Deductive Translation Validation for a Subset of Higher Order Logic

Guodong Li and Konrad Slind

School of Computing, University of Utah

Abstract. We discuss a proof-producing compiler for a subset of higher order logic. The translation validation is automatic, and is based on Hoare rules derived from a compositional semantics for sequences of instructions for an ARM-like machine. Partial and total correctness are dealt with. The main focus is on issues in the intermediate level and back-end of the compiler.

1 Introduction

It is well-known that higher order logic (HOL) has a simple purely functional programming language built into it. Most algorithms can be represented by functional programs in HOL, *i.e.*, as mathematical functions whose properties can be transparently stated and proved correct using ordinary mathematics. This feature is a strength of higher order logic and is routinely exploited in verifications carried out in any HOL implementation. As a striking example, Gonthier's report [6] on his formal proof of the Four Colour Theorem illustrates the use of functional programs in the formalization; he strongly advocates proving their correctness and then incorporating them into the internal reduction relation on which the logic is based.

How else can formalized programs[1] be used? Once one has gone to the effort to define a program and prove its correctness it seems sensible to exploit it as widely as possible. An obvious idea is to push the program out into the 'real world'. This may be accomplished by using 'cut-and-paste' to submit the source text of a formalized program to a compiler or, better yet, by having the theorem prover itself 'prettyprint' the program to concrete syntax which may then be compiled [1]. Recent interesting applications of such a facility include [2] which reflexively applies the generated programs to theorem proving tasks. Moreover, logics implemented in a programming language that supports such conversions, *e.g.*, Lisp for PVS and ACL2, can provide seamless support for such activities. In ACL2, this facility provides a convenient way to obtain high speed simulators from formalizations [8]. In these applications, the original formal program is being translated, essentially unaltered, to the syntax of a high-level programming language, and 'off-the-shelf' compilers are then applied. Thus there is only a small semantic gap between the logical function and the exported program.

[1] We will systematically refer to mathematical functions as programs.

However, suppose we want to translate directly to assembly language or even hardware. Now the semantic gap widens to a yawning gulf: the prettyprinting step essentially becomes full-blown compilation. A formal connection between the source and the results of compilation then becomes necessary in order to convince skeptics or to meet the requirements of high-assurance applications. It is this problem that we tackle in this paper. The technology used to provide the required assurance is *compilation-by-proof*: a subset of the logic itself is compiled to assembly language by running proofs. This has several benefits:

1. The results of compilation are formally proved correct automatically. The fully mechanical translation validation goes with the compilation.
2. There is no fixed syntax for source programs, nor do they have a mandated operational semantics. This supports much flexibility and allows the meaning of the function to be transparent.
3. Proofs about the original source program may be conducted in ordinary mathematics—often the appropriate level of generality—and transfer immediately to the generated assembly. This seems far less daunting than conducting proofs at the assembly level.

Compilation of logic functions by proof has already been investigated in a prototype hardware compiler which synthesizes FPGA implementations from first-order HOL functions [7]. It is our intention in this paper to provide similar functionality for software, namely to use automated proof to translate programs written in a subset of the computable functions expressible in HOL to implementations in ARM assembly.

The front end of the compiler is implemented by a sequence of source-to-source mappings, each implemented via deductive proof steps. The subsequent translation from the intermediate format to assembly utilizes compiler verification techniques similar to those found in, for example [10, 12, 14]. Our compiler is a hybrid of *translation validation* [16] and somewhat more conventional compiler verification technology; one novelty in the back end is that, in contrast to previous work which is typically based on relating the operational or denotational semantics of the source and target object languages, our method performs the validation on the fly by mechanized axiomatic semantics.

For lack of space, we will focus on the intermediate level and back-end of our compiler. A conventional small-step operational semantics for a machine similar to the ARM is defined in Section 3. In Section 4, a semantics for the intermediate format is derived from a compositional semantics for ARM instruction sequences. This allows the derivation of a Hoare logic for the intermediate format. In Section 4.3 that logic is specialized to yield a Hoare logic supporting the task of translation validation, *i.e.*, showing that compilation steps preserve the semantics of the original function. The rules of this logic are suitable for automation by LCF-style tactics and we give some basic examples to indicate how this works.

2 Source Language and Back-end Input Form

Ultimately, our goal is to compile the functional language that dwells in the HOL logic. Roughly speaking, this comprises all ML-style pure terminating functional programs, *i.e.*, those (computable) functions that can be expressed by well-founded recursion in higher order logic [19]. However, in this paper we focus on a simpler subset of higher order logic, which we call *Back-end Input Form (BIF)*. The translation from a source program to *BIF* is validated in the front-end, and there may be multiple source languages that target *BIF*.

BIF is a simple polymorphically-typed functional language handling first order tail-recursive equations where variables range over tuples of elements from types that can be directly represented in machine words for the ARM, *e.g.*, booleans and 32-bit words. 'Let'-binding and function call are also supported. For example, the *BIF* of an arbitrary sequence of encryption rounds in the TEA block cipher [21] is shown in Figure 1.

```
DELTA = 0x9e3779b9w
ShiftXor (x, s, k0, k1) =
  ((x ≪ 4) + k0) # (x + s) #
  ((x ≫ 5) + k1)
Round ((y, z), (k0, k1, k2, k3), s) =
  let s' = s + DELTA in
  let y' = y + ShiftXor (z, s', k0, k1)
  in  ((y', z + ShiftXor (y', s', k2, k3)),
       (k0, k1, k2, k3), s')
Rounds (n, s : state) =
  if n = 0w then s
  else Rounds (n − 1w, Round s)
```

```
Rounds(r0, (r8, r5), (r4, r3, r2, r6), r7) =
let v9 = (op =) (r0, 0w)
in if v9 then ((r8, r5), (r4, r3, r2, r6), r7)
   else let m2 = (op −) (r0, 1w) in
        let m4 = (op +) (r7, 2654435769w) in
        let r1 = ShiftXor (r5, m4, r4, r3) in
        let r9 = (op +) (r8, r1) in
        let r1 = ShiftXor (r9, m4, r2, r6) in
        let r1 = (op +) (r5, r1) in
        let ((m5, m3), (m1, m0, m6, r1), r0) =
          Rounds (m2, (r9, r1), (r4, r3, r2, r6), m4)
        in ((m5, m3), (m1, m0, m6, r1), r0)
```

Fig 1. Rounds in *BIF* (left) and after CPS conversion and register allocation (right)

Various well-known source-to-source translations are employed at the *BIF* level: the input is translated to an equivalent combinatory format, then the combinator form is mapped to *Administrative Normal Form* by performing a CPS transformation. Finally, a standard graph-colouring register allocation phase is invoked. Interestingly, to formally prove the post-register allocation program is equivalent to the original function is very simple, amounting to not much more than checking that the two expressions are α-equivalent. This nice trick was first noticed by Hickey and Nogin [9] and also used by Leroy [12]. It allows the results of standard register allocation algorithms to be used, without having to verify their correctness. In the rest of the paper when mentioning a source program we usually refer to a *BIF* in ANF form and after register allocation.

3 Target Machine

Our goal is to perform a proof to reconcile the source function—an extensional object defined without reference to any notion of evaluation—with the effect of running assembly code on the ARM machine. The underlying basis for this proof

is a version of the ARM machine. (We are currently refining it to the detailed semantics for ARM provided by Fox [5].)

Syntax	**Meaning**
$k \quad ::= 0, 1, \ldots$	Natural numbers (e.g. for addresses)
$v \quad ::= 0w, 1w, \ldots$	Word constants
$r \quad ::= r_0 \mid r_1 \mid ... \mid r_{14}$	Register 0—14
$m[.] ::= m[k] \mid m[r] \mid m[r, k]$	Memory slots, undirect addressing is supported
$d \quad ::= v \mid r \mid m[.]$	Operands
$op \quad ::= mov, add, sub, \ldots$	Operator:move,addition,subtraction,...
$rop ::= eq \mid ne \mid gt \mid lt \mid ge \mid le \mid al \mid nv$	Relation operators: $=, \neq, >, <, \geq, \leq, always, never$
$cnd ::= (d, rop, d)$	Conditions
$asg ::= op\ d \mid op\ d\ d \mid op\ d\ d\ d$	Assignment instruction, abbreviated as $op\ d\ \{\ldots\}$
$cmp ::= cmp\ d\ d \mid tst\ d\ d$	Arithmetic and logical comparison
$jcc ::= b\{rop\}\ (+ \mid -) \mid Addr$	Conditional jumps

Fig.2 ARM instructions

The standard format of an ARM instruction is: $operator\{cond\}\{S\}\ op_1\ op_2$. The *cond* field controls conditional execution of the instruction, it is omitted for unconditional execution; the condition flag is updated if the S field is set; op_1 and op_2 are the destination operand and source operands respectively. Our object language consists of a subset of ARM instructions, the syntax and the semantics domains of which are shown in figure 2 and below respectively.

Name	**Domain Construction**
values, v	$\mathsf{Val} = \text{word32}$
address, k	$\mathsf{Addr} = \mathbb{N}$
program, ρ	$\mathbb{P} = \text{Instr list}$
instr. buffer, π	$\mathsf{IB} = \mathsf{Addr} \rightarrow \text{Instr}$
register buffer, $regs$	$\mathsf{RB} = \{0,1,...,14\} \rightarrow \text{Instr}$
data memories, mem	$\mathsf{DM} = \text{Addr} \rightarrow \mathsf{Val}$
data state, ω	$\Omega = \mathsf{RB} \times \mathsf{DM}$
pc set, $\Cup$	$\mathsf{S}_{pc} = \mathsf{Addr}\ \text{set}$
runtime state, σ	$\Sigma = (\mathsf{Addr} \times \mathsf{Val} \times \Omega) \times \{\mathsf{Addr}\}$

The runtime state of the machine is held in a 5-tuple $((pc, cpsr, regs, mem), \Cup)$ where $pc : \mathsf{Val}$ is a program counter, $cpsr : \mathsf{Val}$ is a process status register, *regs* is a finite map from registers to their contents, *mem* is the memory (a finite map from addresses to their contents), and $\Cup$ records the set of addresses of instructions executed (used for deriving compositional semantics). The convention is: temporaries are allocated into registers r_0-r_9 while r_{10}-r_{14} are reserved for special purpose use. Specifically, r_{10} is used to transfer memory values, while r_{11}, r_{12}, r_{13}, and r_{14} store the values `fp` (frame pointer), `ip` (intra-procedure register pointer), `sp` (stack pointer) and `lr` (link register), respectively. Note that we distinguish data memory from instruction memory (also known as the *instruction buffer*, which is modelled as a function mapping an address to an instruction).

The operational semantics for the language is represented by the `step` function, which executes the instruction pointed to by the *pc*. Instructions are loaded

into the instruction buffer through the upload function. The decode function implements the operational semantics. Note that the pc of an executed instruction is recorded in the $\Cup$ field right after the execution. To model infinite executions of the machine we take advantage of the fact that an unrestricted while loop may be defined in higher order logic [1]. The runTo function continues making steps until it reaches a specific ending point k. An ARM program ρ is executed until the first position beyond the code area is reached. Their definitions and the operational semantics are shown in figure 4.

Notation	**Meaning**
$\epsilon : \mathsf{Val}$	arbitrary value
S	an intermediate representation structure(tree)
f^n	apply function f n times
$f'x$	apply finite map f to argument x
$f \oplus (x, y)$	update finite map f by (x, y), $\doteq \lambda a.$ if $a = x$ then y else $f'a$
${}^{\rho}_{\pi}\Delta^k$	load into an instr. buffer a program, i.e. *upload* ρ π k
$\omega[[\boldsymbol{d}]]$	the value of the variable pair $\boldsymbol{d}$ in ω
$\langle d_1, rop, d_2\rangle : bool$	the value of evaluating d_1 rop d_2
$\lfloor\omega\rfloor$	retrieve the data state from a runtime state
$\|\rho\|$	return the number of instr. in ρ
$\rho_1 \uplus \rho_2$	append ρ_2 to ρ_1
$P[x \leftarrow y]$	substitution of y for x in P

Fig.3 Frequently Used Notations

$\mathsf{upload}\ \rho\ \pi\ k \doteq \quad \mathtt{if}\ null\ \rho\ \mathtt{then}\ \pi\ \mathtt{else}\ \mathsf{upload}\ (tl\ \rho)\ (\lambda i.\mathtt{if}\ i = k\ \mathtt{then}\ hd\ \rho\ \mathtt{else}\ \pi\ i)\ (k+1)$
$\mathsf{step}\ \pi\ ((pc, \omega), \Cup) \doteq (\mathsf{decode}\ \omega\ (\pi\ pc), \Cup \cup \{pc\})$
$\mathsf{runTo}\ \pi\ k\ \sigma \doteq \mathtt{if}\ \sigma.pc = k\ \mathtt{then}\ \sigma\ \mathtt{else}\ \mathsf{runTo}\ \pi\ k\ (\mathsf{step}\ \pi\ \sigma)$
$\mathsf{run\_arm}\ \rho\ \sigma \doteq \mathsf{runTo}\ ({}^{\rho}_{\lambda k.\epsilon}\Delta^{\sigma.pc})\ (\sigma.pc + \|\rho\|)\ \sigma$

Instruction	**Operational Semantics**
Assignment: $op\ x\ \boldsymbol{y}$	$(pc, cpsr, \omega) \rightarrow (pc+1, cpsr, \omega \oplus (x, \langle op\ \boldsymbol{y}\rangle))$
Comparison: $cmp\ d_1\ d_2$	$((pc, cpsr, \omega) \rightarrow (pc+1, cpsr \leftarrow \langle(d_1, eq, d_2)\rangle, \omega))$
Jump: $b\{rop\}\ (+/-)\ k$	$(pc, cpsr, \omega) \rightarrow \begin{cases} (pc\,(+/-)\,k\,, cpsr, \omega) & \mathtt{if}\ \langle rop\ cpsr\rangle \\ (pc+1\,, cpsr, \omega) & \text{otherwise} \end{cases}$

where $\langle op\ \boldsymbol{y}\rangle$ gives the value of the operation on $\boldsymbol{y}$, $cpsr \leftarrow \langle(d_1, eq, d_2)$ sets the comparison results, and $\langle rop\ cpsr\rangle$ returns the result of previous comparison

Fig.4 Operation Semantics of the Machine Language

4 Intermediate Representation

Validation of the translation from high-level language programs to low-level codes requires the ability to reason about low-level language programs. However, low-level languages like ARM are widely believed to be difficult to reason about due to low-level code being flat and to the prominent presence of unrestricted jumps. Fortunately, a compilation produces low level code structurally by combining smaller pieces of code together to generate larger code. Technically, we can formulate a structured version for a sequence of low level instructions and

develop a compositional semantics for it such that the compositional semantics agrees with the standard non-compositional small-step operational semantics of that sequence.

In [18] a compositional natural semantics along with a Hoare logic of a structured version SGoto is developed for the basic low-level language Goto, then the compilation from WHILE to SGoto is given. In [20] a continuation-style logic with a rather sophisticated interpretation of Hoare triples involving explicit fixpoint approximations is proposed. We now discuss a compositional semantics for the flat code, introduced to facilitate reasoning.

4.1 Compositional Semantics

The flat code corresponding to a source program p is generated in accordance to the control flow structures of p. Five structures are recognized during compilation: BLK (Basic Block), SC (Sequential Composition), CJ (Conditional Jump), TR (Tail Recursion) and FC (Function Call). They are defined by the following datatype *ir* (for *intermediate representation*).

$$\begin{array}{l} \mathsf{ir} = \mathsf{BLK}\ \texttt{of}\ \mathsf{assign\ list} \quad \mid \mathsf{CJ}\ \texttt{of}\ \mathsf{condition} * \mathsf{ir} * \mathsf{ir} \\ \quad \mid \mathsf{TR}\ \texttt{of}\ \mathsf{condition} * \mathsf{ir} \quad \mid \mathsf{SC}\ \texttt{of}\ \mathsf{ir} * \mathsf{ir} \\ \quad \mid \mathsf{FC}\ \texttt{of}\ (\mathsf{exp\ list} * \mathsf{exp\ list}) * \mathsf{ir} * (\mathsf{exp\ list} * \mathsf{exp\ list}) \end{array}$$

A BLK structure is just a list of atomic assignments. An FC consists of an argument passing pair (the first component is for the caller, the second component for is the callee), a body *ir*, and a result passing pair. An *ir* will never contain any comparison or jump instructions. We prefix the operator in an assigment with "m" to indicate that it is on the *ir* level ("m" stands for "macro"). For example, the *BIF* and the *ir*-representation of the factorial function are

$$\begin{array}{l} fact\ (x, a) \doteq \texttt{if}\ x = 0w\ \texttt{then}\ a\ \texttt{else}\ fact\ (x - 1w, x \times a) \\ fact\_ir = \mathsf{SC}\ (\mathsf{TR}\ (r_0, eq, 0w)\ (\mathsf{BLK}\ [msub\ r_3\ r_0\ 1w;\ mmul\ r_2\ r_0\ r_1; \\ \qquad\qquad mmov\ r_0\ r_3;\ mmov\ r_1\ r_2]))(mmov\ r_2\ r_1)) \end{array}$$

As the *ir* is just a high-level representation of the structures of the flat code, what's the semantics of an *ir* program? The answer depends on how to translate *ir*s. The semantics of an *ir* program is determined by the operational semantics of the translated code; and different translators lead to different semantics for the same *ir* program. A translator from *ir*-trees can be easily written and is shown below (we discuss function call later).

$$\begin{array}{l} \mathsf{translate}(\mathsf{BLK}\ (stm :: stmL)) \doteq \mathsf{translate\_assignment}\ stm :: \mathsf{translate}(\mathsf{BLK}\ stmL) \\ \mathsf{translate}(\mathsf{BLK}\ []) \doteq [] \\ \mathsf{translate}(\mathsf{SC}\ S_1\ S_2) \doteq \mathsf{mk\_SC}\ (\mathsf{translate}\ S_1)\ (\mathsf{translate}\ S_2) \\ \mathsf{translate}\ (\mathsf{CJ}\ cond\ S_{true}\ S_{false}) \doteq \mathsf{mk\_CJ}\ cond\ (\mathsf{translate}\ S_{true})\ (\mathsf{translate}\ S_{false}) \\ \mathsf{translate}\ (\mathsf{TR}\ cond\ S_{body}) \doteq \mathsf{mk\_TR}\ cond\ (\mathsf{translate}\ S_{body}) \end{array}$$

where $\mathsf{mk\_SC}\ \rho_1\ \rho_2 \doteq \rho_1 \uplus \rho_2$ and

$$\begin{array}{l} \mathsf{mk\_CJ}\ (v_1, rop, v_2)\ \rho_t\ \rho_f \doteq \\ \quad (\mathsf{cmp}\ v_1\ v_2) :: \\ \quad (\mathsf{b}\{rop\}\ +\ (\|\rho_f\| + 2)) :: \rho_f \uplus \\ \quad [\mathsf{bal}\ +\ (\|\rho_t\| + 1)]\ \uplus \rho_t \end{array} \qquad \begin{array}{l} \mathsf{mk\_TR}\ (v_1, rop, v_2)\ \rho \doteq \\ \quad (\mathsf{cmp}\ v_1\ v_2) :: \\ \quad (\mathsf{b}\{rop\} + (\|\rho\| + 2)) :: \rho \uplus \\ \quad [\mathsf{bal} - (|\rho| + 2)] \end{array}$$

An ir-level execution runs the result of translate (i.e. flat code) on the machine, and it observes the data state after the execution

$$\mathsf{run\_ir}\ S\ \omega \doteq \lfloor \mathsf{run\_arm}\ (\mathsf{translate}\ S)\ ((0, 0w, \omega), \{\}) \rfloor$$

The following theorem is essentially trivial, but it serves to organize the translation validation proof: on the left hand size, the definition of translate is used to compile the program, and on the right, rules for run_ir (see below) are used to push the property down to individual ARM instructions.

$$\vdash_{hol}\ \forall P \forall S \forall \omega. P(\lfloor \mathsf{run\_arm}\ (\mathsf{translate}\ S)\ (pc, c, \omega) \rfloor) = P(\mathsf{run\_ir}\ S\ \omega) \tag{1}$$

To support reasoning about execution of flat code, we use the following definition of Hoare triples:

$$\{P\}\ S\ \{Q\} \doteq \forall \omega. P\ \omega \Rightarrow Q(\mathsf{run\_ir}\ S\ \omega)$$

(This is a partial correctness specification, total correctness will be discussed in the next section.) We have derived the following Hoare rules in HOL to infer properties of decomposable structures (note that in our definition the body of a TR structure keeps running when the condition doesn't hold):

$$\frac{\{P\}\ S_1\ \{Q\} \quad \{R\}\ S_2\ \{T\} \quad Q \Rightarrow R}{\{P\}\ (\mathsf{SC}\ S_1\ S_2)\ \{T\}}\ \mathsf{SC} \qquad \frac{\{P\}\ S\ \{P\}}{\{P\}\ (\mathsf{TR}\ C\ S)\ \{P \wedge C\}}\ \mathsf{TR}$$

$$\frac{\{P \wedge C\}\ S_t\ \{Q\} \quad \{P \wedge \neg C\}\ S_f\ \{Q\}}{\{P\}\ (\mathsf{CJ}\ C\ S_t\ S_f)\ \{Q\}}\ \mathsf{CJ1} \qquad \frac{\{P\}\ S_t\ \{Q\} \quad \{P\}\ S_f\ \{R\}}{\{P\}\ (\mathsf{CJ}\ C\ S_t\ S_f)\ \{\mathtt{if}\ C\ \mathtt{then}\ Q\ \mathtt{else}\ R\}}\ \mathsf{CJ2}$$

Fig 5. Hoare rules for ir

By application of these rules we obtain a derived evaluation semantics for ir-trees (as HOL theorems), which indicates our translator works properly:

$\vdash_{hol}$ run_ir (BLK($stm :: stmL$)) $= \lambda\omega.$run_ir (BLK $stmL$) (mdecode ω stm)
$\vdash_{hol}$ run_ir (BLK []) $= \lambda\omega.\omega$
$\vdash_{hol}$ run_ir (BLK($stm :: stmL$)) $= \lambda\omega.$run_ir (SC (BLK[stm]) (BLK $stmL$))
$\vdash_{hol}$ run_ir (SC S_1 S_2) $= \lambda\omega.$run_ir S_2 (run_ir S_1 ω)
$\vdash_{hol}$ run_ir (CJ $cond$ S_t S_f) $= \lambda\omega.$if eval_cond $cond$ ω then run_ir S_t ω else run_ir S_f ω
$\vdash_{hol}$ run_ir(TR $cond$ S_{body}) $= \lambda\omega.$while $(\lambda\sigma.\neg(eval\_cond\ cond\ \sigma))$ $(\lambda\sigma.$run_ir S_{body} $\sigma)$ ω

4.2 Well-formedness and Total Correctness

We now describe definitions and theorems used to derive the Hoare rules; they are mostly self-explanatory. Theorem (2) says a ARM program is free to be uploaded into any area of the instruction buffer without affecting the execution result. As the basic composition theorem, (3) enables us to break the simulation of a large code segment into multiple simulations of small code segments such that the sequential composition of the simulations of these small code segments exhibits the same behaviors as that of the large one. Together with (4), it leads

to (5), which performs composition on data states. Theorem (6) shows that it doesn't matter whatever the initial instruction buffer is.

$$\vdash_{hol}\ \forall\rho\forall k_1\forall k_2\forall\pi\forall i.\, i < \|\rho\| \Rightarrow (\overset{\rho}{\pi}\Delta^{k_1}\ (k_1+i) = (\overset{\rho}{\pi}\Delta^{k_2}\ (k_2+i)) \tag{2}$$

$$\vdash_{hol}\ \forall i\forall k\forall\pi\forall s_0\forall \mathbb{U}_0\ \forall s_1\forall \mathbb{U}_1\ .\mathtt{let}\ (s_1,\mathbb{U}_1) = \mathsf{runTo}\ \pi\ j\ (s_0,\mathbb{U}_0)\ \mathtt{in}$$
$$k \notin (\{s_0.pc\} \cup \mathbb{U}_1) \Rightarrow (\mathsf{runTo}\ \pi\ k\ (s_0,\mathbb{U}_0) = \mathsf{runTo}\ \pi\ k\ (s_1,\mathbb{U}_1)) \tag{3}$$

$$\vdash_{hol}\ \forall\rho\forall\rho_1\forall\rho_2\forall k\forall\pi\forall\sigma.\,\mathsf{closed}\ \rho \wedge \mathsf{terminated}\ \rho \wedge (k + \|\rho_1\| = \sigma.pc) \Rightarrow$$
$$\mathsf{runTo}\ (\overset{\rho_1 \uplus \rho \uplus \rho_2}{\pi}\Delta^{k})\ (\sigma.pc + \|\rho\|)\ \sigma = \mathsf{runTo}\ \overset{\rho}{\pi}\Delta^{\sigma.pc}\ (\sigma.pc + \|\rho\|)\ \sigma \tag{4}$$

$$\vdash_{hol}\ \forall\rho\forall\rho'\forall k_0\forall k_1\forall c_0\forall c_1\forall c_2\forall\pi\forall\omega.\,\mathsf{well\_formed}\ \rho \wedge \mathsf{well\_formed}\ \rho' \Rightarrow$$
$$\lfloor \mathsf{runTo}\ \overset{\rho\uplus\rho'}{\pi}\Delta^{k_0}\ (k_0 + \|\rho\| + \|\rho'\|)\ ((k_0,c_0,\omega),\{\})\rfloor =$$
$$\lfloor \mathsf{runTo}\ (\overset{\rho'}{\pi}\Delta^{k_2})\ (k_2 + \|\rho'\|)\ ((k_2,c_2,\lfloor \mathsf{runTo}\ (\overset{\rho}{\pi}\Delta^{k_1})\ (k_1+\|\rho\|)\ ((k_1,c_1,\omega),\{\})\rfloor,\{\})\rfloor \tag{5}$$

$$\vdash_{hol}\ \forall\rho\forall\pi_0\forall\pi_1\forall\sigma.\,\mathsf{well\_formed}\ \rho \Rightarrow$$
$$\mathsf{runTo}\ \overset{\rho}{\pi_0}\Delta^{\sigma.pc}\ (\sigma.pc + \|\rho\|)\ \sigma = \mathsf{runTo}\ \overset{\rho}{\pi_1}\Delta^{\sigma.pc}\ (\sigma.pc + \|\rho\|)\ \sigma \tag{6}$$

Our translation always generates code satisfying some 'good' properties. An ARM program is *well-formed* if and only if any execution of it will access only its own instructions, any execution is terminated and the data state after an execution is independent of the initial values of *pc* and *cpsr*. We have shown that all programs generated by our compiler are well-formed.

$$\mathsf{well\_formed}\ \rho \doteq \mathsf{closed}\ \rho \wedge \mathsf{terminated}\ \rho \wedge \mathsf{status\_independent}\ \rho$$
$$\mathsf{closed}\ \rho \doteq \forall s\forall\pi\forall k.k \in (\mathsf{runTo}\ (\overset{\rho}{\pi}\Delta^{s.pc})\ (s.pc + \|\rho\|)\ (s,\{\})).\mathbb{U} \Rightarrow$$
$$s.pc \le k\ \wedge\ k < s.pc + \|\rho\|$$
$$\mathsf{terminated}\ \rho \doteq \forall\sigma\forall\pi.\exists n.((\mathsf{step}\ (\overset{\rho}{\pi}\Delta^{s.pc}))^n\ \sigma).pc = \sigma.pc + \|\rho\|$$
$$\mathsf{status\_independent}\rho \doteq \forall\omega\forall k_0\forall k_1\forall c_0\forall c_1\forall\pi.$$
$$\langle \mathsf{runTo}\ (\overset{\rho}{\pi}\Delta^{k_0})\ (k_0+\|\rho\|)\ ((k_0,c_0,\omega),\{\})\rangle = \langle \mathsf{runTo}\ (\overset{\rho}{\pi}\Delta^{k_1})\ (k_1+\|\rho\|)\ ((k_1,c_1,\omega),\{\})\rangle$$

$$\vdash_{hol}\ \forall S.\mathsf{well\_formed}\ (\mathsf{translate}\ S)$$

In particular, the total correctness specification requires proving the object code terminates. This problem is especially sophisticated for tail recursion programs. The key is to find a well-founded (`WF`) relation between the intial state and the state after an execution of the code. Fortunately, HOL-4 provides a number of basic and advanced means of specifying wellfounded relations and proving termination. The following theorem reduces the problem of proving the termination of flat code to a proof of the termination of the original *BIF* logic function, which can be discharged automatically, or interactively if need be, by the TFL package [19] in HOL-4.

$$\vdash_{hol}\ (\forall\omega.\,cnd_f\ (prj_f\ \omega) = \mathtt{eval\_cond}\ cnd\ \omega) \wedge (\forall\omega.prj_f\ (\mathsf{run\_ir}\ S\ \omega) = f\ (prj_f\ \omega)) \wedge$$
$$(\exists R.\,\mathsf{WF}\ R \wedge (\forall t_0\forall t_1.\,\neg(cnd_f\ t_0) \Rightarrow R\ (f\ t_0)\ t_0)) \Rightarrow$$
$$\mathsf{WF}(\lambda\omega_1\lambda\omega_0.\neg(\mathtt{eval\_cond}\ cnd\ \omega_0) \wedge (\omega_1 = \mathsf{run\_ir}\ S\ \omega_0))$$

4.3 Translation Validation over IR

We now have to bridge the semantic gap between the original *BIF* function g with inputs $\boldsymbol{i}$ and outputs $\boldsymbol{o}$, and the results obtained about S, the *ir* representation of g, embodied in Theorem 1. The statement to be proved amounts to the following:

$$\vdash_{hol}\ \forall\omega.(\mathsf{run\_ir}\ S\ \omega)[[\boldsymbol{o}]] = g\ (\omega[[\boldsymbol{i}]])$$

In other words, the theorem asserts that the translated code has the same semantics as the original source program.

We now specialize the axiomatic semantics of *ir*s to translation validation, obtaining a refined set of Hoare rules. A *projective* Hoare rule, denoted by

$$S \vdash p \hookrightarrow \xi \ \uparrow \ (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow q$$

says: provided that predicate p holds in state ω, and inputs $\boldsymbol{i}$ have initial values $\boldsymbol{v}$ in ω, and the variables in stack ξ have values $\boldsymbol{x}$, then in the state after the execution of S, predicate q holds, and the values left in outputs $\boldsymbol{o}$ are equal to applying the function f to the initial values $\boldsymbol{v}$, and all variables in ξ have the same values $\boldsymbol{x}$ as in ω. Formally,

$$\begin{array}{l} S \vdash p \hookrightarrow \xi \ \uparrow \ (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow q \doteq \\ \quad \forall \boldsymbol{x} \forall \boldsymbol{v} \forall \omega . (p\ \omega) \wedge (\boldsymbol{i}_f\ \omega = \boldsymbol{v}) \wedge (\omega[[\xi]] = \boldsymbol{x}) \Rightarrow \\ \quad \texttt{let}\ \omega' = \texttt{run\_ir}\ S\ \omega\ \texttt{in}\ (q\ \omega') \wedge (\boldsymbol{o}_f\ \omega' = f\ \boldsymbol{v}) \wedge (\omega'[[\xi]] = \boldsymbol{x}) \end{array}$$

where function i_f and o_f project from a data state the values of vector $\boldsymbol{i}$ and $\boldsymbol{o}$. Such a rule is obtained by instantiating the P and Q in $\{P\}\ S\ \{Q\}$ to, for any $\boldsymbol{x}$ and $\boldsymbol{v}$, be $\lambda\omega.p\ \omega \wedge (\omega[[\xi]] = \boldsymbol{x}) \wedge (\omega[[\boldsymbol{i}]] = \boldsymbol{v})$ and $\lambda\omega.q\ \omega \wedge (\omega[[\xi]] = \boldsymbol{x}) \wedge (\omega[[\boldsymbol{o}]] = f\ \boldsymbol{v})$ respectively. If the judgement embodied by a projective Hoare rule holds on the S derived from original *BIF* function g, then the synthesized function f should be equivalent to g and, indeed this is easy to prove automatically since they are quite similar.

The projective Hoare rules utilize the following definitions. The product of two vectors makes a new vector: $\boldsymbol{v_1} \times \boldsymbol{v_2} \doteq (\boldsymbol{v_1}, \boldsymbol{v_2})$; the dot product of a function and vector gives a new function: $(\lambda \boldsymbol{x}.f\ \boldsymbol{x}) \odot \boldsymbol{v} \doteq \lambda(\boldsymbol{x}, \boldsymbol{v}).(f\ \boldsymbol{x}, \boldsymbol{v})$. A vector and a projective function are interchangeable. When both p and q are $\lambda\omega.true$, a rule may be simplified to be $S \vdash \xi \ \uparrow \ (\boldsymbol{i}, f, \boldsymbol{o})$. The projective Hoare rules used for mechanical reasoning are (all have been proved in HOL):

$$\frac{S_1 \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i_1}, f_1, \boldsymbol{o_1}) \hookrightarrow Q \qquad S_2 \vdash Q \hookrightarrow \xi \uparrow (\boldsymbol{o_1}, f_2, \boldsymbol{o_2}) \hookrightarrow R}{\texttt{SC}\ S_1\ S_2 \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i_1}, f_2 \circ f_1, \boldsymbol{o_2}) \hookrightarrow R}\ \text{sc}$$

$$\frac{S_1 \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, f_1, \boldsymbol{o}) \hookrightarrow Q \qquad S_2 \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, f_2, \boldsymbol{o}) \hookrightarrow Q}{\texttt{CJ}\ cnd\ S_1\ S_2 \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, (\texttt{if}\ \langle cnd \rangle\ \texttt{then}\ f_1\ \texttt{else}\ f_2), \boldsymbol{o}) \hookrightarrow Q}\ \text{cj}$$

$$\frac{S \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{i}) \hookrightarrow P}{\texttt{TR}\ cnd\ S \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, (\texttt{while}\ (\neg \langle cnd \rangle)\ f), \boldsymbol{i}) \hookrightarrow P}\ \text{tr}$$

$$\frac{S \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q \qquad R \Rightarrow P}{S \vdash R \hookrightarrow \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q}\ \text{strengthen} \qquad \frac{S \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q \qquad Q \Rightarrow R}{S \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow R}\ \text{weaken}$$

$$\frac{S \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q \qquad g\ \boldsymbol{i'} = f\ \boldsymbol{i}}{S \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i'}, g, \boldsymbol{o}) \hookrightarrow Q}\ \text{shuffle} \qquad \frac{S \vdash P \hookrightarrow \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q \qquad \mathsf{intact}(\boldsymbol{\alpha}, P, Q)}{S \vdash P \hookrightarrow (\boldsymbol{\alpha}; \xi) \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q}\ \text{push}$$

$$\frac{S \vdash P \hookrightarrow (\xi_1; \boldsymbol{\alpha}; \xi_2) \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q}{S \vdash P \hookrightarrow (\xi_1; \boldsymbol{\alpha}; \xi_2) \uparrow (\boldsymbol{i} \times \boldsymbol{\alpha}, f \odot \boldsymbol{\alpha}, \boldsymbol{o} \times \boldsymbol{\alpha}) \hookrightarrow Q}\ \text{pick} \qquad \frac{S \vdash P \hookrightarrow (\xi_1; \boldsymbol{\alpha}; \xi_2) \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q}{S \vdash P \hookrightarrow (\xi_1; \xi_2) \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \hookrightarrow Q}\ \text{remove}$$

where $\mathsf{intact}(\boldsymbol{\alpha}, P, Q) \doteq \forall \omega . (P\ \omega\ \Rightarrow\ \omega[[\boldsymbol{\alpha}]] = \boldsymbol{v})\ \wedge\ (Q\ \omega\ \Rightarrow\ \omega[[\boldsymbol{\alpha}]] = \boldsymbol{v})$

Rules sc, cj and tr are self-explanatory; strengthen and weaken are traditional pre-condition strengthening and post-condition weakening Hoare rules respectively. push and remove are used to push a variable into a stack and delete a variable

from the stack respectively. The empty stack is represented by $\rrbracket$. A stack ξ can be divided into multiple parts: $\xi_1, \xi_2, ...$ The pushing of value v into ξ leads to $v;\xi$, while the removing of v from $\xi_1; v; \xi_2$ results in $\xi_1; \xi_2$. The picking of v from $\xi_1; v; \xi_2$ returns v with the stack intact. Rule shuffle is to restructure the input vector. A basic block is simulated as a whole as it is a macro instruction, thus there exists no rule for it.

In the examples we give to illustrate the use of these rules, some functions are used frequently:

$f_+ = \lambda(a,b).a + b \quad f_\times = \lambda(a,b).a * b \quad f_{id} = \lambda a.a$
$\mathsf{fst} = \lambda(a,b).a \quad \mathsf{snd} = \lambda(a,b).b$

Successful application of these rules for automatic translation validation requires the ability to accurately identify the inputs, outputs and the stack variables for an *ir* structure. An intuitive way to do so for an assignment instruction is to let the inputs and outputs be the source and destination of this instrution, and the values of all variables to be used later in the entire scope (i.e. those still "alive") should be stored in the stack. For instance, consider the two basic blocks: $blk_1 \doteq \mathsf{BLK}\ [madd\ r_2\ r_4\ r_2]$ and $blk_2 \doteq \mathsf{BLK}[mmul\ r_1\ r_2\ r_1]$. One validation tree for the sequential compostion of these two blocks is shown below, which implies the semantics function of $\mathsf{SC}\ blk_1\ blk_2 = \mathsf{BLK}\ [madd\ r_2\ r_4\ r_2; mmul\ r_1\ r_2\ r_1]$ is $\lambda((v_0,v_1),v_2).v_2 \times (v_0 + v_1)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{blk_1 \vdash (r1;\rrbracket) \uparrow ((r_4,r_2),f_+,r_2)}{blk_1 \vdash (r1;\rrbracket) \uparrow ((r_4,r_2),r_1),(\lambda((v_0,v_1),v_2).(v_0+v_1,v_2)),(r_2,r_1)}\ \text{pick}}{blk_1 \vdash \rrbracket \uparrow ((r_4,r_2),r_1),(\lambda((v_0,v_1),v_2).(v_0+v_1,v_2)),(r_2,r_1)}\ \text{remove} \quad \dfrac{blk_1 \vdash \rrbracket \uparrow ((r_1,r_2),f_\times,r_1) \qquad (\lambda(v_0,v_1).v_1 \times v_0)\ (r_2,r_1) = f_\times\ (r_1,r_2)}{blk_1 \vdash \rrbracket \uparrow ((r_2,r_1),(\lambda(v_0,v_1).v_1 \times v_0),r_1)}\ \text{shuffle}}{\mathsf{SC}\ blk_1\ blk_2 \vdash \rrbracket \uparrow (((r_4,r_2),r_1),\ (\lambda(v_0,v_1).v_1 \times v_0) \circ (\lambda((v_0,v_1),v_2).(v_0+v_1,v_2)),\ r_1)}\ \text{sc}$$

The `pick` and `remove` operation could be avoided: another derivation can be made by setting both the output of blk_1 and the input of blk_2 to be (r_2, r_1); then by applying the *sc* rule once we obtain the same result. Our compiler finds for an *ir* structure the appropriate inputs and outputs according to the inputs and outputs of the *ir*s surrounding it (if such exist) and tries to match them as much as possible, thus eliminating unneccessary application of the `pick`, `remove` and `shuffle` rule. The stack will not store those temporary variables that won't be used in the subsequent control flow.

The Compiler From an initial *BIF* definition of function g in HOL, our compiler translates to ANF and does register allocation (deductively). The resulting function, still in *BIF*, is analyzed to obtain the *ir* format S. Then Theorem 1 is instantiated with S, and (a) rewriting with translate produces the flat code *flatcode* and (b) bottom-up application of the projective rules synthesizes f. The equality of f and g (both HOL functions) is then proved automatically, and finally the compiler returns the theorem

$$\vdash_{hol}\ \forall\omega.(\lfloor \mathsf{run\_arm}\ flatcode\ (0,0w,\omega)\rfloor)\ [[\boldsymbol{o}]] = f\ (\omega[[\boldsymbol{i}]])$$

The compilation and validation process is fully automatic;[2] sample outputs produced by the compiler may be found in Appendix A.2.

Application of projective rules is controlled by the *ir* structure, annotated with inputs, outputs and context information (so called *annotated ir*), and the verifier utilizes this information to guide the symbolic simulation and the application of rules. Control flow rules `sc`, `cj` and `tr` are applied on structures `SC`, `CJ` and `TR` respectively. For instance, when reasoning about a (CJ *cond* S_1 S_2) structure, we first reason about S_1 and S_2 separately, then apply the `cj` rule. The application of data flow rules `pick`, `remove` and `shuffle` are guided by the "use" and "def" information of a structure maintained by the compiler.

Example 1: Factorial Function
Let

$$body \doteq \mathsf{BLK}[msub\ r_3\ r_0\ 1w;\ mmul\ r_2\ r_0\ r_1;\ mmov\ r_0\ r_3;\ mmov\ r_1\ r_2]$$
$$blk_1 \doteq \mathsf{BLK}[mmov\ r_2\ r_1]$$
$$f_1 \doteq \lambda(v_0, v_1).(v_0 - 1w, v_0 + v_1) \qquad f_2 \doteq \mathsf{while}\ (\neg\langle(r_0, ne, 0w)\rangle)\ f_1$$

Symbolic evaluation of the instructions in *body* and blk_1 proves

$$\vdash_{hol}\ \forall\omega.\mathsf{run\_ir}\ body = \omega \oplus (r_3, \omega[[r_0]] - 1w) \oplus (r_2, \omega[[r_0]] + \omega[[r_1]]) \oplus (r_0, \omega[[r_0]] - 1w) \oplus (r_1, \omega[[r_0]] + \omega[[r_1]])$$
$$\vdash_{hol}\ \forall\omega.\mathsf{run\_ir}\ blk_1 = \omega \oplus (r_2, \omega[[r_0]])$$

From these theorems and by the definition of the projective Hoare rules we have

$$rule_1 = body\ \vdash\ \beth\ \uparrow\ ((r_0, r_1), f_1, (r_0, r_1)) \qquad rule_2 = blk_1\ \vdash\ \beth\ \uparrow\ ((r_0, r_1), \mathsf{snd}, r_2))$$

Then the derivation of the specification on $fact_{ir}$ is

$$\dfrac{\dfrac{rule_1}{\mathsf{TR}\ (r_0, ne, 0w)\ body\ \vdash\ \beth\ \uparrow\ ((r_0, r_1), f_2, (r_0, r_1))}\ \text{tr} \qquad rule_2}{\mathsf{SC}\ (\mathsf{TR}\ (r_0, ne, 0w)\ body)\ blk_1\ \vdash\ \beth\ \uparrow\ ((r_0, r_1), \mathsf{snd} \circ f_2, r_2)}\ \text{sc}$$

The semantics function in this specification

$$\mathsf{snd} \circ f_2 = (\lambda(v_0, v_1).v_1) \circ (\mathsf{while}\ (\neg\langle(r_0, ne, 0w)\rangle)\ (\lambda(v_0, v_1).(v_0 - 1w, v_0 + v_1)))$$

assembles the internal representation of the definition of $fact$ and their equivalence is verified automatically by applying α conversion and function composition, hence verifying the correctness of $fact$'s translation.

$$\vdash_{hol} \forall\omega.\ \mathsf{run\_ir}\ fact\_ir\ \omega = \lfloor(\mathsf{run\_arm}\ (\mathsf{translate}\ fact\_ir)\ ((0, 0w, \omega), \{\})\rfloor[[r_2]] = fact(\omega[[r_0]], \omega[[r_1]])$$

where translate $fact\_ir$ generates, by the definition of translate, the following ARM flat code with inputs (r_0, r_1) and outputs r_2

```
[cmp r0,r1; beq + 6; sub r3,r0,1w; mul r2,r0,r1; mov r0,r3; mov r1,r2; bal - 6; mov r2,r1]
```

Example 2 : Conditional Jump

Source Function: $cj\_f\ (a, b) \doteq$ `let` $c = a + 1w$ `in if` $a = 1w$ `then` c `else` $c + b$

The derivation applies cj once and sc once

$$blk_1 \doteq \mathsf{BLK}\ [madd\ r_2\ r_0\ 1w] \qquad blk_2 \doteq \mathsf{BLK}\ [madd\ r_2\ r_2\ r_1]$$
$$f_1 \doteq \lambda(v_0, v_1, v_2).\mathsf{if}\ v_0 = 1w\ \mathsf{then}\ v_2\ \mathsf{else}\ v_2 + v_1$$

[2] Source code along with examples is included in the 'examples/dev/sw/working' directory in the HOL-4 distribution (`http://hol.sourceforge.net`).

$$\frac{\mathsf{BLK}[] \vdash \beth \uparrow ((r_0, r_1, r_2), (\lambda(v_0, v_1, v_2).v_2), r_2) \qquad blk_2 \vdash \beth \uparrow ((r_0, r_1, r_2), (\lambda(v_0, v_1, v_2).v_2 + v_1), r_2)}{\mathsf{CJ}\ (r_0, eq, 1w)\ (\mathsf{BLK}[])\ blk_2 \vdash \beth \uparrow ((r_0, r_1, r_2), f_1, r_2)}\ \text{cj}$$

$$\frac{blk_1 \vdash \beth \uparrow ((r_0, r_1), (\lambda(v_0, v_1).(v_0, v_1, v_0 + 1w)), (r_0, r_1, r_2))}{\mathsf{SC}\ blk_1\ (\mathsf{CJ}\ (r_0, eq, 1w)\ (\mathsf{BLK}[])\ blk_2) \vdash \beth \uparrow ((r_0, r_1), (f_1 \circ (\lambda(v_0, v_1).(v_0, v_1, v_0 + 1w))), r_2)}\ \text{sc}$$

The resulting semantics function is equivalent to cj_f:

$(\lambda(v_0, v_1, v_2).\mathsf{if}\ v_0 = 1w\ \mathsf{then}\ v_2\ \mathsf{else}\ v_2 + v_1) \circ (\lambda(v_0, v_1).(v_0, v_1, v_0 + 1w))$
$= \lambda(v_0, v_1).\mathsf{if}\ v_0 = 1w\ \mathsf{then}\ v_0 + 1w\ \mathsf{else}\ (v_0 + 1w) + v_1 = cj\_f$

4.4 Function Call

Function calls are compiled into a callee-saves style calling convention complying with the ARM procedure call standard. The entire procedure for a function call consists of pre-call processing, callee execution and post-call processing:

Pre-call processing	**Post-call processing**
1. the caller pushes the arguments into ξ	1. the callee pushes the results into ξ
2. the callee saves registers's values	2. the callee restores values stored in ξ
3. the callee pops the arguments out from ξ	3. the caller obtains the results from ξ

We provide two ways to validate the translation of a FC structure. The first one converts this structure into SC structures and then applies the SC rules (where copy *dst src* is a list of instructions for copying the values of *src* to *dst*). Then the verification for a fc structure is accomplished by applying the sc rule twice.

$\mathsf{FC}\ (caller.\boldsymbol{i}, callee.\boldsymbol{i})\ body\ (caller.\boldsymbol{o}, callee.\boldsymbol{o}) \rightarrow$
$\quad \mathsf{SC}\ (\mathsf{SC}\ (\mathsf{copy}\ callee.\boldsymbol{i}\ caller.\boldsymbol{i})\ body)\ (\mathsf{copy}\ caller.\boldsymbol{o}\ callee.\boldsymbol{o})$

The second way predefines a routine to pass arguments and return results, and gives proofs showing that this routine does work. The main theorem is a single fc rule (where $args = (caller.\boldsymbol{i}, f, callee.\boldsymbol{i})$ and $results = (caller.\boldsymbol{o}, f, callee.\boldsymbol{o})$).

$$\frac{S \vdash P \hookrightarrow \xi \uparrow (callee.\boldsymbol{i}, f, callee.\boldsymbol{o}) \hookrightarrow Q}{\mathtt{FC}\ args\ S\ results \vdash P[callee.\boldsymbol{i} \leftarrow caller.\boldsymbol{i}] \hookrightarrow \xi \uparrow (caller.\boldsymbol{i}, f, caller.\boldsymbol{o}) \hookrightarrow Q[callee.\boldsymbol{o} \leftarrow caller.\boldsymbol{o}]}$$

This rule results from the fact that the caller's stack and callee's stack locate in *separate* areas in the memory. Two areas in the memory are *separate* if the domains of them don't intersect, thus modification of one area won't affect the other. The notation $mem \mid D$ represents the projection of the contents in mem onto domain D. Since in the routine argument/result passing and callee's computation are performed at different areas from the caller's stack, the proof of the `fc` rule is straightforward. Furthermore, we introduce another *ir* level to hide memory from the programs and manage heap and stacks explicitly[3].

$\mathsf{separate}\ D_1\ D_2 \doteq D_1 \cap D_2 = \emptyset$
$\vdash_{hol}\ \mathsf{separate}\ D_1\ D_2 \Rightarrow \forall i \in D_2\, \forall v.mem \mid D_1 = (mem \oplus (i, v)) \mid D_1$

Example 3 : Function Call (using the first way)

Source Functions : $\quad f_1 \doteq \lambda x.x + x + 1w \quad f_2 \doteq \lambda x.x * f_1\ x$

[3] See the `IL1` and `FunCall` theories in the HOL-4 distribution for details.

The pre-call processing *pre* and post-call processing *post* perform the argument passing (from r_0 to r_0) and result passing (from r_0 to r_1) respectively. The value of r_0 will be used after the call, thus it is stored in the stack in *pre*, then in *post* it is restored from the stack.

$pre \doteq \mathsf{BLK}\ [msub\ sp\ sp\ 1w; mstr\ [sp]\ r_0; msub\ sp\ sp\ 1w; mmov\ ip\ sp;$
$\quad mpush\ sp\ \{r_0, fp, ip, lr, pc\}; msub\ fp\ ip\ 1w; mldr\ r_0\ [ip,\ 1]; madd\ ip\ ip\ 1w]$
$body \doteq \mathsf{BLK}\ [madd\ r_0\ r_0\ r_0; madd\ r_0\ r_0\ 1w]$
$post \doteq \mathsf{BLK}\ [madd\ sp\ fp\ 3w; mstr\ [sp]\ r_0; msub\ sp\ sp\ 1w; mldr\ r_1\ [sp, 1];$
$\quad madd\ sp\ sp\ 1w; msub\ sp\ fp\ 4w; mpop\ sp\ r_0, fp, sp, pc]$
$blk_1 \doteq \mathsf{BLK}\ [mmul\ r_0\ r_0\ r_1]$
$S_1 \doteq \mathsf{SC}\ pre\ body \qquad S_2 \doteq \mathsf{SC}\ S_1\ post \qquad g_1 \doteq \lambda v_0.v_0 + v_0 + 1w$
$g_2 \doteq \lambda(v_0, v_1).(f_1\ v_0, v_1)\ p_1 \doteq \lambda\omega.\omega_0[[r_0]] = \boldsymbol{x}\ p_2 \doteq \lambda\omega.\omega_0[[fp - 3]] = \boldsymbol{x}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{pre \vdash p_1 \hookrightarrow \beth \uparrow (r_0, f_{id}, r_0) \hookrightarrow p_2 \qquad body \vdash p_2 \hookrightarrow \beth \uparrow (r_0, g_1, r_0) \hookrightarrow p_2}{S_1 \vdash p_1 \hookrightarrow \beth \uparrow (r_0, g_1 \circ f_{id}, r_0) \hookrightarrow p_2}\ \text{sc} \quad post \vdash p_2 \hookrightarrow \beth \uparrow (r_0, f_{id}, r_1) \hookrightarrow p_1}{S_2 \vdash p_1 \hookrightarrow \beth \uparrow (r_0, f_{id} \circ g_1 \circ f_{id}, r_1) \hookrightarrow p_1 \qquad \mathsf{intact}(r_0, p_1, p_1)}\ \text{sc}}{S_2 \vdash (r_0; \beth) \uparrow (r_0, f_{id} \circ g_1 \circ f_{id}, r_1)}\ \text{push}}{S_2 \vdash (r_0; \beth) \uparrow ((r_0, r_0), g_2, (r_1, r_0))}\ \text{pick}}{S_2 \vdash \beth \uparrow ((r_0, r_0), g_2, (r_1, r_0))\quad \text{(spec 1)}}\ \text{remove}$$

$$\dfrac{\dfrac{blk_1 \vdash \beth \uparrow ((r_1, r_0), f_\times, r_0) \qquad \text{(spec 1)}}{\mathsf{SC}\ S_2\ blk_1 \vdash \beth \uparrow ((r_0, r_0), f_\times \circ g_2, r_0)}\ \text{sc} \qquad \mathsf{BLK}[] \vdash \beth \uparrow (r_0, (\lambda v_0.(v_0, v_0)), (r_0, r_0))}{\mathsf{SC}\ (\mathsf{BLK}[])\ (\mathsf{SC}\ S_2\ blk_1) \vdash \beth \uparrow (r_0, (f_\times \circ g_2 \circ (\lambda v_0.(v_0, v_0))), r_0)}\ \text{sc}$$

Clearly the way we treat a function call is to unroll it at the IR level. One consequence of this is that a function will be unrolled multiple times in the proof if it has multiple call sites. Fortunately, after the unrolled version is verified, we can perform an additional code deployment step to extract the instructions of each callee's body and relocate them to other positions. All occurrences of the code of the same callee should be relocated to the same position, thus only one copy is left. The following theorem justifies this relocating (we assume that the code of the callee ρ_{callee} has included proper returning mechanism, e.g. the last instruction is $ldmfd\ sp!, \{\ldots, fp, sp, pc\}$, so that the *pc* will point to the first instruction after the code of the function call):

$\vdash_{hol} \lfloor \mathsf{run\_arm}\ (\rho_1 \uplus \rho_{callee} \uplus \rho_2 \uplus \rho_{callee} \uplus \rho_3)\ \sigma \rfloor =$
$\quad \lfloor \mathsf{run\_arm}\ (\rho_1 \uplus [bl\ +\ (\|\rho_2\| + \|\rho_3\| + 2)] \uplus \rho_2 \uplus [bl\ +\ (\|\rho_3\| + 1)] \uplus \rho_3 \uplus \rho_{callee}]\ \sigma \rfloor$

5 Related Work

As mentioned, we have also developed a separate hardware compiler for a similar source language [7]. Compilation in that system proceeds essentially by refinement steps: control structures in logic are refined by unclocked circuits implementing those structures, and those circuits can be further refined to be clocked circuits. Finally, the circuits are prettyprinted to Verilog.

Hickey and Nogin [9] have implemented a compiler from a full higher order, untyped, functional language which generates x86 code, based entirely on higher-order rewrite rules. The compiler is written in the MetaPRL logical framework. They don't verify the rewrite rules, and how to apply them in a correct order to produce object code aren't specified either.

Leroy [12] verifies a compiler for a subset of `C` in the Coq system. This work is thoroughly based in operational semantics, and it is really a hybrid of compiler verification and translation validation. For example, Leroy uses a graph colouring register allocator as an oracle, sidestepping a difficult correctness proof. A purely operational semantics based development is that of Klein and Nipkow [10] which gives a thorough formalization of a Java-like language, including a compiler. However, that compiler targets fairly high-level code, and assumes an unbounded number of registers. Compilation from C-like programs to DLX assembly code is verified using the Isabelle/HOL theorem prover [11].

There has recently been a large amount of work on verification of low-level languages, originally prompted by the ideas of proof carrying code and typed assembly language [15]. We are currently investigating links with recent work on Hoare Logics for assembly language, *e.g.*, [4, 13] and also extensions such as Separation Logic [17]. Of course, compiler verification itself is a venerable topic, with far too many publications to survey (see Dave's bilbliography [3]). Restricting to assembler verification, one of the most relevant works for us is by Moore [14].

6 Summary and Future Work

We have shown how a tail-recursive functional language over simple data (i.e. the *BIF*) can be compiled by proof. This allows a smooth passage from recursively defined logical functions to ARM assembly language that is guaranteed to implement those functions. The size of the development of this back-end excluding data structures and non-core components is shown below.

Main Component	**#Definition**	**#Theorem**	**#Line(approx.)**
CPS/ANF Conversion	10	14	900
Machine Model	96	116	1700
Compositional Semantics	14	48	1750
IR and Axiomatic Semantics	96	56	1450
Rules	24	42	600
Function Call	62	51	1350
Mechanical Reasoning (tactics)			850
Compilation Facilities			1550
Total	302	327	10150

Currently, we are strengthening the front end translation to support higher order functions and ML-style datatypes. In the back end, we are finalizing a better approach to the heap, the stack, and function call. In the future, we must of course tackle memory allocation and garbage collection. Finally, we are investigating the formalization of properties of programs that manipulate shared mutable data structures by extending Hoare logic to separation logic.

References

1. Stefan Berghofer and Tobias Nipkow, *Executing Higher Order Logic*, Types for Proofs and Programs, International Workshop, (TYPES 2000), no. 2277.
2. Amine Chaieb and Tobias Nipkow, *Verifying and reflecting quantifier elimination for Presburger arithmetic*, Proceedings of 12th Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2005), 2005.
3. Maulik A. Dave, *Compiler verification: a bibliography*, ACM SIGSOFT Software Engineering Notes **28** (2003), no. 6.
4. Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni, *Modular verification of assembly code with stack-based control abstractions.*, PLDI, 2006, pp. 401–414.
5. Anthony Fox, *Formal verification of the ARM6 micro-architecture*, Tech. Report 548, University of Cambridge Computer Laboratory, November 2002.
6. Georges Gonthier, *A computer checked proof of the four colour theorem*, Tech. report, Microsoft Research Cambridge, 2005.
7. M. Gordon, J. Iyoda, S. Owens, and K. Slind, *Automatic formal synthesis of hardware from higher order logic*, Proceedings of Fifth International Workshop on Automated Verification of Critical Systems (AVoCS), ENTCS, vol. 145, 2005.
8. D. Greve, M. Wilding, and D. Hardin, *High-speed, analyzable simulators*, Computer-Aided Reasoning Case Studies (M. Kaufmann, P. Manolios, and J Moore, eds.), Kluwer Academic Publishers, 2000, pp. 113–135.
9. Jason Hickey and Aleksey Nogin, *Formal compiler construction in a logical framework*, Journal of Higher-Order and Symbolic Computation (2006), to appear.
10. Gerwin Klein and Tobias Nipkow, *A machine-checked model for a java-like language, virtual machine and compiler*, TOPLAS, to appear.
11. Dirk Leinenbach, Wolfgang Paul, and Elena Petrova, *Towards the formal verification of a c0 compiler: Code generation and implementation correctnes*, Software Engineering and Formal Methods (SEFM), 2005.
12. Xavier Leroy, *Formal certification of a compiler backend, or: programming a compiler with a proof assistant*, Proceedings of POPL 2006, ACM Press, 2006.
13. John Matthews, Sandip Ray J Strother Moore, and Daron Vroon, *symbolic simulation approach to assertional program verification*, 2006.
14. J Strother Moore, *Piton: A mechanically verified assembly-level language*, Automated Reasoning Series, Kluwer Academic Publishers, 1996.
15. Greg Morrisett, David Walker, Karl Crary, and Neal Glew, *From System F to typed assembly language*, ACM Transactions on Programming Languages and Systems **21** (1999), no. 3, 527–568.
16. A. Pnueli, M. Siegel, and E. Singerman, *Translation validation*, Proceedings of TACAS '98, 1998.
17. John C. Reynolds, *Separation logic: A logic for shared mutable data structures*, LICS, 2002, pp. 55–74.
18. Ando Saabas and Tarmo Uustalu, *A compositional natural semantics and hoare logic for low- level languages*, SOS 2005.
19. Konrad Slind, *Reasoning about terminating functional programs*, Ph.D. thesis, Institut für Informatik, Technische Universität München, 1999.
20. Gang Tan and Andrew W. Appel, *A compositional logic for control flow*, Verification, Model Checking and Abstract Interpretation (VMCAI 2006), LNCS.
21. David Wheeler and Roger Needham, *TEA, a tiny encryption algorithm*, Fast Software Encryption: Second International Workshop, 1999.

A Appendix

A.1 Compilation of TEA

The *BIF* of the TEA [21] block cipher includes the following first order logic functions (pretty-printed to be in conventional ARM format):

```
ShiftXor (x,s,k0,k1) =
  ((x << 4) + k0) # (x + s) # ((x >> 5) + k1)

Round ((y,z),(k0,k1,k2,k3),s):state =
  let s' = s + DELTA in
  let y' = y + ShiftXor(z, s', k0, k1)
  in
   ((y', z + ShiftXor(y', s', k2, k3)),
    (k0,k1,k2,k3),s')

Rounds (n,s:state) =
  if n=0w then s else Rounds (n-1w, Round s)

TEAEncrypt (keys,txt) =
   let (cipheredtxt,keys,sum) =
         Rounds(32w,(txt,keys,0w))
   in cipheredtxt
```

Our compiler generates the following object code:

```
Name : TEAEncrypt
Arguments : r0 r1 r2 r3 r4 r5
Modified Registers : r0 r1 r2 r3 r4 r5 r6
Returns : r6 r5
Body:
0:  mov ip, sp
1:  stmfd sp!, {fp,ip,lr,pc}
2:  sub fp, ip, #1i
3:  sub sp, sp, #7i
4:  mov r10, #0iw
5:  str r10, [sp]
6:  sub sp, sp, #1i
7:  stmfd sp!, {r4,r5,r0,r1,r2,r3}
8:  mov r10, #32iw
9:  str r10, [sp]
10:  sub sp, sp, #1i
11:  bl + (6)
12:  add sp, sp, #8i
13:  ldmfd sp, {r6,r5,r4,r3,r2,r1,r0}
14:  add sp, sp, #7i
15:  sub sp, fp, #3i
16:  ldmfd sp, {fp,sp,pc}
17:  mov ip, sp
18:  stmfd sp!, {r0,r1,r2,r3,r4,r5,r6,r7,
                 r8,r9,fp,ip,lr,pc}
19:  sub fp, ip, #1i
20:  sub sp, sp, #7i
21:  ldmfd ip, {r0,r8,r5,r4,r3,r2,r6,r7}
22:  add ip, ip, #8i
23:  cmp r0, #0iw
24:  beq + (37)
25:  sub r1, r0, #1iw
26:  str r1, [fp, #~11]
27:  add r1, r7, #2654435769iw
28:  str r1, [fp, #~12]
29:  sub sp, sp, #1i
30:  stmfd sp!, {r4,r3}
31:  ldr r10, [fp, #~12]
32:  str r10, [sp]
33:  str r5, [sp, #~1]
34:  sub sp, sp, #2i
35:  bl + (32)
36:  add sp, sp, #4i
37:  ldr r1, [sp, #1]
38:  add sp, sp, #1i
39:  add r9, r8, r1
40:  sub sp, sp, #1i
41:  stmfd sp!, {r2,r6}
42:  ldr r10, [fp, #~12]
43:  str r10, [sp]
44:  str r9, [sp, #~1]
45:  sub sp, sp, #2i
46:  bl + (21)
47:  add sp, sp, #4i
48:  ldr r1, [sp, #1]
49:  add sp, sp, #1i
50:  add r1, r5, r1
51:  ldr r10, [fp, #~12]
52:  str r10, [sp]
53:  sub sp, sp, #1i
54:  stmfd sp!, {r9,r1,r4,r3,r2,r6}
55:  ldr r10, [fp, #~11]
56:  str r10, [sp]
57:  sub sp, sp, #1i
58:  ldmfd sp, {r0,r8,r5,r4,r3,r2,r6,r7}
59:  add sp, sp, #8i
60:  bal - (37)
61:  mov r1, r6
62:  mov r0, r7
63:  add sp, fp, #16i
64:  stmfd sp!, {r8,r5,r4,r3,r2,r1,r0}
65:  sub sp, fp, #13i
66:  ldmfd sp, {r0,r1,r2,r3,r4,r5,r6,r7,
                 r8,r9,fp,sp,pc}
67:  mov ip, sp
68:  stmfd sp!, {r0,r1,r2,r3,r4,fp,ip,lr,pc}
69:  sub fp, ip, #1i
70:  ldmfd ip, {r0,r1,r2,r3}
71:  add ip, ip, #4i
72:  lsl r4, r0, #4i
73:  add r2, r4, r2
74:  add r1, r0, r1
75:  eor r1, r2, r1
76:  asr r0, r0, #5i
77:  add r0, r0, r3
78:  eor r0, r1, r0
79:  add sp, fp, #6i
80:  str r0, [sp]
81:  sub sp, sp, #1i
82:  sub sp, fp, #8i
83:  ldmfd sp, {r0,r1,r2,r3,r4,fp,sp,pc}
```

A.2 Real Outputs of examples

This section shows the real ouputs in HOL-4 of the correctness statement for the examples appeared in this paper

Example 1 (Factorial)

```
|- !st.
(get_st (run_arm
   [((CMP,NONE,F),NONE,[REG 0; WCONST 0w],NONE);
    ((B,SOME EQ,F),NONE,[],SOME (POS 6));
    ((SUB,NONE,F),SOME (REG 3),[REG 0; WCONST 1w],NONE);
    ((MUL,NONE,F),SOME (REG 2),[REG 0; REG 1],NONE);
    ((MOV,NONE,F),SOME (REG 0),[REG 3],NONE);
    ((MOV,NONE,F),SOME (REG 1),[REG 2],NONE);
    ((B,SOME AL,F),NONE,[],SOME (NEG 6));
    ((MOV,NONE,F),SOME (REG 2),[REG 1],NONE)]
   ((0,0w,st),{}))<MR R2> =
 fact (st<MR R0>,st<MR R1>)
```

Example 2

```
|- !st.
(get_st (run_arm
   [((ADD,NONE,F),SOME (REG 2),[REG 0; WCONST 1w],NONE);
    ((CMP,NONE,F),NONE,[REG 0; WCONST 1w],NONE);
    ((B,SOME EQ,F),NONE,[],SOME (POS 3));
    ((ADD,NONE,F),SOME (REG 2),[REG 2; REG 1],NONE);
    ((B,SOME AL,F),NONE,[],SOME (POS 1))]
   ((0,0w,st),{}))<MR R2> =
 cj_f (st<MR R0>,st<MR R1>))
```

Example 3 (Function Call)

```
|- !st.
 proper st ==>
(get_st (run_arm
  [((SUB,NONE,F),SOME (REG 13),[REG 13; WCONST 1w],NONE);
   ((STR,NONE,F),SOME (REG 0),[MEM (13,POS 0)],NONE);
   ((SUB,NONE,F),SOME (REG 13),[REG 13; WCONST 1w],NONE);
   ((MOV,NONE,F),SOME (REG 12),[REG 13],NONE);
   ((STMFD,NONE,F),SOME (WREG 13),
      [REG 0; REG 11; REG 12; REG 14; REG 15],NONE);
   ((SUB,NONE,F),SOME (REG 11),[REG 12; WCONST 1w],NONE);
   ((LDR,NONE,F),SOME (REG 0),[MEM (12,POS 1)],NONE);
   ((ADD,NONE,F),SOME (REG 12),[REG 12; WCONST 1w],NONE);
   ((ADD,NONE,F),SOME (REG 0),[REG 0; REG 0],NONE);
   ((ADD,NONE,F),SOME (REG 0),[REG 0; WCONST 1w],NONE);
   ((ADD,NONE,F),SOME (REG 13),[REG 11; WCONST 3w],NONE);
   ((STR,NONE,F),SOME (REG 0),[MEM (13,POS 0)],NONE);
   ((SUB,NONE,F),SOME (REG 13),[REG 13; WCONST 1w],NONE);
   ((LDR,NONE,F),SOME (REG 1),[MEM (13,POS 1)],NONE);
   ((ADD,NONE,F),SOME (REG 13),[REG 13; WCONST 1w],NONE);
   ((SUB,NONE,F),SOME (REG 13),[REG 11; WCONST 4w],NONE);
   ((LDMFD,NONE,F),SOME (WREG 13),
     [REG 0; REG 11; REG 13; REG 15],NONE);
   ((MUL,NONE,F),SOME (REG 0),[REG 0; REG 1],NONE)]
    ((0,0w,st),{}))<MR R0> =
 f2 (st<MR R0>)))
```

A.3 Mechanical Reasoning on the Example 1 (Factorial Function)

We show in this section details of how our mechanical verifier applies symbolic simulation and projective Hoare rules to accomplish the translation validation of the factorial function.
An *annotated ir* tree for this source function is generated during the compilation

```
SC(TR((REG 0, eq, WCONST0i),
         \konst{BLK}([{dst = [REG 3], oper = msub, src = [REG 0, WCONST1i]},
              {dst = [REG 2], oper = mmul, src = [REG 0, REG 1]},
              {dst = [REG 0], oper = mmov, src = [REG 3]},
              {dst = [REG 1], oper = mmov, src = [REG 2]}],
             {context = [], fspec = |- T, ins = PAIR(REG 0, REG 1),
              outs = PAIR(REG 0, REG 1)}),
          {context = [NA], fspec = |- T, ins = PAIR(REG 0, REG 1),
           outs = PAIR(REG 0, REG 1)}),
       BLK([{dst = [REG 2], oper = mmov, src = [REG 1]}],
           {context = [], fspec = |- T, ins = PAIR(REG 0, REG 1),
            outs = REG 2}),
       {context = [], fspec = |- T, ins = PAIR(REG 0, REG 1), outs = REG 2})
```

The verifier applies rules according to the structure of this *annotated ir*. At first it descends to the body of the TR structure and simulates this block symbolically to get a specification:

```
Simulating a MSUB instruction
Simulating a MMUL instruction
Simulating a MMOV instruction
Simulating a MMOV instruction
> val it =
    |- let ir = BLK
                [MSUB R3 (MR R0) (MC 1w); MMUL R2 (MR R0) (MR R1);
                 MMOV R0 (MR R3); MMOV R1 (MR R2)]
       in
           PSPEC ir ((\st. T),(\st. T)) (\st. T)
             ((\st. (st<MR R0>,st<MR R1>)), (\(v0,v1). (v0 + 4294967295w,v0 * v1)),
              (\st. (st<MR R0>,st<MR R1>))) /\ WELL_FORMED ir : thm
```

Then the `tr` rule is applied to obtain:

```
 |- let ir = TR (REG 0,EQ,WCONST 0w)
                (BLK
                   [MSUB R3 (MR R0) (MC 1w); MMUL R2 (MR R0) (MR R1);
                    MMOV R0 (MR R3); MMOV R1 (MR R2)])
    in
        PSPEC ir ((\st. T),(\st. T)) (\st. T)
          ((\st. (st<MR R0>,st<MR R1>)),
           WHILE ($~ o (\(v0,v1). v0 = 0w)) (\(v0,v1). (v0 + 4294967295w,v0 * v1)),
           (\st. (st<MR R0>,st<MR R1>))) /\ WELL_FORMED ir : thm
```

Next the block following the `TR` structure is simulated to generate another specification

```
Simulating a MMOV instruction
 |- let ir = BLK [MMOV R2 (MR R1)] in
         PSPEC ir ((\st. T),(\st. T)) (\st. T)
           ((\st. (st<MR R0>,st<MR R1>)),(\(v0,v1). v1),(\st. st<MR R2>)) /\
         WELL_FORMED ir : thm
```

And then the `sc` rule is applied to get the final specification

```
 |- let ir = SC
                (TR (REG 0,EQ,WCONST 0w)
                   (BLK
                      [MSUB R3 (MR R0) (MC 1w); MMUL R2 (MR R0) (MR R1);
                       MMOV R0 (MR R3); MMOV R1 (MR R2)]))
                (BLK [MMOV R2 (MR R1)])
       in
           PSPEC ir ((\st. T),(\st. T)) (\st. T)
             ((\st. (st<MR R0>,st<MR R1>)), (\(v0,v1). v1) o
              WHILE ($~ o (\(v0,v1). v0 = 0w)) (\(v0,v1). (v0 + 4294967295w,v0 * v1)),(\st. st<MR R2>)) /\
           WELL_FORMED ir : thm
```

Finally the semantics function contained in this specification is proved to be equal to the source function with the assistance of pre-proved theorems about the `WHILE` combinator.

```
 |- (\(v0,v1). v1) o WHILE ($~ o (\(v0,v1). v0 = 0w)) (\(v0,v1). (v0 + 4294967295w,v0 * v1)) =
       fact : thm
```