The Hume Programing Language

David Oberhollenzer

Supervisor: Prof. Dr. Georg Moser

June 19, 2015

Overview

- Named after scottish philosopher David Hume (1711-1776)
- *"Higher-order Unified Meta-Environment"*
- Functional
- Strongly typed
- Designed for resource limited systems (e.g. embedded)
- Designed for programs that must meet *correctness* and *performance* criteria (e.g. safety critical, hard realtime)

Functional Programing Languages

- Based on λ-Calculus
- Computation treated as evaluation of functions
- Avoids state changes and mutable data
- Examples: Haskell, OCaml
- Refere to course *"Functional Programing"* at UIBK

Strong Typing

- Data has distinct type
- Data can only be copied to memory region of compatible type
- No implicit type conversion
- No pointer arithmetic (could be used to bypass type system)

History

- Version 0.1 of the *"Hume Report"* released in November 2000
- Core definition finalized and reference implementation available since November 2006
- Currently 4 known implementations

History

- Reference interpreter, Heriot-Watt University
- Hume to C compiler, Heriot-Watt University
- *"hic"* interpreter, LASMEA (french research institute)
- Abstract machine interpreter, St Andrews University

Language Overview

- Data types have to be explicitly specified
- Functional language allows easier proofs for termination, correctness, run-time and memory consumptions
- Expressions can have *predefined limits* for *hard realtime* and memory consumption

5 Levels of Hume

- HW-Hume
  - Pattern matching on tuples of bits
  - Describes circuit wirings
- FSM-Hume
  - Adds first-order (non-recursive) functions to HW-Hume
  - Adds conditional expressions and local definitions
- Template-Hume
  - Adds *predefined* higher-order functions to FSM-Hume
  - Allows polymorphism and inductive data structures
- PR-Hume
  - Adds *user defined* higher order functions to Template-Hume
  - Allows primitive recursive
  - Termination is still decidable
- Full-Hume (Turing-complete language)
  - Adds unrestricted recursion to PR-Hume

Basic Data Types in Hume

- **bool** - boolean value
- **char** - 8 bit Latin-1 character
- **word** - data word with arbitrary number of bits
- **int** - 2's complement integer with arbitrary number of bits
- **nat** - Natural number (unsigned integer) with arbitrary number of bits
- **float** - Floating point number with specifyable bit size
- **string** - String of characters

Structued Data Types in Hume

- **vector** - Fixed length, same data type (similar to C array)
- **tuple** - Fixed length, different types
- **list** - Variable length, same data type
- **Discriminated union** - Can hold a single value of different types

Operators

Operator	Description	Allowed types
div mod	integer divison, modulo	int, nat, workd
+ - *	addition, subtraction, product	float, int, nat, word
unary -	flip sign	float, int, word
**	power	float, int, nat, word
< <= == >= > !=	comparison	all basic and extened types
@ ++ length	select, concatenate, length	string and vectors
&& || not	logical and, or invert	bool
lshl lshr ashl ashr rotl rotr bittest bitclr bitset ^& ^| ^~	bitwise operations	word

Operators

Operator	Description	Allowed types
/	floatingpoint division	float
sin cos tan asin acos atan	trigonometric functions	
sinh cosh tanh atan2		
log log10 ln exp	logarithm and exponentiation	
sqrt	square root	

Code Example - 99 bottles of beer

```
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);
```

Syntactical Overview

- Toplevel construct is *box*
- Has input and output variables
- Body consists of pattern matching constructs
- Can specify timeout to allow constraints on run time
- Can specify handlers for system exceptions (e.g. division by 0, timeout)
- Can contain "wiring" rules
  - Connect box I/O with expressions
  - Connect box I/O with *instantiations* of other boxes
  - Connect box I/O with *instantiations* of itself (recursion)
- Box I/O can be conneted to special streams to access I/O devices

Syntactical Overview

- Type and variable declarations similar to other FP languages
- Example: data Tree a = Leaf a | Node (Tree a) (Tree a);
- Example: type IntTree = Tree (int 32);
- Expression syntax similar to Stanford ML and Haskell

Hume vs OCaml

- Both are functional
- Both can be compiled
- No box construct or "wiring"
- No explcitly typed variables
- Not tuned for hard realtime & memory limits

Hume vs C

- C is an imperative language
- Very low level
- Easy to reach performance and memory goals
- Hard to prove
- Not intended for hardware design
- Hume has more low-level data types and bit fiddling operators
- Hume has specialized operators, C operator behaviour depends on type

Hume vs VHDL

- Intended for hardware design
- Basded on Ada
- Mostly imperative
- Also contains typical features from functional languages
- Functional language seems much better match for HW design

Literature

- The Hume Report, Version 1.1, Kevin Hammond, Greg Michaelson, Robert Pointon, March 2007
- Functional Programming (in OCaml), 5th Edition, Christian Sternagel and Harald Zankl, September 2012
- "Functional programming", English Wikipedia, Version of June 5th 2015
- "Strong and weak typing", English Wikipedia, Version of May 8th 2015

Questions?