{ is a character}}{\varnothing \vdash c : \mathsf{char}} \quad \text{(Char)}$$

$$\frac{s \texttt{ is a string}}{\varnothing \vdash s : \mathsf{string}} \quad \text{(String)}$$

$$\frac{}{x{:}A \vdash x : A} \quad \text{(Var)}$$

$$\frac{\mathrm{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash x \ \mathrm{op} \ y : \mathsf{int}} \quad \text{(PrimBOp Int)}$$

$$\frac{\text{op} \in \{\texttt{-}\}}{x{:}\mathsf{int} \vdash \ \text{op}\ y : \mathsf{int}} \qquad \text{(PrimUOp Int)}$$

$$\frac{\text{op} \in \{\texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}\}}{x{:}\mathsf{float}, y{:}\mathsf{float} \vdash x\ \text{op}\ y : \mathsf{float}} \qquad \text{(PrimBOp Float)}$$

$$\frac{\text{op} \in \{\texttt{-.}\}}{x{:}\mathsf{float} \vdash \ \text{op}\ y : \mathsf{float}} \qquad \text{(PrimUOp Float)}$$

$$\frac{\text{op} \in \{\texttt{==}, \texttt{>=}, \texttt{<=}\}}{x{:}A, y{:}A \vdash x\ \text{op}\ y : \mathsf{bool}} \qquad \text{(PrimBOp Eq)}$$

$$\frac{\text{op} \in \{\texttt{and}, \texttt{or}\}}{x{:}\mathsf{bool}, y{:}\mathsf{bool} \vdash x\ \text{op}\ y : \mathsf{bool}} \qquad \text{(PrimBOp Bool)}$$

$$\frac{\text{op} \in \{\texttt{not}\}}{x{:}\mathsf{bool} \vdash \ \text{op}\ y : \mathsf{bool}} \qquad \text{(PrimUOp Bool)}$$

$$\frac{\Sigma(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \to C\right) \qquad k \geq 0 \qquad k = a}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash \mathit{fid}\ y_1 \cdots y_k : C} \qquad \text{(App)}$$

$$\frac{\Sigma(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \to C\right) \qquad k \geq 1 \qquad k < a}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash \mathit{fid}\ y_1 \cdots y_k : A_{k+1}, \ldots, A_a \to C} \qquad \text{(Under App)}$$

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \left(\_;\_; A_1, \ldots, A_a \to C\right) \qquad a \geq 1 \qquad k > a \\ y_1{:}A_1, \ldots, y_a{:}A_a \vdash \mathit{fid}\ y_1 \cdots y_a : C \qquad x \text{ is fresh} \\ x{:}C, y_{a+1}{:}A_{a+1}, \ldots, y_k{:}A_k \vdash x\ y_{a+1} \cdots y_k : E\end{array}}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash \mathit{fid}\ y_1 \cdots y_k : E} \qquad \text{(Over App)}$$

$$\frac{D = A_1, \ldots, A_a \to C \qquad k \geq 1 \qquad k = a}{z{:}D, y_1{:}A_1, \ldots, y_k{:}A_k \vdash z\ y_1 \cdots y_k : C} \qquad \text{(App Var)}$$

$$\frac{\begin{array}{c}D = A_1, \ldots, A_a \to C \qquad k \geq 1 \qquad k < a \\ B = A_{k+1}, \ldots, A_a \to C\end{array}}{z{:}D, y_1{:}A_1, \ldots, y_k{:}A_k \vdash z\ y_1 \cdots y_k : B} \qquad \text{(Under App Var)}$$

$$\frac{\begin{array}{c}D = A_1, \ldots, A_a \to C \qquad a \geq 1 \qquad k > a \\ y_1{:}A_1, \ldots, y_a{:}A_a \vdash z\ y_1 \cdots y_a : C \qquad x \text{ is fresh} \\ x{:}C, y_{a+1}{:}A_{a+1}, \ldots, y_k{:}A_k \vdash x\ y_{a+1} \cdots y_k : E\end{array}}{z{:}D, y_1{:}A_1, \ldots, y_k{:}A_k \vdash z\ y_1 \cdots y_k : E} \qquad \text{(Over App Var)}$$

The rules for function application, closure-creation (or under application) and over-application are repeated twice: once for applications to normal functions, and once for applications to *closures* arising from higher-order calls through variables representing parameters to function calls. While we have tried to minimise the number of rules that are required, we must treat the application of literal functions and called variables separately, since the enriched type systems for the resource analyses require slightly different costs in each case.

$$\frac{c \in \mathsf{Con} \qquad C = \mu X.\{\cdots \mid c : B_1, \ldots, B_k \mid \cdots\} \quad A_i = B_i \vee (A_i = C \wedge B_i = X) \text{ (for } i = 1, \ldots, k)}{x_1{:}A_1, \ldots, x_k{:}A_k \vdash c\ x_1 \ldots x_k : C} \tag{Constr}$$

$$\frac{\Gamma \vdash e_t : A \qquad \Gamma \vdash e_f :}{\Gamma, x{:}\mathsf{bool} \vdash \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A} \tag{Conditional}$$

$$\frac{\forall i\,.\, \begin{cases} A \vdash pat_i\ ?\triangleright\ \Delta_i \\ \Gamma, \Delta_i \vdash e_i : B \end{cases}}{\Gamma, x{:}A \vdash \texttt{case}\ \mathrm{ck}\ x\ \texttt{of}\ pat_1\ \texttt{->}\ e_1 \mid \cdots \mid pat_k\ \texttt{->}\ e_k : B} \tag{Case}$$

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

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

The type rules for the let-expression seem somewhat complicated due to the fact that they allow later definitions to depend on earlier ones. This requires the use of the sharing relations, since in the enriched type systems for the resource analyses need to be aware of multiple uses of each entity. However, each reference must have the same type in this simple version of the type system.

Exceptions We require one ordinary HUME type to be distinguished as the type of exceptions (*Err*). There is nothing special about this *user-defined type*, whose form will generally be similar to $\mu X.\{\texttt{PatternMatchFailure|DivisionByZero int|}\cdots\}$. The only pecularity about *Err* is that the argument type to all `raise` expressions must be equal to the argument type of all exception handlers. It is therefore possible to refer to type *Err* throughout an entire program. Consequently, this distinguished type is formally a part of the pervasive signature Σ, as for all normal definitions in a program. However, for the sake of brevity and readability, we will often omit it explicitly in the signature.

$$\frac{\Gamma \vdash e : \mathit{Err}}{\Gamma \vdash \texttt{raise}\ \mathit{exn}\ e : A} \tag{Raise}$$

$$\frac{\Gamma \vdash e : A \quad \Delta \vdash \texttt{raise}\ \mathit{exn}\ e_x : A}{\Gamma, \Delta \vdash e\ \texttt{within}\ q\ \texttt{time raise}\ \mathit{exn}\ e_x : A} \tag{Within Time}$$

$$\frac{\Gamma \vdash e : A \qquad \Delta \vdash \texttt{raise}\ exn\ e_x : A}{\Gamma, \Delta \vdash e\ \texttt{within}\ q\ \texttt{stack raise}\ exn\ e_x : A} \quad \text{(Within Stack)}$$

$$\frac{\Gamma \vdash e : A \qquad \Delta \vdash \texttt{raise}\ exn\ e_x : A}{\Gamma, \Delta \vdash e\ \texttt{within}\ q\ \texttt{heap raise}\ exn\ e_x : A} \quad \text{(Within Heap)}$$

Substructural rules It is a noteworthy feature of our type system that all* substructural type rules have been made explicit, where substructural type rules are type rules which are not syntax-driven, i.e. have the same expression occuring in one of their premises. Removing the rules for weakening and sharing (also known as *contraction*) leaves a *linear type system*. The reason behind this design decision is that we require to control the number of references to an object in our type based resource analyses, so that we can determine when sharing occurs through the duplication of references, for example. Furthermore, explicit substructural rules allow the use of simpler rules overall, since this approach avoids merging these properties into all other rules, with consequent obfuscation. Note that additional substructural rules will be introduced in the annotated type system in section 3.1.

$$\frac{\Gamma \vdash e : C}{\Gamma, x{:}A \vdash e : C} \quad \text{(Weak)}$$

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

where the *sharing* relation $\curlyvee(A \,|\mathcal{A})$ is a relation between a types and multisets of types:

$$\curlyvee(A \,|\mathcal{A}) \text{ iff } \forall x \in \mathcal{A} \,.\, x = A$$

that is all elements (if any) of the multiset are equal to the single given type. While this definition may seem somewhat contrived at this point, it will become an important part of the annotated type system for the analysis, where this relation is redefined as partial map, mapping a single type and a multiset of types to set of numerical constraints.

2.2.2 Type Rules for Pattern Matches

The type rule for case-expressions requires another construct, dealing with pattern matches:

$$A \vdash pat\ ?\triangleright\ \Gamma$$

where *pat* is a HUME pattern to be tested against an object of type A. If the pattern matches successfully, then the resulting bindings are given by the context Γ.

$$\frac{}{\mathsf{unit} \vdash \texttt{()}\ ?\triangleright\ \varnothing} \quad \text{(Pattern Unit)}$$

$$\frac{b \in \mathbb{B}}{\mathsf{bool} \vdash b\ ?\triangleright\ \varnothing} \quad \text{(Pattern Bool)}$$

*That is all substructural properties except for permutation, which is built-in since we have, from the outset, defined contexts to be partial maps. Hence, we do not distinguish between the two contexts $x{:}A, y{:}B$ and $y{:}B, x{:}A$.

$$\frac{n \in \mathbb{Z}}{\mathsf{int} \vdash n \ ?\triangleright \ \varnothing} \quad \text{(Pattern Int)}$$

$$\frac{r \in \mathbb{R}}{\mathsf{float} \vdash r \ ?\triangleright \ \varnothing} \quad \text{(Pattern Float)}$$

$$\frac{c \ \texttt{is a character}}{\mathsf{char} \vdash c \ ?\triangleright \ \varnothing} \quad \text{(Pattern Char)}$$

$$\frac{s \ \texttt{is a string}}{\mathsf{string} \vdash s \ ?\triangleright \ \varnothing} \quad \text{(Pattern String)}$$

$$\frac{}{A \vdash x \ ?\triangleright \ x{:}A} \quad \text{(Pattern Var)}$$

$$\frac{}{A \vdash \_ \ ?\triangleright \ \varnothing} \quad \text{(Pattern Wild)}$$

$$\frac{\forall i \,.\, (B_i \vdash pat_i \ ?\triangleright \ \Gamma_i)}{\mu X.\{\cdots \mid c : B_1, \ldots, B_k \mid \cdots\} \vdash c \ pat_1 \cdots pat_k \ ?\triangleright \ \Gamma_1, \cdots, \Gamma_k} \quad \text{(Pattern Constr)}$$

2.2.3 Type Rules for Boxes and Declarations

A HUME program consists of a number of box and function declarations. Formally, the program is defined by its signature Σ, which consist of two mappings: one mapping function-identifiers to their declaration (a triple consisting of: the functions body, a list of argument variables, and the functions type) and another mapping box-identifiers to box definitions.

Our translation to core HUME greatly simplifies the handling of boxes and thus the nature of box definitions, since instead of dealing with an arbitrary number of input and output wires and the possibilities of a wire being empty, a box is transformed into a single case expression which receives only a single input *bundle*. A bundle is a non-recursive datatype having only one unique constructor, which has as many arguments as the original box had input wires. These argument types are themselves non-recursive constructors, each having two unique constructors, one of which has no argument (indicating that no value is present in the wire), and one which has precisely one argument of the same type as the original wire. In this manner, all the input is encapsulated in a single type. A similar technique is used for the output. More precisely, if the original box definition was represented by a 7-tuple consisting of: two ordered lists of typed wire variables for input and output respectively, a list containing pairs of pattern-lists and expressions, the box-flag determining fair or unfair matching; the type of the possible exceptions and the exception handler; then the transformed box definition is a 6-tuple consisting of an expression, an argument variable, the fairness flag and the exception handler. Hence the box definition is much more similar to a function definition and much easier to manage within our analysis. In detail, the transformation simplifies

$$\Sigma(box) = \begin{pmatrix} [y_1{:}A_1, \ldots, y_k{:}A_k]; \\ [z_1{:}C_1, \ldots, z_h{:}C_h]; \\ \big[(pat_{(1,1)}, \ldots, pat_{(1,k)}; e_1), \ldots, (pat_{(1,1)}, \ldots, pat_{(j,k)}, e_j)\big]; \\ \text{fairness}; B_x; \ e_x \end{pmatrix}$$

to the more concise form

$$\Sigma(box) = \big(e_{box}; y; A_{box} \rightarrowtail C_{box}; \text{fairness}; B_x; e_x\big)$$

where

$$
\begin{aligned}
A_{box} = \;& \mu Y.\{\text{Bundle-ik} : \{\text{Wire-i1} : A_1 \mid \text{None-i1}\}, \ldots, \{\text{Wire-ik} : A_k \mid \text{None-ik}\}\} \\
C_{box} = \;& \mu Z.\{\text{Bundle-oh} : \{\text{Wire-o1} : C_1 \mid \text{None-o1}\}, \ldots, \{\text{Wire-oh} : C_h \mid \text{None-oh}\}\} \\
e_b = \;& \texttt{case boxcase } y \texttt{ of} \\
& \quad \text{Bundle-ik (Wire-i1 } pat_{(1,1)}) \cdots (\text{Wire-ik } pat_{(1,k)}) \texttt{ -> } e_1 \\
& \mid \text{Bundle-ik (Wire-i1 } pat_{(2,1)}) \cdots (\text{Wire-ik } pat_{(2,k)}) \texttt{ -> } e_2 \\
& \quad \vdots \\
& \mid \text{Bundle-ik (Wire-i1 } pat_{(j,1)}) \cdots (\text{Wire-ik } pat_{(j,k)}) \texttt{ -> } e_j
\end{aligned}
$$

and each constructor Bundle-ik, Bundle-oh, Wire-i1, ..., Wire-ik, Wire-o1, ..., Wire-oh is assumed to be unique to box *box*, except for the wire constructors which may lead to another box inside the HUME program.

We can now define when a HUME program is well-typed. 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

$$\Sigma(fid) = \big(e_f; [y_1, \ldots, y_k]; A_1, \ldots, A_k \rightarrowtail C\big)$$

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

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

$$\Sigma(box) = e_b; y; A \rightarrowtail C; \text{fairness}; B_x; e_x$$

there exists a finite type derivation such that $y{:}A \vdash e_b : C$ and $err{:}Err \vdash e_x : C$

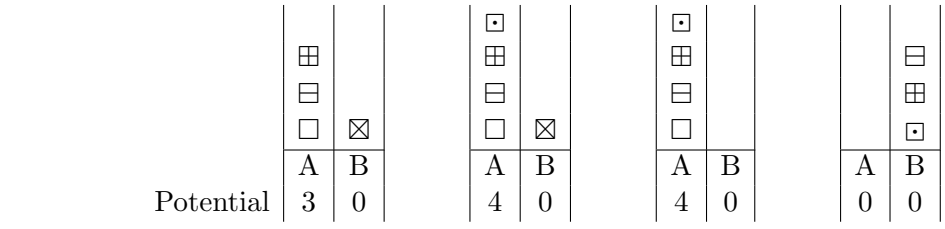
c) For all pairs of boxes sharing a wire, type assigned to each wire is identical.

2.3 Basic Principle of an Amortised Analysis

In amortised analysis, data structure(s) are assigned an arbitrary non-negative number, representing the *potential*, usually denoted by Φ. The amortised cost of an operation is its total cost (time or space) plus the difference in potential before and after the operation. The sum of the amortised costs plus the potential of the initial data structure then bounds (from above) the actual cost of a sequence of operations. If the potential is chosen carefully, then the amortised cost of individual operations can be either zero or a constant even when their actual cost is difficult to determine.

The simplest example to show the usefulness of an amortised analysis is an implementation of a queue using two stacks A and B. Enqueuing is performed on stack A, and dequeuing is performed on stack B, unless B is empty, in which case the entire contents of A are moved to B prior to dequeuing. A sequence of example operations on such a simulated queue is shown in Figure 2. The costs shown in this example here are for the time to execute a queue operation, but the principle can equally be used for a different resource or other countable properties as well. We have chosen to use a time rather than heap metric here since this conveys the intuition behind our approach in a straightforward way: it is easy to see that enqueuing (`enqu( )`) has a time cost of one; while dequeuing (`dequ( )`) has a variable cost, which is often one but may sometimes be proportional to the size of A.

This variable cost is unpleasant, since costing a sequence of queue operations will apparently require us to track the current state of stack A. However, if we decree that the size of A is the potential of the data then enqueuing will always have an amortised cost of 2 (one for the actual cost, one for the



Operation	$\overrightarrow{\texttt{enqu}(\boxdot)}$	$\overrightarrow{\texttt{dequ}(\,) = \boxtimes}$	$\overrightarrow{\texttt{dequ}(\,) = \square}$
Actual Cost	1	1	5
Change of Potential	1	0	-4
Amortised Cost	2	1	1

Figure 2: Simulating a Queue by using two Stacks

increase in potential) and dequeuing will always have an amortised cost of 1, since the cost of moving A over to (the empty stack) B cancels out against the decrease in potential. Thus, the actual cost of a sequence of operations is bounded by the initial size of A, plus twice the number of enqueue operations, plus the number of dequeue operations. In this case, we can also see this directly by observing that each element is moved exactly three times: once into A, once from A to B, once out of B. Therefore we require only the initial state, but do not have to model the state changes in order to determine the cost of a sequence of operations. In the above queue example, both stacks have the same type, but each element of A contributes 1 to the overall potential, whereas each element of B contributes a potential of 0. Our idea is now to record this information within the type by adding a number to each type constructor, which denotes the potential carried by each constructor of that type. So given an element of such an annotated type, we can compute its assigned potential by summing over all nodes times their assigned factor.

It is important to note that, while our analysis is entirely done at compile-time, the absolute potential can in fact be only computed at runtime, when the concrete data and its layout is known. However, we will never actually compute the potential, but rather concern ourselves with the hypothetical *change* of potential along all possible paths of computations, as expressed in our annotated type rules, which we will now define below.

3 Costing Heap-space

Let CV be an infinite set of *resource variables* disjoint from the identifier sets $\mathsf{Var}, \mathsf{Id}, \mathsf{Con}, \mathsf{Exn}$. Resource variables are always assumed to range over $\mathbb{Q}^+$. The *annotated types* of HUME are then given by the following grammar:

$$\begin{array}{lll} A & ::= & B \mid C \mid X \mid \mu X.\{c_1{:}q_1; A_{(1,1)}, \ldots, A_{(1,j_1)} \mid \ldots \mid c_k{:}q_k; A_{(k,1)}, \ldots, A_{(k,j_k)}\} \\ B & ::= & \mathsf{unit} \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{char} \mid \mathsf{bool} \mid \mathsf{string} \\ C & ::= & \forall \alpha \in \psi. A_1, \ldots, A_k \xrightarrow[m']{m} A \end{array}$$

where $c_i \in \mathsf{Con}$ are constructor labels, m, m', q_i are resource variables belonging to the set CV and which range over $\mathbb{Q}^+$, $\alpha \subset \mathsf{CV}$ is a set of resource variables and ψ is a set of linear inequalities composed of non-negative rational constants and resource variables (not necessarily contained in α) both ranging over $\mathbb{Q}^+$. We may sometimes omit both α and ψ if both sets are empty. There are two differences from the basic system described in Section 2.2:

a) Each constructor now carries a resource variable ranging over $\mathbb{Q}^+$, which represents the potential associated with an instance of that constructor applied to the supplied annotated types.

b) Function types now carry two annotations, representing: i) the base potential that is required to call the function; and ii) the amount of potential released upon return from the call.

   Furthermore, it is possible to bind certain resource variables, so that one and the same function may be used in a way that is *resource polymorphic*. For example, the identity function on integer lists could be assigned the type

$$\forall \{x, y, z\} \in \{x \geq y\}.\, \mu X.\{\mathrm{nil} : 0 \,|\, \mathrm{cons} : z; \mathsf{int}, X\} \xrightarrow[y]{x} \mu X.\{\mathrm{nil} : 0 \,|\, \mathrm{cons} : z; \mathsf{int}, X\}$$

   which is obviously much more general than any type that is confined to concrete values, such as $\mu X.\{\mathrm{nil} : 0 \,|\, \mathrm{cons} : 42; \mathsf{int}, X\} \xrightarrow[7]{7} \mu X.\{\mathrm{nil} : 0 \,|\, \mathrm{cons} : 42; \mathsf{int}, X\}$.

Following the ideas of Section 2.3, we can now assign runtime values a *potential* based on their enriched HUME type. The potential of a runtime value, v, of enriched HUME type, A, is written $\Phi_\eta(v : A)$, and is the sum over all annotations associated with each constructor occuring within value v or reached by reference within heap η from v. Given a context, Γ, and a suitable environment, $\mathcal{V}$, we can naturally extend this definition by

$$\Phi_\eta(\mathcal{V} : \Gamma) := \sum_{x \in \mathrm{dom}(\Gamma)} \Phi_\eta(\mathcal{V}(x) : \Gamma(x))$$

Where "suitable" means that the named values in environment $\mathcal{V}$ correspond to the types given to them by context Γ, including all referenced values in the heap η. We will formalize this notion later by

$$\eta \vDash \mathcal{V} : \Gamma$$

However, for the time being, the notion of a well-typed memory configuration should be intuitively clear. Note that we never actually need to compute the potential: our analysis is carried out entirely at compile time and our result will quantify over all possible runtime values of a certain enriched HUME type. Returning to our example of integer-labelled binary trees, any tree of the type

$$\mu Y.\{\texttt{Leaf}{:}p; \mathsf{int} \mid \texttt{Node}{:}q; \mathsf{int}, Y, Y\}$$

carries a potential of q for each leaf and a potential of p for each node. An example showing how to compute the overall potential for elements of this type is given in Figure 3. It is important to note that the potential is linearly distributed over a data structure, hence trees with shared values (i.e. where

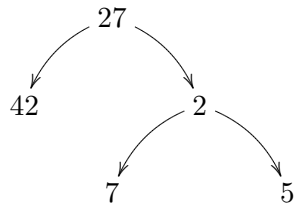
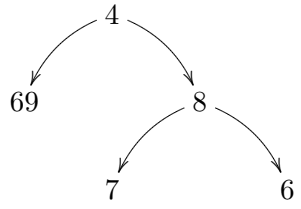
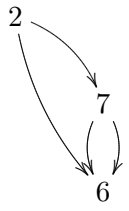
Enriched HUME Type	Value Instance in Memory	Potential Φ
$\mu Y.\{\texttt{Leaf}{:}0; \mathsf{int} \mid \texttt{Node}{:}3; \mathsf{int}, Y, Y\}$		$3 \cdot 0 + 2 \cdot 3 \quad = \ 6$
$\mu Y.\{\texttt{Leaf}{:}4; \mathsf{int} \mid \texttt{Node}{:}1; \mathsf{int}, Y, Y\}$		$3 \cdot 4 + 2 \cdot 1 \quad = 14$
$\mu Y.\{\texttt{Leaf}{:}4; \mathsf{int} \mid \texttt{Node}{:}1; \mathsf{int}, Y, Y\}$		$3 \cdot 4 + 2 \cdot 1 \quad = 14$

Figure 3: Integer labelled binary trees and their potential.

some nodes are pointed to by more than one parent node) carry the same potential as trees without sharing. However, recall again that we will never have to actually compute the potential. The potential is entirely calculated at compile time, and never at runtime. So figure 3 just illustrates the underlying meaning of the notion "potential" and nothing else.

3.1 Annotated Type Rules for Expressions

The type rules for expressions have the form

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

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 $\in \mathbb{Q}^+$. The meaning of the statement relies on the notion of a *valuation* $\boldsymbol{v}$ mapping all resource variables to $\mathbb{Q}^+$: For all *valuations* $\boldsymbol{v}$ such that the constraint set $\boldsymbol{v}(\psi)$ is satisfied, the HUME expression e has type $\boldsymbol{v}(A)$ in the context $\boldsymbol{v}(\Gamma)$. The enriched typing statement also tells us about the resource usage of expression e: For all memory configurations consisting of environment $\mathcal{V}$ and heap η fitting the context Γ, executing expression e will require at most $\boldsymbol{v}(m) + \Phi_\eta\big(\mathcal{V} : \boldsymbol{v}(\Gamma)\big)$ heap units. Furthermore, if the computation finishes with heap η' and result value v, then at least $\boldsymbol{v}(m') + \Phi_{\eta'}\big(v : \boldsymbol{v}(\Gamma)\big)$ heap units remain unused after the computation has finished. We formalise the correctness of the rules by 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 the four statements above are all satisfied, then it follows that for all $r \in \mathbb{Q}^{+}$ and $m \in \mathbb{N}$ such that $m \geq \boldsymbol{v}(q) + \Phi_{\eta}\big(\mathcal{V} : \boldsymbol{v}(\Gamma)\big) + r$ holds, there exists $m' \in \mathbb{N}$ satisfying $m' \geq \boldsymbol{v}(q') + \phi_{\eta'}\big(v{:}\boldsymbol{v}(A)\big) + r$ and also $\eta, \mathcal{V} \vdash^{p}_{p'} \vdash^{m}_{m'} e \rightsquigarrow_{\diamond} v, \eta'$ for some $p, p' \in \mathbb{Q}^{+}$.

Recall that the statment $\eta, \mathcal{V} \vdash^{p}_{p'} \vdash^{m}_{m'} e \rightsquigarrow_{\diamond} v, \eta'$ was defined in the formal HUME operational semantics (D12) and specifies that e evaluates successfully in the given configuration if at least m heap-units and p stack-units are available in the beginning and that at least m' heap-units and p' stack-units are unused after the evaluation is finished.

Note: The statement given in premise 1.II *measures* the resource consumption of the evaluation of e on a given *specific* input, whereas the statement in premise 1.I, which we will define shortly, *bounds* the resource consumption of expression e for *all* well-typed inputs. We now give the annotated type rules for HUME expressions for bounding Heap-space consumption:.

$$\frac{}{\varnothing \vdash^{2}_{0} \texttt{()} : \mathsf{unit} \mid \emptyset} \tag{Unit}$$

$$\frac{b \in \mathbb{B}}{\varnothing \vdash^{2}_{0} b : \mathsf{bool} \mid \emptyset} \tag{Bool}$$

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash^{2}_{0} n : \mathsf{int} \mid \emptyset} \tag{Int}$$

$$\frac{r \in \mathbb{R}}{\varnothing \vdash^{2}_{0} r : \mathsf{float} \mid \emptyset} \tag{Float}$$

$$\frac{c \ \texttt{is a character}}{\varnothing \vdash^{2}_{0} c : \mathsf{char} \mid \emptyset} \tag{Char}$$

$$\frac{s \ \texttt{is a string}}{\varnothing \vdash^{2+\lceil |s|/4 \rceil}_{0} s : \mathsf{string} \mid \emptyset} \tag{String}$$

$$\frac{}{x{:}A \vdash^{0}_{0} x : A \mid \emptyset} \tag{Var}$$

$$\frac{\mathrm{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash^{2}_{0} x \ \mathrm{op} \ y : \mathsf{int} \mid \emptyset} \tag{PrimBOp Int}$$

$$\frac{\mathrm{op} \in \{\texttt{-}\}}{x{:}\mathsf{int} \vdash^{2}_{0} \ \mathrm{op} \ y : \mathsf{int} \mid \emptyset} \tag{PrimUOp Int}$$

$$\frac{\mathrm{op} \in \{\texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}\}}{x{:}\mathsf{float}, y{:}\mathsf{float} \vdash^{2}_{0} x \ \mathrm{op} \ y : \mathsf{float} \mid \emptyset} \quad \textsc{(PrimBOp Float)}$$

$$\frac{\mathrm{op} \in \{\texttt{-.}\}}{x{:}\mathsf{float} \vdash^{2}_{0} \ \mathrm{op} \ y : \mathsf{float} \mid \emptyset} \quad \textsc{(PrimUOp Float)}$$

$$\frac{\mathrm{op} \in \{\texttt{==}, \texttt{>=}, \texttt{<=}\}}{x{:}A, y{:}A \vdash^{2}_{0} x \ \mathrm{op} \ y : \mathsf{bool} \mid \emptyset} \quad \textsc{(PrimBOp Eq)}$$

$$\frac{\mathrm{op} \in \{\texttt{and}, \texttt{or}\}}{x{:}\mathsf{bool}, y{:}\mathsf{bool} \vdash^{2}_{0} x \ \mathrm{op} \ y : \mathsf{bool} \mid \emptyset} \quad \textsc{(PrimBOp Bool)}$$

$$\frac{\mathrm{op} \in \{\texttt{not}\}}{x{:}\mathsf{bool} \vdash^{2}_{0} \ \mathrm{op} \ y : \mathsf{bool} \mid \emptyset} \quad \textsc{(PrimUOp Bool)}$$

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

$$\frac{\begin{array}{c}\Sigma(\mathit{fid}) = \big(\_;\_;\forall\alpha \in \psi.A_1, \ldots, A_a \xrightarrow[m']{m} C\big) \qquad k \geq 1 \qquad k < a \\ \forall i \leq k \,.\, \big(\phi_i = \curlyvee(A_i \,|\, A_i, A_i)\big) \qquad \beta = \alpha \setminus \mathrm{FV}\big(A_1, \ldots, A_k\big)\end{array}}{y_1{:}A_1, \ldots, y_k{:}A_k \vdash^{4+k}_{0} \mathit{fid} \ y_1 \cdots y_k : \forall\beta \in \psi.A_{k+1}, \ldots, A_a \xrightarrow[m']{m} C \mid \bigcup_i \phi_i} \quad \textsc{(Under App)}$$

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

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

$$\frac{\begin{array}{c}D = \forall\alpha \in \psi.A_1, \ldots, A_a \xrightarrow[m']{m} C \qquad k \geq 1 \qquad k < a \\ \forall i \leq k \,.\, \big(\phi_i = \curlyvee(A_i \,|\, A_i, A_i)\big) \qquad \beta = \alpha \setminus \mathrm{FV}\big(A_1, \ldots, A_k\big) \qquad B = \forall\beta \in \psi.A_{k+1}, \ldots, A_a \xrightarrow[m']{m} C\end{array}}{z{:}D, y_1{:}A_1, \ldots, y_k{:}A_k \vdash^{4+k}_{0} z \ y_1 \cdots y_k : B \mid \bigcup_i \phi_i} \quad \textsc{(Under App Var)}$$

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

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

The rule Constr plays a crucial role in our annotated HUME type system, since it is within this rule where free potential is bound to a data structure. The counterpart to this rule is the pattern matching rule Pattern Constr, which releases bound potential again. The pattern matching rules are explained in section 3.2 and occur as premises within the rules Case Expr/Fun/Box.

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

$$\frac{\text{ck} \in \{\texttt{exprcase}, \texttt{funcase}\} \qquad \forall i \,. \begin{cases} A \vdash^{m_{(i-1)}}_{m_i} pat_i \;?\triangleright\; \Delta_i; \pi_i; \psi_i \\ \Gamma, \Delta_i \vdash^{m'_i}_{m''} e_i : B \mid \phi_i \\ \chi_i = \{\pi_i + m_i \geq m'_i\} \end{cases}}{\Gamma, x{:}A \vdash^{m_0}_{m''} \texttt{case ck } x \texttt{ of } pat_1 \texttt{ -> } e_1 \texttt{ | } \cdots \texttt{ | } pat_k \texttt{ -> } e_k : B \mid \bigcup_i \psi_i \cup \phi_i \cup \chi_i} \quad \text{(Case Expr/Fun)}$$

$$\frac{\text{ck} = \texttt{boxcase} \qquad \forall i \,. \begin{cases} A \vdash^{m_{(i-1)}}_{m_i} pat_i \;?\triangleright\; \Delta_i; \pi_i; \psi_i \\ \Gamma, \Delta_i \vdash^{m'_i}_{m''} e_i : B \mid \phi_i \\ \chi_i = \{\pi_i + m_i \geq m'_i + m_0\} \end{cases}}{\Gamma, x{:}A \vdash^{m_0}_{m''} \texttt{case ck } x \texttt{ of } pat_1 \texttt{ -> } e_1 \texttt{ | } \cdots \texttt{ | } pat_k \texttt{ -> } e_k : B \mid \bigcup_i \psi_i \cup \phi_i \cup \chi_i} \quad \text{(Case Box)}$$

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 m_0 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.

$$\frac{\forall i \,. \begin{cases} \Delta_i = \{x_1{:}A^i_1, \ldots, x_{(i-1)}{:}A^i_{(i-1)}\} \restriction \mathrm{FV}(e_i) \\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\}) \\ \psi_i = \curlyvee(A_i \,|\, \mathcal{A}_i) \\ \Delta_i, \Gamma_i \vdash^{m_{(i-1)}}_{m_i} e_i : A_i \mid \phi_i \end{cases}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash^{m_0}_{m_{k+1}} \texttt{LET } x_1 = e_1, \ldots, x_k = e_k \texttt{ IN } e : A \mid \bigcup_i \phi_1 \cup \psi_i} \quad \text{(Ghost let)}$$

$$\frac{\forall i \,. \begin{cases} \Delta_i = \{x_1{:}A^i_1, \ldots, x_{(i-1)}{:}A^i_{(i-1)}\} \restriction \mathrm{FV}(e_i) \\ \mathcal{A}_i = \biguplus_j \mathrm{ran}(\Delta_j \restriction \{x_i\}) \\ \psi_i = \curlyvee(A_i \,|\, \mathcal{A}_i) \\ \Delta_i, \Gamma_i \vdash^{m_{(i-1)}}_{m_i} e_i : A_i \mid \phi_i \end{cases}}{\Gamma_1, \ldots, \Gamma_{k+1} \vdash^{m_0}_{m_{k+1}} \texttt{let } x_1 = e_1, \ldots, x_k = e_k \texttt{ in } e_{k+1} : A_{k+1} \mid \bigcup_i \phi_i \cup \psi_i} \quad \text{(Let)}$$

Exceptions

$$\frac{\Gamma \vdash^{m}_{m'+2} e : \mathit{Err} \mid \psi}{\Gamma \vdash^{m}_{m'} \texttt{raise } \mathit{exn}\ e : A \mid \psi} \quad \text{(Raise)}$$

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

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

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

Substructural rules

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

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

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

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

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

Here, $\curlyvee(A\,|A_1, A_2\,)$ is defined for all zero-order types to be the set of constraints generated by splitting type A into A_1 and A_2. It is defined only for those cases where A_1 and A_2 are identical to A other than renaming of the resource variables. The set of constraints comprises the constraint $a = a_1 + a_2$ for each resource variable a that is free in A with a_1, a_2 being their renamed counterparts in A_1 and A_2 respectively. If a certain data structure is used several times, then its assigned potential must be split among those several uses, so that the overall potential remains constant despite multiple uses of that data structure. For example, we have

$$\curlyvee\left(\mu Y.\{\texttt{Leaf}{:}a; \mathsf{int} \mid \texttt{Node}{:}b; \mathsf{int}, Y, Y\} \,\middle|\, \begin{matrix}\mu Y.\{\texttt{Leaf}{:}a_1; \mathsf{int} \mid \texttt{Node}{:}b_1; \mathsf{int}, Y, Y\},\\ \mu Y.\{\texttt{Leaf}{:}a_2; \mathsf{int} \mid \texttt{Node}{:}b_2; \mathsf{int}, Y, Y\}\end{matrix}\right) = \left\{ \begin{matrix} a = a_1 + a_2, \\ b = b_1 + b_2 \end{matrix} \right\}$$

Reconsidering the instantiated types from figure 3 again, by using rule Share we can exchange a single reference to an object of type $\mu Y.\{\texttt{Leaf}{:}4; \mathsf{int} \mid \texttt{Node}{:}4; \mathsf{int}, Y, Y\}$ for two handles to the very same object having types $\mu Y.\{\texttt{Leaf}{:}0; \mathsf{int} \mid \texttt{Node}{:}3; \mathsf{int}, Y, Y\}$ and $\mu Y.\{\texttt{Leaf}{:}4; \mathsf{int} \mid \texttt{Node}{:}1; \mathsf{int}, Y, Y\}$ respectively. We extend the sharing definition to function types in the following way by

$$\curlyvee\Big(\forall \alpha \in \psi.A_1, \ldots, A_k \xrightarrow[p']{p} C \,\Big|\, \forall \alpha \in \psi.A_1, \ldots, A_k \xrightarrow[p']{p} C\,,\ \forall \beta \in \phi.B_1, \ldots, B_k \xrightarrow[q']{q} D\Big) = \emptyset$$

if and only if all variables in β are fresh and there exists a substitution $\sigma{:}\mathsf{CV} \to \mathsf{CV}$ such that $\alpha = \sigma(\beta)$, $\psi = \sigma(\phi)$, $p = \sigma(q), p' = \sigma(q')$ $\forall i \,.\, A_i = \sigma(B_i)$ and $C = \sigma(D)$. In other words, such that the second type is derived as a renaming of all bound variables to fresh ones, leaving all free variables unchanged. It is straightforward to extend this definition to $\curlyvee(A \,|A_1, \ldots, A_k)$, e.g. for base types we have that for each resource variable a which occurs free in A we obtain the constraint $a = a_1 + \cdots + a_k$, with a_i being the renamed counterpart of a within A_i. While the subtyping of annotated types may seem at first to be slightly counter-intuitive, it is justified by the observation that it is always safe to drop potential. For example, if $A <: B$ then type A has more ($\geq$) potential than type B. Since the subtype relation will be used on annotated types containing resource variables, it is also extended to a partial mapping, which returns the constraints that are associated with those resource variables that are contained in the annotated types. The mapping $A <: B$ generates the constraints required to restrict the annotated type A to the annotated type B. It is derived from the previous mapping by $A <: B \approx \curlyvee(A \,|B, C)$ for a suitable type C containing fresh resource variables. However, rather than introducing fresh resource variables (which would be redundant ["slack"] in the corresponding linear program), we omit these and turn the equalities into inequalities, e.g.

$$\mu X.\{\texttt{Nil}{:}a \mid \texttt{Cons}{:}c; \mathsf{int}, X\} <: \mu X.\{\texttt{Nil}{:}b \mid \texttt{Cons}{:}d; \mathsf{int}, X\} \quad = \quad \{a \geq b, \qquad c \geq d\}$$

which is equivalent to

$$\curlyvee\left(\mu X.\{\texttt{Nil}{:}a \mid \texttt{Cons}{:}c; \mathsf{int}, X\} \,\middle|\, \begin{matrix} \mu X.\{\texttt{Nil}{:}b \mid \texttt{Cons}{:}d; \mathsf{int}, X\}\,, \\ \mu X.\{\texttt{Nil}{:}x \mid \texttt{Cons}{:}y; \mathsf{int}, X\} \end{matrix}\right) \quad = \quad \{a = b + x, \qquad c = d + y\}$$

if x and y are slack variables, i.e. do not occur anywhere else in the whole generated linear program. Note that this augmented type system will reject more terms as untypable when compared with the basic HUME type system. In general, we expect only terms to be typable whose resource consumption is linear in the size of their inputs, a restriction which seems acceptable in the setting of embedded software.

The definition of $\textsc{Size} : \mathsf{Con} \to \mathbb{Q}^+$ allows us to plug in different costs for certain constructors, if required. The default value for Heap-space consumption returned for a constructor with k arguments is $3 + k$. The only exception is for constructors representing the "none" value $*$ (used to indicate that no output is required on some wire). For these constructors, Size is defined to be 1, corresponding to the Heap-space cost specified in the formal HUME Semantics (Deliverable D12). This is ensured by the transformation to Core-Hume described in Section 2.1.1.

3.2 Annotated Type Rules for Pattern Matches

The annotated type rules dealing with pattern matches have the form

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

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.

For the Heap-space analysis, the annotated pattern match rules are extremely simple since pattern-matching in HUME never alters the Heap. However, pattern matching is significantly more complex in terms of both Stack-space usage and Worst-Case Execution Time, and we therefore include these rules both for completeness and for consistency with our other analyses.

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

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

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

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

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

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

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

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

$$\frac{\forall i\,.\,\left(B_i \vdash^{m_{i-1}}_{m_i}\ pat_i\ ?\triangleright\ \Gamma_i; \pi_i; \phi_i\right)}{\mu X.\{\cdots \mid c : q; B_1, \ldots, B_k \mid \cdots\} \vdash^{m_0}_{m_k}\ c\ pat_1 \cdots pat_k\ ?\triangleright\ \Gamma_1, \cdots, \Gamma_k; q + \Sigma_i \pi_i; \bigcup_i \phi_i} \quad \text{(Pattern Constr)}$$

Substructural Rules

$$\frac{A \vdash^{n}_{n'}\ pat\ ?\triangleright\ \Gamma; \pi; \phi}{A \vdash^{m}_{m'}\ pat\ ?\triangleright\ \Gamma; \pi; \phi \cup \{m \geq n, m - n \geq m' - n'\}} \quad \text{(Pattern Relax Heap)}$$

3.3 Annotated Type Rules for Boxes and Declarations

In addition to our definitions in section 2.2.3, we require that the resource variables assigned to the constructors of bundle types and the "None-constructors" of wire types are restricted to the value 0. Bundles cannot transmit any potential since they are an artifact of our simplification to core HUME. Wires may communicate potential between connected boxes, but it is obvious that no potential can be communicated if the wire does not hold an actual value. 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

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

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

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

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

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

c) For all pairs of boxes sharing a wire, the underlying unannotated type assigned to each wire is identical, but the resource variables are different and unique within the program. Since we are dealing with Heap-space, the potential transmitted through each wire is not required to match up, since the Heap memory of a box is reset completely after each execution of a box. This is specific to the Heap-space analysis only.

While we are already able to apply our analysis to several programs, we can improve the accuracy of the result that we obtain by introducing resource-polymorphism for the function definitions. In order to achieve this, we first need to identify all mutually-recursive components in the call graph first using standard techniques. Starting with the group of function definitions that do not depend on any other function definitions, we construct an enriched typing just for this block. We then gather the entire constraint-set in one set ψ and collect all occurring resource variables in a set α. The signatures for the functions contained in this block this block can then be updated by replacing $\Sigma(\mathit{fid}) = \big(e_f; [y_1, \ldots, y_k]; A_1, \ldots, A_k \xrightarrow{m}_{m'} C\big)$ with $\Sigma(\mathit{fid}) = \big(e_f; [y_1, \ldots, y_k]; \forall\alpha \in \psi. A_1, \ldots, A_k \xrightarrow{m}_{m'} C\big)$. If a function of this block is used more than once in one of the following blocks of function definition or eventually in the box definitions, then our definition of the sharing relation $\curlyvee(\cdot\,|\,\cdot)$ ensures that each copy receives a fresh set of resource variables for the bound resource variables in α. Each use of the function then inserts a copy of the required constraints, thereby permitting the resource polymorphism that was outlined in Section 3. We then repeat this process for the next block of mutually recursive functions, and so on. Note that resource polymorphism is unnecessary at the box level, since any wire can only be connected to at most two boxes. However, the use of resource polymorphism for box definitions might be sensible where box templates are used to aid the programmer defining a large group of mostly identical boxes.

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 heap consumption is given in terms of the following (rich) type of the main function:

```
ARTHUR3 typing for resource "Heap":
  0, (float<10>) -6/0-> float<0> ,0
```

This type indicates that the execution of the main function will require $10n + 6$ heap units, for an input value of n. 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 can now examine the costs of this program in more detail. Allocating the variables `?z_1` and `?z_2` requires 2 heap units in each case. This is due to the "boxed" representation of values in the heap, which means that for each heap-value a tag is used to indicate its type. In the meantime we have also produced an early prototype of an unboxed implementation of the Hume Abstract Machine (HAM). However, initially we model the cost-transparent unboxed version of the HAM, and leave the treatment of an optimised HAM, as well as the treatment of program optimisations, as further work.

In the then-branch of the conditional the remaining costs reduce to 2 heap units for the allocation of the constant 1.0 as the result value. In the recursion case (else-branch), again 2 heap units are required for `?z_3`. Subsequently, the subtraction operation on `n` will free as many resources as are attached to this variable (say p). 2 heap units are needed to record the results of the subtraction (`?z_4`) and the final multiplication as overall result. The function call requires 6 heap units (1 for the function, 1 for its argument and 1 for the result). Combining these subsidiary results, using max to statically model the costs of the two branches we obtain the following costs for the factorial function:

$$4 + \max(2, 2 - p + 2 + 2 + 6)$$

We obtain the best solution to this inequation in the case where both branches require exactly the same resources, i.e. for $p = 10$. This gives 6 for the base case, and 10 for each step in the recursion, arriving at the formula shown in the rich type above. It is interesting to observe that tighter resource bounds can be inferred for this very simple function if it is rewritten as:

```
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. In the absence of such sharing, a function `fac` with rich type `float<p> -x/y-> float<0>` has the following overall costs:

$$x \geq \max(2 - (0 * p) + y, 2 + 2 - (1 * p) + x - y + 2 + y)$$

and it is easy to see that $p = 6$ is a solution (in fact, all $p \geq 6$ are solutions, but we are looking for the smallest solution to compute the "best" resource bound).

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

```
ARTHUR3 typing for resource "Heap":
  0, (float<6>) -2/0-> float<0> ,0
```

i.e. for an input n, in total $6n + 2$ heap units are needed.

4.2 Example 2: 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 heap 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 "Heap":
  0, (flist[2;float<0>,#|0]) -2/0-> float<0> ,0
```

For each `Cons` node in the list, represented as `float<0>,#` in the type, 2 heap units are needed to perform the computation. This corresponds to the allocation of Heap locations to record the result of the recursive call and costs associated with the final addition operation. In addition, a further 2 heap units are required in the non-recursive case, as shown by the 2 in the function type `-2/0->`, to record the result of matching the function input. It follows that the heap consumption is $2n + 2$ for an input list of length n. Empirical results from the Hume Abstract Machine (HAM) interpreter with profiling support confirm the bounds derived by these types, up to a constant factor (identical for all programs), representing the constant costs associated with program setup, which we have not yet incorporated into the analysis.

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 (in total 140 inequalities in 153 variables are generated):

```
ARTHUR3 typing for resource "Heap":

Box: mult2
 ?v_8: wire1float[8;float<0>|*], ?v_9: wire1float[0;float<9>|*]
---0/0--->
 ?v_1: wire1float[0;float<0>|*], ?v_2: wire1float[0;float<0>|*], ?v_3: wire1float[0;float<9>|*]

Box: itermult
 ?v_1: wire1float[0;float<0>|*], ?v_2: wire1float[0;float<0>|*], ?v_3: wire1float[0;float<9>|*],
 ?v_4: wire1float[0;float<0>|*], ?v_5: wire1float[1;float<0>|*], ?v_6: wire1float[0;float<9>|*]
---8/0--->
 ?v_4: wire1float[0;float<0>|*], ?v_5: wire1float[1;float<0>|*], ?v_6: wire1float[0;float<9>|*],
 ?v_7: wire1float[0;float<0>|*]
```

Significantly, the `itermult` box requires heap that is linear in the size of the data that is carried on the `?v_3` and `?v_6` wires. These correspond to the bound variable `y`, over which the inter-box iteration is performed. In addition, the box requires constant space for initialising the values on the feedback-wires between the two boxes. 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 (in total there are 47 inequalities in 54 variables):

```
ARTHUR3 typing for resource "Heap":
Box: mult13
    ?v_1: wire1float[12;float<0>|*], ?v_3: wire1float[0;float<10>|*]
   ---0/0--->
    ?v_2: wire1float[0;float<0>|*], ?v_4: wire1float[0;float<0>|*]
```

As expected, the costs are linear in the second input to the box, `j`, over which the recursion is performed and constant over the other input wire: for an input `j` of size n the box needs $10n + 12$ units of heap 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 4. 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.

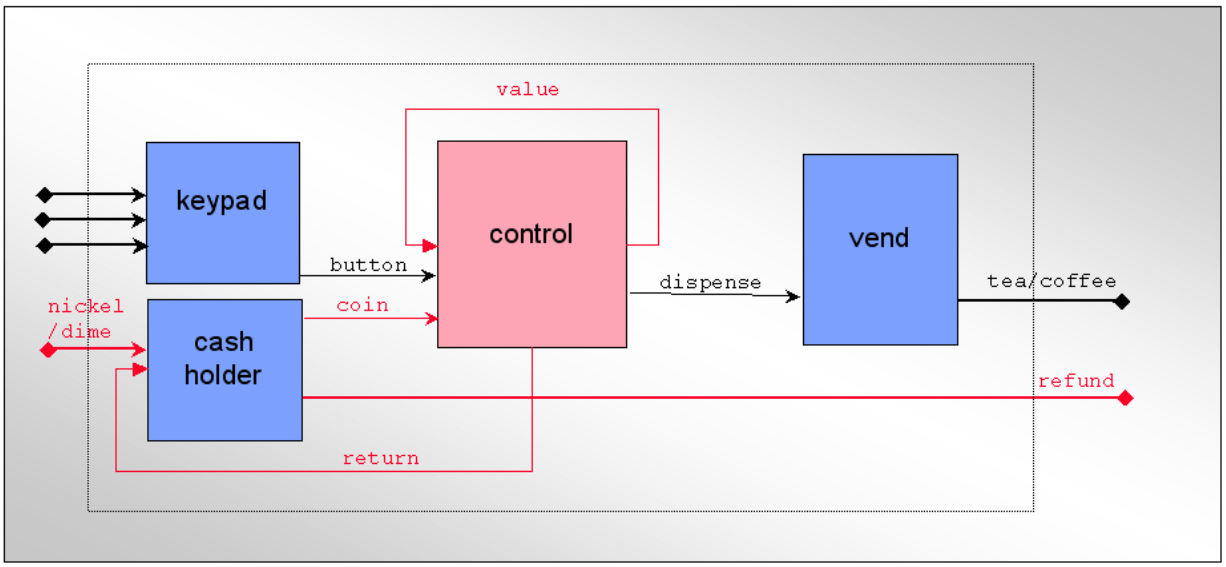


Figure 4: Hume example: vending machine box diagram

```
-- 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 (in total 512 inequalties in 398 variables have been generated; solving them takes about 0.16s on a 1.7GHz Pentium laptop, with 1GB main memory and 2MB cache):

```
ARTHUR3 typing for resource "Heap":
```

```
Box: inp
    ?v_6: wire1char[17;char|*]
   ---2/0--->
    ?v_2: wire1coins[8;coins[0|0]|*], ?v_1: wire1buttons[16;buttons[0|0|0]|*]

Box: coffee
    ?v_2: wire1coins[8;coins[0|0]|*], ?v_1: wire1buttons[16;buttons[0|0|0]|*],
    ?v_5: wire1float[0;float<0>|*]
   ---0/0--->
    ?v_3: wire1drinks[5;drinks[0|0]|*], ?v_5: wire1float[0;float<0>|*],
    ?v_4: wire1float[9;float<0>|*]

Box: outp
    ?v_3: wire1drinks[5;drinks[0|0]|*], ?v_4: wire1float[9;float<0>|*]
   ---0/0--->
    ?v_7: wire1int[0;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 "Heap":
  0, (float<0>,float<2>,flist[0|16;float<0>,#]) -14/12-> flist[0|0;float<0>,#] ,0
```

It follows the heap consumption for `mapit` is $16n + 2m + 14$ for an input list of length n and a value m for the variable `si`. This example demonstrates that we can capture linear bounds, even if they involve several function parameters. We also see that 12 heap units become free at the end of the evaluation. While this is less relevant on the box-level, where the heap 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 heap space analysis for HUME, built on a generic framework that will also allow the construction of analyses of stack and time costs (deliverables D5 and D14). The prototype 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 3.1. 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.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, March 2006.

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