HW-Hume in Isabelle

Chunxu Liu and Greg Michaelson

School of Mathematical and Computer Science
Heriot-Watt University, Riccarton, Scotland.
Email: {chunxu,greg}@macs.hw.ac.uk

Abstract. HW-Hume is the decidable Hume level oriented to direct implementation in hardware. As a first stage in the development of a verified compiler from HW-Hume to Java, we have implemented the semantics of HW-Hume in the Isabelle/HOL theorem prover, enabling the automatic proof of correctness of programs in a Floyd/Hoare style.

1 Introduction

A *verified* compiler gives guarantees that compilation preserves meaning from source to target, but not that the source program satisfies its specification. That is, given:

$M_S : S \rightarrow D$ - meaning of source programs in language S
$M_T : T \rightarrow D$ - meaning of target programs in language T
$C_{S-T} : S \rightarrow T$ - compiler from S to T

where D is some domain of meanings, we wish to guarantee for program P_S in source language S that:

$M_T(C(P_S)) = M_S(P_S)$

In contrast, a verifying compiler gives guarantees that a source program satisfies its specification, but not that the target is a true translation of the source.

Verified compilation has a long but surprisingly thin pedigree, since McCarthy and Painter's seminal work 40 years ago for arithmetic expressions [9]. They defined very simple source and target languages and their semantics, gave rules from source to target and proved them correct. Almost all subsequent work has followed this basic approach. However, where McCarthy and Painter used an ad hoc notation and constructed proofs by hand, there has been a growing trend to the use of formal, and possibly executable, notations, and of automated theorem proving technology.

Palsberg [14] designed, implemented and proved the correctness of a compiler generator, called Cantor, that accepts action semantic descriptions.

Stepney [17] discusses compilation from the simple high-level imperative source language Tosca to the low level target language Aida. Prolog and Z are used as the meta-languages to define the denotational semantics of both Tosca and Aida, with translation templates from each source language syntax structure to the equivalent target language structure. While this highly ambitious work for an industrial client, used executable notations to deliver a working verified compiler, almost all proofs were conducted painstakingly by hand.

Stringer-Calvert [18] extends Stepney's work in his PhD thesis, presenting an overview of the development of a demonstrably correct compiler by what he terms the DCC

method, which has three components: Specification, Implementation and Proof. The correctness of the DCC compiler is proved mechanically, using PVS [16]. of programming language, and targets used an abstract RISC machine language.

Curzon [3, 4] presents a formal machine-checked verification of a simple compiler specification using the HOL [5] theorem prover, thus combining a unitary notation and proof tool. He has implemented a tool that executes the verified compiler specification using formal proof. He also discusses bootstraping a correct compiler implementation.

Most recently, Klein and Nipkow [7] have used Isabelle/HOL [13] to prove the correctness of a compiler from the Java subset Jinja to JVM code.

We are exploring the development of a provably correct or verified compiler from HW-Hume [10, 11], the decidable but impoverished Hume layer oriented to hardware realisation, to Java. We have already constructed a prototype HW-Hume to Java compiler, discussed in [8].

While our work is are strongly influenced by Stepney, and by Klein and Nipkow, there are important differences. Where Stepney proved correctness via denotational semantics, we intend to use an operational semantics. While this enables us to retain close correspondence with our prototype HW-Hume to Java compiler, proofs may be longer than for denotational semantics, as operational semantics includes considerably more detail of evaluation. Similarly, Klein and Nipkow use Isaballe to also prove that source programs are well formed and type correct. We are only concerned with proving the correctness of translation and assume that some prior analysis has established other properties.

As a vital core stage in our work, we have embedded the semantics of HW-Hume in Isabelle, enabling proof of correctness. In the following sections we provide overviews of HW-Hume and it semantics, and of Isabelle/HOL. We then present the realisation of the HW-Hume semantics in Isabelle/HOLs, and discuss the proof of correctness of two HW-Hume exemplars. Finally, we consider how we intend to complete our formally verified HW-Hume to Java compiler.

2 HW-Hume Semantics

2.1 HW-Hume Abstract Syntax

Figure 1 shows the abstract syntax of HW-Hume. A HW-Hume program is built from one or more box(s), one or more wire(s) and optional initial declarations, const declarations and type declarations. The program execution is independent of the order of box, wire and init declarations.

2.2 HW-Hume Execution Model

Figure 2 shows the HW-Hume execution model which is based on non-terminating, round-robin, one-shot scheduling of boxes.

prog	::= $decl_1$ ″;″ ...″;″ $decl_n$	$n \geq 1$
decl	::= *box* | *wire* | *init* | *constdecl* | *typedecl*	
box	::= ″**box**″ *boxid ins outs* ″**match**″ *matches*	
ins/outs	::= ($ioid_1$ :: $type_1$, ...$ioid_n$ :: $type_1$)	$n \geq 1$
matches	::= $match_1$ ″|″ ... ″|″ $match_n$	$n \geq 1$
match	::= *patt* ″−>″ *expr*	
wire	::= ″**wire**″ $link_1$ ″**to**″ $link_2$	
init	::= *link* ″=″ *value*	
link	::= *boxid* ″.″ *ioid*	
constdecl	::= *constid* ″=″ *value*	
typedecl	::= *typeid* ″=″ *type*	
patt	::= *intliteral* | * | _ | _* | *varid* | (*patt*,*patt*)	
expr	::= *intliteral* | * | *varid* | (*expr*,*expr*)	
val	::= *intliteral* | (*val*,*val*)	
type	::= *int* | *typeid* | (*type*,*type*)	

Fig. 1. HW-Hume Abstract Syntax

```
for each box
    state ← RUNNABLE
forever
    for each box
        if state != BLOCKED then
            state ← MATCHFAIL
            for each match
                if some pattern matches input values then
                    consume values from input wires
                    evaluate associated expression
                    generate output wires
                    state ← SUCCESS
                    stop match loop
    for each box
        if state!=MATCHFAIL then
            if output wires can be established to their input wires then
                establish output wiress
                state ← RUNNABLE
            else
                state ← BLOCKED
```

Fig. 2. HW-Hume Execution Model

3 Isabelle Overview

Isabelle is a popular generic interactive theorem prover which supports a variety of logics. Isabelle/HOL [13] is the specialization for HOL(Higher-Order Logic) that is based on Gordon's HOL system [5], which itself is based on Church's original paper [2]. As Tobias Nipkow [13] said:

HOL = Functional Programming + Logic

Isabelle conforms largely to standard mathematical notation. As a generic proof assistant for logic, Isabelle is a powerful system for implementing logic formalisms.

Isabelle Proof General is a generic interface for proof assistants in Isabelle which supports a print mode for X Symbol tokens. In Isabelle, we can write

`1<^bsub>v\<^esub>` or `1<^isub>v` or `1<^sub>v`

In the Isabelle Proof General environment or an Isabelle print document, this will be displayed as 1_v which is easy to read.

In the sequel, we use Isabelle for Isabelle/HOL.

This section introduces Isabelle basic types with their primitive operations and further non-standard notation.

base types:

- *bool* — the type of truth.
- *nat* — the type of natural numbers.

constructor types:

- *list* — the type of lists with the type *'a list*. *nat list* means that every element of list has the type *nat*. Empty list is []. The infix operator @ concatenates two lists. The infix operator # inserts a element to the beginning of a list. $xs!n$ is the nth-element of xs (starting with 0).
- *set* — the type of sets with the type *'a set*. *nat set* means that every element of the set has the type *nat*. Empty set is {}. We write $\emptyset$ instead of {} for friendly reading.
- *option* — defined by
  datatype *'a option = None | Some 'a*
  It adjoins a new element *None* to a type *'a*. We write $\lfloor x \rfloor$ instead of *Some x* for succinctness. $the\ \lfloor x \rfloor = x$, where *the* is a function.
- *Pairs* — ordered pairs. (a_1, a_2) is of type $\tau_1 \times \tau_2$, where a_1 is of type τ_1 and a_2 is of type τ_2. $fst(a_1, a_2) = a_1$, $snd(a_1, a_2) = a_2$, where *fst* and *snd* are functions.
- *Tuples* — defined by *Pairs* nested. (a_1, a_2, a_3) stands for $(a_1, (a_2, a_3))$. So
  $fst(a_1, a_2, a_3) = a_1, \;\; fst(snd(a_1, a_2, a_3)) = a_2$.

function types:

- *total function* — denoted by $\Rightarrow$. Like SML, $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ means $\tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$. $[\tau_1, \tau_2, ..., \tau_n] \Rightarrow \tau$ is abbreviation of $\tau_1 \Rightarrow \tau_2 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$. Its update is $f(x := y)$ where $f :: {'a} \Rightarrow {'b}$, x with type $'a$ and y with type $'b$.

- *partial function* — defined by $'a \Rightarrow\ 'b\ option$. *None* represents undefinedness, $f\,x{=}\lfloor x \rfloor$ means that x is mapped to y. The domain of f is defined $dom\ f \equiv \{a \mid f\ \ x \neq None\}$. We write $'a \rightharpoonup\ 'b$ instead of $'a \Rightarrow\ 'b\ option$. *empty* is defined by $\lambda x.None$. Its updates is $f(x := \lfloor y \rfloor)$. We abbraviate $f(x := \lfloor y \rfloor)$ to $f(x \mapsto y)$. Such functions are called maps. The infix operator ++ overwrites map m_1 with m_2.
  i.e. $m_1 ++ m_2 \equiv \lambda x.\ case\ m_2\ x\ of\ None \Rightarrow m_1\ x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor$
- *Inductive definitions* — a method to define a function. In fact, a function from set A to set B is defined by a relation set $C \subseteq A \times B$ when set C satisfies some properties. Many datatype are inductive defined. A structural oprational semantics [6] is inductive definition of an evaluation relation. The inductive definitions [1] specifies the least set R closed under given collection rules. Applying a rule to elements of R yields a result within R. Milner [12] implemented one of the first inductive definitions. Isabelle provides commands for formalizing inductive definitions. Paulson [15] proved a fixedpoint with inductive definitions in Isabelle. Klein and Nipkow [7] inductively defined Jinja semantics and proved correctness of Jinja compiler. We mainly present HW-Hume semantics with inductive definitions.

type variables: denoted by $'a$, $'b$ etc. Like SML, they give rise to polymorphis types.

inference rule: $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ means $A_1 \Longrightarrow (A_2 \Longrightarrow A_3)$.
$[\![A_1; A_2; ...; A_n]\!] \Longrightarrow A$ is abbreviation of $A_1 \Longrightarrow (A_2 \Longrightarrow (... \Longrightarrow (A_n \Longrightarrow A)...))$.
It means "If A_1 and A_2 and ... and A_n then A".
i.e. inference rule

$$\frac{A_1\ \ A_2\ \ ...\ \ A_n}{A}$$

4 HW-Hume Semantics in Isabelle

4.1 HW-Hume Abstract Syntax in Isabelle

Figure 3 shows the abstract syntax of HW-Hume in Isabelle. After precompilation, all constid in const declarations are replaced by their values, and all typeid in type declarations are replaced by their types. The program execution is independent of the order of box and wire declarations. So we define a HW-Hume program as

"**types** $HProg = HBox\ \ list \times HWire\ \ list$"

"*HInit list*" is initial values. A box comprises a boxid, a series of inputs, a series of outputs and a series of matches. Hence we define a box as

"**types** $HBox = BName \times HIO\ \ list \times HIO\ \ list \times HMatch\ \ list$"

The first *HIO list* is for input and the second is for output.

4.2 HW-Hume States

Figure 4 shows the definition of the HW-Hume states — dynamic environment. Each box in a HW-Hume program has its own input and output. We define $hLocation = INOUT \times BName \times LName$) to denote them. The states is a map from hLocation to value. At the end of each cycle, a HW-Hume program checks the wire of each box's

```
types BName = string                -- "names for box"
types LName = string                -- "names for io (link)"
types VName = string                -- "names for variable"
datatype HP = I | W | WI            -- "ignore,wild,wild ignore."
            | HPInt int             -- "nat value"
            | HPVar VName           -- "variable"
            | HPPair HP HP          -- "pair"
datatype HE = I                     -- "ignore"
            | HEInt int             -- "int value"
            | HEVar VName           -- "variable"
            | HEPair HE HE          -- "pair"
datatype HV = HVInt int             -- "int value"
            | HVPair HV HV          -- "pair"
datatype HT = Z                     -- "nat"
            | HTPair HT HT          -- "pair"
types HIO    = LName × HT
types HMatch = HP × HE
types HBox   = BName × HIO list × HIO list × HMatch list
types HWire  = (BName × LName) × (BName × LName)
types HProg  = HBox list × HWire list
types HInit  = BName × LName × HV
```

Fig. 3. HW-Hume Abstract Syntax

output and try to transfer their values to some box's input. If there is an output which cannot be transferred into a box, that box is blocked. Conversely, if an output is still mapped to a value, that box is blocked. We define a function *BnoEmpty* to check if a box is blocked or not. The function *Init_hStete* sets the initial states. The function *UpdateS* updates state by transferring each box's output which can be transferred to its target.

4.3 HW-Hume Big Step Semantics

We next present the HW-Hume big step semantics in inductive definitions judgement form.

Cycles is the top-step of running a Hume program. Its goal is to run all boxes N times in repeated cycles. When all cycles finish, the program halts. The global states (*hState*) identify the program states.

We use *RC* defined in Figure 4 as a running result. The possible results of *RunC* are error (*Cer hErr*) or success. When success, the result is *Cs hState*.

We inductively define *RunC* as well. The *RunC B* is the least relation set closed under given rules below.

We define the *RunC* judgement form $P \vdash_c \langle n, s\rangle \Rightarrow \langle rc\rangle$. This is an abbreviation of $(n, s, rc) \in RunC\ P$. The P has type *HProg*; n has type *hCNum*; rc has type *RC*.

We define *Cycle* induction rules below.

Global State:	
datatype $INOUT = LI \mid LO$	
types $hLocation = INOUT \times BName \times LName$	
types $hState = hLocation \rightharpoonup HV$	
Local State:	
datatype $hErr = MPVer \mid MIPer \mid EEer \mid MOEer$	
types $hVar = VName \rightharpoonup HV$	
types $hCNum = nat$	
Running Result States	
datatype $RC = Cer\ hErr \mid Cs\ hState$	Run Cycles (RunC)
datatype $RB = Ber\ hErr \mid Bs\ hState$	Run Box and Boxes (RunB, RunBL)
datatype $RM = Mer\ hErr \mid Ms\ hState$	Run Match (RunM)
datatype $MIP = IPer\ hErr \mid IPf \mid IPs\ hState \times hVar$	Match in to pattrn (MatchIP)
datatype $MPV = PVer\ hErr \mid PVf \mid PVs\ hVar$	Match pattrn to value (MatchPV)
datatype $EE = Eer\ hErr \mid Es\ HE$	Evalate expression (EvalE)
datatype $MOE = OEer\ hErr \mid OEs\ hState$	Match out to expression (MatchOE)

Fig. 4. HW-Hume State

- C0_s — Running a program 0 time, i.e. 0 cycle. s is not changed.

$$\frac{n = 0}{P \vdash_c \langle n, s\rangle \Rightarrow \langle Cs\ \ s\rangle}$$

- C1_e — Calling *RunBL* error, so running a program 1 cycle error.

$$\frac{n = 1 \quad \vdash_{bl} \langle P2BL\ P, s\rangle \Rightarrow \langle Ber\ \ er\rangle}{P \vdash_c \langle n, s\rangle \Rightarrow \langle Cer\ \ er\rangle}$$

- C1_s — Calling *RunBL* success, running a program 1 cycle success.

$$\frac{n = 1 \quad \vdash_{bl} \langle P2BL\ P, s\rangle \Rightarrow \langle Bs\ \ s'\rangle \quad s'' = UpdateS\ \ s'\ P}{P \vdash_c \langle n, s\rangle \Rightarrow \langle Cs\ \ s''\rangle}$$

- C_R_e — More than one cycle, recursive, the first cycle error.

$$\frac{n > 1 \quad P \vdash_c \langle 1, s\rangle \Rightarrow \langle Cer\ \ er\rangle}{P \vdash_c \langle n, s\rangle \Rightarrow \langle Cer\ \ er\rangle}$$

- C_R — More than one cycle, recursive, the first cycle is success. Result depends on remaining cycles.

$$\frac{n > 1 \quad P \vdash_c \langle 1, s\rangle \Rightarrow \langle Cs\ \ s'\rangle \quad n' = n - 1 \quad P \vdash_c \langle n', s'\rangle \Rightarrow \langle rc\rangle}{P \vdash_c \langle n, s\rangle \Rightarrow \langle rc\rangle}$$

We show below the result theorems:

theorem *EvalE_Result*: $bn, s, lv \vdash_e \langle e\rangle \Rightarrow \langle ee\rangle \Longrightarrow$
$(ee = Eer\ \ EEer) \vee (\exists es.(ee = Es\ \ es) \wedge (HEhasVar\ \ es = False))$

consts $RunC\ :: HProg \Rightarrow (hCNum \times hState \times RC)\ set$
syntax $RunC :: HProg \Rightarrow hCNum \Rightarrow hState \Rightarrow RC \Rightarrow bool$
$(_ \vdash_c \langle _, _\rangle \Rightarrow \langle _\rangle\ \ [0, 0, 0, 0]\ 81)$
translations $P \vdash_c \langle n, s\rangle \Rightarrow \langle rc\rangle\ \ \rightleftharpoons\ \ (n, s, rc) \in RunC\ \ P$

consts $RunBL\ :: (HBox\ \ list\ \times\ hState\ \times\ RB)\ set$
syntax $RunBL :: HBox\ \ list\ \Rightarrow\ hState\ \Rightarrow\ RB\ \Rightarrow\ bool$
$(\vdash_{bl} \langle _, _\rangle\ \Rightarrow\ \langle _\rangle\ \ [0, 0, 0]\ 81)$
translations $\vdash_{bl} \langle bl, s\rangle\ \Rightarrow\ \langle rb\rangle\ \ \rightleftharpoons\ \ (bl, s, rb) \in RunBL$

consts $RunB\ :: HBox \Rightarrow (hState \times RB)\ set$
syntax $RunB :: HBox \Rightarrow hState \Rightarrow RB \Rightarrow bool$
$(_ \vdash_b \langle _\rangle \Rightarrow \langle _\rangle\ \ [0, 0, 0]\ 81)$
translations $B \vdash_b \langle s\rangle \Rightarrow \langle rb\rangle\ \ \rightleftharpoons\ \ (s, rb) \in RunB\ \ B$

consts $RunM\ :: BName \Rightarrow LName\ \ list \Rightarrow LName\ \ list \Rightarrow$
$(HMatch\ \ list\ \times\ hState\ \times\ RM)\ set$
syntax $RunM :: BName \Rightarrow LName\ \ list \Rightarrow LName\ \ list \Rightarrow$
$HMatch\ \ list \Rightarrow hState \Rightarrow RM \Rightarrow bool$
$(_, _, _ \vdash_m \langle _, _\rangle \Rightarrow \langle _\rangle\ \ [0, 0, 0, 0, 0, 0]\ 81)$
translations $bn, ilnl, olnl \vdash_m \langle ml, s\rangle \Rightarrow \langle rm\rangle\ \ \rightleftharpoons\ \ (ml, s, rm) \in RunM\ \ bn\ \ ilnl\ \ olnl$

consts $MatchIP\ :: BName \Rightarrow$
$(LNname\ \ list \times HP \times hState \times hVar \times MIP)\ set$
syntax $MatchIP :: BName \Rightarrow$
$(LNname\ \ list \Rightarrow HP \Rightarrow hState \Rightarrow hVar \Rightarrow MIP \Rightarrow boot$
$(_ \vdash_{i-p} \langle _, _, _, _\rangle \Rightarrow \langle _\rangle\ \ [0, 0, 0, 0, 0, 0]\ 81)$
translations $bn \vdash_{i-p} \langle ilnl, p, s, lv\rangle \Rightarrow \langle mip\rangle\ \ \rightleftharpoons\ \ (ilnl, p, s, lv, mip) \in MatchIP\ \ bn$

consts $MatchPV\ :: (HP \times HV \times hVar \times MPV)\ set$
syntax $MatchPV :: HP \Rightarrow HV \Rightarrow hVar \Rightarrow MPV \Rightarrow bool$
$(\vdash_{p-v} \langle _, _, _\rangle \Rightarrow \langle _\rangle\ \ [0, 0, 0, 0]\ 81)$
translations $\vdash_{p-v} \langle p, v, lv\rangle \Rightarrow \langle mpv\rangle\ \ \rightleftharpoons\ \ (p, v, lv, mpv) \in MatchPV$

consts $EvalE\ :: BName \Rightarrow hState \Rightarrow hVar \Rightarrow (HE \times EE)\ set$
syntax $EvalE :: BName \Rightarrow hState \Rightarrow hVar \Rightarrow HE \Rightarrow EE \Rightarrow bool$
$(_, _, _ \vdash_e \langle _\rangle \Rightarrow \langle _\rangle\ \ [0, 0, 0, 0, 0]\ 81)$
translations $bn, s, lv \vdash_e \langle e\rangle \Rightarrow \langle ee\rangle\ \ \rightleftharpoons\ \ (e, ee) \in EvalE\ bn\ s\ lv$

consts $MatchOE\ :: BName \Rightarrow (LName\ \ list \times HE \times hState \times MOE)\ set$
syntax $MatchOE :: BName \Rightarrow (LName\ \ list \times HE \times hState \times MOE \Rightarrow bool$
$(_ \vdash_{o-e} \langle _, _, _\rangle \Rightarrow \langle _\rangle\ \ [0, 0, 0, 0, 0]\ 81)$
translations $bn \vdash_{o-e} \langle olnl, e, s\rangle \Rightarrow \langle moe\rangle\ \ \rightleftharpoons\ \ (olnl, e, s, moe) \in MatchOE\ \ bn$

Fig. 5. Inductive Definitions and Judgement Forms

Given a box name bn, global state s and local state lv, an expression e will evaluate to ee provided either ee is error or there is some final success expression es which has no free variables.

theorem *MatchOE_Result*: $bn \vdash_{o-e} \langle olnl, e, s \rangle \Rightarrow \langle moe \rangle \Longrightarrow$
$(moe = OEer\ \ MOEer) \vee (\exists oes.(moe = OEs\ \ oes))$

Given a box name bn, then the associated output list $olnl$, expression e and global state s and evaluates to moe such that either moe is error or there is some success global state oes corresponding to it.

theorem *MatchPV_Result*: $\vdash_{p-v} \langle p, v, lv \rangle \Rightarrow \langle mpv \rangle \Longrightarrow$
$(mpv = PVer\ \ MPVer) \vee (mpv = PVf) \vee (\exists lv'.(mpv = PVs\ \ lv'))$

The pattern p, value v and local state lvs evaluates to mpv which is either error or Match Fail or success local state lv'.

The other theorems below are similar.

theorem *MatchIP_Result*: $bn \vdash_{i-p} \langle ilnl, p, s, lv \rangle \Rightarrow \langle mip \rangle \Longrightarrow$
$(mip = IPer\ \ MPVer) \vee (mip = IPer\ \ MIPer) \vee (mip = IPf) \vee (\exists ips.(mip = IPs\ \ ips))$

theorem *RunM_Result*: $bn, ilnl, olnl \vdash_{m} \langle ml, s \rangle \Rightarrow \langle m \rangle \Longrightarrow (\exists pls.(m = Ms\ \ pls)) \vee$
$(m = Mer\ \ MPVer) \vee (m = Mer\ \ MIPer) \vee (m = Mer\ \ EEer) \vee (m = Mer\ \ MOEer)$

theorem *RunB_Result*: $B \vdash_{b} \langle s \rangle \Rightarrow \langle rb \rangle \Longrightarrow (\exists bs.(rb = Bs\ \ bs)) \vee$
$(rb = Ber\ \ MPVer) \vee rb = Ber\ \ MIPer) \vee rb = Ber\ \ EEer) \vee rb = Ber\ \ MOEer)$

theorem *RunBL_Result*: $\vdash_{bl} \langle bls \rangle \Rightarrow \langle rb \rangle \Longrightarrow (\exists bs.(rb = Bs\ \ bs)) \vee$
$(rb = Ber\ \ MPVer) \vee rb = Ber\ \ MIPer) \vee rb = Ber\ \ EEer) \vee rb = Ber\ \ MOEer)$

theorem *RunC_Result*: $P \vdash_{c} \langle n, s \rangle \Rightarrow \langle rc \rangle \Longrightarrow (\exists s'.(rc = Cs\ \ s')) \vee$
$(rc = Cer\ \ MPVer) \vee rc = Cer\ \ MIPer) \vee rc = Cer\ \ EEer) \vee rc = Cer\ \ MOEer)$

5 Proving HW-Hume Program Correctness

Now, we prove two HW-Hume programs correct from the HW-Hume semantics in Isabelle.

5.1 Proving Swap

The first example is "*Swap*" which is very simple. There is only one box and one wire. The box Swap's out("*o*") is a link to its in("*i*"). The "*Swap.o*" and "*Swap.i*" are tuples. The matching rule "(x,y) -> (y,x)" will swap the values of "*Swap.i*". Figure 6 depicts the following code:

```
box Swap
  in  (i::(Bit,Bit))
  out (o::(Bit,Bit))
match
  (x,y) -> (y,x);
wire Swap
  (Swap.o initially (0,1))
  (Swap.i);
```

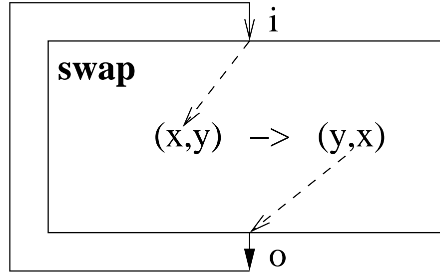

Fig. 6. Swap

With our HW-Hume abstract syntax in Isabelle, this program is presented as:

```
constdefs Swap_P :: HProg
Swap_P ≡
(
  [ (* Box *)
    (
      ''Swap'',
      [(''i'', (Z,Z)_t)],   (* in *)
      [(''o'', (Z,Z)_t)],   (* out *)
      [
       ( (''x''_pv, ''y''_pv)_p,   (''y''_ev, ''x''_ev)_e )
      ]
    )
  ],
  [ (* Wire *)
    ( (''Swap'', ''o''),   (''Swap'', ''i'') )
  ],
)
constdefs Swap_Init :: HInit list
Swap_Init ≡
(
  [ (* Init *)
    ( (''Swap'', ''i'', (0_v, 1_v)_v )
  ]
)
```

After initialisation, "*Swap.i*" has value "(0,1)". After running box "*Swap*" once, "*Swap.o*" got value "(1,0)". Then, at the end of the cycle, it will be transfered to "*Swap.i*". So before the second cycle, "*Swap.i*" has value "(1,0)". Before the third cycle, "*Swap.i*" has value "(0,1)", and so on.

In Isabelle, the global states of HW-Hume is defined as $hState = hLocation \rightharpoonup HV$. We can get the initial state by applying function *Init_hState* to *Swap_Init*. We define

two states:

constdefs $\mathit{Swap_S0} :: \mathit{hState}$

$\mathit{Swap_S0} \equiv \mathit{empty}((LI, ''Swap'', ''i'') \mapsto (0_v, 1_v)_v)$

constdefs $\mathit{Swap_S1} :: \mathit{hState}$

$\mathit{Swap_S1} \equiv \mathit{empty}((LI, ''Swap'', ''i'') \mapsto (1_v, 0_v)_v)$

Simply, we have a lemma:

lemma $\mathit{SwapInit}: \quad \mathit{Init_hState\ Swap_Init} = \mathit{Swap_S0}$

i.e. *Swap_S0* equals the initial state.

With our inductive defined HW-Hume semantics in Isabelle, we prove below lemmas by recursively calling inductive rules.

lemma $\mathit{Swap_S0_Cycle_0}: \quad \mathit{Swap_P} \vdash_c \langle 0, \mathit{Swap_S0}\rangle \Rightarrow \langle \mathit{Cs}\ \ \mathit{Swap_S0}\rangle$

lemma $\mathit{Swap_S1_Cycle_0}: \quad \mathit{Swap_P} \vdash_c \langle 0, \mathit{Swap_S1}\rangle \Rightarrow \langle \mathit{Cs}\ \ \mathit{Swap_S1}\rangle$

For cycle 0 with state $Swap_S0$ or $Swap_S1$, the result is success with that state.

The other lemmas are similar.

lemma $\mathit{Swap_S0_Cycle_1}: \quad \mathit{Swap_P} \vdash_c \langle 1, \mathit{Swap_S0}\rangle \Rightarrow \langle \mathit{Cs}\ \ \mathit{Swap_S1}\rangle$

lemma $\mathit{Swap_S1_Cycle_1}: \quad \mathit{Swap_P} \vdash_c \langle 1, \mathit{Swap_S1}\rangle \Rightarrow \langle \mathit{Cs}\ \ \mathit{Swap_S0}\rangle$

lemma $\mathit{Swap_S0_Cycle_2}: \quad \mathit{Swap_P} \vdash_c \langle 2, \mathit{Swap_S0}\rangle \Rightarrow \langle \mathit{Cs}\ \ \mathit{Swap_S0}\rangle$

lemma $\mathit{Swap_S1_Cycle_2}: \quad \mathit{Swap_P} \vdash_c \langle 2, \mathit{Swap_S1}\rangle \Rightarrow \langle \mathit{Cs}\ \ \mathit{Swap_S1}\rangle$

Finally, we have a theorem:

theorem *Swap_Cycles*:

$\mathit{Swap_P} \vdash_c \langle 2 * n, \quad \mathit{Init_hState}\ \ \mathit{Swap_Init}\rangle \Rightarrow \langle \mathit{Cs}\ \ \mathit{Swap_S0}\rangle$

$\mathit{Swap_P} \vdash_c \langle 2 * n + 1, \mathit{Init_hState}\ \ \mathit{Swap_Init}\rangle \Rightarrow \langle \mathit{Cs}\ \ \mathit{Swap_S1}\rangle$

Based on theorem *Swap_Cycles*, when we run *Swap* $2 * n$ times, "*Swap.i*" has value "(0,1)"; when we run *Swap* $2 * n + 1$ times, "*Swap.i*" has value "(1,0)".

5.2 Proving Adder

The second example is "*Adder*". There are three boxes and four wires. The box "*gen*" outputs from "(0,0,0)" to "(1,1,1)" in each cycle repetitively. This output ("*gen.t*") is linked to the in of box "*adder*". Matching rules of the box "*adder*" calculate full bit addition of "*adder.i*" by truth table. The result "*adder.o*" will be transfered to box "*output*". In the original version, the "*output*" is standard output. Because we cannot at present accommodate I/O in our semantics in Isabelle, we simulate standard output by box "*output*.

In each cycle, "*gen.t*" is a value from "(0,0,0)" to "(1,1,1)". At the end of a cycle, the value is transferred to "*adder.i*". On the next cycle, the full bit addition is stored in "*adder.o*". At the end of the cycle, the value is transferred to "*output.i*". Then in the third cycle, the box "*output*" throws it away. Figure 7 depicts the following code:

```
box gen
in  (i::(Bit,Bit,Bit))
out (o::(Bit,Bit,Bit), t::(Bit,Bit,Bit))
match
```

```
  (0,0,0) -> ((0,0,1),(0,0,0)) |
  (0,0,1) -> ((0,1,0),(0,1,0)) |
  (0,1,0) -> ((0,1,1),(1,0,0)) |
  (0,1,1) -> ((1,0,0),(1,1,0)) |
  (1,0,0) -> ((1,0,1),(0,0,1)) |
  (1,0,1) -> ((1,1,0),(0,1,1)) |
  (1,1,0) -> ((1,1,1),(1,0,1)) |
  (1,1,1) -> ((0,0,0),(1,1,1));

box adder
in  (i::(Bit,Bit,Bit))
out (o::(Bit,Bit))
match
  (0,0,0) -> (0,0) |
  (0,1,0) -> (1,0) |
  (1,0,0) -> (1,0) |
  (1,1,0) -> (0,1) |
  (0,0,1) -> (1,0) |
  (0,1,1) -> (0,1) |
  (1,0,1) -> (0,1) |
  (1,1,1) -> (1,1) ;

box output
in  (i::Bit, eat::(Bit,Bit))
out (o::Bit)
match
  (*,eat) -> (*);

wire gen    (gen.o initially (0,0,0)) (gen.i, adder.i trace);
wire adder  (gen.t)                   (output.eat trace);
wire output (output.o, adder.o)       (output.i);
```

With our HW-Hume abstract syntax in Isabelle, this program is presented as:

constdefs *Adder_P* :: *HProg*
Adder_P $\equiv$
(
 [(* *Box* *)
  (
   ''gen'',
   $[(''i'',\ (Z,Z,Z)_t)]$, (* *in* *)
   $[(''o'',\ (Z,Z,Z)_t),\ (''t'',\ (Z,Z,Z)_t)]$, (* *out* *)
   [
    $(\ (0_p,0_p,0_p)_p,\quad ((0_e,0_e,1_e)_e,(0_e,0_e,0_e)_e)_e\ ),$
    ...
    $(\ (1_p,1_p,1_p)_p,\quad ((0_e,0_e,0_e)_e,(1_e,1_e,1_e)_e)_e\ )$
   ]

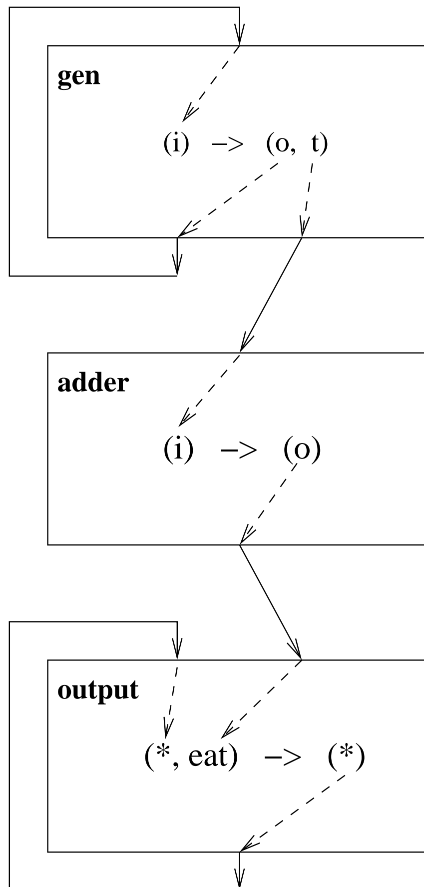

Fig. 7. Adder

```
    ),
    (
      ''adder'',
      [(''i'', (Z,Z,Z)_t)],  (* in   *)
      [(''o'', (Z,Z)_t)],         (* out  *)
      [
        ( (0_p, 0_p, 0_p)_p,    (0_e, 0_e)_e ),
         ...
        ( (1_p, 0_p, 0_p)_p,    (1_e, 0_e)_e ),
         ...
      ]
    ),
    (
      ''output'',
      [(''i'', Z),  (''eat'', (Z,Z)_t)],  (* in   *)
      [(''o'', Z)],                            (* out  *)
      [
        ( (HP.I, ''eat''_pv)_p,    HE.I )
      ]
    )
  ],
  [ (* Wire *)
    ( (''gen'',      ''o'' ),   (''gen'',      ''i''   ) ),
    ( (''gen'',      ''t'' ),   (''adder'',   ''i''   ) ),
    ( (''adder'',  ''o'' ),   (''output'', ''eat'') ),
    ( (''output'', ''o'' ),   (''output'', ''i''   ) )
  ],
)
constdefs Adder_Init :: HInit list
Adder_Init  ≡
(
  [ (* Init *)
    ( (''gen'',''i'', (0_v, 0_v, 0_v)_v )
  ]
)
```

We define global states of *Adder* in Isabelle as:

```
constdefs Adder_S0' :: hState
    Adder_S0' ≡ empty(
      (LI, ''gen'', ''i'') ↦ (0_v, 0_v, 0_v)_v
    )
```

We define state *Adder_S0′* as a mapping from link *gen.i* to the value $(0, 0, 0)$.

```
    constdefs Adder_S0 :: hState
    Adder_S0 ≡ empty(
      (LI, ''gen'', ''i'') ↦ (0_v, 0_v, 0_v)_v,
```

$(LI, ''adder'', ''i'') \mapsto (1_v, 1_v, 1_v)_v,$
$(LI, ''output'', ''eat'') \mapsto (0_v, 1_v)_v$
)

State *adder_S*0 maps link *gen.i* to $(0, 0, 0)$, *adder.i* to $(1, 1, 1)$ and *output.eat* to (0,1).

Other state definitions are similar.

constdefs $Adder_S1' :: hState$
$Adder_S1' \equiv empty($
$(LI, ''gen'', ''i'') \mapsto (0_v, 0_v, 1_v)_v,$
$(LI, ''adder'', ''i'') \mapsto (0_v, 0_v, 0_v)_v$
)

constdefs $Adder_S1 :: hState$
$Adder_S1 \equiv empty($
$(LI, ''gen'', ''i'') \mapsto (0_v, 0_v, 1_v)_v,$
$(LI, ''adder'', ''i'') \mapsto (0_v, 0_v, 0_v)_v,$
$(LI, ''output'', ''eat'') \mapsto (1_v, 1_v)_v$
)
...

constdefs $Adder_S7 :: hState$
$Adder_S7 \equiv empty($
$(LI, ''gen'', ''i'') \mapsto (1_v, 1_v, 1_v)_v,$
$(LI, ''adder'', ''i'') \mapsto (1_v, 0_v, 1_v)_v,$
$(LI, ''output'', ''eat'') \mapsto (0_v, 1_v)_v$
)

With our inductively defined HW-Hume semantics in Isabelle, we prove the following lemmas by recursively calling inductive rules.

lemma $AdderInit: \ Init_hState\ Adder_Init = Adder_S0'$
lemma $Adder_S0'_Cycle_1: \ Adder_P \vdash_c \langle 1, Adder_S0' \rangle \Rightarrow \langle Cs\ Adder_S1' \rangle$
lemma $Adder_S1'_Cycle_1: \ Adder_P \vdash_c \langle 1, Adder_S1' \rangle \Rightarrow \langle Cs\ Adder_S2 \rangle$
...
lemma $Adder_S6_Cycle_1: \ Adder_P \vdash_c \langle 1, Adder_S6 \rangle \Rightarrow \langle Cs\ Adder_S7 \rangle$
lemma $Adder_S7_Cycle_1: \ Adder_P \vdash_c \langle 1, Adder_S7 \rangle \Rightarrow \langle Cs\ Adder_S0 \rangle$

Finally, we have a theorem:

theorem *Adder_Cycles*:
$Adder_P \vdash_c \langle 0, Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S0' \rangle$
$Adder_P \vdash_c \langle 1, Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S1' \rangle$
$Adder_P \vdash_c \langle 0 + 8 * (n+1), Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S0 \rangle$
$Adder_P \vdash_c \langle 1 + 8 * (n+1), Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S1 \rangle$
$Adder_P \vdash_c \langle 2 + 8 * n, Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S2 \rangle$
$Adder_P \vdash_c \langle 3 + 8 * n, Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S3 \rangle$
$Adder_P \vdash_c \langle 4 + 8 * n, Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S4 \rangle$
$Adder_P \vdash_c \langle 5 + 8 * n, Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S5 \rangle$
$Adder_P \vdash_c \langle 6 + 8 * n, Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S6 \rangle$
$Adder_P \vdash_c \langle 7 + 8 * n, Init_hState\ Adder_Init \rangle \Rightarrow \langle Cs\ Adder_S7 \rangle$

This set of thereoms characterises the number of cycles from the initial state to evaluate the program characterised by the abstract syntax *Adder_Init* to reach the different possible result states.

6 Results

We have carried out the above proofs using Isabelle on a Pentium 4 based PC running Linux Red Hat 8, with 1GB of RAM and a 1.6GH CPU. Table 6 shows the size of the proofs and the time taken.

	Swap	Adder
Boxes	1	3
Wires	1	4
States	2	11
Match Rules	1	17
Proof Size (lines)	200	2000
Run Time (secs)	10	1020

Fig. 8. Isabell proofs.

From these initial results, we observe that the proof sizes and times appear to rise very rapidly with increases in the size of programs. This is unsurprising: at present we are essentially performing naive proofs for all possible transition sequences.

7 Conclusions

We have presented the embedding of HW-Hume in Isabelle and the proof of correctness of two programs. While we have demonstrated that our approach is promising, initial results suggest that we must explore more abstract tactics to make mechanical correctness proof useable for realistic Hume programs composed from many multi-state/multi-wire boxes,

The work discussed here is central to our longer term goal of a verified compiler from HW-Hume to Java. We recently decided to adopt Jinja as the target language, to enable an integrated approach in Isabelle. We have also modified our original HW-Hume to Java compiler to generate Jinja, and are currently formalising the translation from HW-Hume to Jinja. Time permitting, the next stages would be to embed this formalisation in Isabelle and to explore automatic support for proof that compilation maintains semantic consistency.

Acknowlegements

This research is partly supported by EU FP6 EmBounded. We would like to thank our colleagues in the wider Hume project for valuable discussion.

References

1. P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.
2. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
3. P. Curzon. Compiler correctness and input/output. Technical report, Computer Laboratory, University of Cambridge, November 1992.
4. P. Curzon. The Verified Compilation of Vista Programs. Technical report, Computer Laboratory, University of Cambridge, January 1994.
5. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.
6. Matthew Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
7. Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.
8. C. Liu and G. Michaelson. Translating Hume to Java. In H-W. Loidl, editor, *Draft Proceedings of 5th Symposium on Trends in Functional Programming, Ludwig-Maximillian's Universitat, Munich, Germany*, pages 113–128, November 2004.
9. J. McCarthy and J. Painter. Correctness of a compiler for Arithmetic Expressions. In J. T. Schwarz, editor, *Proceedings of Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.
10. G. Michaelson and Kevin Hammond. The Hume Language Definition and Report, Version 0.3. Technical report, Heriot-Watt University and University of St Andrews, 2004.
11. 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.
12. Robin Milner. *How to Derive Inductions in LCF*. Edinburgh University Press, U.K., 1980.
13. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher Order Logic*, volume 2283 of *LNCS*. Springer, 2002. `http://www.in.tum.de/~nipkow/LNCS2283`.
14. J. Palsberg. A provably correct compiler generator. In *Proceedings, ESOP '92, 4th European Symposium on Programming, Rennes, France*, February 1992.
15. Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 187–211. MIT Press, Cambridge, MA, USA, 2000.
16. SRI-Formalware. *The PVS Specification and Verification System*. SRI International Computer Science Laboratory, 2006. `http://pvs.csl.sri.com/`.
17. Susan Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1993.
18. D. W. J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*. PhD thesis, University of York, March 1998.