

Hume

Jorge Coelho, Luís Miguel Pinho

Polytechnic Institute of Porto

Portugal

Introduction

Hume (Higher-order Unified Meta-Environment) [1] is an academic project developed at St. Andrews University and Herriot-Watt University that aims to build a strongly typed, functionally-based language and associated tools for developing resource-limited systems. It is known that properties such as program equivalence, termination and time and space use are undecidable for Turing complete languages. Also, languages in which these properties are decidable namely, finite state machines, lack expressiveness. With Hume, developers want to achieve a high level of expressiveness and provide strong guarantees for properties like execution time and space usage. This is achieved by a separation between coordination and expression aspects of the language. Coordination, which describes a finite number of communication processes, can always be described by a finite state machine. On the other hand, the expression aspects of the language are described in several sub layers that represent a tradeoff between expressiveness and ease of costing and should be used in a balanced way between the need for proofs and the need for the language capabilities.

Description

Hume programs consist of concurrent "boxes" describing computations. Boxes describe an abstract notion of process and specify the links between its input and output. These links are called "wires". A box reads inputs and produces outputs simultaneously. Each box has a buffer to hold the output until the respective consumer frees it. Boxes are scheduled in a round-robin fashion, following the order of declarations. A box is scheduled only when all inputs are available and executes as an atomic non preemptable task to produce output values. This approach guarantees a high degree of abstraction which is an important property to enable program analysis and manipulation. The language is divided in two parts, the Coordination Layer and the Expression layer.

Coordination Layer

The coordination layer is designed to have statically provable properties. It is a finite state notation for the description of multiple interacting processes built from expressions defined in the expression layer. Due to its nature it is easy to verify properties at this level such as the absence of deadlock, livelock or resource starvation. Boxes describe an abstract notion of process and specify the links between its input and output channels in terms of functional patterns and expressions. It is possible to connect the output of a box

to one of its own inputs and this can be used to implement state variables as one typically finds in imperative languages. Let's present a simple example:

```
stream output to "std_out";
box adder
  in (x :: int 32)
  out (y :: int 32, shown :: (int 32, char))
match
  z -> (z+1, (z, '\n'));
wire adder
  (adder.y initially 0)
  (adder.x, output);
```

Here we define the box "adder" which accepts one integer and increments it. The input and output type are declared following "in" and "out" respectively. The box works by matching the input with the options below "match" and evaluating the expression on the right of the -> sign of the matched line which is the result. The box is wired in such a way that the output is connected to the input and it is initially 0. The result is also sent to the standard output and thus, running this program will produce integers starting with 0 in the standard output. Boxes also provide exception handling facilities including timeouts and system exceptions.

Expression Layer

The Expression layer defines different levels of expressiveness of the language. These are described by:

- HW-Hume: a hardware description language capable of describing both synchronous and asynchronous hardware circuits, with pattern matching on tuples of bits, but with no other data types or operations;
- FSM-Hume: a hardware/software language HW-Hume plus first-order functions, conditionals expressions and local definitions;
- Template-Hume: a language for template-based programming FSM-Hume plus predefined higher-order functions, polymorphism and inductive data structures, but no user-defined higher-order functions or recursive function definitions;
- PR-Hume: a language with decidable termination Template-Hume plus user-defined primitive recursive and higher-order functions, and inductive data structure definitions;
- Full-Hume: a Turing-complete language PR-Hume plus unrestricted recursion in both functions and data structures.

Going up in this layered approach to the language results in reducing the properties that can be proved about the program. A detailed description of each of these layers can be found in [4]. It is possible to perform static analysis of both expression and coordination aspects. Boxes can be proved to be deterministic, terminating and bounded in space and time, giving these bounds automatically. Wiring can be checked for process equivalence and safety. Worst-case resource usage can be computed before programs are run [3]. These properties are not guaranteed for Full-Hume since it is Turing-complete. The idea is that the language is restricted by using only a subset where for example recursion is forbidden or alternatively employ static analysis. Proof exists that FSM-Hume is finite state [6] and thus if programs are restricted to this subset it is possible to perform time

and space analysis. Some tools for time and space analysis are under development. One example is Hume Amortized Resource Analysis which has an online version [2] for the proof of the concept. Further information about cost analysis for Hume can be found in [5].

Concurrency

The concurrency model of Hume uses microthreads to process each iteration of a box. Communication between threads is asynchronous by means of buffers. A microthread is scheduled if the inputs of a box become available, or if it has been scheduled to run, and then runs to completion. The inputs are matched against the rule patterns, matching rules are executed and the results are placed on the output buffers. Blocking may occur if outputs from previous executions have not yet been consumed. If not the microthread completes the current execution. There is no support to control the time related behavior of applications.

Exceptions

Exceptions are supported both at system and process level, including the ability to set timeouts for expression computations. Exception handlers can be declared for a box and will deal with exceptions raised by the evaluation of the right-hand side of the box rules. We could add a handler to the former example in order to take some action in case we get an overflow:

```
stream output to "std_out";

box adder
in (x :: int 32)
out (y :: int 32, shown :: (int 32, char))
match
z -> (z+1, (z, '\n'));
handler
  Overflow -> *
wire adder
  (adder.y initially 0)
  (adder.x, output);
```

Here, whenever it is not possible to add due to overflow the defined handler deals with this situation ignoring the error (* - represents a null value). Hume defines handlers for taking actions whenever space (StackOverflow and HeapOverflow) or time (Timeout) is exceeded. One strong limitation is that handlers cannot perform any computation which means, it is not possible to have dynamic expressions on the right hand side of the handler.

Conclusions and Future

Hume is still in a very early stage of its development. There are some preliminary versions of compilers and interpreters available from the Hume webpage [1] which don't implement all the features defined in the Hume report, namely exceptions and timeouts.

Although targeted for real-time embedded systems, there is no support to real-time specific scheduling policies, apart the only policy implemented (round-robin).

Furthermore, time related control (such as simple periods) is not considered. This is the major drawback of the language. It is not clear, how different policies could be implemented without impairing the language's features. The language could also be improved by providing a strong support for exception handling (not only the defined static exceptions).

Several different groups are working in different aspects of the language and thus implementing their own core of features for testing purposes. Hume would benefit if a single path of development was defined and avoiding the creation of different branches where work is replicated and different incompatible versions can arise.

Links and References

- [1] The hume webpage. <http://www-fp.cs.st-andrews.ac.uk/hume/index.shtml>.
- [2] Hume amortised resource analysis. <http://www-fp.cs.st-andrews.ac.uk/embounded/software/cost/cost.cgi>.
- [3] Armelle Bonenfant, Christian Ferdinand, Kevin Hammond, and Reinhold Heckmann. Worst-case execution times for a purely functional language. In *Proc. Implementation of Functional Languages (IFL 2006)*, volume 4449 of *Lecture Notes in Computer Science*. Springer, 2007. To appear.
- [4] Kevin Hammond, Greg Michaelson, and Pedro Vasconcelos. Bounded space programming using finite state machines and recursive functions: the hume approach. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 2006. Submitted.
- [5] Steffen Jost and Kevin Hammond. Validation of the prototype worst-case execution time (wcet) analysis. Technical report, School of Computer Science, Univ. of St Andrews, Scotland, 2007.
- [6] Greg Michaelson, Kevin Hammond, and Jocelyn Sérot. FSM-Hume is finite state. In Stephen Gilmore, editor, *Trends in Functional Programming, Volume 4*, volume 4 of *Trends in Functional Programming*, pages 19-28. Intellect, 2005.