Trusted Source Translation of a
Total Function Language

Guodong Li and Konrad Slind

School of Computing, University of Utah
{ligd, slind}@cs.utah.edu

Abstract. We present a trusted source translator that transforms total functions defined in the specification language of the HOL theorem prover to simple intermediate code. This translator eliminates polymorphism by code specification, removes higher-order functions through closure conversion, interprets pattern matching as conditional expressions, etc. The target intermediate language can be further translated by proof to a simple imperative language. Each transformation is proven to be correct automatically. The formalization, implementation and mechanical verification of all transformations are done in HOL-4.

1 Introduction

Giving realistic programming languages such as C and Java correct semantics is difficult. It is more difficult to make such semantics tractable so that we can reason about non-trivial programs in a formal setting. Some widely used functional languages have been given a formal semantics, *e.g.* Scheme has a denotational semantics [19] and ML has a formal operational semantics [14]. However, these semantics do not as yet provide a practical basis for formal reasoning about programs in the languages, although they are extremely valuable as reference documents and for proving meta-theorems (like type preservation).

In order to allow formal reasoning to the maximum extent, we can program applications directly in logic, and then compile the logic to realistic platforms for execution. Specifically, we can specify both the algorithms and the mathematics needed for their verification in higher order logic, and then compile the verified algorithms to low level platforms which are also modeled in the same logic.

The specification language we use is the Total Functional Language (TFL) [20], which is a pure, total functional programming layer on top of higher order logic and implemented in both the HOL-4 [17] and Isabelle [16] systems. TFL enables abstract algorithms to be specified in a mixture of mathematics and programming idioms and then reasoned about using a theorem prover. Roughly speaking, this language comprises ML-style pure terminating functional programs, *i.e.*, those (computable) functions that can be expressed by well-founded recursion in higher order logic. Features like type inference, polymorphism, higher-order functions and pattern matching make it a comfortable setting in which to program. This language can express a very wide range of algorithms. The trade-off is that the compilation of logic specifications written in this language is fairly complicated.

We have developed a software compiler [11,12], which produces assembly code, and a hardware compiler [21], which synthesizes Verilog netlists, for a small subset of TFL. This subset, named HOL-, is a simple monomorphically-typed functional language handling first order equations where variables range over tuples of booleans and 32-bit words. The software compiler performs normalization, inline expansion, nested function hoisting, register allocation and code generation to convert HOL- programs into assembly. Many transformations are implemented as rewrite rules [12]. The correctness of each transformation is proven on the fly: after a program is translated, a theorem is given as by-product that states the equivalence of the transformed code and this program.

This paper presents an extension of these compilers by strengthening the front end translation to support polymorphism, higher order functions, algebraic datatypes, pattern matching and other advanced features in TFL. As far as we know, it is the first verified translator that compiles logic specifications coded in such advanced functional languages as TFL. We also present an approach to translate HOL- into a simple imperative language. The mechanical correctness proof is performed deductively by synthesizing functions from imperative code and showing that these functions are equivalent to the original HOL- functions. This enables safe source translations from ML-like functional languages to imperative languages.

A TFL program is converted into an equivalent HOL- program via a sequence of transformations, the correctness of each of which is proved automatically in the logic system. Although standard compilation techniques developed for functional programming may be applied here, new challenges are posed due to the fact that (i) the source language is not visible in the logic — it is the logic itself that is taken as the source language; (ii) TFL programs have a set-theoretic semantics rather than an operational or denotational semantics; (iii) all transformations must be formalized and verified in the logic that is compiled. Since TFL and HOL-programs are not defined as datatypes and do not have an evaluation semantics, widely-used techniques that base on structural induction on syntax datatypes and rule-induction over evaluation relations cannot be applied here.

The main contribution of our work is that we construct and verify compilations for logic specifications written in the term language of a widely-used theorem prover. Users can model an application directly in HOL and prove properties about it, then our compilers translate it to low level code with a certificate (proof) showing that this code correctly implements the application. As a consequence, the execution of this code will always guarantee the properties proven on the original application.

2   TFL and HOL-

Both TFL and HOL- are subsets of the higher order logic built in HOL, thus their syntax and the semantics have already been defined. Programs written in them are simply mathematical functions defined in the HOL logic. It is this feature that enables us to use standard mathematics to prove properties of these

programs directly in the logic system. This supports much flexibility and allows the meaning of a program to be transparent. In particular, two programs are equivalent when the mathematical functions represented by them are equal.

One immediate advantage of taking TFL 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 forms, ending in HOL-.

TFL is a polymorphic, higher order, pure and terminating functional language supporting algebraic datatypes and pattern matching. Its syntax is shown in Figure 1, where $[term]_{separator}$ means a sequence of $term$'s separated by the $separator$. HOL- is a simple typed functional language handling first order equations over nested tuples of basic types. Clearly HOL- is a subset of TFL.

τ	$::= T \mid t \mid \tau\ D$	primitive type, type variable and algebraic type
	$\mid \tau\ \#\ \tau \mid \tau \rightarrow \tau$	tuple type and arrow(function) type
at_c	$::= id \mid id\ \text{of}\ [\tau]_\Rightarrow$	algebraic datatype clause
at	$::= \text{datatype}\ id = [at_c]_|$	algebraic datatype
	$\mid [at]_;$	mutually recursive datatype
pt	$::= i \mid v \mid c\ \overrightarrow{pt}$	pattern
e	$::= i : T \mid v : \tau$	constant and variable
	$\mid \overrightarrow{e}$	tuple, i.e.$[e]_,$
	$\mid p\ e$	primitive application
	$\mid c\ e$	constructor application
	$\mid f_{id}$	function identifier
	$\mid e\ e$	composite application
	$\mid \text{if}\ e\ \text{then}\ e\ \text{else}\ e$	conditional
	$\mid \text{case}\ e\ \text{of}\ [(c\ e) \rightsquigarrow e]_|$	case splitting
	$\mid \text{let}\ v = e\ \text{in}\ e$	let binding
	$\mid [\lambda v].\ e$	anonymous function
f_{decl}	$::= f_{id}\ ([pt]_,) = e$	pattern matching clause
	$\mid [f_{decl}]_\wedge$	function declaration
	$\mid v = e$	top level variable declaration

Fig. 1. The syntax of the total function language TFL

The translator performs transformations that are familiar from existing functional language compilers except that it does so by proof for the term language of HOL. TFL's high-level features such as polymorphism, higher-order functions, pattern matching and composite expressions need to be expressed in terms of HOL's much lower-level structures:

- The translator removes polymorphism from TFL programs by making duplications of polymorphic datatype declarations and functions for each distinct combination of instantiating types.
- The translator names intermediate computation results and makes the evaluation order explicit by performing a continuation-passing-style (CPS)

transformation. TFL expressions and functions are simplified to forms suitable for subsequent transformations.
– The translator applies defunctionalization to remove higher-order functions by creating algebraic datatypes to represent function closures and type based dispatch functions to direct the control to top level function definitions.
– The translator converts pattern matching first into nested case expressions, then into explicit conditional expressions.

All intermediate forms of a program are still mathematical functions defined in HOL. The correctness proof of a transformation of a source program p proceeds, in a *translation validation* [18] style, by showing the generated program q computes the same mathematical function as p. Note that the built-in type checker in HOL will type check both p and q to ensure their type safety. Two techniques are used to generate correctness proofs:

1. The transformation is implemented as a rewrite rule based on a theorem that is proven once and for all. In many cases we just need to instantiate this theorem by the input program. Examples include the normalization.
2. A per-run correctness check is performed to show that the transformation ensures semantics preserving on the given program. In general, we convert the source program p into p' using some algorithm, and then compare p and p' w.r.t their semantics. Examples include the monomorphisation and defunctionalization.

3   Trusted Transformation

In this section we describe algorithms of a series of syntax directed transformations; we also show how to prove the correctness of them.

3.1   Monomorphisation

This transformation eliminates polymorphism and produces a simply-typed intermediate form that enables good data representations. The basic idea is to duplicate a datatype declaration at each type used and a function declaration at each type used, resulting in multiple monomorphic clones of this datatype and function. This step paves the way for subsequent conversions such as the type based defunctionalization. Although this seems to lead to code explosion in theory, it is manageable in practice (MLton, a fancy ML compiler, uses similar techniques and reports maximum increase of 30% in code size).

The first step is to build an instantiation map that enumerates for each datatype and function declaration the full set of instantiations for each polymorphic type. A TFL program will be type checked by the HOL system and be annotated with polymorphic type identifiers such as $'a, 'b, \ldots$ when it is defined. In particular, type inference has been done for (mutually) recursive functions. The remaining task is to instantiate the generic types of a function with the actual types of arguments at its call sites.

The notation used in this section is as follows. A substitution rule $R = (t \hookrightarrow \{T\})$ maps an abstract type t to a set of its type instantiations; an instantiation set $S = \{R\}$ is a set of substitution rules; and an instantiation map $M = \{z \hookrightarrow S\}$ maps a datatype or a function z to its instantiation set S. We write $M.y$ for the value at field y in the map M; if $y \notin Dom\ M$ then $M.y$ returns an empty set. The union of two substitution sets $S_1 \cup_s S_2$ is $\{t \hookrightarrow S_1.t \cup S_2.t \mid t \in Dom\ S_1 \cup Dom\ S_2\}$. We write $\bigcup_s \{S\}$ for the combined union of a set of substitution rules. The union of two instantiation maps $M_1 \bigcup_m M_2$ is defined similarly. The composition of two instantiation sets S_1 and S_2, denoted as $S_1\ o_r\ S_2$, is $\{t \hookrightarrow \bigcup \{S_2.t \mid t \in Dom\ S_1\} \mid z \in Dom\ S_1\}$. And, the composition of an instantiation map M and a set S is defined as $M\ o_m\ S = \{z \hookrightarrow M.z\ o_r\ S \mid z \in Dom\ M\}$.

The instantiation information of each occurrence of a polymorphic function and datatype is coerced into an instantiation map during a syntax directed bottom-up traversal. The main conversion rules Γ and Δ shown in Fig. 2 build the instantiation map by investigating types and expressions respectively. The rule for a single variable/function declaration is trivial and omitted here: we just need to walk over the right hand side of its definition. If a top level function f is called in the body of another function top level g, then g must be visited first to generate an instantiation map M_g, and then f is visited to generate M_f; finally these two maps are combined to a new one, *i.e.* $((M_f \circ M_g.f) \cup_m M_g)$. The clauses in mutually recursive functions can be visited in an arbitrary order.

$$
\begin{aligned}
\Gamma[[\tau]] &= \{\}, \quad \text{for } \tau \in \{T, t\} \\
\Gamma[[\tau\ D]] &= \{D \hookrightarrow \mathsf{match\_tp}\ (\mathsf{at\_tp}\ D)\ \tau\} \\
\Gamma[[\tau_1\ op_t\ \tau_2]] &= \Gamma[[\tau_1]] \cup_m \Gamma[[\tau_2]], \quad \text{for } op_t \in \{\#, \rightarrow\} \\
\Delta[[i]] &= \{\} \\
\Delta[[v : \tau]] &= \Gamma[[\tau]] \\
\Delta[[[e],]] &= \bigcup_m \{\Gamma[[e]]\} \\
\Delta[[p\ e]] &= \Delta[[e]] \\
\Delta[[(c : \tau)\ e]] &= \{\mathsf{con2tp}\ c \hookrightarrow \mathsf{match\_tp}\ (\mathsf{con2tp}\ c)\ \tau\} \\
&\quad \cup_m \Gamma[[\tau]] \cup_m \Delta[[e]] \\
\Delta[[(f : \tau)\ e]] &= \{f_{id} \hookrightarrow \mathsf{match\_tp}\ f_{id}\ \tau\} \cup_m \Gamma[[\tau]] \cup_m \Delta[[e]] \\
\Delta[[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]] &= \Delta[[e_1]] \cup_m \Delta[[e_2]] \cup_m \Delta[[e_3]] \\
\Delta[[\text{case } e_1 \text{ of } [((c : \tau)\ e_2) \rightsquigarrow e_3]_|]] &= \Delta[[e_1]] \cup_m \bigcup_m \{\{\mathsf{con2tp}\ c \hookrightarrow \mathsf{match\_tp}\ (\mathsf{con2tp}\ c)\ \tau\} \\
&\quad \cup_m \Delta[[e_2]] \cup_m \Delta[[e_3]]\} \\
\Delta[[\text{let } v = e_1 \text{ in } e_2]] &= (\Delta[[e_1]]\ o_m\ \Delta[[e_2]].v) \cup_m \Delta[[e_2]] \\
\Delta[[[\lambda v.]^* e]] &= \Delta[[e]]
\end{aligned}
$$

Fig. 2. Build instantiation maps for polymorphic components

This algorithm makes use of a couple of auxiliary functions provided by the HOL system. Function $\mathsf{con2tp}\ c$ maps a constructor c to the datatype to which it belongs; $\mathsf{at\_tp}\ D$ returns σ if there is a datatype definition $\mathsf{datatype}\ \sigma = D$ of $\ldots$; when x is either a function name or a constructor, $\mathsf{match\_tp}\ x\ \tau$ matches the original type of x (*i.e.* the type when x is defined) with τ and returns a substitution set.

After the final instantiation map is obtained, we duplicate a polymorphic datatype/function for all combinations of its type instantiations, and replace each call of the polymorphic function with the call to its monomorphic clone with respect to the type. The automatic correctness proof for the transformation is trivial: each duplication of a polymorphic function computes the same function on the arguments of the instantiating types.

Now we give a simple example to illustrate the transformation.

$$\text{datatype } \sigma = C \text{ of } 'a \,\#\, 'b \qquad f \,(x :' a) = x$$
$$g \,(x :' c, y :' d) = \text{let } (h :' d \to ('c \,\#\, 'd) \,\sigma) = \lambda z :' d.$$
$$(C : ('c \,\#\, 'd) \to ('c \,\#\, 'd) \,\sigma) \,((f :' c \to' c) \,x, (f :' d \to' d) \,z) \text{ in } h \, y$$
$$j = (g \,(1 : num, \bot : bool), \; g \,(\bot : bool, \top : bool))$$

The algorithm builds the following instantiation maps:

Investigate j : $M_j = \{g \hookrightarrow \{'c \hookrightarrow \{bool, num\}, 'd \hookrightarrow \{bool\}\}\}$
Investigate g : $M_g = \{f \hookrightarrow \{'a \hookrightarrow \{'c,' d\}\}, \sigma \hookrightarrow \{'a \hookrightarrow \{'c\}, 'b \hookrightarrow \{'d\}\}\}$
Compose M_g and M_j : $M_{g \circ j} = M_g \circ M_j.g =$
    $\{ f \hookrightarrow \{'a \hookrightarrow \{bool, num\}\}, \sigma \hookrightarrow \{'a \hookrightarrow \{bool, num\}, 'b \hookrightarrow \{bool\}\} \}$
Union M_g and $M_{g \circ j}$: $M_{\{g,j\}} = M_g \cup_m M_j =$
    $\{ f \hookrightarrow \{'a \hookrightarrow \{bool, num\}\}, g \hookrightarrow \{'c \hookrightarrow \{bool, num\}, 'd \hookrightarrow \{bool\}\},$
    $\sigma \hookrightarrow \{'a \hookrightarrow \{bool, num\}, 'b \hookrightarrow \{bool\}\} \}$
Investigate f : no changes, $M_{\{f,g,j\}} = M_{\{g,j\}}$

Then for datatype σ, function f and function g, a monomorphic clone is created for each combination of instantiating types. Calls to the original functions are replaced with the appropriate copies of the right type. For example, function j is converted to $j = (g_{num\#bool} \,(1, \bot), \; g_{bool\#bool} \,(\bot, \top))$, where $g_{num\#bool}$ and $g_{bool\#bool}$ are the two clones of g. The correctness of j's conversion is proved based on the theorems showing that g's copies compute the same function as g with respect to the instantiating types: $\Vdash_{thm} g_{num\#bool} = g \; \wedge \; g_{bool\#bool} = g$.

3.2   Normalization

This transformation bridges the gap between the form of expressions and control flow structures in TFL and HOL-. A TFL program is converted to a simpler form such that: (1) the arguments to function and constructor applications are atoms like variables or constants; (2) discriminators in case expressions are also simple expressions; (3) compound expressions nested in an expression are lifted to make new 'let' bindings; (4) curried functions are uncurried to a sequence of simple functions that take a single tupled argument. Primitive expressions such as arithmetic and logical expressions on atoms need not to be converted.

A continuation-passing-style (CPS) transformation is performed to normalize TFL programs. The essence is to sequentialize the computation of TFL expressions by introducing variables for intermediate results, and the control flow is pinned down into a sequence of elementary steps. It extends the one in our software compiler [12] by addressing higher level structures specific to TFL. In the following rules, $C \, e \, k$ denotes the application of the continuation k to an expression e, and its value is equal to $k \, e$. After the conversion, we rewrite with the

theorem $C\ e\ k = $ let $x = e$ in $k\ x$ to obtain 'let'-based normal forms.

$C\ [[e]]\ k = k\ e,$ when e is a primitive expression
$C\ [[\lambda\overrightarrow{v}.e]] = \lambda\overrightarrow{v}.\lambda k.\ C\ [[e]]\ k$
$C\ [[(e_1, e_2)]]\ k = C\ [[e_1]]\ (\lambda x.\ C\ [[e_2]]\ (\lambda y.\ k\ (x,y)))$
$C\ [[op\ e]]\ k = C\ [[e]]\ (\lambda x.\ k\ (op\ x))$ when $op \in \{p, c, f_{id}\}$
$C\ [[(e_1\ e_2)]]\ k = C\ [[e_1]]\ (\lambda x.\ C\ [[e_2]]\ (\lambda y.\ k\ (x\ y)))$
$C\ [[$let $v = e_1$ in $e_2]]\ k = C\ [[e_1]]\ (\lambda x.\ C\ [[e_2]]\ (\lambda y.\ k\ y))$
$C\ [[$if e_1 then e_2 else $e_3]]\ k = $
    $C\ [[e_1]]\ (\lambda x.\ k\ ($if x then $C\ [[e_2]]\ (\lambda x.x)$ else $C\ [[e_3]]\ (\lambda x.x)))$
$C\ [[$case e_1 of $c\ e_{2_1} \rightsquigarrow e_{3_1} \mid c\ e_{2_2} \rightsquigarrow e_{3_2} \mid \ldots]]\ k = $
    $C\ [[e_1]]\ (\lambda x.\ (C\ [[e_{2_1}]]\ (\lambda y_1.\ C\ [[e_{2_2}]]\ (\lambda y_2.\ \ldots,$
      $k\ ($case x of $c\ y_1 \rightsquigarrow C\ [[e_{3_1}]]\ (\lambda x.x) \mid c\ y_2 \rightsquigarrow C\ [[e_{3_2}]]\ (\lambda x.x) \mid \ldots)))))$

The following example illustrates this transformation, where c_1 and c_2 are the two constructors of a datatype.

Original: $f\ (x, y, z) = $ case $x - y - z$ of $c_1\ a \Rightarrow f(x - 1, a, y) \mid c_2\ b \Rightarrow b + y$
Converted: $f\ (x, y, z) = $
    let $v_1 = x - y - z$ in
        case v_1 of $c_1\ a \Rightarrow$ let $v_2 = x - 1$ in $f(v_2, a, y) \mid c_2\ b \Rightarrow b + y$

3.3   Defunctionalization

In this section we convert higher-order functions into equivalent first-order functions and hoist nested functions to the top level through a type based closure conversion. After the conversion, no nested functions exist; and function call is made by dispatching on the closure tag followed by a top-level call.

Function closures are represented as algebraic data types in a way that, for each function definition, a constructor taking the free variables of this function is created. For each arrow type we create a dispatch function, which converts the definition of a function of this arrow type into a closure constructor application. A nested function is hoisted to the top level with its free variables to be passed as extra arguments. After that, the calling to the original function is replaced by a calling to the relevant dispatch function passing a closure containing the values of this function's free variables. The dispatch function examines the closure tag and passes control to the appropriate hoisted function. Thus, higher order operations on functions are replaced by equivalent operations on first order closure values.

As an optimization, we first run a pass to identify all 'targeted' functions which appear in the arguments or outputs of other functions and record them in a side effect variable `Targeted`. Non-targeted functions need not to be closure converted, and calls to them are made as usual. During this pass we also find out the functions to be defined at the top level and record them in `Hoisted`. Finally `Hoisted` contains all top level functions and nested function to be hoisted.

The conversion works on simple typed functions obtained by monomorphisation. We create a closure datatype and a dispatch function for each of the arrow types that targeted functions may have. A function definition is replaced by a binding to an application of the corresponding closure constructor to this function's free variables. Suppose the set of targeted functions of type τ is

$\{f_i\ x_i = e_i \mid i = 1, 2, \dots\}$, then the following algebraic datatype and dispatch function are created, where tp_of and fv return the type and free variables of a term respectively (and the type builder Γ will be described below):

$$clos_\tau = cons_{f_1}^\tau \text{ of } \Gamma[[\text{tp\_of (fv } f_1)]] \mid cons_{f_2}^\tau \text{ of } \Gamma[[\text{tp\_of (fv } f_2)]] \mid \dots$$
$$(dispatch_\tau(cons_{f_1}^\tau, x_1, y_1) = (f_1 : \Gamma[[\tau]])\ (x_1, y_1))\quad \wedge$$
$$(dispatch_\tau(cons_{f_2}^\tau, x_2, y_2) = (f_2 : \Gamma[[\tau]])\ (x_2, y_2))\quad \wedge$$
$$\dots$$

As shown in Fig. 3, the main translation algorithm inspects the references and applications of targeted functions and replaces them with the corresponding closures and dispatch functions. Function Γ returns the new types of variables. When walking over expressions, Δ replaces calls to unknown functions (*i.e.* those not presented in <u>Hoisted</u>) with calls to the appropriate dispatch function, and calls to known functions with calls to hoisted functions. In this case the values of free variables are passed as extra arguments. Function references are also replaced with appropriate closures. Finally <u>Redefn</u> contains all converted functions, which will be renamed and redefined in HOL at the top level.

$$
\begin{aligned}
&\Gamma[[v : T]] &&= T\\
&\Gamma[[v : \tau_1 \to \tau_2]] &&= \text{if } v \in \underline{\texttt{Targeted}} \text{ then } clos_{\tau_1 \to \tau_2} \text{ else } \tau_1 \to \tau_2\\
&\Gamma[[v : \tau\ D]] &&= \Gamma[[\tau]]\ \overline{D}\\
&\Gamma[[[v],]] &&= [\Gamma[[v]]],\\
&\Delta[[v : \tau]] &&= \text{if } v \in \underline{\texttt{Targeted}} \text{ then } cons_v^\tau \text{ else } v : clos_\tau\\
&\Delta[[[e],]] &&= [\Delta[[e]]],\\
&\Delta[[p\ e]] &&= p\ (\Delta[[e]])\\
&\Delta[[c\ e]] &&= c\ (\Delta[[e]])\\
&\Delta[[(f : \tau)\ e]] &&= \text{if } f \in \underline{\texttt{Hoisted}} \text{ then (new\_name\_of } f)\ (\Delta[[e]], \text{fv } f)\\
&&&\quad\ \ \text{else } dispatch_\tau\ (f : clos_\tau, \Delta[[e]])\\
&\Delta[[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]] &&= \text{if } \Delta[[e_1]] \text{ then } \Delta[[e_2]] \text{ else } \Delta[[e_3]]\\
&\Delta[[\text{case } e_1 \text{ of } [c\ e_2 \rightsquigarrow e_3]_|]] &&= \text{case } \Delta[[e_1]] \text{ of } [(\Delta[[c\ e_2]]) \rightsquigarrow \Delta[[e_3]]]_|\\
&\Delta[[\text{let } f = \lambda\overrightarrow{v}.\,e_1 \text{ in } e_2]] &&= (\Phi[[f\ \overrightarrow{v} = e_1]]\ ;\ \Delta[[e_2]])\\
&\Delta[[\text{let } v = e_1 \text{ in } e_2]] &&= \text{let } v = \Delta[[e_1]] \text{ in } \Delta[[e_2]]\quad \text{when } e_1 \text{ is not a } \lambda \text{ expression}\\
&\Phi[[f_{id}\ (\overrightarrow{v} : \tau) = e]] =\\
&\quad \text{let } e' = \Delta[[e]] \text{ in}\\
&\quad \underline{\texttt{Redefn}} := \underline{\texttt{Redefn}} + (f_{id} \hookrightarrow \underline{\texttt{Redefn}}.f_{id} \cup \{(f_{id} : \tau \to \Gamma[[\text{tp\_of } e']])\ \overrightarrow{v} = e'\}\\
&\Phi[[[f_{decl}]_\wedge]] = [\Phi[[f_{decl}]]];
\end{aligned}
$$

Fig. 3. Remove higher order functions through closure conversion

Now we show the technique to prove the equivalence of a source function f to its converted form f'. We say that a variable $v' : \tau'$ corresponds $v : \tau$ iff: (1) $v = v'$ if both τ and τ' are closure type or neither of them is. (2) $\forall x \forall x'.\ dispatch_{\tau'}(v', x') = v\ x$ if v' is a closure type and v is an arrow type, and x' corresponds to x; or vice versa. Then f' is equivalent to f iff they correspond to each other. The proof process is simple, as it suffices to simply rewrite with the old and new definitions of the functions.

As an example, the following higher order program

$$f\ (x : num) = x * 2 < x + 10$$
$$g\ (s : num \rightarrow bool, x : num) =$$
$$\quad \text{let } h_1 = \lambda y.\, y + x \text{ in if } s\ x \text{ then } h_1 \text{ else let } h_2 = \lambda y.\, h_1\ y * x \text{ in } h_2$$
$$k\ (x : num) = \text{if } x = 0 \text{ then } 1 \text{ else } g\ (f, x)\ (k\ (x - 1))$$

is closure converted to

datatype $clos_{\tau_1} = cons_f^{\tau_1}$
datatype $clos_{\tau_2} = cons_{h_1}^{\tau_2}$ of $num \mid cons_{h_2}^{\tau_2}$ of num
$dispatch_{\tau_1}\ (cons_f^{\tau_1} : clos_{\tau_1}, x : num) = f'\ x\ \ \wedge\ \ f'\ x = x * 2 < x + 10$
$dispatch_{\tau_2}\ (cons_{h_1}^{\tau_2}\ y : clos_{\tau_2},\ x : num) = h_1'\ (y, x))\ \wedge$
$dispatch_{\tau_2}\ (cons_{h_2}^{\tau_2}\ y : clos_{\tau_2},\ x : num) = h_2'\ (y, x))\ \wedge$
$h_1'\ (y, x) = y + x\ \ \wedge\ \ h_2'\ (y, x) = h_1'(y, x) * x$
$g'\ (s : clos_{\tau_1}, x : num) = \text{if } dispatch_{\tau_1}\ (s, x) \text{ then } cons_{h_1}^{\tau_2}\ x \text{ else } cons_{h_2}^{\tau_2}\ x$
$k'\ (x : num) = \text{if } x = 0 \text{ then } 1 \text{ else } g\ (cons_f^{\tau_1}, x), (k'\ (x - 1))$
where τ_1 and τ_2 stand for arrow types $num \rightarrow bool$ and $num \rightarrow num$ respectively

And the following theorems (which are proved automatically) justify the correctness of this conversion:

$\Vdash_{thm}\ f = f'\qquad \Vdash_{thm}\ k' = k$
$\Vdash_{thm}\ (\forall x.\, dispatch_{\tau_1}\ (s', x) = s\ x) \Rightarrow \forall x \forall y.\, dispatch_{\tau_2}\ (g'\ (s', x), y) = (g\ (s, x))\ y$

3.4   Pattern Matching

This conversion to nested case expressions is based on Augustsson's original work [1], which was adapted by Slind [20] for function description in HOL. A pre-processing pass is first performed to deal with incomplete and overlapping patterns: incomplete patterns are made complete by adding rows for all missing constructors; and overlapping patterns are handled by replacing a value with possible constructors. Note that this approach may make the pattern exponentially larger because no heuristics are used to choose the "best" order in which subterms of any term are to be examined.

The translation rule Δ shown below converts patterns $[pat_i \rightsquigarrow rhs_i]_|$ into a nested case expression. It takes two arguments: a stack of variables that are yet to be matched, and a matrix whose rows correspond to the clauses in the pattern. All rows are of equal length, and the elements in a column should have the same type.

Conversion Δ proceeds from left to right, column by column. At each step the first column is examined. If each element in this column is a variable, then the head variable z in the stack is substituted for the corresponding v_i for the right hand side of each clause. If each element in the column is the application of a constructor for type τ, and τ contains constructor $C_1, \ldots, C_n$, then the rows are partitioned into n groups of size $k_1, \ldots, k_n$ according to the constructors. After partitioning, a row $(C(\bar{p}) :: pats;\ rhs)$ has its lead constructor discarded, resulting in a row expression $(\bar{p} \,@\, pats; rhs)$. Here :: is the list constructor, and @ appends the second list to the first one. If constructor C_i has type $\tau_1 \rightarrow$

$\cdots \to \tau_j \to \tau$, then a set ν_i of new variables $v_1, \ldots, v_j$ are pushed onto the stack. Finally the results for all groups are combined into a case expression on the head of the stack.

$$\Delta \begin{pmatrix} z :: stack \\ \overline{v_1 :: pats_1 \rightsquigarrow rhs_1,} \\ \cdots \\ v_n :: pats_n \rightsquigarrow rhs_n \end{pmatrix} = \Delta \begin{pmatrix} stack \\ \overline{pats_1 \rightsquigarrow rhs_1[z \leftarrow v_1],} \\ \cdots \\ pats_n \rightsquigarrow rhs_n[z \leftarrow v_2] \end{pmatrix}, \text{ and}$$

$$\Delta \begin{pmatrix} z :: stack \\ \overline{C_1\ \overline{p_{11}} :: pats_{11} \rightsquigarrow rhs_{11},} \\ \cdots \\ C_n\ \overline{p_{1k_1}} :: pats_{1k_1} \rightsquigarrow rhs_{1k_1} \\ C_n\ \overline{p_{n1}} :: pats_{n1} \rightsquigarrow rhs_{11}, \\ \cdots \\ C_n\ \overline{p_{1k_n}} :: pats_{nk_n} \rightsquigarrow rhs_{nk_n} \end{pmatrix} = \mathsf{tp\_case}\ (\lambda \nu_1.M_1)\ \ldots\ (\lambda \nu_n.M_n)\ z$$

$$\text{where } M_i = \Delta \begin{pmatrix} \nu_1 :: stack \\ \overline{\overline{p_{i1}}\ @\ pats_{k1} \rightsquigarrow rhs_{k1},} \\ \cdots \\ \overline{p_{ik_i}}\ @\ pats_{ik_i} \rightsquigarrow rhs_{ik_i} \end{pmatrix} \text{ for } i = 0, \ldots, n$$

When a datatype tp with n constructors is declared, a case expression theorem $\forall x.\, tp\_case\ f_1\ \ldots\ f_n\ (C_i\ x) \equiv f_i\ x$ for $i = 1, \ldots, n$ is stored in HOL. For example, the case expression for natural number is $(\mathsf{num\_case}\ b\ f\ 0 = b)\ \wedge$ $(\mathsf{num\_case}\ b\ f\ (\mathsf{Suc}\ n) = f\ n)$.

For example, this step translates the Greatest Common Divisor function gcd to a form taking only one argument:

$gcd\ (0, y) = y \qquad gcd\ (\mathsf{Suc}\ x, 0) = \mathsf{Suc}\ x$
$gcd\ (\mathsf{Suc}\ x, \mathsf{Suc}\ y) = \mathsf{if}\ y \leq x\ \mathsf{then}\ gcd\ (x - y, \mathsf{Suc}\ y)\ \mathsf{else}\ gcd\ (\mathsf{Suc}\ x, y - x)$
$\qquad \Rightarrow$
$gcd\ z = \mathsf{pair\_case}\ (\lambda v\ v_1.\ \mathsf{num\_case}\ v_1 (\lambda v_2.\ \mathsf{num\_case}\ (\mathsf{Suc}\ v_2)$
$\quad (\lambda v_3.\ \mathsf{if}\ v_3 \leq v_2\ \mathsf{then}\ gcd\ (v_2 - v_3, \mathsf{Suc}\ v_3)\ \mathsf{else}\ gcd\ (\mathsf{Suc}\ v_2, v_3 - v_2))\ v_1)\ v)\ z$

In the next step case expressions are interpreted as conditional expressions based on the following theorem

$\mathsf{tp\_case}\ (\lambda x.f_1\ x)\ (\lambda x.f_2\ x)\ \ldots\ z =$
$\mathsf{if\ is\_}C_1\ z\ \mathsf{then}\ f_1\ (\mathsf{destruct}_{C_1}\ z)\ \mathsf{else\ if\ is\_}C_2\ z\ \mathsf{then}\ f_2\ (\mathsf{destruct}_{C_2}\ z)\ \mathsf{else} \ldots$

where operator $\mathsf{is\_}C_i$ tells whether a variable matches the ith constructor C_i, i.e. $\mathsf{is\_}C_i\ (C_j\ x) = \top$ iff $i = j$; and operator $\mathsf{destruct}_{C_i}$ is the destructor function for constructor C_i. For example, $\mathsf{destruct}_{\mathsf{Suc}}\ (\mathsf{Suc}\ x) = x$. These operators will be implemented as datatype access operations in later compilation phases. In addition, an optimization is performed to tuple variables: if an argument x has type $\tau_1 \# \ldots \# \tau_n$, then it is replaced by a tuple of new variables $(x_1, \ldots, x_n)$. Superfluous branches and 'let' bindings are also removed. In this manner the gcd function is converted to

$$gcd\ (z_1, z_2) = \text{if } z_1 = 0 \text{ then } z_2$$
$$\text{else let } v_2 = \text{destruct}_{Suc}\ z_1 \text{ in}$$
$$\text{if } z_2 = 0 \text{ then Suc } v_2 \text{ else let } v_3 = \text{destruct}_{Suc}\ z_2 \text{ in}$$
$$\text{if } v_3 \le v_2 \text{ then } gcd\ (v_2 - v_3, \text{Suc } v_3) \text{ else } gcd\ (\text{Suc } v_2, v_3 - v_2)$$

4   Producing-by-Proof Imperative Code

Porting pure and terminating ML programs into TFL is easy due to the high similarity in the syntax and semantics of ML and TFL. One of the main issues — the termination proof of the imported ML program — is handled by proving that the generated TFL function is total. Moreover, the imported programs will be type checked by HOL. As the translation from TFL to HOL- eliminates features pertaining to functional languages such as higher order functions and nested expressions, it is natural to consider translating HOL- to realistic imperative languages such as C and Java.

We have developed a method in our software compiler [11] that translates simple normal forms obtained from HOL- programs to a low level imperative language HSL (Heap and Stack Level). HSL supports various structured control statements including blocks, sequential compositions, conditionals, tail recursions, and function calls. However, since HSL is designed to couple tightly with the targeted machine language and accesses registers and heaps directly, it is not a good candidate as the target imperative language.

We extend HSL to a higher level imperative language IL (for *Imperative Language*). The global variables of IL correspond to top level variables in HOL-; and local variables in IL correspond to the administrative redexes (*i.e.* left hand sides) of 'let' expressions in HOL-. IL also inherits the datatypes from HOL-, thus no datatype representation is needed. What's more important is our augmentation of the reasoning mechanism: we maintain a set of separate logic judgments rather than just one judgment (as we did in [11]) and use them to reason about programs. The syntax of IL's control flow structures is shown below.

$$
\begin{array}{lll}
s ::= & v := s & \text{assignment} \\
& |\ \texttt{return}\ v & \text{return} \\
& |\ s; s & \text{sequential statement} \\
& |\ \texttt{IF}\ e\ \texttt{THEN}\ s\ \texttt{ELSE}\ s & \text{conditional jump} \\
& |\ \texttt{WHILE}\ e\ s & \text{loop} \\
& |\ v :=_f\ p_{id}\ s & \text{function call} \\
p ::= & p_{id}\ (\overrightarrow{v}) = s & \text{programs}
\end{array}
$$

We first define an operational semantics (omitted here due to lack of space) for IL and then derive an axiomatic semantics from it. Each axiomatic semantics rule is specified as a Hoare triple $\{precondition\}\ program\ \{postcondition\}$:

$$\frac{\{P\}\ S_1\ \{Q\} \quad \{R\}\ S_2\ \{T\} \quad Q \Rightarrow R}{\{P\}\ (S_1\ ; S_2)\ \{T\}} \qquad \frac{\{P\}\ S\ \{P\}}{\{P\}\ (\texttt{WHILE}\ C\ S)\ \{P \wedge \neg C\}}$$

$$\frac{\{P \wedge C\}\ S_t\ \{Q\} \quad \{P \wedge \neg C\}\ S_f\ \{Q\}}{\{P\}\ (\texttt{IF}\ C\ \texttt{THEN}\ S_t\ \texttt{ELSE}\ S_f)\ \{Q\}} \qquad \frac{\{P\}\ S_t\ \{Q\} \quad \{P\}\ S_f\ \{R\}}{\{P\}\ (\texttt{IF}\ C\ \texttt{THEN}\ S_t\ \texttt{ELSE}\ S_f)\ \{\texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ R\}}$$

In order to connect the semantics of a IL program s with that of a HOL- function f, we introduce the following rule to characterize s's axiomatic semantics as a set of predicates (where $\sigma\langle x \rangle$ returns the value of variable x in state σ; and eval $S\ \sigma$ returns the new state after S's execution):

$$s \vdash \{(\bar{i}_k, f_k\ \bar{i}_k, \bar{o}_k)\} \doteq \forall k \forall \sigma \forall \bar{v}_k.(\sigma\langle \bar{i}_k \rangle = \bar{v}_k) \Rightarrow ((\text{eval } S\ \sigma)\langle \bar{o}_k \rangle = f_k\ \bar{v}_k)$$

The kth predicate $(\bar{i}_k,\ f_k\ \bar{i}_k,\ \bar{o}_k)$ specifies that: if inputs $\bar{i}_k$ have initial values $\bar{v}_k$, then in the state after the execution of s, the values left in outputs $\bar{o}_k$ are equal to applying the function f_i to the initial values $\bar{v}_k$. Such a rule is obtained by instantiating the P and Q in $\{P\}\ s\ \{Q\}$ to $\lambda\sigma.\forall k.\sigma\langle \bar{i}_k \rangle = \bar{v}_k$ and $\lambda\sigma.\forall k.\sigma\langle \bar{o}_k \rangle = f_i\ \bar{v}_k$ respectively. We also write e_k for $f_k\ \bar{i}_k$ if the context is clear. If the judgment embodied by a predicate synthesizes f on inputs $\bar{i}$ and outputs $\bar{o}$, then we claim that s correctly implements f with respect to $\bar{i}$ and $\bar{o}$.

In a preprocessing step, tail recursive HOL- functions are rewritten to equivalent 'while' forms [12], where while $c\ f \doteq \lambda x.\text{if } \neg c\ x \text{ then } x \text{ else while } (f\ x)$. Currently this preprocessing admits only tail recursive programs; mutually recursive functions are not supported yet.

We derive a couple of rules to mechanically synthesize for an IL program the functions it correctly implements. The rules utilize the following definitions. Notation Δ converts a HOL- variables and program fragments to TFL terms. $\bigcup(\bar{i}_k)$ constructs a tuple from the union of $\bar{i}_k$ for all k. As usual, $[\texttt{let } o_i = f_k\ \bar{i}_k]_{\texttt{in}}$ stands for a chain of 'let' bindings: $\texttt{let } o_1 = f_1\ i_1 \texttt{ in let } o_2 = f_2\ i_2 \texttt{ in} \ldots$. Rule refl, assgn and return build basic predicates in accordance to TFL 's semantics. Rule cond, while and application are used to synthesize functions for conditional statements, loops and function calls respectively. The predicate sets are manipulated by the union rule union and the elimination rule elim. As the inputs and outputs of s are tuples of arbitrary arity, we provide a shuffle rule to change the structures of them. This rule is particularly useful when we need to match the inputs and outputs of a synthesized function with those of the original function. The sequential composition rule seq is the most complicated one. For each variable $o1_k$ in s_2's inputs, this rule looks up a predicate $(\bar{i1}_k, e1_k, \bar{o1}_k)$ in Σ_1, and inserts a let binding of $e1_k$ to $\bar{o1}_k$ into the composed expression.

$$\frac{}{\vdash \{(\bar{i}, \bar{i}, \bar{i})\}}\ \text{refl} \qquad \frac{}{out := f\ in \vdash \{(\Delta\ in, \Delta\ (f\ in), \Delta\ out)\}}\ \text{assgn}$$

$$\frac{}{\texttt{return } out \vdash \{(\Delta\ out, \Delta\ out, \Delta\ out)\}}\ \text{return}$$

$$\frac{s_1 \vdash \Sigma_1 \quad s_2 \vdash \Sigma_2 \quad \{(\bar{i1}_k, e1_k, \bar{o1}_k)\} \subseteq \Sigma_1 \quad (\bar{o1}, e2, \bar{o2}) \in \Sigma_2}{s_1\ ;\ s_2 \vdash \{(\bigcup(\bar{i1}_k), [\texttt{let } o1_k = e1_k]_{\texttt{in}}\ e2, \bar{o2})\}}\ \text{seq}$$

$$\frac{s_1 \vdash \Sigma_1 \quad s_2 \vdash \Sigma_2 \quad (\bar{i}, e1, \bar{o}) \in \Sigma_1 \quad (\bar{i}, e2, \bar{o}) \in \Sigma_2}{\texttt{IF } cnd \texttt{ THEN } s_1 \texttt{ ELSE } s_2 \vdash \{(\bar{i}, \texttt{if } (\Delta\ cnd) \texttt{ then } e1 \texttt{ else } e2, \bar{o})\}}\ \text{cond}$$

$$\frac{s \vdash \Sigma \quad (\bar{i}, e, \bar{i}) \in \Sigma}{\texttt{WHILE } cnd\ s \vdash \{(\bar{i}, \texttt{while } (\Delta\ cnd)\ e, \bar{i})\}}\ \text{while}$$

$$\frac{s \vdash \Sigma \cup \{(callee.\bar{i}, e, callee.\bar{o})\} \quad (\Delta\ caller.i) = caller.\bar{i} \quad (\Delta\ caller.o) = caller.\bar{o}}{caller.o :=_f s\ caller.i \vdash \{(caller.i, e[callee.\bar{i} \leftarrow caller.\bar{i}], caller.\bar{o})\}}\ \text{application}$$

$$\frac{s \vdash \Sigma_1 \quad s \vdash \Sigma_2}{s \vdash \Sigma_1 \cup \Sigma_2} \text{ union} \quad \frac{s \vdash \Sigma \cup \{(\bar{i}, e, \bar{o})\}}{s \vdash \Sigma} \text{ elim} \quad \frac{s \vdash \Sigma \cup \{(\bar{i}, f\,\bar{i}, \bar{o})\} \quad g\,\bar{i'} = f\,\bar{i}}{s \vdash (i', g\,i', \bar{o})} \text{ shuffle}$$

Basically, a predicate set records the values of live variables during the execution by relating them with other variables' old values. These rules are applied to build relations between specific inputs and outputs during the execution. The application of them is syntax directed, and proceeds in a bottom-up manner. For example, given the following IL program p produced from HOL- function f,

$$p\,(a,b) = c := 2a + b; \text{ IF } c^2 > 1000 \text{ THEN return } c \text{ ELSE } \{c := c * b; \text{ return } c\}$$
$$f\,(a,b) = \text{let } c = 2a + b \text{ in if } c^2 > 1000 \text{ then } c \text{ else let } c = c * b \text{ in } c$$

we first apply rules refl, assgn and return to get $c := c * b \vdash \{((b,c), c * b, c)\}$ and return $c \vdash \{(c,c,c)\}$. Then by applying the seq rule once we have $(c := c * b; \text{ return } c) \vdash \{((b,c), \text{let } c = c * b \text{ in } c, c)\}$. Similarly return $c \vdash \{((b,c), c, c)\}$ is derived. According to the cond rule we have $(\text{IF } c^2 > 1000 \text{ THEN return } c \text{ ELSE } \{c := c * b; \text{ return } c\}) \vdash \{((b,c), \text{if } c^2 > 1000 \text{ then } c \text{ else let } c = c * b \text{ in } c, c)\}$. For brevity we denote it as $S \vdash \{((b,c), e, c)\}$. Now investigating the remaining statement $c := 2a + b$ will generate $c := 2a + b \vdash \{(b,b,b), ((a,b), 2a+b, c)\}$. Then applying the seq once we have $c := 2a + b; S \vdash \{((a,b), \text{let } b = b \text{ in let } c = 2a + b \text{ in } e, c)\}$. Finally, after the superfluous 'let' binding of b is removed through β-reduction, the synthesized function is equal to f. The derivation is syntax-directed and automatic.

This reasoning mechanism can be improved by adopting Myreen and Gordon's idea that uses separation logic [15] to reason about assembly language. We are considering porting their method into our setting to verify the translation from HOL- to IL.

5   Related Work

There has been much work on translating functional languages; one of the most influential has been the paper of Tolmach and Oliva [22] which developed a translation from SML-like functional language to Ada. Our monomorphisation and closure conversion methods are similar, *i.e.*, removing polymorphism by code specialization and higher-order functions through closure conversion. However, we target logic specification languages and perform correctness proofs on the transformations. Our work can be regarded as an extension of theirs by now verifying the correctness of these two conversions.

Hickey and Nogin [7] worked in MetaPRL to construct a compiler from a full higher order, untyped, functional language to Intel x86 code, based entirely on higher-order rewrite rules. A set of unverified rewriting rules are used to convert a higher level program to a lower level program. They use higher-order abstract syntax to represent programs and do not define the semantics of these programs. Thus no formal verification of the rewriting rules is done.

Hannan and Pfenning [6] constructed a verified compiler in LF for the untyped λ-calculus. The target machine is a variant of the CAM runtime and differs greatly from real machines. In their work, programs are associated with operational

semantics; and both compiler transformation and verifications are modeled as deductive systems. Chlipala [4] further considered compiling a simply-typed λ-calculus to assembly language. He proved semantics preservation based on denotational semantics assigned to the intermediate languages. Type preservation for each compiler pass was also verified. The source language in these works is the bare lambda calculus and is thus much simpler than TFL, thus their compilers only begin to deal with the high level issues we discuss in this paper.

Compared with Chlipala [4] who gives intermediate languages dependent types, Benton and Benton [2] interprets types as binary relations. They proved a semantic type soundness for a compiler from a simple imperative language with heap-allocated data into an idealized assembly language.

Leroy [3,10] verified a compiler from a subset of C, *i.e.* 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 the rewriting steps.

A purely operational semantics based development is that of Klein and Nipkow [8] 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. The Isabelle/HOL theorem prover is also used to verify the compilation from a type-safe subset of C to DLX assembly code [9], where a big step semantics and a small step semantics for this language are defined. In addition, Meyer and Wolff [13] derive in Isabelle/HOL a verified compilation of a lazy language (called MiniHaskell) to a strict language (called MiniML) based on the denotational semantics of these languages. Of course, compiler verification itself is a venerable topic, with far too many publications to survey (see Dave's bibliography [5]).

6 Conclusions and Future Work

We have presented an approach to construct and mechanically verify a translator from TFL to HOL-. The outputs of this translator can be compiled to assembly code and hardware using the verified compilers for HOL- we developed in previous work [11,12,21]. Thus users can write logic specifications in an expressive language TFL and obtain certified low level implementations automatically.

Currently, we are augmenting the compiler to tackle garbage collection, as well as performing a variety of optimizations on intermediate code. We also consider translating by proof a large subset of Java into TFL.

References

1. Augustsson, L.: Compiling pattern matching. In: Conference on Functional Programming Languages and Computer Architecture (1985)
2. Benton, N., Zarfaty, U.: Formalizing and verifying semantic type soundness of a simple compiler. In: 9th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP 2007) (2007)

3. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 460–475. Springer, Heidelberg (2006)

4. Chlipala, A.: A certified type-preserving compiler from lambda calculus to assembly language. In: Conference on Programming Language Design and Implementation (PLDI 2007) (2007)

5. Dave, M.A.: Compiler verification: a bibliography. ACM SIGSOFT Software Engineering Notes 28(6), 2–2 (2003)

6. Hannan, J., Pfenning, F.: Compiler verification in LF. In: Proceedings of the 7th Symposium on Logic in Computer Science (LICS 1992) (1992)

7. Hickey, J., Nogin, A.: Formal compiler construction in a logical framework. Journal of Higher-Order and Symbolic Computation 19(2-3), 197–230 (2006)

8. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Transactions on Programming Languages and Systems (TOPLAS) 28(4), 619–695 (2006)

9. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In: 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005) (2005)

10. Leroy, X.: Formal certification of a compiler backend, or: programming a compiler with a proof assistant. In: Symposium on the Principles of Programming Languages (POPL 2006), ACM Press, New York (2006)

11. Li, G., Owens, S., Slind, K.: Structure of a proof-producing compiler for a subset of higher order logic. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 205–219. Springer, Heidelberg (2007)

12. Li, G., Slind, K.: Compilation as rewriting in higher order logic. In: 21th Conference on Automated Deduction (CADE-21) (July 2007)

13. Wolff, B., Meyer, T.: Tactic-based optimized compilation of functional programs. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 201–214. Springer, Heidelberg (2006)

14. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML, revised edition. MIT Press, Cambridge (1997)

15. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 568–582. Springer, Heidelberg (2007)

16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)

17. Norrish, M., Slind, K.: HOL-4 manuals (1998-2006), http://hol.sourceforge.net/

18. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, Springer, Heidelberg (1998)

19. http://swissnet.ai.mit.edu/~jaffer/r5rs-formal.pdf

20. Slind, K.: Reasoning about terminating functional programs, Ph.D. thesis, Institut für Informatik, Technische Universität München (1999)

21. Slind, K., Owens, S., Iyoda, J., Gordon, M.: Proof producing synthesis of arithmetic and cryptographic hardware. Formal Aspects of Computing 19(3), 343–362 (2007)

22. Tolmach, A., Oliva, D.P.: From ML to Ada: Strongly-typed language interoperability via source translation. Journal of Functional Programming 8(4), 367–412 (1998)