Seminar Report

Hume

David Oberhollenzer
David.Oberhollenzer@student.uibk.ac.at

31 July 2015

Supervisor: Georg Moser

Abstract

Hume is a functional programming language designed for applications with hard constraints on memory usage and execution time. Hume is specifically aimed at easier, automated verification of those properties. This report outlines the syntax and semantics of Hume, how automated verification of time and memory limits are achieved and compares Hume with other programming languages aimed at subsets of the problems Hume addresses.

Contents

1 Introduction

For real-time and safety-critical systems, it is vital to have guarantees [1] on execution time and resource usage. Many attempts have been made in the past at developing systems for automatic verification of those properties that analyze binaries generated by high level language compilers. Typically, imperative languages have to be largely restricted to a non turing-complete subset, with even further restrictions on loop and branching constructs, to allow automatic verification.[4]

Hume (Higher-order Unified Meta-Environment) is a strongly typed, functional language, named after the Scottish philosopher David Hume (1711-1776). Hume tries to address those issues by language design and specifically developed tools for automatic program verification. Specifically, the language is split in carefully designed layers in an attempt to make the automatically provable, non-turing-complete subset of the language as big as possible.[1]

In contrast to traditional, imperative languages that treat program execution as sequential processing of instructions, functional languages are based on lambda calculus, combinatory logic and term rewriting. Being more abstract and closer to mathematics simplifies program verification for functional programs in contrast to imperative languages.[3]

2 History

The design of the Hume language is governed by the *„Hume Report"*. The first version, version 0.1, was released in November 2000, specifying the first draft of the language and the support tools that would have to be developed.

The core language definition was finalized in November 2006, along with the reference implementation released in the same year.

3 Memory and Time Constraints

In the past, multiple approaches have been made at estimating the worst case execution time (WCET) of programs written in imperative programming languages by analyzing the compiled binaries of the programs. Typically those approaches require that the program being analyzed is known to halt and impose restrictions on loop and branch constructs, as well as limits on recursion. An automated tool then constructs control flow graphs from the binary and estimates the WCET from the longest possible path.[4]

While those approaches are rather generic regarding the programming languages that can be analyzed, those approaches have two main drawbacks. First of all, the programming language used has to be largely restricted in expressiveness[1][2]. Second, processors of modern, non-microcontroler platforms have complex mechanisms such as multiple levels of memory caches, deep pipelines, branch predication and run-time

[1] preferable formerly proven

instruction reordering that on the one hand increase average case execution time but make WCET estimation very complex. Despite analyzing the processor dependent binaries, the actual execution time largely depends on the hardware details and all possible states that the processor could have been in prior execution of the code[4].

A notable result of this research is the *aiT* tool, developed by AbsInt GmbH that is used in industry for WCET analysis of real-time applications. The tool uses processor specific estimation models and research is currently focusing on automatically generating such models from VHDL definitions of processors.[4]

The approach taken by the Hume Research Program is to develop a programming language (Hume) that is specifically designed for easy verification and static analysis of worst case execution time and memory consumption, as well as to develop the necessary tools for analyzing Hume programs. For easy analysis, Hume is composed of multiple layers, where only the top most layer completes the language to a turing-complete programming language. For the layers below termination is decidable and static analysis can be performed easily, without restricting the language too much.[1]

However, when working out the WCET for Hume programs run on actual hardware (as opposed to the virtual machine developed for the reference implementation), the same problems as described before arise. Actual WCET estimates of Hume programs on hardware have so far been performed on the Power PC G4 and Renesas M32C microcontroller platforms, using the *aiT* tool.[1][2]

When using the full, turing-complete version of Hume, termination is no longer decidable and automated tools can no longer guarantee worst case execution time and memory consumption. To mitigate that problem, Hume supports annotating expressions with absolute time or memory limits that are checked at run time. If evaluation of an expression exceeds its limits, the run time environment interrupts executions and jumps into a user specified trap handler.[1]

3.1 Language Layers

The Hume language is divided into five layers, each layer extending the previous one, up to the topmost layer that makes the language turing-complete.

- **HW-Hume** The bottom most layer. This layer only allows pattern matching on tuples of bits, as well as describing circuit wiring of synchronous or asynchronous hardware circuits using wiring rules.
- **FSM-Hume** This layer adds fist-order functions, conditional expressions and local definitions to HW-Hume.
- **Template-Hume** This layer adds pre-defined (no user-defined) higher-order functions, polymorphism and inductive data structures to FSM-Hume.
- **PR-Hume** This layer adds user-defined primitive recursive and higher-order functions, as well as inductive data structure definitions to Template-Hume. However, termination is still decidable for programs using only this layer of Hume.

- **Full Hume** This layer adds unrestricted recursion for data types and functions to PR-Hume in order to extend it to a turing complete language.

4 Language Overview

This section roughly outlines the structure of the Hume programming language. It is not intended as a complete and exhaustive definition of the Hume language. For that purpose, the interested reader may referrer to the *Hume Report*.

4.1 Basic Data Types

Hume does not support pointer data types and has no notion of undefined values. In addition to the data types listed below, Hume has the types *bit* which is a synonym for *word 1* and *byte* which is a synonym for *word 8*.

bool	boolean value (constants true or false)
char	8 bit Latin-1 character
word ⟨size⟩	a word that is the specified number of bits wide
nat ⟨size⟩	a natural number (unsigned) that is the specified number of bits wide
int ⟨size⟩	a twos complement integer number that is the specified number of bits wide
float ⟨size⟩	a floating pointer number of specified bit size
string [⟨size⟩]	a string of characters with optionally specifiable size

For all basic types, Hume supports the comparison operators $<, <=, ==, !=, >=$ and $>$ that result in boolean values. There boolean data type supports C style && (logic and) and || (logic or) operators, as well as a textual „not" operator for logic inversion.

There are a number of textual bit shift, rotate, set and test operators for the *word* data type.

The *word*, *int*, *nat* and *float* types support $+ - * divmod$ operators, as well as the $**$ operator for powers. The integer types are types are specified to use two's complement arithmetic. The *int*, *word* and *float* types support unary, inversion, whereas while the *nat* type does not, as it is by definition unsigned. For the *float* type, Hume defines a number of trigonometric functions, logarithms and square root as operators.

The *string* data type supports the additional operators @ (element selection by index), ++ (concatenation) and *length*.

4.2 Structured Data Types

In addition to the basic data types, Hume has built in support for *vectors* (fixed length sequence of the same type), *tuples* (fixed length sequence of different types), *lists* (variable length sequence of the same type) and *discriminated unions*.

Similar to strings, *vectors* and *lists* support the operators @, ++ and *length*. The vector type supports additional operators for mapping, folding and constructing vectors,

whereas *lists* support the *hd* and *tl* operator for splitting lists into head and tail, as well as a colon list constructor as opposed to a keyword in the case of vectors.

The Hume language allows pattern matching on tuples, lists and discriminated unions.

4.3 Syntactical Overview

The syntax of Hume largely borrows from Haskell[1]. An interesting deviation that stems from the fact that the lowest layer of the language is designed as a hardware description language, is what Hume referrers to as the *box* construct.

The box is the top most construct of a Hume program. It has a name, a set of inputs, a set of outputs, local variable definitions and optional constraints on execution time and memory limits evaluated at run time.

The body of a box consists of pattern matching constructs and wiring rules (how to connect inputs with expressions or outputs and expression results with outputs). Furthermore, the inputs and outputs of boxes can be connected to *instances* of other boxes, *instances* of the box itself (recursion) or possible I/O devices external to the program.

The lexical grammar of boxes is outlined below:

```
<boxdecl> ::= <prelude> <body>

<prelude> ::= "box" <boxid> "in" <inoutlist> "out" <inoutlist>
              [ "within" <constraint> ] [ "handles" <exnidlist> ]

<inoutlist> ::= "(" <inout1> "," ... "," <inoutn> ")"

<inout> ::= <varids> "::" <type> [ "timeout" <cexpr> ]

<varids> ::= <varid1> "," ... "," <varidn>
```

The lexical grammar of wiring rules is outlined below:

```
<wiredecl> ::= "template" <templateid> <prelude> <body>

<wiringdecl> ::=   "wire" <boxid> <sources> <dests>
                 | "wire" <link> "to" <link>
                 | "replicate" <boxid> "as" <boxid> [ "*" <natconst> ]
                 | "instantiate" <templateid> "as"
                    <boxid> [ "*" <natconst> ]

<sources>/<dests> ::= "(" <link1> "," ... "," <linkn> ")"
```

```
<link> ::= <connection> | <strid> | <portid>

<connection> ::= <boxid> "." <varid>
```

Notably, the keyword *within* allows the specifications of timeouts on operations.

5 Examples

5.1 99 Bottles of Beer

The example program below illustrates a possible implementation of the 99 bottles of beer song:

```
box bottles
in (n::int 64)
out (n'::int 64,v::string)
match
0 -> (99,("No more bottles of beer on the wall, ",
          "no more bottles of beer.\n",
          "Go to the store and buy some more, ",
          "99 bottles of beer on the wall.\n") as string) |
n -> (n-1,(n," bottles of beer on the wall, ",
           n," bottles of beer.\n",
           "Take one down and pass it around, ",
           n-1,"bottles of beer on the wall.\n") as string);

wire bottles (bottles.n' initially 99) (bottles.n,output);
```

The box „bottles" is defined was having a 64 bit integer input *n*, a 64 bit integer output *n'* and a string output, *v*.

The body of the box consists of a simple pattern matching rule, mapping an input of 0 to a tuple containing the number 99 and the final phrase of the song, and mapping any other non-zero number to a tuple of the decremented number and the non-final phrase of the song.

Finally, the output of the box is recursively *wired* to itself, with an initial beer bottle count of 99 to generate the song.

5.2 Using Time Constraints

The example below illustrates how execution time constraints can be modeled using the keyword *within*:

```
box b
in (v::int 32)
out (v'::int 32)
match
x -> very_complex_function x within 20ns
handle Timeout -> 0;
```

In this example, the input of a box is mapped to the result of a function called „very_complex_function" that is expected to deliver a result *within* a maximum of 20 nano seconds. Should the evaluation of the expression at run time take longer than the given timeout value, execution is aborted and the **Timeout** handler is called. In the last line, the *handle* keyword is used to register a timeout handler that (in this examples) evaluates to constant zero.

6 Comparison to other languages

6.1 Hume and OCaml

While Hume and OCaml are both functional programming languages, OCaml has not been designed with robustness or automatically proof able run time costs in mind. In contrast to Hume, OCaml also does not have hardware design features, like the box or wiring constructs in Hume. Other than that, the expression syntax of OCaml[2] and Hume are very similar.

6.2 Hume and C

The C programming language is designed for low level systems programming. Due to its low level nature, C is theoretically also suitable for development of fast, memory conserving applications.

In contrast to Hume, C is an imperative programming language. Program execution in C is modeled directly after program execution on hardware, executing a sequence of imperative statements that perform calculations or explicitly access memory. Arbitrary jumps can be performed within statement sequences. Statement sequence are grouped in „functions" that can be explicitly called from expressions.

In contrast to Hume, the C programming language has a weaker type system and permits direct access and manipulation of memory contents, including type unsafe pointer casts and access. Also in contrast to *many* functional programming languages, functions in

[2] as taught at the University Of Innsbruck

C represent a finite block of executable code and cannot be copied, modified, instantiated or computed and returned from functions[3].

While it is possible to write C programs that meet hard memory or execution time limits, those properties are much harder to prove for C than for a functional language. The closest to automated run-time verification for C programs are the static binary analysis approaches discussed in section 3.

6.3 Hume and VHDL

VHDL (VHSIC Hardware Description Language) is a strongly typed, parallel programming language developed for the design of integrated circuits.

Similar to VHDL, the lowest layer of Hume is also intended for hardware design. The box constructs used by Hume are very similar to the entity and architecture constructs in VHDL. An entity in VHDL defines input and output interfaces of component and an architecture defines its behavior. The actual „program" code is written inside the body of an architecture. Also, Hume shares many of its shift, rotate and bit manipulation operators with VHDL.

To illustrate some of that, a listing of a 4 bit parallel adder, defined in VHDL is provided:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder is port ( clk  : in std_logic;
                       a    : in std_logic_vector(3 downto 0);
                       b    : in std_logic_vector(3 downto 0);
                       q    : out std_logic_vector(3 downto 0) );
end adder;

architecture adder_arch of adder is
    signal q_s : signed(3 downto 0);
    q <= std_logic_vector(q_s);

    adding_proc:
    process (clk)
    begin
        if rising_edge(clk) then
            q_s <= ('0' & signed(a)) + ('0' & signed(b));
        end if;
    end process;
end adder_arch;
```

[3]However, C does support pointers to functions

It can be seen that VHDL as a hardware description language is much more verbose than HW-Hume. The equivalent of the *wire* keyword in Hume is the left double arrow operator in VHDL.

A major drawback of VHDL in contrast to Hume (apart from it's much more verbose syntax) is the context dependent meaning of operators in VHDL. For instance the left arrow operator seen in the example **sometimes** causes synchronization depending on whether it is used inside an *if* block or a *case-esac* pattern matching block[4]. An other example visible in the sample code is the use of the & operator for both concatenation, bit wise *and* and.

In contrast to HW-Hume and FSM-Hume, VHDL itself is already a turing-complete language and was not split into layers with decidable properties, which gives rise to a number of problems, specifically that it is very easy to design constructs that can no longer be synthesized to hardware, as well as making formal verification just as problematic as in traditional, imperative languages, as outlined in section 3.

In contrast to Hume, however, VHDL is standardized and supported by the tool chains of most FPGA[5] vendors, as well as by a large number of ASIC[6] vendors, whereas at the time of writing, there is no known FPGA tool chain that supports Hume.

7 Conclusion

An overview over the Hume programming language was presented, as well as the motivation for developing Hume, namely automated formal verification of worst case execution time and memory consumption of programs developed for safety critical, real-time applications. Furthermore, the verification methods of Hume were compared to existing tools and research for WCET estimation of programs written in traditional, imperative languages and finally, Hume was compared to languages with similar aspects or backgrounds.

In conclusion, despite its automated formal verification tools and properties, in the near future, Hume is unlikely to set foot in the embedded world that it specifically is marketed for. Mainly because large memory footprint of the reference implementation (circa 15 megabyte at the time of writing) in comparison to C that runs natively on a broad variety of platforms, is standardized and has next to no overhead.

However, it would be interesting to see the hardware definition subset of the Hume language being supported by FPGA tool chains, as it provides a much simpler and cleaner alternative to the two dominating languages, VHDL and Verilog. Due to the proprietary nature of FPGAs and ASIC design, this is also unlikely to happen any time soon.

[4]Pattern matching in VHDL uses Algol style case-esac syntax
[5]Field Programmable Gate Array
[6]Application Specific Integrated Circuits

References

[1] Kevin Hammond Greg Michaelson and Robert Pointon. The hume report, version 1.1, March 2007.

[2] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann 0001. "carbon credits" for resource-bounded computations using amortised analysis. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 354–369. Springer, 2009.

[3] Christian Sternagel and Harald Zankl. Functional programming (in ocaml), 5th edition, September 2012.

[4] Reinhard Wilhelm and Daniel Grund. Computation takes time, but how much? *Commun. ACM*, 57(2):94–103, February 2014.