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

Specific Targeted Research Project (STReP)
FET Open

D24 (WP7c): Hume Machine Code Compiler

Due date of deliverable: July 2007
Actual submission date: January 2008

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: Heriot-Watt University Revision: 1.10

Purpose: The purpose of this deliverable is to provide a stable, complete and reliable machine code compiler for the Hume language, conforming to the language specification and formal semantics and being compatible with the resource analysis being developed elsewhere in the EmBounded project. The compiler starts from provably correct HAM code generated by the Hume to HAM translator presented in Deliverable D23 (WP7c) [PS08] and will be integrated with the compiler analyses described in Deliverable D25 (WP7c).

Results: The main result of this deliverable is a working Hume to object code compiler incorporating the Hume to HAM translator and the compiler analyses. These are all supplied on a single Hume compiler installation disc.

Conclusion: The machine code compiler has been written and thoroughly tested as part of the EmBounded project. It has been used to verify the results of the resource analysis and has also been applied in the development of many applications programs.

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)	

Hume Machine Code Compiler

Robert F. Pointon <rpointon@macs.hw.ac.uk>
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland

Norman Scaife <Norman.Scaife@lasmea.univ-bpclermont.fr>
Laboratoire LASMEA, Blaise Pascal University

Abstract

This deliverable describes how the machine code compiler has been developed to meet the EmBounded project criteria. The project objectives are stated and brief explanations of how these objectives were met are presented. The positioning of this deliverable is discussed in Section 5. The evidence supplied in support of these statements is listed. Finally, we cite academic papers related to this work.

Major Revisions		
Revision	Date	Changes
1.9	14 Aug. 2009	table-of-contents (addressing Review Report Year 4)
1.7	26 May 2008	appendix A on ham-to-C translation (addressing Review Report Year 3)
1.6	6 Feb. 2008	initial version

Contents

1 Introduction

This deliverable, D24, details the Hume machine code compiler, its relationship to the Hume language development environment and its relationship with the Hume to HAM translator documented in Deliverable D23 [PS08] and with the compiler analyses explained in Deliverable D25.

The compiler plays a central role in this environment since a robust, complete, reliable and usable implementation is needed by the development of the analysis tools, both resource analysis and other analyses such as verification, and by the numerous applications being developed as part of EmBounded and elsewhere.

Prior to the EmBounded project there were various experimental (interpreted) implementations of the Hume programming language and we used these as the starting point for the current compiler.

2 Objectives

This work is towards achieving part of the EmBounded project overall objective 6: *development of underpinning specification, implementation and support environment for the Hume language.*

This means maintaining consistency between the various components that make up the Hume language; specifications, analyses and implementations. It also means developing robust tools for program development and for runtime monitoring and analysis.

The machine code compiler is an essential part of this Hume development environment. It must conform to:

- the formal semantics for the Hume language, Deliverable D12 (WP7a) and at least one target architecture must be implemented (actually the Renesas M32C) to allow proper testing of the compiler.
- the subset of the Hume language currently implemented by the resource analysis (Deliverables D05 [JLH07b], D11 [JLH07a], and D16 [JH07]) as closely as possible but may go further in implementing advanced features which the analysis cannot yet handle.

Our objectives are therefore:

1. Update the existing experimental compiler to cover the complete language definition plus extensions required by the compilation process and by the resource analysis.
2. Provide a reliable and usable implementation of the Hume language in the form of a compiler to machine code.
3. Improve on the execution times for the existing experimental Hume implementations.
4. Provide features essential to significant embedded and real-time programming applications examples.
5. Target a hardware platform suitable for testing Hume in an embedded environment.
6. Provide peripheral support for Hume programming in an embedded environment where the resulting code has to interface with other real-time code and systems.
7. Make the resulting implementation compatible with the resource analysis tools also being developed under EmBounded.

3 Meeting the objectives

To meet these objectives, we started from the existing prototype Hume intepreter and developed tools for translating Hume abstract machine (HAM) code into C and then provided the necessary compile-time and runtime support for our target architecture. To provide a more complete development environment, we also developed various support libraries and the foreign function interface.

In summary:

1. We have developed all the Hume compiler components to cover the full range of Hume constructs.
2. The compiler and associated tools have been thoroughly tested against a test suite of example programs and in the field by applications developers leading to a solid and dependable implementation of the machine code compiler.
3. Formally specified high-level semantics-preserving compiler optimisations have been incorporated into the compiler chain leading to highly competitive performance for a compiler of this type.
4. We provide support for interrupt handling for all of our supported architectures. Similarly, the initial round-robin scheduler has been rewritten to use a super-step scheduler which gives better performance and better conformance to the Hume semantics.
5. We have implemented a machine code generator for the Renesas M32C embedded controller. This includes support for all on-board features such as ADC/DAC and interrupts.
6. A substantial, but not necessarily complete implementation of basic support libraries is provided. These include support for common data structures such as lists and vectors as well as more low-level support for native types such as different integer sizes and floating point operations. The compiler is also provided with a full foreign function interface to the C language.
7. The compiler itself implements a range of analyses which helped in developing the compiler itself and can be used by application developers in debugging and analysis work. The compiler is also compatible with the more restricted subset of Hume implemented as part of the resource analysis. Some minor differences in syntax and type-checking remain to be ironed out but the core semantics are the same.

4 Architectures

Hume program development took place on multiple architectures:

- x86 – desktop development using the GCC compiler under linux and Mac OSX.
- PPC – desktop development using the GCC compiler under Mac OSX.
- M32C – embedded development using the IAR compiler tool suite.

The common use of x86 and PPC on the desktop enabled rapid program development and from a compilation viewpoint allowed us to verify that there were no endian issues in the generated code. The use of the Renesas M32C/85 provided testing against a commercial compiler and the development of I/O libraries specific to the embedded environment.

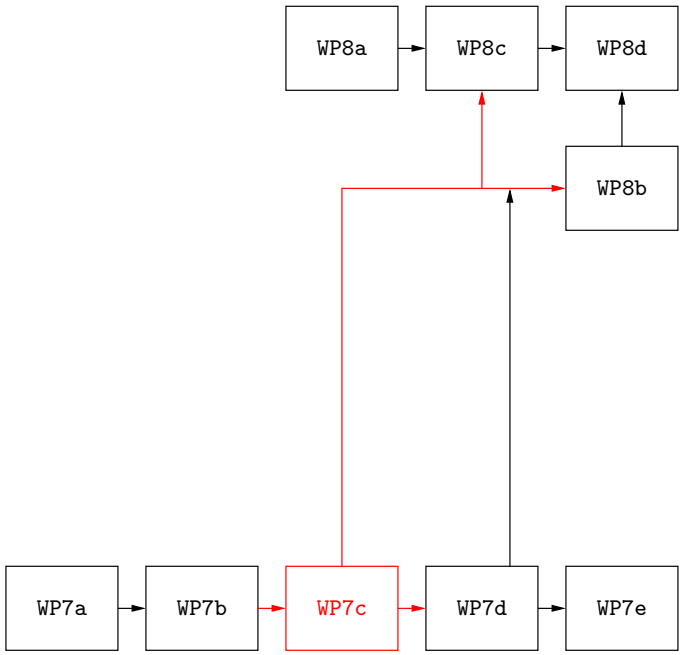

Figure 1: Inter-task dependencies for EmBounded

5 Relation with other deliverables

Figure 1 positions workpackage WP7c in the overall EmBounded task diagram. WP7c depends on WP7b (Formal translation from Hume to HAM instructions) and is required by WP7d (Hume model-checking), WP8b (Real-time computer vision algorithms) and WP8c (Evaluation of Hume). Three software deliverables have been produced in WP7c: D23 (Hume to HAM translator); D24 (Machine Code Generators); and D25 (Analyses integrated into the Compiler). A CD has been produced containing this software, and the software has also been made publicly available on the project web site (`http://www.embounded.org`).

Since work on the Hume compiler is central to the entire EmBounded project, a detailed diagram of all the interdependencies from these deliverables would be infeasibly complex. Instead, we present a table of dependencies based upon the requirements of the deliverables. Table 1 shows the deliverables which are interdependent with this deliverable.

6 Material provided

We provide the following material as evidence of our having met the objectives in this deliverable:

- CD containing the sources and installation instructions for the `humec` compiler implementation. A single CD has been produced containing all software produced by D23, D24 and D25. This software comprises a complete tool chain taking Hume source files as inputs and generating binary files for the chosen target architecture. The tools in the tool chain are:
  - Updated implementation of `phamc`, the Hume to HAM translator with support for the `humec` compiler (D23), and including integrated analyses (D25).
  - The implementation of the HAM to C compiler `ham2c` (D24).
  - Sources for the runtime system needed by the C compiler when linking the generated object code (D24).

Deliverable	Title	Notes
D03 (WP7b)	Formal translation from Hume to HAM	Compiler based upon formal translation
D05 (WP4a)	Report on stack size analysis	Compatibility with resource-analysis tools
D07 (WP8a)	Real-time testbed applications	Most of the applications work is based on the machine code compiler
D11 (WP4b)	Report on heap space analysis	Compatibility with resource-analysis tools
D12 (WP7a)	Formal semantics for Hume	Basis for the compilation scheme
D16 (WP3c)	Report on WCET analysis	Compatibility with resource-analysis tools
D23 (WP7c)	Hume to HAM compiler	Pre-processing required by the machine-code compiler
D25 (WP7c)	Analyses integrated into compiler	Analyses highly interdependent with the compiler
D26 (WP7d)	Translator to support model-checking	Compatibility with verification tools
D33 (WP8c)	Report evaluating Hume	The compiler is evaluated along with the other tools in the project
D34 (WP7e)	Hume problem-solving environment	Compiler an integral part of the Hume PSE

Table 1: Deliverables dependent upon/depending on D24 (WP7c)

  - An overall management program called `humec` which calls the other tools in the chain in the right order for producing binary files.

- Installation guide and User manual for the `phamc`, `ham2c` and `humec` programs.
- Copies of the academic papers produced by the EmBounded project that are related to the development and use of the Hume to HAM translator and machine code compiler [HFH+06b, HFH+06a, Poi06, MP06].

A Notes on the ham2c program

A.1 Files

File	Description
`main.c`	Coordination of program
`parse.c`	Parses ham to ops
`ops.c`	Definition of all known ops
`boxes.c`	Boxes and i/o
`functions.c`	Functions/strings/constructors/exceptions
`output.c`	Code generation
`coord.c`	Code generation for scheduling and i/o
`util_unix.c`	Utilities
`macros.c`	Loads macros from "`arch_macro*.txt`"
`io.c`	Load io definitions from "`arch_io_*.txt`"
`typing.c`	Handles discriminated unions
`ubcopy.c`	Copy code generation
`ubcmp.c`	Compare code generation
`ubprint.c`	Pretty print for debug code generation
`ubsize.c`	Determines exact size of bounded data structures
`cconsts.c`	Removes and caches constants
`filter.c`	Tidy up of ops

A.2 Type Specialised Code

Files: `typing.c, ubcopy.c, ubcmp.c, uprint.c, ubsize.c`

In unboxing it is necessary to use type information to guide the traversal of data structures, this can either be achieved by interpreting a type signature at runtime, or compiling the type to produce specialised code. The humec uses a combination of both. The circumstances where type information is required are:

- copying heap-to-wire
- printing for debugging
- comparing
- reading from a string/file
- showing to produce a string
- ccall for the packing/unpacking or arguments and the result

General requirements are:

- closures need not be considered
- exceptions are only required when printing
- discriminated unions are only required when comparing, printing, or copying
- high performance is desirable when copying
- int/float precision is only required for ccall

- word precision is only required for ccall, read, or show.

Due to the fact that simple types (no discriminated unions) are involved with reading, showing, and ccall means that interpreting a type signature at runtime is sufficient. Note that this approach was already previously being used for reading and ccall-unpacking in boxed humec. The type signature can easily be represented as a finite string which is both easily human and machine readable.

The high performance desired for copying and the general discriminated unions of copying, comparing and printing, makes these cases good candidates for generating specialised code. Extra effort is put into copying to ensure that tail-recursive structures (list) are handled efficiently.

The code generation for each follows an identical pattern:

- The type signature is first normalized. Normalization depends on the particular operation, for example, when copying, all the simple types (int, word, bool, char, float, zero tuple, and enumerations) can be treated the same way without consideration of precision, i.e. as the int type, but when printing the precision can still be ignored though it is necessary to format each type differently. Concrete examples for `"v5t3i8bf16"` would be:
  - `"vt3iii"` copy
  - `"vt3iif"` compare
  - `"vt3ibf"` show
  - `"v5t3ibf"` read
- The type is looked up in a map to see if it has already been processed.
- The type is traversed, recursively checking subtypes first, to emit specialised code (Noteworthy is the need for type substitutions for parametric discriminated unions).

A.3 Example Code

Consider the definition for list, and the specialisation of for a list of boolean:

```
data list a = nil | cons a (list a);
type blist = list bool;
```

A.3.1 Compare

Compare produces a function for each type where the function is capable of all comparison operations, i.e. ==, <=, <, >=, >, and !=.

```
static res_e cmp1(HEAP x, HEAP y, boolean less) {
  if(less && (x.lv < y.lv)) return RES_LESS;
  return (x.lv == y.lv)?RES_EQ:RES_NEQ;
}
static res_e cmp0(HEAP x, HEAP y, boolean less) {
  uint16 ex = ordFor(x, 1);
  uint16 ey = ordFor(y, 1);
  if(less && (ex < ey)) return RES_LESS;
  if(ex != ey) return RES_NEQ;
  if(!ISPTR(x)) return RES_EQ;
  uint8 res;
  res = cmp1(x.hp[0], y.hp[0], less);
  if(res != RES_EQ) return res;
  res = cmp0(x.hp[1], y.hp[1], less);
  if(res != RES_EQ) return res;
  return RES_EQ;
}
```

In terms of usage the code for each comparator operation is as follows:

Operator	Code
==	`(cmp0(stack[sp], stack[sp-1], FALSE) == RES_EQ)`
<=	`(cmp0(stack[sp], stack[sp-1], TRUE) != RES_NEQ)`
<	`(cmp0(stack[sp], stack[sp-1], TRUE) == RES_LESS)`
>=	`(cmp0(stack[sp-1], stack[sp], TRUE) != RES_NEQ)`
>	`(cmp0(stack[sp-1], stack[sp], TRUE) == RES_LESS)`
!=	`(cmp0(stack[sp], stack[sp-1], FALSE) != RES_EQ)`

However, when comparing trivial types such as integer or float, then the obvious C code is used, e.g. ==, would give `(stack[sp].lv == stack[sp-1].lv)` for integer comparison.

A.3.2 Print

Print produces human readable, Hume language like output for all data types.

```
static void dump2(FILE *file, HEAP v) {
  fputs((v.lv==0)?"false":"true", file);
}
static void dump1(FILE *file, HEAP v) {
  if(!ISPTR(v)) {
    fprintf(file, "nil");
  return;
  }
  fputs("(cons", file);
  fputs(" ", file);
  dump2(file, v.hp[0]);
  fputs(" ", file);
  dump1(file, v.hp[1]);
  fputs(")", file);
}
```

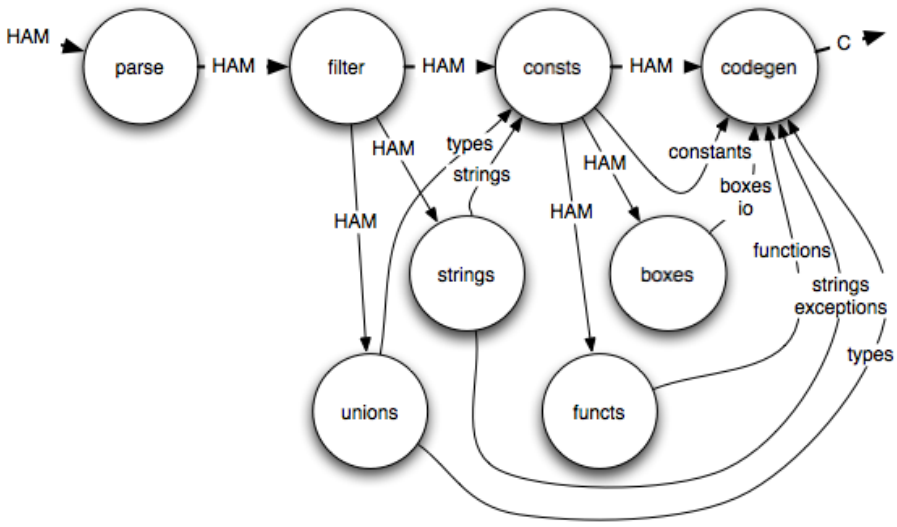

Figure 2: ham2c data-flow

A.3.3 Copy

Copy contains some optimisation, for example it generates loops for for tail-recursive types.

```
static void copy1(HEAP **hp, HEAP *dest, HEAP src) {
  HEAP result;
tailrecurse:
  if(!ISPTR(src)) { *dest = src; return; }
  result.hp = *hp;
  *dest = result;
  (*hp) += 2;
  result.hp[0] = src.hp[0];
  dest = &result.hp[1];
  src = src.hp[1];
  goto tailrecurse;
}
```

A.4 Tidying up Ham ops

File: `filter.c`

- removes `OP_MKTUPLE/OP_UNPACK` pairs as an optimisation
- removes empty box "`*_init`" operations
- removes deadcode trailing `OP_TAILCALL` and `OP_CHECKOUTPUTS`
- removes instruction specific to hami in terms of faux-io boxes and exceptions
- combines all `OP_CONSUME/OP_MAYBECONSUME` into one instruction (`OP_CONSUMESET`)
- combines all `OP_MATCHAVAILABLE` into a single instruction (`OP_AVAILSET`)
- ensures that `OP_AVAILSET` proceeds `OP_MATCHRULE`, `OP_CONSUMESET` follows `OP_MATCHEDRULE`, and that redundant match instructions are removed

- inserts `OP_GOTO` and eliminates duplicate code so that each box output is through one instruction
- splits `OP_WITHINSPACE/ OP_DONEWITHINSPACE` into separate `OP_WITHINHEAP/ OP_DONEWITHINHEAP` and `OP_WITHINSTACK/ OP_DONEWITHINSTACK` instructions

A.5 Data Flow

In Figure 2 the data-flow from the original HAM through to the final C is shown. When generating boxed code the "unions" stage becomes unnecessary and is only used to verify wires size. When generating unboxed code the "strings" stage becomes unnecessary as all strings will be handled by "cconsts", and the union/exceception information instead comes from "unions".

References

[HFH+06a] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hoffmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards formally verifiable resource bounds for real-time embedded systems. In *Proc. Workshop on Innovative Techniques for Certification of Embedded Systems*, 2006. 6

[HFH+06b] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hofman, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards formally verifiable WCET analysis for a functional programming language. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, number 06902 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. 6

[JH07] S. Jost and K. Hammond. Validation of the Prototype Worst-Case Executation Time (WCET) analysis. EmBounded Project Deliverable, October 2007. Deliverable D16. 2

[JLH07a] S. Jost, H-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D11. 2

[JLH07b] S. Jost, H-W. Loidl, and K. Hammond. Report on Stack-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D5. 2

[MP06] Greg Michaelson and Robert Pointon. Recursion, iteration and hume scheduling. In *Trends in Functional Programming (TFP'06)*, Nottingham UK, April 2006. 6

[Poi06] Robert F. Pointon. Compiling vs costing - experiences with hume. In *Implementation of Functional Languages (IFL'06)*, Budapest Hungary, September 2006. 6

[PS08] R.F. Pointon and N. Scaife. Hume to HAM Translator. EmBounded Project Deliverable, January 2008. Deliverable D23. (document), 1