Compiling from Higher Order Logic

Konrad Slind

School of Computing, University of Utah

June 17, 2008

Acknowledgements

Anthony Fox, Mike Gordon, Guodong Li, Magnus Myreen, Scott Owens

FP in TP Choices

- Deep embedding.
  - Datatype of programs + inductively defined evaluation, typing, *etc.* relations.
  - PL is the principal object of study
  - Supported pretty well in various systems: Coq, HOL, Twelf, Isabelle/HOL, PLT-Redex
  - Examples: μ-Java, RSR6, SML, OCaml-Light, C, C++, ...
  - But: proving properties of individual programs is hard
- Shallow embedding.
  - Use built-in functions of the logic.
  - No single type of programs
  - Individual programs are the main objects of interest

FP in TP Choices

- Deep embedding.
  - Datatype of programs + inductively defined evaluation, typing, *etc.* relations.
  - PL is the principal object of study
  - Supported pretty well in various systems: Coq, HOL, Twelf, Isabelle/HOL, PLT-Redex
  - Examples: μ-Java, RSR6, SML, OCaml-Light, C, C++, ...
  - But: proving properties of individual programs is hard
- Shallow embedding.
  - Use built-in functions of the logic.
  - No single type of programs
  - Individual programs are the main objects of interest

HOL

HOL is essentially Church's Simple Type Theory

- HOL = simply typed λ-calculus + logic
- ML-style types: **bool**, $\alpha \rightarrow \beta$, $\alpha * \beta$, α **list**, algebraic datatypes, lazy lists
- But also $\mathbb{R}$ and lots of other incomputable stuff
- Terms: variables, constants, applications, λ-abstractions
- Classical logic defined on top.
- Logic of total functions

Recursion

Recursive functions can be defined with a 'controlled' recursion combinator—$\mathbf{WFREC}_{\prec}$:

Theorem (Wellfounded Recursion)

$$\mathbf{WF}(\prec) \Rightarrow (\mathbf{WFREC}_{\prec}\ F)\ x = F\ ((\mathbf{WFREC}_{\prec}\ F)\ |_{\{y \mid y \prec x\}})\ x$$

Systems like HOL and Isabelle/HOL manipulate input recursion equations into a form where the WF recn. theorem can be instantiated and massaged into a useful form.

Example

Consider

$$\textbf{variant}\ x\ \ell = \mathtt{if}\ \textbf{mem}\ x\ \ell\ \mathtt{then}\ \textbf{variant}\ (x+1)\ \ell\ \mathtt{else}\ x$$

- Translate into functional (Augusstson's pattern-matching translation)
- Instantiate F in theorem.
- Extract termination conditions
- Find termination relation $\prec$
- Prove **WF**$(\prec)$
- Prove termination conditions

Much of this can be automated. Works for mutual, nested, and higher-order recursions.

Recursion Induction

Allows one to prove a property P of a function by assuming P holds for each recursive call and then showing that P holds for the entire function.

Theorem (variant-induction)

$$\forall P.\ (\forall x\ \ell.\ (\mathbf{mem}\ x\ \ell \Rightarrow P\ (x+1)\ \ell) \Rightarrow P\ x\ \ell) \Rightarrow \forall x\ \ell.\ P\ x\ \ell$$

- Automatically derived from recursion equations (using termination).
- Proving correctness of **variant** is much easier with **variant**-induction than with $\mathbb{N}$-induction.

Upshot

Verification methodology for functional programs modelled with the built-in functions of the logic:

- Define program
- The logic framework has thus taken care of lexing, parsing, type inference, and overload resolution
- Prove termination. (Obligation; can be deferred)
- Recursion equations now usable
- Apply custom induction theorem to prove properties

Provocations

"I want to verify programs, not algorithms!"
–A. Tolmach

"WYSINWYG" –Tom Reps

Compilation

Perhaps the most widely used tool in CS are compilers.

Since compilers are crucial infrastructure, compiler verification is important.

There are at least three main themes in verifying compilation:

- User sprinkles assertions throughout code; compiler attempts to automatically prove them.
- Formalize and verify a compiler
- Translation validation

Verified Compilation

- Verified compiler: formalize source, target, and compilation algorithm as function from source to target. Then verify.
- Examples: McCarthy-Painter, ..., Klein-Nipkow, X. Leroy *et al*, ...
- Translation validation: run compiler; then prove that output code is equivalent to input.
- Examples: Pnueli, Siegel, and Singerman (TACAS'98), Necula (PLDI 2000), Li, Owens, and Slind (ESOP'07)

Verified Compilation

- Verified compiler: formalize source, target, and compilation algorithm as function from source to target. Then verify.
- Examples: McCarthy-Painter, ..., Klein-Nipkow, X. Leroy *et al*, ...
- Translation validation: run compiler; then prove that output code is equivalent to input.
- Examples: Pnueli, Siegel, and Singerman (TACAS'98), Necula (PLDI 2000), Li, Owens, and Slind (ESOP'07)

Compilers in theorem provers

- Hickey and Nogin (HOUFL to x86)
  - Higher-order rewrite rules in Meta-PRL basis for compilation.
  - Rules not verified
- Leroy (Clight to PPC)
  - Clight compiler as Coq function
  - Big-step operational semantics of subset of C
  - Formalized compiler in Coq and proved it correct
- Iyoda, Gordon, and Slind (subset of HOL to hardware)
- Li, Owens, Myreen, Fox, Slind (same subset to software)

Example

Accumulator-style 32-bit factorial:

$$\vdash \textbf{fac32}(n, acc) = \\ \quad \texttt{if } n = 0w \texttt{ then } acc \texttt{ else } \textbf{fac32}(n - 1w, acc * n)$$

Compiler returns a theorem:

```
|- ARM_PROG
    (R 0w r0 * R 1w r1 * ~S * R30 14w lr)
          L0:  CMP r0, #0
          L1:  MULNE r1, r0, r1
          L2:  SUBNE r0, r0, #1
          L3:  BNE L0
          L4:  MOV pc, lr
  (~R 14w * ~S * ~R 0w * R 1w (fac32(r0,r1)) ...)
```

Discussion

$$\vdash \texttt{ARM\_PROG}\ (\mathit{pre})\quad \mathit{ARMcode}\quad (\mathit{post})$$

is a theorem in the HOL logic, automatically proved.

Based on following formal theories

- ARM μ-architecture (Fox)
- ARM ISA (Fox)
- μ-arch. implements ISA (Fox)
- Hoare Logic (with separating conjunction) for ARM (Myreen)

Proposed methodology

- Specify functional programs as logic functions
- Prove correctness properties (no operational semantics!)
- Translate to low-level executable format (h/w, assembly) by proof
- Thus execution returns answers meeting the correctness properties

Compiling Logic?

Instead of compiling **programs**, we compile **logic definitions** (mathematical functions).

In other words, the source language is a subset of the functions expressible in the proof assistant (HOL-4).

This is unusual, since such functions

- have no ASTs visible in the logic (shallow embedding)
- have no operational semantics

What's a compiler writer to do?

Compiler

It turns out that things don't change very much: one of the themes of TV is that one can use standard algorithms and 'just' check the results.

- Start with a (recursive) function already defined in HOL-4.
- Now we try to do as much as possible by source-to-source translation.
- These translations are *semantic* versions of the standard syntax manipulations
- Theme: maintenance of equality, by proof, from starting program

Source Language

First order tail recursive functions over nested tuples of base types (**nat** and **word32**).

For example, the TEA block cipher can be defined in this syntax (all variables have type **word32**):

$$\textbf{ShiftXor}\,(x, s, k_0, k_1) = (x \ll 4 + k_0) \oplus (x + s) \oplus (x \ll 5 + k_1)$$

$$
\begin{array}{l}
\textbf{Rounds}\,(n, (y, z), (k_0, k_1, k_2, k_3), s) = \\
\quad \texttt{if}\ n = 0w\ \texttt{then}\ ((y, z), (k_0, k_1, k_2, k_3), s)\ \texttt{else} \\
\quad \textbf{Rounds}\ \ (n - 1w, \\
\qquad\qquad\qquad \texttt{let}\ s' = s + 2654435769w\ \texttt{in} \\
\qquad\qquad\qquad \texttt{let}\ y' = y + \textbf{ShiftXor}(z, s', k_0, k_1) \\
\qquad\qquad\qquad \texttt{in}\ ((y', z + \textbf{ShiftXor}(y', s', k_2, k_3)), (k_0, k_1, k_2, k_3), s')
\end{array}
$$

$$
\begin{array}{l}
\textbf{Encrypt}(keys, txt) = \\
\quad \texttt{let}\ (ctxt, keys, sum) = \textbf{Rounds}(32w, (txt, keys, 0w)) \\
\quad \texttt{in}\ ctxt
\end{array}
$$

Source Language

First order tail recursive functions over nested tuples of base types (**nat** and **word32**).

For example, the TEA block cipher can be defined in this syntax (all variables have type **word32**):

$$\textbf{ShiftXor}\ (x, s, k_0, k_1) = (x \ll 4 + k_0) \oplus (x + s) \oplus (x \ll 5 + k_1)$$

$$
\begin{array}{l}
\textbf{Rounds}\ (n, (y, z), (k_0, k_1, k_2, k_3), s) = \\
\quad \texttt{if}\ n = 0w\ \texttt{then}\ ((y, z), (k_0, k_1, k_2, k_3), s)\ \texttt{else} \\
\quad \textbf{Rounds}\ \ (n - 1w, \\
\qquad\qquad\quad \texttt{let}\ s' = s + 2654435769w\ \texttt{in} \\
\qquad\qquad\quad \texttt{let}\ y' = y + \textbf{ShiftXor}(z, s', k_0, k_1) \\
\qquad\qquad\quad \texttt{in}\ ((y', z + \textbf{ShiftXor}(y', s', k_2, k_3)), (k_0, k_1, k_2, k_3), s')
\end{array}
$$

$$
\begin{array}{l}
\textbf{Encrypt}(keys, txt) = \\
\quad \texttt{let}\ (ctxt, keys, sum) = \textbf{Rounds}(32w, (txt, keys, 0w)) \\
\quad \texttt{in}\ ctxt
\end{array}
$$

Compiler passes

- Flattening
- Unique naming
- Inlining
- Register allocation

Flattening

A uniform way to achieve this is with the CPS transformation. Although usually understood syntactically, it can also be defined as a higher order function:

$$\mathbf{C}\ e\ f = f(e)$$

Resulting rewrite rules:

[C_intro]	$e \longleftrightarrow \mathbf{C}\ e\ (\lambda x.x)$
[C_binop]	$\mathbf{C}\ (e_1\ \mathbf{op}_b\ e_2)\ k \longleftrightarrow \mathbf{C}\ e_1\ (\lambda x.\,\mathbf{C}\ e_2\ (\lambda y.\,\mathbf{C}\ (x\ \mathbf{op}_b\ y)\ k))$
[C_pair]	$\mathbf{C}\ (e_1, e_2)\ k \longleftrightarrow \mathbf{C}\ e_1\ (\lambda x.\,\mathbf{C}\ e_2\ (\lambda y.\,\mathbf{C}\ (x, y)\ k))$
[C_let_ANF]	$\mathbf{C}\ (\mathbf{let}\ v = e\ \mathbf{in}\ f\ v)\ k \longleftrightarrow \mathbf{C}\ e\ (\lambda x.\,\mathbf{C}\ (f\ x)\ (\lambda y.\,k\ y))$
[C_abs]	$\mathbf{C}\ (\lambda v.\,f\ v)\ k \longleftrightarrow \mathbf{C}\ (\lambda v.\,(\mathbf{C}\ (f\ v)\ (\lambda x.\,x)))\ k$
[C_app]	$\mathbf{C}\ (f\ e)\ k \longleftrightarrow \mathbf{C}\ f\ (\lambda g.\,\mathbf{C}\ e\ (\lambda x.\,\mathbf{C}\ (g\ x)\ (\lambda y.\,k\ y)))$

Flattening

Let's look at C_binop:

$$\mathbf{C}\ (e_1\ \mathbf{op}\ e_2)\ k \ \longleftrightarrow\ \mathbf{C}\ e_1\ (\lambda x.\, \mathbf{C}\ e_2\ (\lambda y.\, \mathbf{C}\ (x\ \mathbf{op}\ y)\ k))$$

Its effect as a rewrite rule is to push occurrences of **C** deeper into the compound expression, building up an incomprehensible linear structure.

Eventually, rewriting stops and we introduce **lets** :

$$\mathbf{C}\ e\ k \ \longleftrightarrow\ \mathtt{let}\ x = e\ \mathtt{in}\ k\ x$$

Flattening

Let's look at C_binop:

$$\mathbf{C}\ (e_1\ \mathbf{op}\ e_2)\ k \ \longleftrightarrow\ \mathbf{C}\ e_1\ (\lambda x.\,\mathbf{C}\ e_2\ (\lambda y.\,\mathbf{C}\ (x\ \mathbf{op}\ y)\ k))$$

Its effect as a rewrite rule is to push occurrences of **C** deeper into the compound expression, building up an incomprehensible linear structure.

Eventually, rewriting stops and we introduce **lets** :

$$\mathbf{C}\ e\ k \ \longleftrightarrow\ \mathtt{let}\ x = e\ \mathtt{in}\ k\ x$$

Flattening

Let's look at C_binop:

$$\mathbf{C}\ (e_1\ \mathbf{op}\ e_2)\ k \ \longleftrightarrow\ \mathbf{C}\ e_1\ (\lambda x.\, \mathbf{C}\ e_2\ (\lambda y.\, \mathbf{C}\ (x\ \mathbf{op}\ y)\ k))$$

Its effect as a rewrite rule is to push occurrences of **C** deeper into the compound expression, building up an incomprehensible linear structure.

Eventually, rewriting stops and we introduce **lets** :

$$\mathbf{C}\ e\ k \ \longleftrightarrow\ \texttt{let}\ x = e\ \texttt{in}\ k\ x$$

Example

Recall **ShiftXor**:

$$\vdash \textbf{ShiftXor}\ (x, s, k_0, k_1) = (x \ll 4 + k_0) \oplus (x + s) \oplus (x \ll 5 + k_1)$$

which our compiler flattens to the equal form

$$
\begin{array}{l}
\vdash \textbf{ShiftXor}(v_1, v_2, v_3, v_4) = \\
\quad \texttt{let } v_5 = v_1 \ll 4 \texttt{ in} \\
\quad \texttt{let } v_6 = v_5 + v_3 \texttt{ in} \\
\quad \texttt{let } v_7 = v_1 + v_2 \texttt{ in} \\
\quad \texttt{let } v_8 = v_6 \oplus v_7 \texttt{ in} \\
\quad \texttt{let } v_9 = v_1 \ll 5 \texttt{ in} \\
\quad \texttt{let } v_{10} = v_9 + v_4 \texttt{ in} \\
\quad \texttt{let } v_{11} = v_8 \oplus v_{10} \\
\quad \texttt{in } v_{11}
\end{array}
$$

Variable handling

The underlying deductive machinery of HOL-4 ensures that variables are automatically renamed, as needed to avoid name capture.

We also remove spurious bindings (var-var) and useless bindings with

$$\begin{array}{lcl} \textbf{let } x = v \textbf{ in } e[x] & \longleftrightarrow & e[v] \\ \textbf{let } x = e_1 \textbf{ in } e_2 & \longleftrightarrow & e_2 \end{array}$$

We also uniquely name each introduced **let** variable. This is just an α-conversion, and so preserves equality.

Variable handling

The underlying deductive machinery of HOL-4 ensures that variables are automatically renamed, as needed to avoid name capture.

We also remove spurious bindings (var-var) and useless bindings with

$$\begin{array}{lcl} \textbf{let } x = v \textbf{ in } e[x] & \longleftrightarrow & e[v] \\ \textbf{let } x = e_1 \textbf{ in } e_2 & \longleftrightarrow & e_2 \end{array}$$

We also uniquely name each introduced **let** variable. This is just an α-conversion, and so preserves equality.

Variable handling

The underlying deductive machinery of HOL-4 ensures that variables are automatically renamed, as needed to avoid name capture.

We also remove spurious bindings (var-var) and useless bindings with

$$\begin{array}{lcl} \textbf{let } x = v \textbf{ in } e[x] & \longleftrightarrow & e[v] \\ \textbf{let } x = e_1 \textbf{ in } e_2 & \longleftrightarrow & e_2 \end{array}$$

We also uniquely name each introduced **let** variable. This is just an α-conversion, and so preserves equality.

Inlining

This is just expansion of definitions, so trivially preserves equality. Framework automatically takes care of avoiding name clashes.

‘Small’ functions are inlined.

Recursive functions when inlined, are unrolled a small number of times.

Inlining opens up possibilities for constant folding and removing trivial bindings.

Upshot: what Norman Ramsey said.

Register allocation

Now the function has been translated to a form close to being processable by a machine.

Each **let** binding can be regarded as performing a machine operation or subroutine call and storing the result in a register.

But we have the unrealistic assumption that there are an unbounded number of registers. Enter register allocation.

Big advantage of TV: can use off-the-shelf register allocation algorithms and just verify the results of the allocation.

In previous work, we used a standard graph-colouring algorithm.

Register allocation

The gap between an unbounded number of virtual registers and a fixed number of real registers is bridged by use of memory.

Nice trick from Jason Hickey: use a naming convention on variables to say which are really registers and which are memory locations.

- v_i is a variable waiting to be allocated
- r_j is a register
- m_k is a memory location

Round definition

```
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')
```

Before register allocation

```
|- Round ((v1,v2),(v3,v4,v5,v6),v7) =
     let v8 = v7 + DELTA in
     let v9 = ShiftXor (v2,v8,v3,v4) in
     let v10 = v1 + v9 in
     let v11 = ShiftXor (v10,v8,v5,v6) in
     let v12 = v2 + v11
      in
        ((v10,v12),(v3,v4,v5,v6),v8)
```

After register allocation

Four available registers:

```
|- Round ((r0,r1),(r2,r3,m1,m2),m3) =
      let m4 = r2 in
      let r2 = m3 in
      let r2 = r2 + DELTA in
      let m3 = r3 in
      let r3 = ShiftXor (r1,r2,m4,m3) in
      let r0 = r0 + r3 in
      let r3 = ShiftXor (r0,r2,m1,m2) in
      let r1 = r1 + r3
       in
          ((r0,r1),(m4,m3,m1,m2),r2)
```

Lessons

Most compilation steps can be expressed as rewrite rules (local transformations).

Some transformations require proofs that $p_i = p_{i+1}$, where p_i is the whole program.

What then?

Have extended input language to

- polymorphism
- higher order functions
- user datatypes; complex pattern-matching

(See paper in TACAS 2008)

But also need to deal with generating code and embrace (finally) the operational semantics of the underlying machine.

Dealing with the machine

Arcane amount of detail, dealt with by proof automation for Hoare/Separation Logic

- generate code blindly following post-register allocated function
- apply Hoare rules following structure of the HOL function
  - sequential composition
  - conditional branches
  - loops (use induction theorem(s) for recursive functions to prove loop equal to recursion)
  - subroutines

Decompiling to Logic

Suppose you have to verify some assembly $\mathcal{A}$.

Wouldn't it be nice to automatically map $\mathcal{A}$ to a logic function **f** such that

$$\vdash \forall x.\ P\ (\mathbf{f}\ x)$$

would formally imply that P holds of any evaluation of $\mathcal{A}$.

Myreen has implemented decompilers from ARM, IA-32, and PPC to HOL. Has applied this in proof of correctness of a Cheney-style garbage collector, written in ARM.

See his webpage for details.

Future Work

- Still a bit of work to do to get an end-to-end compiler.
- Reduce the various types in the final program to a uniform encoding.
- Front end handles recursive datastructures, but back end needs a (verified) runtime system.
- Possibly utilize work of Myreen on verified g.c. and lisp interpreter
- Find interesting applications

Summary

- People have been writing and proving correctness of functional programs in theorem provers for quite a while.
- Compiling such functions by proof offers new opportunities in verified compilation.
- A theorem prover can be a good environment for writing a compiler, especially if proofs are important.
- Brings together kinds of verification: recursion/induction, Separation Logic.
- Delaying entry into world of operational semantics may have benefits.

THE END