Structure of a Proof-Producing Compiler for a Subset of Higher Order Logic

Guodong Li, Scott Owens, and Konrad Slind

School of Computing, University of Utah

Abstract. We give an overview of a proof-producing compiler which translates recursion equations, defined in higher order logic, to assembly language. The compiler is implemented and validated with a mix of translation validation and compiler verification techniques. Both the design of the compiler and its mechanical verification are implemented in the same logic framework.

1 Introduction

Most compilers are used to compile programs. However, it also makes sense to execute logic [1], and thus to compile logic. This is the basis for logic programming, where search for solutions to problems phrased as logic formulas is the dominant paradigm [12]. In this paper we address another—hitherto unexploited—opportunity for logic compilation; namely, the term language that dwells within higher order logic [16,17]. This language comprises, roughly speaking, ML-style pure terminating functional programs, *i.e.*, those (computable) functions that can be expressed by well-founded recursion in higher order logic [21]. Features like type inference, polymorphism, and pattern matching make this subset a comfortable setting in which to program. Although this language does not contain all computable functions, it does express a very wide range of algorithms and, of course, the logic provides a setting for correctness proofs of such programs.

Compilation techniques developed for functional programming may be applied to translate these programs to machine code. However, since we are in a formal setting, it is natural to ask for more, namely the formal correctness of compilation. There are two main approaches to achieving this high level of assurance: compiler verification and translation validation. Compiler verification proceeds by formally specifying, in the object logic, the source and target languages, along with the compilation algorithm. Then the correctness of the compiler is proven once and forall: a single object logic theorem establishes that all successful runs of the compiler generate correct code. In contrast, translation validation [18] does a per-run correctness proof. Its main advantage is that only the results of compilation steps need to be verified, which can at times be far simpler than verifying the algorithms performing the compilation.

We have built a proof-producing compiler for a simple subset of higher order logic terms in the HOL-4 proof system [17]. The compiler is mainly based on translation validation, but compiler verification techniques such as those found in

R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 205–219, 2007.

[9,11,14] are also used. A run of the compiler returns an (automatically proved) theorem expressing the correctness of the compilation run; from this theorem the generated code, for an ARM-like machine, can be directly read-off.

The task of compiling the term language of a logic using the logic itself poses a couple of novel challenges: first, the source language is not visible in the logic; second, there is no notion of evaluation for the logic. Source functions have a set-theoretic semantics which has to be reconciled with the operational semantics of the target machine.

In the remainder of the paper, we give an overview of the structure of the compiler, and summarize our experiences to date.[1]

2 Overview

One immediate advantage of taking logic terms as the source language is that many front end tasks are already provided by the HOL-4 system: lexical analysis, parsing, type inference, overloading resolution, function definition, and termination proof (needed to admit recursive functions, since HOL is a logic of total functions). The result of all this activity is a valid HOL function definition, embodied in a possibly recursive equation. From this starting point, a sequence of proof-based transformations pass through intermediate languages, ending in assembly. We will describe four intermediate languages: *HOL-* (HOL Minor), *ANF/ACF* (Administrative Normal Form / A Combinator Form), *HSL* (Heap and Stack Level), and *CFL* (Control Flow Level). HOL-, ANF and ACF programs are simply HOL functions, with no attached operational semantics. It is this feature that enables us to use standard mathematics to prove properties of HOL- and ANF programs directly in HOL. HSL and CFL, on the other hand, are imperative languages represented with syntax trees and operational semantics.

The translation from a source function to HOL- is performed and validated in the front-end in an *ad hoc* manner; in fact there may be multiple source languages that target HOL-. The translation from HOL- to ANF/ACF is mainly expressed as a collection of verified rewrite rules. Currently, the translation from HOL- to ANF/ACF includes performing closure conversion, CPS conversion, and register allocation in that order. ANF is used for the compilation to HSL, while ACF is for the validation of such compilation. ACF is obtained from ANF through verified rewriting. The result is a theorem equating the original function with the ACF translation of its body.

An ANF-format program is converted (not by proof) to a corresponding HSL program, which is in turn converted to its CFL by laying out the heap and the stacks. Finally, the CFL is translated to ARM-like object code by linearizing the control-flow structures. Roughly speaking, the path from HOL- to HSL proceeds by translation validation, while the other steps rely on compiler verification techniques.

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

Since we do not have an evaluation semantics for HOL- or ACF, widely-used techniques for proving semantics preservation for the translation, *e.g.*, simulation arguments based on rule-induction over the evaluation relation, are not applied to validate the translation from ACF to HSL. Instead, we derive a collection of Hoare rules from the operational semantics of HSL and show that this semantics agrees with the ACF level function by bottom-up reasoning. Thus, for an ACF function g with inputs $\boldsymbol{i}$ and outputs $\boldsymbol{o}$, and the HSL program S_{hsl} obtained from g, the following statement must be proved (where $\sigma[[v]]$ reads the value of variable v from state σ):

$$\vdash_{thm} \ \forall \sigma_{\text{hsl}}.\ (\mathsf{run}_{\text{hsl}}\ S_{hsl}\ \sigma_{\text{hsl}})\ [[\boldsymbol{o}]] = g\ (\sigma_{\text{hsl}}\ [[\boldsymbol{i}]])$$

HSL states are defined over virtual registers, heap variables and stack variables, while CFL states range over machine registers and machine memory locations. The correctness of the translation from HSL to CFL is phrased by relating the states of these two languages by a relation $\backsimeq$. The correctness statement asserts that the execution of a HSL statement S_{hsl} has the same effect on a HSL state as the execution of its corresponding CFL statement S_{cfl}:

$\vdash_{def}\ (\sigma_{\text{hsl}} \backsimeq \sigma_{\text{cfl}}) \doteq (\forall v \in \sigma_{\text{hsl}}.\sigma_{\text{hsl}}[[v]] = \sigma_{\text{cfl}}[[v']])$ where v' is v's injection into σ_{cfl}
$\vdash_{thm}\ \sigma_{\text{hsl}} \backsimeq \sigma_{\text{cfl}} \Rightarrow (\mathsf{run}_{\text{hsl}}\ S_{\text{hsl}}\ \sigma_{\text{hsl}} \backsimeq \mathsf{run}_{\text{cfl}}\ S_{\text{cfl}}\ \sigma_{\text{cfl}})$

The runtime state σ_{arm} for the machine is a tuple of a program counter (pc), a process status register ($cpsr$), physical registers and physical memory (ω). If a CFL program S_{cfl} is correctly translated to an ARM program S_{arm}, then the execution of S_{cfl} and S_{arm} should result in the same status of registers and memory, thus any property proved at the CFL level can be pushed down to the ARM level:

$$\vdash_{thm}\ \mathsf{run}_{\text{cfl}}\ S_{\text{cfl}}\ \sigma_{\text{cfl}} = (\mathsf{run}_{\text{arm}}\ S_{\text{arm}}\ (pc, pcsr, \sigma_{\text{cfl}})).\omega$$

Collecting all correctness statements together gives the validation proof for the translation from HOL- to ARM: for a HOL- function g with inputs $\boldsymbol{i}$ and outputs $\boldsymbol{o}$, and the final flat code S_{arm} obtained from g, in the state after the execution of S_{arm}, the values left in outputs $\boldsymbol{o}$ are equal to applying the function g to the initial values of inputs $\boldsymbol{i}$ in σ_{arm}

$$\vdash_{thm}\ \forall \sigma_{\text{arm}}.(\mathsf{run}_{\text{arm}}\ S_{\text{arm}}\ \sigma_{\text{arm}})[[\boldsymbol{o}]] = g\ (\sigma_{\text{arm}}[[\boldsymbol{i}]])$$

3 Language Syntax and Semantics

In Figures 1-3 we give the syntax of the intermediate languages. HOL- is a simple polymorphically-typed functional language handling 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, λ expression and function call are also supported.

ANF is obtained from HOL- by performing closure conversion to eliminate higher order functions, and a CPS (continuation-passing style) transformation so that all expressions are flattened and the control flow is pinned down into a sequence of elementary steps. Register allocation is performed on a data structure obtained by analyzing the ANF program. This ANF program is also rewritten to its ACF form that is a 'constructor'-like semantic function.

$op_b ::= +$ | $-$ | $*$ | $\ggg$ | $\gg$ | $\gtrdot$ | $\ll$ | $\&$ | $\mid$ | $\#$		arithmetic / bitwise operator
$op_r ::= =$ | $\neq$ | $<$ | $>$ | $\leq$ | $\geq$		relational operator
$op_l ::= \wedge$ | $\vee$ | $\neg$		logic operator
e	$::= w \mid v$	word and variable identifier
	$\mid \overrightarrow{e}$	tuple, i.e.$(\ldots, e, \ldots)$
	$\mid e\ op_b\ e \mid e\ op_r\ e \mid e\ op_l\ e$	binary operation
	$\mid \lambda v.\, e$	anonymous function
	$\mid$ if e then e else e	conditional
	$\mid$ let $\overrightarrow{v}\ =\ e$ in e	let definition
	$\mid e\ e$	application
	$\mid f$	named function
f	$::= f_{id}\ \overrightarrow{v} = e$	function definition

$x ::= w \mid v$		word and variable identifier
$e ::= \overrightarrow{x}$		tuple
	$\mid$ (op op_b) $x\ \ x \mid$ (op op_r) $x\ \ x$	binary operation
	$\mid$ if v then e else e	conditional on single variable
	$\mid$ let $v\ =\ e$ in e	let assignment to single variable
	$\mid$ let $\overrightarrow{v}\ = f\ e$ in e	function call
$f ::= f_{id}\ \overrightarrow{v} = e$		function definition

$x, f ::=$	similar to x, f in ANF	
y	$::= \overrightarrow{x} \mid y\ op_b\ y$	data processing operation
z	$::= \lambda\overrightarrow{v}.\, y \mid \lambda\overrightarrow{v}.f\ \overrightarrow{x}$	data processing function
c	$::= \lambda\overrightarrow{v}.\, (x, op_r, x)$	conditional function
e	$::= z \mid$ sc $e\ e \mid$ cj $c\ e\ e \mid$ tr $c\ e\ e$	compositional function

Fig. 1. Syntax of HOL- (top), ANF (middle) and ACF (bottom)

As mentioned, HOL-, ANF, and ACF programs are mathematical functions with no associated evaluation semantics. They can be understood as λ expressions, and the order of reductions is not specified on them.

HSL is a simple imperative language that supports various structured control statements including blocks (BLK), sequential composition (SC), conditionals (CJ) and tail recursion (TR), plus an important structure for function call—FC. Variables are divided into register variables, heap (global) variables, and stack (local) variables. A BLK structure is just a list of atomic instructions. An FC structure consists of an argument passing pair (the first component is for the caller, the second component for is the callee), a body statement, and a result passing pair. Heap variables are not allowed in parameters or results since their values are not transferred through the stack. A HSL program will never contain any comparison or jump instructions. Variables are divided into register variables, heap variables and stack variables. Variables in ANF format have been mapped to either register, heap or stack variables by register allocation and interprocedural analysis. In our current implementation, heap (global) variables are replaced with stack variables during closure conversion, thus actually no heap variable appears in the HSL.

CFL explicitly lays outs the heap and stacks for function calls. It specifies machine registers and memory locations for the variables in HSL. A function call in HSL is implemented by dividing the processing into three phases: pre-call processing, function body execution and post-call processing. Pointer registers `hp` (heap pointer), `fp` (frame pointer), `ip` (intra-procedure register pointer), `sp` (stack pointer) and `lr` (link register) are used to control the layout of the heap and stack frames for functions. CFL works over machine registers and memory, thus a (one-to-one) mapping from HSL variables to them is required.

The translation from CFL to the object code simply performs the linearization of control-flow structures. The format of an ARM instruction is: $op\{cond\}\ d_1\ d_2$. The *cond* field controls conditional execution of the instruction, it is omitted for unconditional execution; d_1 and d_2 are the destination operand and source operands respectively.

$\mathtt{op}^{\mathtt{b}}$	$::= add \mid sub \mid mul \mid ror \mid lsr \mid asr \mid$ $\mid lsl \mid and \mid orr \mid eor \mid rsb \mid mla \ldots$	arithmetic and bitwise operators
r	$::= r_0 \mid r_1 \mid \ldots \mid r_8$	register variable
v	$::= r \mid sk[.]$	register and stack variable
y	$::= w \mid r$	word constant and register
x	$::= w \mid v$	constant and variable
$inst$	$::= \mathtt{op}^{\mathtt{b}}\ r\ y\ y$	arithmetic and bitwise operation
	$\mid ldr\ r\ (\widetilde{hp[i]} \mid sk[.]) \mid str\ (hp[i] \mid sk[.])\ r$	access to heap and stack
s	$::= \mathsf{BLK}\ \widetilde{inst}$	basic block containing an instr. list
	$\mid \mathsf{CJ}\ (x, op_r, x)\ s\ s$	conditional jump
	$\mid \mathsf{TR}\ (x, op_r, x)\ s$	tail recursion (loop)
	$\mid \mathsf{FC}\ (\widetilde{x}, \widetilde{v})\ s\ (\widetilde{v}, \widetilde{x})$	function call
p	$::= (\ \overrightarrow{v}, s, \overrightarrow{x}\ )$	programs

r_d	$::= \mathsf{HSL}.r$	data register
r_b	$::= hp \mid fp \mid ip \mid sp \mid lr$	base (pointer) register
r	$::= r_d \mid r_b$	register
m	$::= m[r_b, +i] \mid m[r_b, -i]$	memory location
v, y, x, p	::= similar to v, y, x, p in HSL	
$inst$	$::= \mathtt{op}^{\mathtt{b}}\ r\ y\ y \mid ldr\ r\ m \mid str\ m\ r \mid push\ \widetilde{r} \mid pop\ \widetilde{r}$	single instruction
s	$::= \mathsf{BLK}\ \widetilde{inst} \mid \mathsf{CJ}\ (x, op_r, x)\ s\ s \mid \mathsf{TR}\ (x, op_r, x)\ s$	control flow structures

$$\frac{}{\mathsf{BLK}\ [] \vdash \sigma \rightarrowtail \sigma} \qquad \frac{\mathtt{eval\_inst}\ inst\ \sigma = \sigma_1 \quad \mathsf{BLK}\ instL \vdash \sigma_1 \rightarrowtail \sigma_2}{\mathsf{BLK}\ (inst{::}instL) \vdash \sigma \rightarrowtail \sigma_2}$$

$$\frac{S_1 \vdash \sigma \rightarrowtail \sigma_1 \quad S_2 \vdash \sigma_1 \rightarrowtail \sigma_2}{\mathsf{SC}\ S_1\ S_2 \vdash \sigma \rightarrowtail \sigma_2} \qquad \frac{S_1 \vdash \sigma \rightarrowtail \sigma_1 \quad \mathtt{is\_true}\ (\mathtt{eval\_cond}\ cond\ \sigma)}{\mathsf{CJ}\ cond\ S_1\ S_2 \vdash \sigma \rightarrowtail \sigma_1}$$

$$\frac{S_2 \vdash \sigma \rightarrowtail \sigma_1 \quad \mathtt{is\_false}\ (\mathtt{eval\_cond}\ cond\ \sigma)}{\mathsf{CJ}\ cond\ S_1\ S_2 \vdash \sigma \rightarrowtail \sigma_1} \qquad \frac{\mathtt{is\_true}\ (\mathtt{eval\_cond}\ cond\ \sigma)}{\mathsf{TR}\ cond\ S \vdash \sigma \rightarrowtail \sigma}$$

$$\frac{S \vdash \sigma \rightarrowtail \sigma_1 \quad \mathtt{is\_false}(\mathtt{eval\_cond}\ cond\ \sigma) \quad \mathsf{TR}\ cond\ S \vdash \sigma_1 \rightarrowtail \sigma_2}{\mathsf{TR}\ cond\ S \vdash \sigma \rightarrowtail \sigma_2}$$

$$\frac{\mathtt{copy}\ (\sigma_\epsilon, \sigma)\ (callee.\boldsymbol{i}, caller.\boldsymbol{i}) = \sigma_1 \quad S \vdash \sigma_1 \rightarrowtail \sigma_2 \quad \mathtt{copy}\ (\sigma, \sigma_2)\ (caller.o, callee.i) = \sigma_3}{\mathsf{FC}\ (caller.\boldsymbol{i}, callee.\boldsymbol{i})\ S\ (caller.\boldsymbol{o}, callee.\boldsymbol{o}) \vdash \sigma \rightarrowtail \sigma_3}$$

Fig. 2. Syntax for HSL (top) and CFL (middle), and evaluation rules (bottom) (Note: FC structures only appear in HSL)

In our machine model, the data memory is separated from instruction memory (also known as the *instruction buffer*, which is modeled as a function mapping an address to an instruction). At each step the instruction pointed to by the *pc* is executed. A program is executed until the first position beyond the code area is reached.

$$
\begin{array}{lll}
r & ::= \mathsf{CFL}.r \mid pc & \text{machine register} \\
m, v, y, x & ::= \text{similar to } m, v, y, x \text{ in } \mathsf{CFL} & \\
inst & ::= b\{op_r\} + k \mid b\{op_r\} - k & \text{branch instruction} \\
 & \mid \texttt{cmp}\ y\ y \mid \texttt{tst}\ y\ y & \text{comparison instruction} \\
 & \mid \mathsf{CFL}.inst & \text{operation instruction} \\
p & ::= (\ \overrightarrow{v}, \widetilde{inst}, \overrightarrow{x}\ ) & \text{programs}
\end{array}
$$

$$
\frac{\texttt{eval\_op}\ (op\ y\ \boldsymbol{x})\ \omega = \omega_1}{op\ y\ \boldsymbol{x} \vdash (pc,cpsr,\omega) \rightarrowtail (pc{+}1,cpsr,\omega_1)}
\qquad
\frac{\texttt{update\_cpsr}\ cpsr\ d_1\ d_2 = cpsr_1}{cmp\ d_1\ d_2 \vdash (pc,cpsr,\omega) \rightarrowtail (pc{+}1,cpsr_1,\omega)}
$$

$$
\frac{\texttt{is\_true}\ (\texttt{eval\_cpsr}\ cpsr\ rop)}{b\{rop\}\ (+/-)\ k \vdash (pc,\ cpsr,\ \omega) \rightarrowtail (pc\ (+/-)\ k,\ cpsr,\ \omega)}
$$

$$
\frac{\texttt{is\_false}\ (\texttt{eval\_cpsr}\ cpsr\ rop)}{b\{rop\}\ (+/-)\ k \vdash (pc,cpsr,\omega) \rightarrowtail (pc+1,\ cpsr,\ \omega)}
$$

Fig. 3. Syntax and evaluation rules of the machine language

Since expressions in HOL-, ANF and ACF are simply HOL functions, no explicit definitions for either the syntax or the semantics of them are required. In contrast, the abstract syntax for HSL and CFL is presented as inductive data types, and the operational semantics of them are defined over these data types (note that in our definition the body of a TR structure keeps running when the condition does not hold).

4 Translation and Verification

In this section we discuss the stages of compilation, focusing on how the proofs are organized.

4.1 From HOL- to ANF/ACF

Various well-known source-to-source translations are employed at this level: the input is first transformed to a first order function, then to ANF by performing a CPS transformation. And then a standard graph-colouring register allocation phase is invoked to produce a data structure for generating HSL programs. Finally, ANF is rewritten to ACF, an equivalent combinatory format.

Closure Conversion. Higher order and local functions in HOL- are eliminated by closure conversion, where the free variables for local functions are captured in an environment as passed to the function as an extra argument.

Combinator format. Although we do not have syntax trees for functions at this level, we can define and use 'constructor'-like semantic functions, and use them to implement translation steps. The recursion equation is translated to an equivalent combinatory format based on combinators for sequential composition (Seq), parallel composition (Par), conditionals (Ite), and tail-recursion (Rec). Note that the Seq and Par combinators are sufficient to express `let`-expressions.

$$\begin{array}{ll}
\vdash_{def} \mathsf{Seq}\ f_1\ f_2 \doteq \lambda x.f_2(f_1\ x) & \vdash_{def} \mathsf{Par}\ f_1\ f_2 \doteq \lambda x.(f_1\ x, f_2\ x) \\
\vdash_{def} \mathsf{Ite}\ f_1\ f_2\ f_3 & \doteq \lambda x.\mathsf{if}\ f_1\ x\ \mathsf{then}\ f_2\ x\ \mathsf{else}\ f_3\ x \\
\vdash_{def} \mathsf{Rec}\ f_1\ f_2\ f_3 & \doteq \lambda x.\mathsf{if}\ f_1\ x\ \mathsf{then}\ f_2\ x\ \mathsf{else}\ \mathsf{Rec}\ (f_3\ x) \\
\vdash_{thm} (\lambda x.\mathsf{let}\ v = f_1(x)\ \mathsf{in}\ f_2(x,v)) & \doteq \mathsf{Seq}\ (\mathsf{Par}(\lambda x.x)\ f_1)\ f_2
\end{array}$$

CPS Conversion. Once the program is in combinator format, a CPS translation is applied. CPS is defined semantically: $\mathsf{CPS}\ f \doteq \lambda k\ x.k\ (f\ x)$ specifies the CPS interface to a function. From this definition, it is easy to prove the theorem relating ordinary function application to CPS function application: $\vdash \forall f\ x.\ f\ x = (\mathsf{CPS}\ f)\ (\lambda x.x)\ x$. The CPS transformation phase repeatedly rewrites with the following theorems to push the CPS function down through the combinators:

$$\begin{array}{ll}
\vdash_{thm} \mathsf{CPS}\ (\mathsf{Seq}\ f_1\ f_2) & = \mathsf{CPS_SEQ}\ (\mathsf{CPS}\ f_1)\ (\mathsf{CPS}\ f_2) \\
\vdash_{thm} \mathsf{CPS}\ (\mathsf{Par}\ f_1\ f_2) & = \mathsf{CPS_PAR}\ (\mathsf{CPS}\ f_1)\ (\mathsf{CPS}\ f_2) \\
\vdash_{thm} \mathsf{CPS}\ (\mathsf{Ite}\ e\ f_1\ f_2) & = \mathsf{CPS_ITE}\ (\mathsf{CPS}\ e)\ (\mathsf{CPS}\ f_1)\ (\mathsf{CPS}\ f_2) \\
\vdash_{thm} \mathsf{CPS}\ (\mathsf{Rec}\ e\ f_1\ f_2) & = \mathsf{CPS_REC}\ (\mathsf{CPS}\ e)\ (\mathsf{CPS}\ f_1)\ (\mathsf{CPS}\ f_2)
\end{array}$$

where

$$\begin{array}{ll}
\vdash_{def} \mathsf{CPS_SEQ}\ f_1\ f_2 & \doteq \lambda k\ x.f_1\ (\lambda r.f_2\ k\ r)\ x \\
\vdash_{def} \mathsf{CPS_PAR}\ f_1\ f_2 & \doteq \lambda k\ x.f_1\ (\lambda r_2.f_2\ (\lambda r_1.k\ (r_2, r_1))\ x)\ x \\
\vdash_{def} \mathsf{CPS_ITE}\ e\ f_1\ f_2 & \doteq \lambda k\ x.e\ (\lambda r.\mathsf{let}\ k_1 = k\ \mathsf{in}\ \mathsf{if}\ r\ \mathsf{then}\ f_1\ k_1\ x\ \mathsf{else}\ f_2\ k_1\ x)\ x \\
\vdash_{def} \mathsf{CPS_REC}\ e\ f_1\ f_2 & \doteq \lambda k\ x.k\ (\mathsf{Rec}\ (e\ (\lambda x.x))\ (f_1\ (\lambda x.x))\ (f_2\ (\lambda x.x))\ x)
\end{array}$$

Then the CPS interface from the expression is removed by rewriting with the theorem $\vdash \mathsf{CPS}\ f\ k = \lambda x.\mathsf{let}\ z = f\ x\ \mathsf{in}\ k\ z$ to obtain a readable, let-based A-normal form. There is also a pass to remove all the β-redexes introduced in the CPS translation. The quality of the ANF expression is improved by removing as many tuples as possible, and by removing redundant let expressions that simply rename variables. All phases of transformations are term rewriting with theorems that establish equality for the input and result of each rewriting step.

Register Allocation. This phase converts the ANF form to a data structure suitable for performing register allocation. Interestingly, the graph colouring register allocation algorithm does not have to be verified; instead, the computed colouring can be taken and used to build a term incorporating the required spilling. To formally prove that this new term is equivalent to the original is very simple, amounting to not much more than checking that the two expressions are α-equivalent. In our implementation this task is fulfilled implicitly when we verify the translation from ACF to HSL by comparing the ACF with the synthesized function. This nice trick was first noticed by Hickey and Nogin [8] and is also used by Leroy [11]. It allows the results of standard register allocation algorithms to be used, without having to verify their correctness. The following

example shows the HOL- (left) and an ANF (right) of the TEA block cipher [23] (names of variables spilled begin with m and those in registers begin with r):

```
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)
```

ACF. The ANF is again converted to an equivalent 'constructor'-like semantic function (*i.e.*, ACF) based on combinators for sequential composition (sc), conditionals (cj) and tail-recursion (tr). By definition $\mathsf{sc} = \mathsf{Seq}$ and $\mathsf{cj} = \mathsf{Ite}$; however, tr is a little different from Rec.

$$\vdash_{def} \ \mathsf{tr}\ f_1\ f_2 \doteq \lambda x.\mathsf{if}\ f_1\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{tr}\ (f_2\ x)$$
$$\vdash_{thm} \ (f\ x = \mathsf{if}\ f_1\ x\ \mathsf{then}\ f_2\ x\ \mathsf{else}\ f\ (f_3\ x)) \Leftrightarrow (f = \mathsf{sc}\ (\mathsf{tr}\ f_1\ f_3)\ f_2)$$

4.2 From ACF to HSL

To support reasoning about HSL programs, we use the following Hoare triples:

$$\{P\}\ S\ \{Q\} \doteq \forall \sigma_{\mathrm{hsl}}.P\ \sigma_{\mathrm{hsl}} \Rightarrow Q(\mathsf{run}_{\mathrm{hsl}}\ S\ \sigma_{\mathrm{hsl}})$$

We first derive standard Hoare rules. Then, to bridge the semantic gap between an ACF function g with inputs $\boldsymbol{i}$ and outputs $\boldsymbol{o}$, and the HSL structure S built from g's ANF, we specialize the axiomatic semantics to obtain a refined set of Hoare rules—dubbed the *projective* Hoare rules. A projective Hoare rule says: provided that inputs $\boldsymbol{i}$ have initial values $\boldsymbol{v}$, and any variable x in the live variable set ξ has value k, then in the state σ' after the execution of S, the values left in outputs $\boldsymbol{o}$ are equal to applying the function f to the initial values $\boldsymbol{v}$, and x's value is still k:

$$\begin{array}{l} S \vdash \xi\ \uparrow\ (\boldsymbol{i}, f, \boldsymbol{o}) \doteq \\ \quad \forall x \in \xi\ \forall \boldsymbol{v} \forall k \forall \sigma_{\mathrm{hsl}}.(\boldsymbol{i}_f\ \sigma_{\mathrm{hsl}} = \boldsymbol{v}) \wedge (\sigma_{\mathrm{hsl}}[[x]] = k) \Rightarrow \\ \quad \mathtt{let}\ \sigma'_{\mathrm{hsl}} = \mathsf{run}_{\mathrm{hsl}}\ S\ \sigma_{\mathrm{hsl}}\ \mathtt{in}\ \wedge (\boldsymbol{o}_f\ \sigma'_{\mathrm{hsl}} = f\ \boldsymbol{v}) \wedge (\sigma'_{\mathrm{hsl}}[[x]] = k) \end{array}$$

where functions i_f and o_f project from a data state the values of vector $\boldsymbol{i}$ and $\boldsymbol{o}$. If the judgement embodied by a projective Hoare rule holds on the S derived from 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. Operator `mk_cnd` turns a condition into a condition function. Suppose $\overrightarrow{\xi}$ turns a set ξ into a vector, and $\overleftarrow{\boldsymbol{v}}$ turns a vector $\boldsymbol{v}$ into a set, then the product of a vector and a set

makes a new vector that comprises $\boldsymbol{v_1}$ and all elements in ξ, $\boldsymbol{v_1} \times \xi \doteq (\boldsymbol{v_1}, \overrightarrow{\xi})$. The dot product of a function and a set gives a new function: $(\lambda \boldsymbol{x}.f\ \boldsymbol{x}) \odot \xi \doteq \lambda(\boldsymbol{x}, \overrightarrow{\xi}).(f\ \boldsymbol{x}, \overrightarrow{\xi})$. A vector and a projective function are interchangeable.

$$\frac{s_1 \vdash \xi_1 \uparrow (\boldsymbol{i_1}, f_1, \boldsymbol{o_1}) \quad s_2 \vdash \xi_2 \uparrow (\boldsymbol{o_1}, f_2, \boldsymbol{o_2})}{\mathsf{SC}\ s_1\ s_2 \vdash \xi_1 \cap \xi_2 \uparrow (\boldsymbol{i_1}, \mathsf{sc}\ f_1\ f_2, \boldsymbol{o_2})}\ \text{sc\_rule}$$

$$\frac{s_1 \vdash \xi_1 \uparrow (\boldsymbol{i}, f_1, \boldsymbol{o}) \quad s_2 \vdash \xi_2 \uparrow (\boldsymbol{i}, f_2, \boldsymbol{o})}{\mathsf{CJ}\ cnd\ s_1\ s_2 \vdash \xi_1 \cap \xi_2 \uparrow (\boldsymbol{i}, (\mathsf{cj}\ (\mathtt{mk\_cnd}\ cnd)\ f_1\ f_2), \boldsymbol{o})}\ \text{cj\_rule}$$

$$\frac{s \vdash \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{i})}{\mathsf{TR}\ cnd\ s \vdash \xi \uparrow (\boldsymbol{i}, (\mathsf{tr}\ (\mathtt{mk\_cnd}\ cnd)\ f), \boldsymbol{i})}\ \text{tr\_rule} \qquad \frac{s \vdash \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \quad g\ \boldsymbol{i'} = f\ \boldsymbol{i}}{s \vdash \xi \uparrow (\boldsymbol{i'}, g, \boldsymbol{o})}\ \text{shuffle\_rule}$$

$$\frac{s \vdash \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \quad \xi' \subseteq \xi}{s \vdash \xi \uparrow (\boldsymbol{i} \times \xi', f \odot \xi', \boldsymbol{o} \times \xi')}\ \text{pick\_rule} \qquad \frac{s \vdash \xi \uparrow (\boldsymbol{i}, f, \boldsymbol{o}) \quad \xi' \subseteq \xi}{s \vdash \xi' \uparrow (\boldsymbol{i}, f, \boldsymbol{o})}\ \text{shrink\_rule}$$

$$\frac{s \vdash \xi \uparrow (callee.\boldsymbol{i}, f, callee.\boldsymbol{o}) \quad \overleftarrow{caller.\boldsymbol{o}} \cap \xi' = \phi}{\mathsf{FC}\ (caller.\boldsymbol{i}, f, callee.\boldsymbol{i})\ s\ (caller.\boldsymbol{o}, f, callee.\boldsymbol{o}) \vdash \xi' \uparrow (caller.\boldsymbol{i}, f, caller.\boldsymbol{o})}\ \text{fc\_rule}$$

These rules are used to keep track of how the relation between specific inputs and outputs change during the execution. Rules sc_rule, cj_rule and tr_rule are control flow rules and their meaning is self-explanatory. The live variable set ξ stores the variables that are still live but not modified by the current statement. In other words, when the value of a live variable is not altered by the current statement, it is stored in ξ for future use. A live variable is either in ξ, or in the outputs $\boldsymbol{o}$. When it becomes not live any more, it should be removed from ξ. Maintaining a ξ helps to reduce the number of variables in the inputs and outputs. Rule pick_rule is for extracting variables from the live variable set, while shrink_rule is used to discard variables not live any more from the set. Rule shuffle_rule is to restructure the input vector. Restructuring the ouput vector is accomplished by appending an empty block and applying the shuffle_rule to it. A basic block is simulated as a whole as it is a macro instruction, thus there exists no rule for it.

Application of projective rules is controlled by an annotated structure with inputs, outputs and context information, which guides the symbolic simulation and the application of rules. Control flow rules `sc_rule`, `cj_rule` and `tr_rule` 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` rule. The application of data flow rules `pick_rule`, `shrink_rule` and `shuffle_rule` are guided by the "use" and "def" information of a structure maintained by the compiler.

4.3 From HSL to CFL

The main task for this translation is to implement function calls and map heap variables and stack variables to memory (for wider application we handle heap variables here although they are replaced with stack variables during closure conversion). Obviously the mapping function, $\setminus$, shall be a one-to-one function.

The storage for local (stack) variables is allocated on function entry and released on function exit. In particular, local variables are held in a stack frame that will be "destroyed" on function exit, and the storage for its stack can be

"collected" and reused for other function calls. The memory is modelled as a finite map with addresses ranging from 0 to $2^{32}-1$.

We introduce an injection relation $\backsimeq^{\searrow}$ to relate the states occurring during the execution of HSL code and that of the translated CFL code, where $\searrow$ consists of three injective functions $\searrow_{rg}$, $\searrow_{hp}$ and $\searrow_{sk}$ that map logical registers, heap variables and stack variables to machine registers and memory locations respectively. Of course all procedures use the same $\searrow_{hp}$ as they share the global heap. The correctness statement amounts to showing that the execution of a HSL statement S_{hsl} has the same effect on a HSL state as the execution of its corresponding CFL statement S_{cfl} (notation D_σ and D_S return the domains of the finite maps in σ and the variables accessed by the instruction in S).

$$\vdash_{def} \ \mathsf{one\_one\_inj}\ \sigma_{\text{hsl}}\ \searrow\ \sigma_{\text{cfl}} \doteq \forall v_1, v_2 \in D_{\sigma_{\text{hsl}}}.\ \mathsf{addr}\ \sigma_{\text{cfl}}\ v_1^{\searrow} \neq \mathsf{addr}\ \sigma_{\text{cfl}}\ v_2^{\searrow}$$
$$\vdash_{def} \ \sigma_{\text{hsl}} \backsimeq^{\searrow} \sigma_{\text{cfl}} \ \doteq\ \forall v \in D_{\sigma_{\text{hsl}}}.\ \sigma_{\text{hsl}}[[v]] = \sigma_{\text{cfl}}[[v^{\searrow}]]$$
$$\vdash_{def} \ (S_{\text{hsl}} \equiv^{\searrow} S_{\text{cfl}}) \doteq$$
$$\forall \sigma_{\text{hsl}} \forall \sigma_{\text{cfl}}.\ (D_{S_{\text{hsl}}} = D_{\sigma_{\text{hsl}}} \wedge \sigma_{\text{hsl}} \backsimeq^{\searrow} \sigma_{\text{cfl}}) \Rightarrow (\mathsf{run}_{\text{hsl}}\ S_{\text{hsl}}\ \sigma_{\text{hsl}}\ \backsimeq^{\searrow}\ \mathsf{run}_{\text{cfl}}\ S_{\text{cfl}}\ \sigma_{\text{cfl}})$$

The function addr returns the address of a mapped variable. An address is parameterized by a state containing the values of base registers (e.g. fp and sp). Given an injection $\searrow$, the translation from HSL to CFL for most structures is simple and we just need to replace HSL variables with their mapped machine registers and memory locations. A FC structure will be converted to the sequential composition of pre-call processing, callee's body and post-call processing:

$$r_i^{\searrow} \doteq \searrow_{rg}\ r_i \quad hp[i]^{\searrow} \doteq m[\searrow_{hp}\ i] \quad sk[i]^{\searrow} \doteq m[\searrow_{sk}\ i] \quad S^{\searrow} \doteq \forall v \in D_S.\ S[v \leftarrow v^{\searrow}]$$
$$\Gamma_{\text{hsl}}\ S \doteq S^{\searrow} \quad \text{when } S \text{ is a BLK,SC,CJ or TR structure}$$
$$\Gamma_{\text{hsl}}\ (\mathsf{FC}\ (caller.\boldsymbol{i}, callee.\boldsymbol{i})\ S\ (caller.\boldsymbol{o}, callee.\boldsymbol{o})) \doteq$$
$$\mathsf{SC}\ (\mathsf{SC}\ pre\ (\Gamma_{\text{hsl}}\ S))\ post \quad \text{for valid } pre, post \text{ and } \searrow' \text{ described below}$$

When $\searrow_{sk}$ maps different stack variables to different memory locations, the translation for BLK, SC, CJ and TR structures guarantees semantics preservation. The translation for FC is more complicated: we require that the pre-call processing and post-call processing fulfill the parameter passing and result returning task; and the execution of the pre-call processing, function body and post-call processing should not modify the values of the caller's register and stack variables except for those set to receive results (we name this the *value recovering* property). Assuming that $\searrow$ is an one-to-one injection, we have:

$$\frac{}{(\mathsf{BLK}\ S) \equiv^{\searrow} (\mathsf{BLK}\ S^{\searrow})} \qquad \frac{S_{\text{hsl\_1}} \equiv^{\searrow} S_{\text{cfl\_1}} \quad S_{\text{hsl\_2}} \equiv^{\searrow} S_{\text{cfl\_2}}}{\mathsf{SC}\ S_{\text{hsl\_1}}\ S_{\text{hsl\_2}} \equiv^{\searrow} \mathsf{SC}\ S_{\text{cfl\_1}}\ S_{\text{cfl\_2}}}$$

$$\frac{S_{\text{hsl\_1}} \equiv^{\searrow} S_{\text{cfl\_1}} \quad S_{\text{hsl\_2}} \equiv^{\searrow} S_{\text{cfl\_2}}}{\mathsf{CJ}\ cond\ S_{\text{hsl\_1}}\ S_{\text{hsl\_2}} \equiv^{\searrow} \mathsf{CJ}\ cond^{\searrow}\ S_{\text{cfl\_1}}\ S_{\text{cfl\_2}}} \qquad \frac{S_{\text{hsl}} \equiv^{\searrow} S_{\text{cfl}}}{\mathsf{TR}\ cond\ S_{\text{hsl}} \equiv^{\searrow} \mathsf{TR}\ cond^{\searrow}\ S_{\text{cfl}}}$$

$$\frac{\begin{array}{c}\forall \sigma.\sigma[[caller.\boldsymbol{i}^{\searrow}]] = (\mathsf{run}_{\text{cfl}}\ pre\ \sigma)[[callee.\boldsymbol{i}^{\searrow'}]] \quad S_{\text{hsl}} \equiv^{\searrow'} S_{\text{cfl}} \\ \forall \sigma.\sigma[[\overleftarrow{callee.\boldsymbol{o}}^{\searrow'}]] = (\mathsf{run}_{\text{cfl}}\ post\ \sigma)[[caller.\boldsymbol{o}^{\searrow}]] \\ \forall \sigma.\forall v \in (D^{rg,sk}_{S_{caller}} \setminus \overleftarrow{caller.\boldsymbol{o}}).\ \sigma[[v^{\searrow}]] = (\mathsf{run}_{\text{cfl}}\ (\mathsf{SC}\ (\mathsf{SC}\ pre\ S_{\text{cfl}})\ post)\ \sigma)[[v^{\searrow}]]\end{array}}{\mathsf{FC}\ (caller.\boldsymbol{i}, callee.\boldsymbol{i})\ S_{hsl}\ (caller.\boldsymbol{o}, callee.\boldsymbol{o}) \equiv^{\searrow} \mathsf{SC}\ (\mathsf{SC}\ pre\ S_{\text{cfl}})\ post}$$

There are many ways to guarantee that the value recovering property holds. One of them is to layout the frames of the caller and callee in such a way that their domains do not intersect with each other; and the values of register variables

modified by the callee's execution are recovered on the function entry. This leads to a valid implementation of a frame layout and a function call procedure. The areas in the memory devoted to stack frames (i.e. the activation record) are marked by the *ip*, *fp* and *sp*. When the callee is called, space for results are reserved by growing the stack, then the caller pushes all parameters into the stack; and then the frame for the callee is created. Specifically, when a callee is called, its stack frames shall not be overlapped with the callee's frame.

As indicated by the following rule, an implementation is valid if it ensures that: (1) the parameter/result passing and the body execution do not change the values of stack variables in the caller's frame except those for receiving results (*i.e.*, $caller.\boldsymbol{o}$); (2) all register variables are pushed into memory before parameter passing on function entry and then popped from memory before result passing on function exit. In the following rule, $\sigma\langle v\rangle$ represents reading the value at concrete address v from state σ, and D_r is the abbreviation of $D_{S_{caller}}$.

$$\frac{\begin{array}{c}\sigma_1 = \mathsf{run}_{\mathrm{cfl}}\ pre\ \sigma \quad \sigma_2 = \mathsf{run}_{\mathrm{cfl}}\ S_{\mathrm{cfl}}\ \sigma_1 \quad \sigma_3 = \mathsf{run}_{\mathrm{cfl}}\ post\ \sigma_2 \\ \forall v \in (D_r^{sk})^{\searrow}.\sigma\langle v\rangle = \sigma_1\langle v\rangle \quad \exists x_i.\ \sigma_1\langle x_i\rangle = \sigma[[r_i]] \text{ for } i \in D^{rg}_{S_{callee}} \\ \forall v \in (D_r^{sk})^{\searrow} \cup \{x_i \mid i \in D^{rg}_{S_{callee}}\}.\sigma_2\langle v\rangle = \sigma_1\langle v\rangle \\ \forall v \in (D_r^{sk} \setminus \overleftarrow{caller.\boldsymbol{o}})^{\searrow}.\ \sigma_3\langle v\rangle = \sigma_2\langle v\rangle \quad \forall r_i \in (D_r^{rg} \setminus \overleftarrow{caller.\boldsymbol{o}}).\ \sigma_3[[r_i]] = \sigma_2\langle x_i\rangle\end{array}}{\forall\sigma.\forall v{\in}(D_r^{rg,sk}\setminus\overleftarrow{caller.\boldsymbol{o}}).\ \sigma[[v^{\searrow}]]{=}(\mathsf{run}_{\mathrm{cfl}}\ (\mathsf{SC}\ (\mathsf{SC}\ pre\ S_{\mathrm{cfl}})\ post)\ \sigma)[[v^{\searrow}]]}$$

Complying with these requirements, our implementation compiles function calls into a callee-save style calling convention. Specifically, $\searrow_{sk}{=}\searrow'_{sk}{=}\ \lambda i.(fp, -(i + 12))$, $\searrow_{rg}{=}\searrow'_{rg}{=}\ \lambda r.r$ and $\searrow_{hp}{=}\searrow'_{hp}{=}\ \lambda i.(hp, -i)$. By carefully moving the pointers *fp*, *ip* and *sp* we keep the caller's frame and callee's frame located in separate areas in the memory. All parameters and results are passed through the stack, and the callee saves all data registers (*i.e.*, $r_0 - r_8$) in all cases. This solution is suboptimal but easier to verify. In particular, it allows us, while performing colouring register allocation, not to add interferences between caller-save registers and temporaries that are live across a call.

higher address (32-bit word based address) lower address

← ...	global heap	previous frame	current frame	next frame	... →

	Memory	Addr		Memory	Addr
caller's ip	reserved for pc	i		...	.
caller's fp	saved lr	i-1		stack variable n	j
	save ip	i-2	caller's sp	parameter/result k	j-1
	save fp	i-3		...	.
	stored reg 8	i-4		parameter/result 0	k
	...	.	callee's ip	reserved for pc	k-1
	stored reg 0	i-12	callee's fp	saved lr	k-2
	stack variable 0	i-13		...	.

$pre =$ BLK [*sub sp sp* $(max(\#caller.\boldsymbol{i}, \#caller.\boldsymbol{o}) - \#caller.\boldsymbol{i})$; *push* $caller.\boldsymbol{i}$;
mov ip sp; *sub fp ip* 1; *sub sp sp* 1; *push* $\{r_0, \ldots, r_8, fp, ip, lr\}$;
add sp sp 12; *pop* $callee.\boldsymbol{i}$; *sub sp fp* $(12 + \#stack\_variables)$]

$post =$ BLK [*add sp ip* $\#callee.\boldsymbol{o}$; *push* $callee.\boldsymbol{o}$; *sub sp fp* 12;
pop $\{r_0, \ldots, r_8, fp, ip, lr\}$; *mov sp ip*; *pop* $caller.\boldsymbol{o}$;
sub sp fp $(12 + \#stack\_variables)$]

One subtlety appearing in proofs is that the initial values of hp, sp, ip and fp must be greater than specific values so that the memory can accomodate all stack frames and the areas consumed by pre/post processing.

Both the heap and the stacks are simply finite maps, thus we do not formalize and rely on any heap management and stack property. In [3] a block-base memory model between a machine memory and a high-level view is introduced to manage frame stacks. As in our method, separation is enforced between stack blocks belonging to different function activation records.

4.4 From CFL to ARM

The translation from CFL to ARM proceeds by linearizing the SC, CJ and TR structures. The instructions in basic blocks are already in the right format. Our translation always generates flat code satisfying good properties including: (1) any execution of the translated code will not access beyond its own area in the instruction buffer; (2) the data state after an execution is independent of the initial values of pc and $cpsr$; (3) all executions terminate.

The translation verification for CJ proceeds by case analysis on the condition; while that for TR by the induction on the number of rounds the body is executed. This linearization scheme turns out to be most succinct in terms of the length of generated code. One optimization is performed at the flat code level for function calls: all occurrences of a callee are moved to the same area in the code so that only one copy is left. Unconditional jumps are inserted appropriately. The correctness proof for this relocation is straight forward because the adjusted code runs in the same way as its old version.

$$
\begin{aligned}
\Gamma_{\text{cfl}}\ (\mathsf{BLK}\ (inst :: instL)) &\doteq inst :: \Gamma_{\text{cfl}}(\mathsf{BLK}\ instL)\\
\Gamma_{\text{cfl}}\ (\mathsf{BLK}\ [\,]) &\doteq [\,]\\
\Gamma_{\text{cfl}}\ (\mathsf{SC}\ s_1\ s_2) &\doteq (\Gamma_{\text{cfl}}\ s_1)\ \uplus\ (\Gamma_{\text{cfl}}\ s_2)\\
\Gamma_{\text{cfl}}\ (\mathsf{CJ}\ (v_1, rop, v_2)\ s_t\ s_f) &\doteq \mathsf{let}\ (\rho_t\ \rho_f) = (\Gamma_{\text{cfl}}\ s_t, \Gamma_{\text{cfl}}\ s_f)\ \mathsf{in}\\
&\qquad (\mathsf{cmp}\ v_1\ v_2) :: (\mathsf{b}\{rop\}\ +\ \|\rho_f\| + 2) ::\\
&\qquad \rho_f \uplus [\mathsf{bal}\ +\ \|\rho_t\| + 1] \uplus \rho_t\\
\Gamma_{\text{cfl}}\ (\mathsf{TR}\ (v_1, rop, v_2)\ s) &\doteq \mathsf{let}\ \rho = \Gamma_{\text{cfl}}\ s\ \mathsf{in}\\
&\qquad (\mathsf{cmp}\ v_1\ v_2) :: (\mathsf{b}\{rop\} + \|\rho\| + 2) :: \rho\ \uplus [\mathsf{bal} - (|\rho| + 2)]
\end{aligned}
$$

Note that $\|\rho\|$ returns the number of instructions in ρ, and $\rho_1 \uplus \rho_2$ appends ρ_2 to ρ_1.

Example. With the following abbreviations,

$$
\begin{aligned}
&body \doteq \mathsf{BLK}\ [msub\ r_3\ r_0\ 1w;\ mmul\ r_2\ r_0\ r_1;\ mmov\ r_0\ r_3;\ mmov\ r_1\ r_2]\\
&blk_1 \doteq \mathsf{BLK}\ [mmov\ r_2\ r_1] \quad snd \doteq \lambda(v_0, v_1).v_1\\
&f_1 \doteq \lambda(v_0, v_1).(v_0 - 1w, v_0 + v_1) \quad f_2 \doteq \mathsf{tr}\ (\lambda(v_0, v_1).v_0 = 0w))\rangle)\ f_1
\end{aligned}
$$

the intermediate forms of the factorial function and the derivation of the specification connecting the $fact_{hsl}$ and $fact_{acf}$ (where $Axiom_1 = blk_1 \vdash \{\} \uparrow ((r_0, r_1), snd, r_2)$) are

HOL-: $fact\ (x, a) \doteq \texttt{if}\ x = 0w\ \texttt{then}\ a\ \texttt{else}\ fact\ (x - 1w, x \times a)$
ACF: $fact_{acf} \doteq \mathsf{sc}\ (\mathsf{tr}\ (\lambda(v_0, v_1).v_0 = 0w)\ f_1)\ snd$
HSL: $fact_{hsl} \doteq \mathsf{SC}\ (\mathsf{TR}\ (r_0, eq, 0w)\ body)\ blk_1$
CFL: $fact_{cfl} \doteq \Gamma_{hsl}\ fact_{hsl} = fact_{hsl}$
ARM: $fact_{arm} \doteq \Gamma_{\text{cfl}}\ fact_{cfl} = [cmp\ r_0\ r_1;\ beq\ +\ 6;\ sub\ r_3\ r_0\ 1w;\ mul\ r_2\ r_0\ r_1;$
$mov\ r_0\ r_3;\ mov\ r_1,\ r_2;\ bal\ -\ 6;\ mov\ r_2,\ r_1]$

$$\dfrac{\dfrac{body\ \vdash\ \{\}\ \uparrow\ ((r_0, r_1), f_1, (r_0, r_1))}{\mathsf{TR}\ (r_0, ne, 0w)\ body\ \vdash\ \{\}\ \uparrow\ ((r_0, r_1), f_2, (r_0, r_1))}\ \text{tr\_rule} \qquad Axiom_1}{\mathsf{SC}\ (\mathsf{TR}\ (r_0, ne, 0w)\ body)\ blk_1\ \vdash\ \{\}\ \uparrow\ ((r_0, r_1), fact_{acf}, r_2)}\ \text{sc\_rule}$$

5 Related Work

We have also developed a hardware compiler for a similar source language [7]: it takes in HOL function definitions and emits FPGA-level netlists. Compilation proceeds essentially by refinement steps: control structures in logic are refined by formulas representing unclocked circuits implementing those structures, and those circuit-formulas are further refined to be formulas for clocked circuits.

Hickey and Nogin [8] constructed a compiler from a higher order, untyped, functional language to Intel x86 code, based entirely on higher-order rewrite rules. The compiler is written in the MetaPRL logical framework. A set of rewrite rules are used to convert a higher level program to a lower level program. However, verification of the rules remains to be done. Since their source languages and intermediate representations are similar to ours, we may apply their rules during the translation from HOL to HOL- and then ANF, *e.g.*, the closure conversion and CPS conversion rules; yet our existing verification techniques for these translations are still valid. Similarly, Watson [22] proposes a refinement calculus for the compilation from high-level language to .NET assembly; Sampaio [20] uses term rewriting to convert source programs to their normal forms representing object code. These latter works are not machine automated.

Leroy [2,11] has verified a compiler from a subset of C, Clight, to PowerPC assembly code in the Coq system. The semantics of Clight is completely deterministic and specified as big-step operational semantics. Several intermediate languages are introduced and translations between them are verified. The proof of semantics preservation for the translation proceeds by induction over the Clight evaluation derivation and case analysis on the last evaluation rule used; in contrast, our proofs proceed by verifying that the rewrite rules used are semantics preserving and the execution of programs at different phases has the same effect on the corresponding states. Leroy also uses translation validation to sidestep the difficult correctness proof for register allocation. He relies on an outside verifier to check *a posteriori* the graph colouring register allocator.

A purely operational semantics based development is that of Klein and Nipkow [9] which gives a thorough formalization of a Java-like language. A compiler from this language to a subset of Java Virtual Machine is verified using Isabelle/HOL. However, that compiler targets high-level code than our assembly, for example it assumes an unbounded number of registers. Compilation from a type-safe subset

of C to DLX assembly code has been verified using the Isabelle/HOL theorem prover [10]. A big step semantics and a small step semantics for this language are related by the proof.

There has recently been a large amount of work on verifying 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.*, [5,13] and also extensions such as Separation Logic [19]. Of course, compiler verification itself is a venerable topic, with far too many publications to survey (see Dave's bibliography [4]). Restricting to assembler verification, one of the most relevant works for us is by Moore [14].

6 Conclusions and Future Work

We have presented the design of a compiler for a subset of higher order logic which operates by running proofs. The fact that the source language is not associated with any evaluation semantics makes the translation validation somewhat novel. Our end-to-end, fully automatic compiler successfully bridges the large gap between programs in logic and low level assembly programs.

Currently, the validation of the translation from an ACF program to its HSL program requires the HSL program to inherit ACF's structure, thus restricting the degree of optimizations at the HSL level. In spite of this restriction, many optimizations can be performed in the other levels. For example, optimizations on basic blocks are easy since their validation simply requires symbolic simulation.

Currently, we are strengthening the front end translation to support ML-style datatypes and non-tail recursive functions. We are also augumenting the back end to tackle dynamic memory allocation, as well as changing the current ARM-like target language to the detailed ARM model developed by Fox [6].

Acknowledgements. We thank Thomas Tuerk for his help in refining the definition of the ARM model. We also appreciate the advice from the anonymous reviewers.

References

1. Stefan Berghofer and Tobias Nipkow, *Executing higher order logic*, P. Callaghan, Z. Luo, J. McKinna, R. Pollack, editors, Types for Proofs and Programs, International Workshop (TYPES 2000), 2000.
2. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy, *Formal verification of a C compiler front-end*, 14th International Symposium on Formal Methods (FM 2006), Hamilton, Canada, 2006.
3. Sandrine Blazy and Xavier Leroy, *Formal verification of a memory model for C-like imperative languages*, International Conference on Formal Engineering Methods (ICFEM 2005), Manchester, UK, 2005.
4. Maulik A. Dave, *Compiler verification: a bibliography*, ACM SIGSOFT Software Engineering Notes **28** (2003), no. 6, 2–2.

5. Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni, *Modular verification of assembly code with stack-based control abstractions*, ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06), 2006, pp. 401–414.
6. Anthony Fox, *Formal verification of the ARM6 micro-architecture*, Tech. Report 548, University of Cambridge Computer Laboratory, November 2002.
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 2005), ENTCS, vol. 145, 2005.
8. Jason Hickey and Aleksey Nogin, *Formal compiler construction in a logical framework*, Journal of Higher-Order and Symbolic Computation **19** (2006), no. 2-3, 197–230.
9. Gerwin Klein and Tobias Nipkow, *A machine-checked model for a Java-like language, virtual machine and compiler*, TOPLAS **28** (2006), no. 4, 619–695.
10. Dirk Leinenbach, Wolfgang Paul, and Elena Petrova, *Towards the formal verification of a C0 compiler: Code generation and implementation correctnes*, 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 2005.
11. Xavier Leroy, *Formal certification of a compiler backend, or: programming a compiler with a proof assistant*, Symposium on the Principles of Programming Languages (POPL 2006), ACM Press, 2006.
12. Kim Marriott and Peter J. Stuckey, *Programming with constraints, an introduction*, MIT Press, 1998.
13. John Matthews, J Strother Moore, Sandip Ray, and Daron Vroon, *Verification condition generation via theorem proving*, LPAR 2006 (LNCS 4246), Springer Verlag, 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. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel, *Isabelle/HOL — a proof assistant for higher-order logic*, LNCS, vol. 2283, Springer, 2002.
17. Michael Norrish and Konrad Slind, *HOL-4 manuals*, 1998-2006, Available at `http://hol.sourceforge.net/`.
18. A. Pnueli, M. Siegel, and E. Singerman, *Translation validation*, 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98), 1998.
19. John C. Reynolds, *Separation logic: A logic for shared mutable data structures*, IEEE Symposium on Logic in Computer Science (LICS'02), 2002, pp. 55–74.
20. Augusto Sampaio, *An algebraic approach to compiler design, volume 4 of AMAST series in computing*, World Scientific, 1997.
21. Konrad Slind, *Reasoning about terminating functional programs*, Ph.D. thesis, Institut für Informatik, Technische Universität München, 1999.
22. Geoffrey Watson, *Compilation by refinement for a practical assembly language*, International Conference on Formal Engineering Methods (ICFEM 2003), 2003.
23. David Wheeler and Roger Needham, *TEA, a tiny encryption algorithm*, Fast Software Encryption: Second International Workshop, 1999.