Proof-Producing Synthesis of ML from Higher-Order Logic

Magnus O. Myreen Scott Owens
Computer Laboratory, University of Cambridge, UK
{magnus.myreen,scott.owens}@cl.cam.ac.uk

Abstract

The higher-order logic found in proof assistants such as Coq and various HOL systems provides a convenient setting for the development and verification of pure functional programs. However, to efficiently run these programs, they must be converted (or "extracted") to functional programs in a programming language such as ML or Haskell. With current techniques, this step, which must be trusted, relates similar looking objects that have very different semantic definitions, such as the set-theoretic model of a logic and the operational semantics of a programming language.

In this paper, we show how to increase the trustworthiness of this step with an automated technique. Given a functional program expressed in higher-order logic, our technique provides the corresponding program for a functional language defined with an operational semantics, and it provides a mechanically checked theorem relating the two. This theorem can then be used to transfer verified properties of the logical function to the program.

We have implemented our technique in the HOL4 theorem prover, translating functions to a core subset of Standard ML, and have applied it to examples including functional data structures, a parser generator, cryptographic algorithms, and a garbage collector.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal Methods

General Terms Program Synthesis, Verification

1. Introduction

The logics of most proof assistants for higher-order logic (Coq, Isabelle/HOL, HOL4, PVS, etc.) contain subsets which closely resemble pure functional programming languages. As a result, it has become commonplace to verify functional programs by first coding up algorithms as functions in a theorem prover's logic, then using the prover to prove those logical functions correct, and then simply printing (sometimes called "extracting") these functions into the syntax of a functional programming language, typically SML, OCaml, Lisp, or Haskell. This approach is now used even in very large verification efforts such as the CompCert verified compiler [20] and several projects based on CompCert [1, 29, 38]; it has also been used in database verification [27].

However, the printing step is a potential weak link, as Harrison remarks in a survey on reflection [14]:

> "[...] the final jump from an abstract function inside the logic to a concrete implementation in a serious programming language which *appears to correspond to it* is a glaring leap of faith."

In this paper we show how this *leap of faith* can be made into a trustworthy step. We show how the translation can be automatically performed via proof — a proof which states that (A:) the translation is semantics preserving with respect to the logic and an operational semantics of the target language. Ideally, one could then (B:) run the generated code on a platform which has been proved to implement that operational semantics. This setup provides the highest degree of trust in the executing code without any more effort on the part of programmers and prover users than the current printing/extraction approach.

In previous work, we have shown that A and B are possible for the simple case of an untyped first-order Lisp language [32], i.e. we can synthesise verified Lisp from Lisp-like functions living in higher-order logic; and achieve B by running the generated programs on a verified Lisp implementation [33] which has been proved to implement our operational semantics.

In this paper, we tackle the more complex problem of performing A for higher-order, typed ML-like functions, i.e. we show how semantics preserving translations from higher-order logic into a subset of ML can be performed inside the theorem prover. We believe our method works in general for connecting shallow and deep embeddings of functional programming languages. However, for this paper, we target a specific subset of a Standard ML language, for which we will be able to achieve B in future work with a verified compiler and runtime similar to [6], [9], or [33]. We call our ML subset MiniML and use SML syntax.

1.1 Example

To illustrate what our semantics preserving translation provides, assume that the user defines a summation function over lists using foldl as follows:[1]

$$\mathsf{sum} \;=\; \mathsf{foldl}\ (\lambda(x, y).\ x + y)\ 0$$

This sum function lives in higher-order logic but falls within the subset of the logic that corresponds directly to pure ML. As a result, we can translate sum into ML (like Poly/ML [28], our MiniML supports arbitrary precision integer arithmetic).

```
val sum = foldl (fn (x,y) => x+y) 0
```

For each run, our translation process proves a certificate theorem relating the function in the logic, sum, to the abstract syntax of the ML function, sum, w.r.t. an operational semantics of ML. For sum, this automatically derived certificate theorem states: when the closure that represents sum is applied to an argument of the right type, a list of numbers, then it will return a result, a number, which

[1] Throughout the paper we will typeset higher-order logic equations and definitions in san-serif and MiniML code in typewriter.

ICFP'12 September 10–12, Copenhagen, Denmark.

is exactly the same as the result of applying the HOL function sum to the same input.

The challenge is to do this translation in an easily automated, mechanical manner. In particular, one has to keep track of the relationship between shallowly embedded values, e.g., mathematical functions, and deeply embedded values in the ML semantics, e.g., closures. Our solution involves refinement/coupling invariants and combinators over refinement invariants.

1.2 Contributions

The main contribution of this paper is a new technique by which functions as defined in higher-order logic (HOL) can be translated, with proof, into pure ML equipped with an operational semantics. The ML-like subset of higher-order logic we consider includes:

- total recursive functions,
- type variables,
- functions as first-class values,
- nested pattern matching and user-defined datatypes, and
- partially specified functions, e.g. those with missing pattern match cases.

We also show how our translation technique can be extended with new translations for user-defined operations and types. As an example, we show how to add support for translation of operations over finite sets.

This work improves on the current state of the art of program synthesis from theorem provers (sometimes called program extraction, e.g. extract in Coq, emit-ML in HOL4 and code generation in Isabelle/HOL) by removing that step from the trusted computing base without requiring any additional work from the user. We prove the trustworthiness of the translation with certificate theorems stating that the generated code has exactly the behaviour (including termination) of the original logic function for all inputs where the original function is not partially specified.

We show that our technique is practical with case studies from the HOL4 examples repository, and other examples from the literature, including functional data structures, a parser generator, cryptographic algorithms, and a garbage collector.

Our translator, all of our examples, our semantics for MiniML, and its verified metatheory are all available at `http://www.cl.cam.ac.uk/~mom22/miniml/`.

2. Synthesising Quicksort: an example

Before explaining how our technique works, we first show what it does on a simple, but realistic, example: quicksort. Section 4 presents several larger and more significant examples.

One can define quicksort for lists in higher-order logic as follows.[2] Here ++ appends lists and partition splits a list into two: those elements that satisfy the given predicate, and those that do not.

$$\begin{array}{l}(\mathsf{qsort}\ R\ [\,]\ =\ [\,]) \wedge \\ (\mathsf{qsort}\ R\ (h :: t) = \\ \quad \mathsf{let}\ (l_1, l_2) = \mathsf{partition}\ (\lambda y.\ R\ y\ h)\ t\ \mathsf{in} \\ \quad\quad (\mathsf{qsort}\ R\ l_1)\text{++}[h]\text{++}(\mathsf{qsort}\ R\ l_2))\end{array}$$

Given this definition of the algorithm, one can use HOL to prove the correctness of quicksort:

Theorem 1 (Quicksort correctness)**.** *Given a transitive, total relation R and a list l, qsort returns a sorted permutation of list l.*

[2] In fact, we are re-using Konrad Slind's verified quicksort algorithm from HOL4's library.

Proof. Mechanically verified in HOL4's library: a textbook exercise in program verification. □

Note that this definition and proof could be (and indeed were) developed in HOL4 without any reference to an intended use of the ML synthesis technique presented in this paper.

Given quicksort's definition, our translator can then generate the AST for the following MiniML function (MiniML doesn't have built-in lists; the `Nil`, `Cons`, and `append` constructors and function come from translating the HOL4 list library used by quicksort):

```
fun qsort r = fn l => case l of
 | Nil => Nil
 | Cons(h,t) =>
    let val x' = partition (fn y => r y h) t in
    case x' of
     | Pair(l1,l2) =>
        append (append (qsort r l1) (Cons(h,Nil)))
               (qsort r l2)
    end
```

In the process of generating the above code, the translator also establishes a correspondence between MiniML values and HOL terms and proves the following theorem stating correctness of the translation.

Theorem 2 (Certificate theorem for `qsort`)**.** *When given an application of `qsort` to arguments corresponding to HOL terms, the MiniML operational semantics will terminate with a value that corresponds to the application of HOL function qsort to those terms.*

Proof. Automatically proved as part of the translation, the details of which are the topic of this paper. This proof uses the induction theorem that arises from the definition of qsort in HOL [39]. □

We can use this automatically proved theorem to push the verification result for qsort (Theorem 1) to also apply to the generated MiniML code `qsort`:

Theorem 3 (MiniML quicksort correctness)**.** *If*

1. `qsort` *is bound in the MiniML environment to the implementation listed above,*
2. `leq_R` *is a value that corresponds to a transitive, total HOL relation leq, and*
3. `unsorted_l` *is a value that corresponds to HOL list l,*

then evaluation of MiniML program `qsort leq_R unsorted_l` *terminates with a list value* `sorted_l` *that corresponds to the sorted HOL list* (*qsort leq l*).

Proof. Trivial combination of the two theorems above. □

In summary, we have taken the quicksort algorithm, expressed as a definition in higher-order logic and verified in that setting, and we have generated a pure functional MiniML program and automatically proved that it is correct, according to the operational semantics of MiniML. Note that the meaning of HOL's qsort function is in terms of the proof theory or model theory of higher-order logic, while the MiniML `qsort` function has an operational meaning, which is understood by ML compilers.

3. Overview of approach

In this section, we give a tutorial introduction to our translation approach. Subsequent sections will provide the details (Sect. 5), case studies (Sect. 4) and formal definitions (Sect. 6) that we omit in this section.

3.1 Basic judgements

Our translation from HOL to MiniML derives certificate theorems stated in terms of a predicate called Eval (which is reminiscent of a logical relation).

$$\mathsf{Eval}\ env\ exp\ post$$

Such statements are true if MiniML expression exp evaluates in environment env to some value x and the postcondition $post$ is true for this x, i.e. $post\ x$. Here env is a list of bindings: names are bound to MiniML values (as modelled in our semantics of MiniML), exp is abstract syntax for a MiniML expression and $post$ is a function from MiniML values to $bool$.

Typically, $post$ will be instantiated with a refinement invariant relating a value from HOL to a MiniML value. An example of such an invariant is int. The relation $\mathsf{int}\ n\ v$ is true if integer n is represented in MiniML as value v. With this refinement invariant we can state that the deep embedding of MiniML expression `5` evaluates to 5 in HOL, as follows. We will denote MiniML abstract syntax trees of our MiniML language using SML syntax inside $\lfloor \cdot \rfloor$.

$$\mathsf{Eval}\ env\ \lfloor \texttt{5} \rfloor\ (\mathsf{int}\ 5) \tag{1}$$

We can similarly state that MiniML variable `n` evaluates to the value held in HOL integer variable n by writing:

$$\mathsf{Eval}\ env\ \lfloor \texttt{n} \rfloor\ (\mathsf{int}\ n) \tag{2}$$

From statements such as (1) and (2), we can derive properties of compound expressions, e.g. for addition of numbers:

$$\mathsf{Eval}\ env\ \lfloor \texttt{n+5} \rfloor\ (\mathsf{int}\ (n+5)) \tag{3}$$

3.2 Refinement combinator for functions

The above examples considered simple MiniML expressions that produce concrete values. However, MiniML values can also be closures, such as produced by

```
fn n => n+5
```

To handle closures, we want to combine the refinement invariants for the input and for the output types; in this case both use invariant int. To do this, we have a refinement combinator, $\rightarrow$, which takes two invariants, a and b, as arguments:

$$a \rightarrow b$$

The statement $(a \rightarrow b)\ f\ v$ is true if the value v is a closure such that, when the closure is applied to a value satisfying refinement invariant input a, it returns a value satisfying output b; and furthermore, its input-output relation coincides with f. In other words, when evaluated v corresponds to evaluation of HOL function f. For example, $(\mathsf{int} \rightarrow \mathsf{int})\ (\lambda n.\ n+5)\ v$ specifies that v is a closure in MiniML which has an input-output relation corresponding to the HOL function $\lambda n.\ n+5$.

The $\rightarrow$ refinement combinator can be introduced using a rule for the MiniML closure constructor `fn`. For example, we can derive the following from statement (3) and its assumption on n, i.e. (2).

$$\mathsf{Eval}\ env\ \lfloor \texttt{fn n => n+5} \rfloor\ ((\mathsf{int} \rightarrow \mathsf{int})\ (\lambda n.\ n+5)) \tag{4}$$

Closures that are specified using the $\rightarrow$ combinator can be applied to arguments of the corresponding 'input refinement invariant'. For example, we apply (4) to (1) to arrive at their combination:

$$\mathsf{Eval}\ env\ \lfloor \texttt{(fn n => n+5) 5} \rfloor\ (\mathsf{int}\ ((\lambda n.\ n+5)\ 5))$$

3.3 Type variables and functions as first-class values

The above examples used int as a fixed type/invariant. So how do we translate something that has HOL type α, i.e. a variable type? Answer: for this we use a regular HOL variable for the invariant, e.g. we can use variable a with HOL type: $\alpha \rightarrow ml\_value \rightarrow bool$ as the invariant. (Here and throughout ml_value is the HOL datatype which models MiniML values in HOL as a deep embedding.) The HOL type of int is $int \rightarrow ml\_value \rightarrow bool$, i.e. all that we did was abstract the constant int to a variable a and, similarly in its type, we abstracted the type int to α.

With this variable a ranging over all possible refinement invariants, we can state that MiniML variable `x` evaluates to HOL variable x of type α as follows.

$$\mathsf{Eval}\ env\ \lfloor \texttt{x} \rfloor\ (a\ x)$$

Similarly, we can use the invariant combinator from above to specify that the MiniML value is some closure such that HOL function f of type $\alpha \rightarrow \alpha$ is an accurate representation in the HOL logic.

$$\mathsf{Eval}\ env\ \lfloor \texttt{f} \rfloor\ ((a \rightarrow a)\ f)$$

Since these statements are stated in terms of refinement invariants and $\rightarrow$, we can apply the combinator rules mentioned above. For example, we can derive MiniML code corresponding to a HOL function $\lambda f\ x.\ f\ (f\ x)$ which has an abstract type involving α.

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor \texttt{fn f => fn x => f (f x)} \rfloor \\ \qquad (((a \rightarrow a) \rightarrow a \rightarrow a)\ (\lambda f\ x.\ f\ (f\ x)))\end{array} \tag{5}$$

Evaluation of `fn f => fn x => f (f x)` results in the following closure in our semantics of MiniML:[3]

$$\mathsf{Closure}\ env\ \texttt{"f"}\ \lfloor \texttt{fn x => f (f x)} \rfloor$$

If we assume that MiniML variable name `"twice"` is bound to this value in the evaluation environment env then we can prove, from (5), that the MiniML code `twice` evaluates to a closure with exactly the same behaviour as a HOL function defined by $\mathsf{twice} = \lambda f\ x.\ f\ (f\ x)$.

$$\begin{array}{l}env\ \texttt{"twice"} = \mathsf{Closure}\ twice\_env\ \ldots \implies \\ \mathsf{Eval}\ env\ \lfloor \texttt{twice} \rfloor\ (((a \rightarrow a) \rightarrow a \rightarrow a)\ \mathsf{twice}))\end{array} \tag{6}$$

This is the way we translate non-recursive functions into MiniML.

The example above used variables in place of some refinement invariants. These variables can, of course, be instantiated when combined with Eval-theorems of more specific types. For example, we can plug together (4) and (6) to derive:

$$\begin{array}{l}env\ \texttt{"twice"} = \mathsf{Closure}\ \ldots \implies \\ \mathsf{Eval}\ env\ \lfloor \texttt{twice (fn n => n+5)} \rfloor \\ \qquad ((\mathsf{int} \rightarrow \mathsf{int})\ (\mathsf{twice}\ (\lambda n.\ n+5))))\end{array}$$

3.4 Recursive functions

ML code for non-recursive functions can be derived as shown above. However, recursive functions require some additional effort. To illustrate why, consider the following definition of gcd.

$$\mathsf{gcd}\ m\ n\ =\ \mathsf{if}\ 0 < n\ \mathsf{then}\ \mathsf{gcd}\ n\ (m\ \mathsf{mod}\ n)\ \mathsf{else}\ m$$

If we were to do exactly the same derivation for the right-hand side of the definition of gcd, we would get stuck. The algorithm that the examples above illustrate proceeds in a bottom-up manner: it traverses the structure of the HOL term for which we want to generate MiniML. When translating the right-hand side of a recursive function's definition, what are we to use as the Eval-description of the effect of applying the recursive call? At that stage we would like to have a theorem of the form:

$$\ldots \implies \mathsf{Eval}\ env\ \lfloor \texttt{gcd} \rfloor\ ((\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int})\ \mathsf{gcd})$$

In other words, we would like to assume what we set out to prove.

Our solution is to make a more precise assumption: we formulate the assumption in such a way that it records for what values it

[3] We represent a closure in three parts: an environment, a parameter, and a body expression.

was applied; we then discharge these assumptions using an induction which will be explained later.

We use a new combinator eq to 'record' what values we have assumed that the recursive call is applied to. The definition of eq,

$$\mathsf{eq}\ a\ x = \lambda y\ v.\ (x = y) \wedge a\ y\ v$$

is explained in Section 6.7 together with a more thorough explanation of this example. However, for now, read the following as saying that a call to MiniML gcd has exactly the behaviour of HOL gcd if it is applied to int inputs m and n.

$$\mathsf{Eval}\ env\ \lfloor \texttt{gcd} \rfloor\ ((\mathsf{eq}\ \mathsf{int}\ m \rightarrow \mathsf{eq}\ \mathsf{int}\ n \rightarrow \mathsf{int})\ \mathsf{gcd}) \qquad (7)$$

For the rest of this example we abbreviate (7) as $\mathsf{P}\ m\ n$.

For the recursive call in gcd's right-hand side we can derive the following Eval-theorem. Note how the assumption P mentions exactly what values gcd was called with.

$$\begin{array}{l}\mathsf{P}\ n\ (m \bmod n) \implies \\ \mathsf{Eval}\ env\ \lfloor \texttt{gcd n (m mod n)} \rfloor\ (\mathsf{int}\ (\mathsf{gcd}\ \ n\ (m \bmod n)))\end{array}$$

By making this kind of assumption at every recursive call site, we can proceed with our bottom-up derivation as before. The entire right-hand side of gcd produces the following result:

$$\begin{array}{l}(0 < n \implies \mathsf{P}\ n\ (m \bmod n)) \implies \\ \mathsf{Eval}\ env\ \lfloor \texttt{if 0 < n then gcd ...} \rfloor\ (\mathsf{int}\ (\mathsf{gcd}\ m\ n))\end{array}$$

We now proceed to package the right-hand side of gcd into a closure, very much as we did for twice above, except this time we need a recursive closure (which is described in Section 6.2). We omit the details regarding recursive closures here, but note that the result of this packaging is a theorem:

$$\begin{array}{l}env\ \texttt{"gcd"} = \mathsf{Recclosure}\ \ldots\ \lfloor \texttt{if 0 < n ...} \rfloor \implies \\ \forall m\ n.\ (0 < n \implies \mathsf{P}\ n\ (m \bmod n)) \implies \mathsf{P}\ m\ n\end{array} \qquad (8)$$

We now turn to the phase where we discharge the assumptions that were made at the call sites. For this we will use an induction principle which arises from the totality proof for gcd. All functions in HOL are total, and as a side product of definitions we get an automatically proved induction scheme that is tailored to the structure of the recursion in the definition. The induction scheme that comes out of the definition of gcd is:

$$\begin{array}{l}\forall P.\ (\forall m\ n.\ (0 < n \implies P\ n\ (m \bmod n)) \implies P\ m\ n) \\ \quad \implies (\forall m\ n.\ P\ m\ n)\end{array} \qquad (9)$$

Note that this induction scheme matches the structure of (8) precisely. This means that, by one application of modus ponens of (8) and (9), we arrive at a theorem with a right-hand side: $\forall m\ n.\ \mathsf{P}\ m\ n$. By expanding the abbreviation P (and some simplification to remove eq as explained in Section 6.8), we arrive at the desired certificate theorem for gcd:

$$\begin{array}{l}env\ \texttt{"gcd"} = \mathsf{Recclosure}\ \ldots\ \lfloor \texttt{if 0 < n ...} \rfloor \implies \\ \mathsf{Eval}\ env\ \lfloor \texttt{gcd} \rfloor\ ((\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int})\ \mathsf{gcd})\end{array}$$

To summarise: we use eq together with the custom induction scheme that HOL gives us for each recursive definition to perform translations of recursive HOL functions.

3.5 Datatypes and pattern matching

HOL provides ways of defining ML-like datatypes, e.g. the *list* type can be defined as follows:

$$\mathsf{datatype}\ \alpha\ list\ =\ \mathsf{Nil}\ |\ \mathsf{Cons}\ \mathsf{of}\ \alpha \times (\alpha\ list)$$

These datatypes can be used in ML-like pattern matching. In the following text we will write Cons as :: and Nil as [].

We can support such datatypes in translations by defining a refinement invariant for each datatype that is encountered. For $\alpha\ list$, we define list which takes a refinement invariant a as an argument. We write application of list in post-fix notation, i.e. a list, to make it look like a type. The definition of list can be automatically produced from the datatype definition. Here Conv is a constructor-value from the MiniML semantics (as opposed to, say, a Closure value we saw previously).

$$\begin{array}{rcl}(a\ \mathsf{list})\ []\ v & = & (v = \mathsf{Conv}\ \texttt{"Nil"}\ []) \\ (a\ \mathsf{list})\ (x :: xs)\ v & = & \exists v_1\ v_2.\ (v = \mathsf{Conv}\ \texttt{"Cons"}\ [v_1, v_2]) \\ & & \quad a\ x\ v_1 \wedge (a\ \mathsf{list})\ xs\ v_2\end{array}$$

Based on this definition, we can derive lemmas (see Section 6.6) with which we can translate constructors and pattern matching for this datatype.

However, there is one trick involved: HOL functions that have pattern matching at the top-level tend to be defined as multiple equations. For example, the map function is typically defined in HOL using two equations:

$$\begin{array}{rcl}\mathsf{map}\ f\ []\ & = & []\ \\ \mathsf{map}\ f\ (x :: xs)\ & = & f\ x :: \mathsf{map}\ f\ xs\end{array}$$

In the process of defining this in HOL, the theorem prover reduces the multi-line definition to a single line with a case statement:

$$\mathsf{map}\ f\ xs\ =\ \mathsf{case}\ xs\ \mathsf{of}\ \ldots$$

It is these single-line definitions that we translate into MiniML functions with similar case statements. The translation of case statements will be explained in more detail in Section 6.6.

3.6 Partial functions and under specification

The use of pattern matching leads to partiality.[4] The simplest case of this partiality is the definition of hd for lists, which is defined intentionally with only one case:

$$\mathsf{hd}\ (x :: xs)\ =\ x$$

This definition could equally well have been defined in HOL as:

$$\mathsf{hd}\ xs\ =\ \mathsf{case}\ xs\ \mathsf{of}\ [] \Rightarrow \mathsf{ARB}\ |\ (x :: xs) \Rightarrow x$$

using the special ARB[5] constant in HOL, which cannot be translated into MiniML.

When translating a partial definition into MiniML, we can only prove a connection between MiniML and HOL for certain well-defined input values. For this purpose we use eq from above to restrict the possible input values. The theorem that relates hd to its MiniML counterpart includes a side-condition $xs \neq []$ on the input, which is applied via eq:

$$\begin{array}{l}(env\ \texttt{"hd"} = \ldots) \wedge xs \neq [] \implies \\ \mathsf{Eval}\ env\ \lfloor \texttt{hd} \rfloor\ ((\mathsf{eq}\ (a\ \mathsf{list})\ xs \rightarrow a)\ \mathsf{hd})\end{array}$$

The generated MiniML code includes raise Error in the places where the translation is disconnected from the HOL function.

```
hd xs = case xs of [] => raise Error | ...
```

At the point in the derivation where we require a MiniML value corresponding to ARB, we have a trivially true theorem with false on the left-hand side of an implication.

$$\mathsf{false} \implies \mathsf{Eval}\ env\ \lfloor \texttt{raise Error} \rfloor\ (a\ \mathsf{ARB})$$

This false assumption trickles up to the top level causing the side condition, $xs \neq []$ for hd.

[4] All functions in HOL are total. However, their definitions can omit cases causing their equational specification to appear partial.

[5] ARB is defined non-constructively using Hilbert's arbitrary choice operator.

Translation of recursive partial functions results in recursive side conditions, e.g. the zip function is defined in HOL as:

$$\begin{array}{rcl} \mathsf{zip}\ ([\,],[\,]) & = & [\,] \\ \mathsf{zip}\ (x :: xs, y :: ys) & = & (x, y) :: \mathsf{zip}\ (xs, ys) \end{array}$$

The side condition which is produced for zip is:

$$\begin{array}{rcl} \mathsf{zip\_side}\ ([\,],[\,]) & = & \mathsf{true} \\ \mathsf{zip\_side}\ ([\,], y :: ys) & = & \mathsf{false} \\ \mathsf{zip\_side}\ (x :: xs,[\,]) & = & \mathsf{false} \\ \mathsf{zip\_side}\ (x :: xs, y :: ys) & = & \mathsf{zip\_side}\ (xs, ys) \end{array}$$

These side conditions arise in the derivation as assumptions that are not discharged when the definition-specific induction is applied.

3.7 Equality types

There is another source of partiality: equality tests. MiniML and HOL have different semantics regarding equality. In MiniML, equality of function closures cannot be tested, while equality of functions is allowed in HOL. Whenever an equality is to be translated, we use the following lemma which introduces a condition, EqualityType, on the refinement invariant a for the values that are tested. The definition of EqualityType is given in Section 6.4.

$$\begin{array}{l} \mathsf{Eval}\ env\ \lfloor \mathtt{x} \rfloor\ (a\ x) \wedge \mathsf{Eval}\ env\ \lfloor \mathtt{y} \rfloor\ (a\ y)\ \Longrightarrow \\ \mathsf{EqualityType}\ a\ \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \mathtt{x = y} \rfloor\ (\mathsf{bool}\ (x = y)) \end{array}$$

In contrast to the partiality caused by missing patterns, this form of partiality is neater in that it applies to the refinement invariant, not the actual input values.

For each datatype definition we attempt to prove a lemma which simplifies such equality type constraints, e.g. for the list invariant we can automatically prove:

$$\forall a.\ \mathsf{EqualityType}\ a\ \Longrightarrow\ \mathsf{EqualityType}\ (a\ \mathsf{list})$$

Such lemmas cannot always be proved, e.g. if the datatype contains a function type.

3.8 User-defined extensions

Our approach to supporting user-defined datatypes in Section 3.5 involves machinery which automatically defines new refinement invariants and proves lemmas that can be used in the translation process. The same kind of extensions can also be provided by the user with custom refinement invariants and lemmas for types defined in ways other than datatype (e.g., a quotient construction).

As a simple example, consider the following naive refinement invariant for finite sets represented as lists in MiniML:

$$(a\ \mathsf{set})\ s\ v\ =\ \exists xs.\ (a\ \mathsf{list})\ xs\ v \wedge (s = \mathsf{set\_from\_list}\ xs)$$

Using basic list operations we can prove judgements that can be used for translating basic sets and set operations, e.g. $\{\}$, $\cup$ and $\in$ are implemented by `[]`, `append` and `mem`. The last one also depends on EqualityType a.

$$\mathsf{Eval}\ env\ \lfloor \mathtt{[]} \rfloor\ ((a\ \mathsf{set})\ \{\})$$

$$\begin{array}{l} \mathsf{Eval}\ env\ \lfloor \mathtt{x} \rfloor\ ((a\ \mathsf{set})\ x) \wedge \mathsf{Eval}\ env\ \lfloor \mathtt{y} \rfloor\ ((a\ \mathsf{set})\ y)\ \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \mathtt{append\ x\ y} \rfloor\ ((a\ \mathsf{set})\ (x \cup y)) \end{array}$$

$$\begin{array}{l} \mathsf{Eval}\ env\ \lfloor \mathtt{r} \rfloor\ (a\ r) \wedge \mathsf{Eval}\ env\ \lfloor \mathtt{x} \rfloor\ ((a\ \mathsf{set})\ x)\ \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \mathtt{mem\ r\ x} \rfloor\ (\mathsf{bool}\ (r \in x)) \end{array}$$

The example above is naive and can potentially produce very inefficient code. However, the basic idea can be applied to more efficient data structures, e.g. the datatypes presented in Okasaki's book on functional data structures [36].

We have implemented extensions which can deal with finite sets, finite maps, natural numbers and n-bit machine arithmetic.

4. Case studies

Our translation is implemented (Section 5.1) as an ML program that operates over the HOL4 prover's internal representation of higher-order logic terms, producing HOL4 theorems about MiniML programs (whose semantics we have formally specified in HOL4, see Section 6.2). To demonstrate that it is robust, we have successfully applied it to the following algorithms:

- Miller-Rabin primality test (by Hurd [16])
  This example uses higher-order, recursive, and partial functions, and it requires that all three of these aspects be handled simultaneously.
- An SLR parser generator (by Barthwal [2])
  This is non-trivial algorithm with a long definition: 150 lines in HOL. Its definition makes use of pattern matching.
- AES, RC6 and TEA private key encryption/decryption algorithms (verified by Duan et al. [11])
  These algorithms operate on fixed-size word values, which we support through the technique for user-defined extensions (Section 3.8). We represent fixed-size words as integers in MiniML and use a refinement invariant to make sure the correspondence is maintained.
- McCarthy's 91 function, quicksort (by Slind [39]), and a regular expression matching function (by Owens [37])
  The 91 function and regular expression matcher both have intricate totality proofs, but our technique can easily and automatically prove termination based on the HOL-provided induction principles (which were justified by the original totality proofs).
- A copying Cheney garbage collector (by Myreen [31])
  This is a model of Cheney's algorithm for copying garbage collection — a verified algorithm used in constructing a verified Lisp runtime [33]. It models memory as a mapping from natural numbers to a datatype of abstract memory values.
- Functional data structures from Okasaki's book [36]
  - heap datatypes: leftist, pairing, lazy, splay, binomial
  - set datatypes: unbalanced, red-black
  - sorting algorithms: merge sort
  - list datatypes: binary random-access lists
  - queues datatypes: batched, bankers, physicists, real-time, implicit, Hood-Melville

The algorithms from all but the last point above have been previously verified in HOL4. We have verified 13 of the 15 functional data structures from the last point. These data structures are the examples that Charguéraud [4] uses for his characteristic formula technique (except that we omit the bootstrapped heap and catenable list whose datatypes are not supported by HOL's datatype package). Compared with Charguéraud's verification proofs, ours are similar in length. However, Charguéraud had to use special purpose tactics to deal with his characteristic formulae. In contrast, our verification proofs use only conventional HOL4 tactics. See the related work section for further comparison.

5. Algorithm

We have thus far omitted details and explained our approach through examples. Here, and in the next section, we provide formal definitions and explain that technicalities that earlier text avoided.

We start with an outline of the algorithm for translation. Our method translates one top-level function definition at a time. Each function is translated using the following automatic steps:

Information retrieval. The initial phase collects the necessary information about the function, e.g. is it a constant definition, is it recursive? If it is recursive then the induction theorem associated with its definition is fetched from the context.

Preprocessing. The next step prepares the definition for translation: the definition is collapsed to a single top-level clause, as mentioned in Section 3.5, and certain implicit pattern matching is rewritten into explicit pattern matching, e.g. $\lambda(x, y).\ body$ is expanded into $\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ (x, y) \Rightarrow body$. For the rest of this section, assume that the definition is now of the form:

$$\mathsf{f}\ x_1\ x_2\ \ldots\ x_n = rhs$$

Bottom-up traversal. The next phase takes the right-hand side of the definition to be translated and constructs an Eval-theorem, as demonstrated in Section 3. This theorem is derived through a bottom-up traversal of the HOL expression. At each stage the proof rule or lemma which is applied introduces the corresponding MiniML syntax into the Eval-theorem. The result of this traversal is a theorem where the right-hand side of the HOL function appears together with its derived MiniML counterpart.

$$assumptions \implies \mathsf{Eval}\ env\ derived\_code\ (inv\ rhs)$$

The next phases attempt to discharge the assumptions. Trivial assumptions, such as some EqualityType assumptions, can be discharged as part of the bottom-up traversal.

Packaging. The next phase reduces the rhs to the function constant f. To do this, rules are applied which introduce a λ for each formal parameter, and then perform the following simplification on the right-hand side: the definition is collapsed and eta conversion is performed.

$$\begin{array}{ll} & \lambda x_1\ x_2\ \ldots\ x_n.\ rhs \\ = & \lambda x_1\ x_2\ \ldots\ x_n.\ \mathsf{f}\ x_1\ x_2\ \ldots\ x_n \\ = & \mathsf{f} \end{array}$$

Introduction of λ in the right-hand side of the HOL expression introduces closures on the MiniML side. For recursive functions, the final closure lemma is a special rule for introducing a recursive closure, explained in Section 6.5.

Induction. For recursive functions, the induction theorem associated with the function definition is used to discharge the assumptions that were made at the recursive call sites. The assumptions that the induction theorem fails to discharge are collected and defined to be a side-condition. Such side conditions usually arise from partiality in pattern matching (Section 3.6).

Simplification. As mentioned in Section 3.4, after the induction theorem has been applied the resulting theorem contains redundant occurrences of the eq combinator. These are removed using rewriting as explained in Section 6.8.

Future use. Once the translation is complete, the certificate theorem is stored into the translator's memory. Future translations can then use this certificate theorem in their **Bottom-up traversal** phase, when function constant f is encountered.

5.1 Implementation

Implementing the above algorithm in a HOL theorem prover is straightforward. One writes an ML program which performs the proof steps outlined above. Concretely, this involves writing ML functions that construct elements of type `thm` using the logical kernel's primitives (which correspond to axioms and inference rules of higher-order logic). Following the LCF-approach, this design ensures that all proved theorems are the result of the basic inference rules of higher-order logic.

$$\begin{array}{lcl} t & := & \alpha \mid tc \mid (t_1, \ldots, t_n)tc \mid t_1 \to t_2 \\ p & := & x \mid C\ p_1 \ldots p_n \\ e & := & x \mid \mathsf{ARB} \mid C\ e_1 \ldots e_n \mid \lambda x.e \mid e_1\ e_2 \mid e_1 = e_2 \\ & \mid & e_1 \wedge e_2 \mid e_1 \vee e_2 \\ & \mid & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ & \mid & \mathsf{case}\ e\ \mathsf{of}\ p_1 \Rightarrow e_1\ \mathsf{|}\ \ldots\ \mathsf{|}\ p_n \Rightarrow e_n \\ c & := & C \mid C\ \mathsf{of}\ t_1 \Rightarrow \ldots \Rightarrow t_n \\ d & := & x_1 = c_{11}\ \mathsf{|}\ \ldots\ \mathsf{|}\ c_{1n_1}; \ldots; x_m = c_{m1}\ \mathsf{|}\ \ldots\ \mathsf{|}\ c_{mn_m} \\ & \mid & (x_1\ p_{11} \ldots p_{1n_1} = e_1) \wedge \ldots \wedge (x_m\ p_{m1} \ldots p_{mn_m} = e_m) \end{array}$$

where x ranges over identifiers, C over constructor names, and tc over type constructor names

Figure 1. Core HOL source grammar

$$\begin{array}{lcl} t & := & \ldots \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{num} \mid \mathsf{char} \mid t_1 \times t_2 \mid t\ \mathsf{list} \mid t\ \mathsf{option} \\ p & := & \ldots \\ & \mid & \mathsf{T} \mid \mathsf{F} \mid \mathbb{Z} \mid \mathbb{N} \mid (p_1, p_2) \mid [] \mid p_1 :: p_2 \mid \mathsf{SOME}\ p_1 \mid \mathsf{NONE} \\ e & := & \ldots \\ & \mid & \mathsf{T} \mid \mathsf{F} \mid \mathbb{Z} \mid \mathbb{N} \mid (e_1, e_2) \mid [] \mid e_1 :: e_2 \mid \mathsf{SOME}\ e_1 \mid \mathsf{NONE} \end{array}$$

Figure 2. HOL source grammar after prelude extension

We have implemented our translator in the HOL4 theorem prover. Source code and examples are available at:

`http://www.cl.cam.ac.uk/~mom22/miniml/`

6. Technical details

This section dives into some technical details. We provide definitions and descriptions of the lemmas that are used as part of translations.

6.1 HOL source language

Figure 1 gives the subset of HOL definitions d that we can translate. This grammar describes a subset of the HOL4 logic, it is not deeply embedded in HOL4, nor do we formally reason about it. It includes (possibly mutually) recursive, higher-order functions that operate over (possibly mutually) recursive, user-defined datatypes. The translation will fail if it encounters a term not in this subset (e.g. universal and existential quantifiers, Hilbert's choice) in the definitions being translated. The translator comes with a standard prelude that includes support for booleans, integers, natural numbers, characters, pairs, lists, and options (Figure 2).

6.2 MiniML target language

Figure 3 gives the source grammar for MiniML types t, values v, patterns p, expressions e, type definitions td/c and top-level definitions d. The language is a mostly unsugared subset of core Standard ML. It includes mutually recursive datatype definitions; higher-order, anonymous, and mutually recursive functions; nested pattern matching; and abrupt termination (a simplified `raise`). MiniML integers are arbitrary precision (which is how the Poly/ML compiler implements integers natively, other ML implementations usually support them as a library). Unsupported features are records, mutable references, exception handling, and the module system.

We give MiniML both small-step and big-step call-by-value operational semantics, and a type system. Each of these three has an expression-level and definition-level component; here we only present the expression level, but see `http://www.cl.cam.ac.uk/~mom22/miniml/` for complete definitions as well as HOL4 proofs of the theorems below, at both levels. The type system is typical. Figure 4 gives the auxiliary definitions needed to support the semantics (in this figure we abbreviate ml_value to v), and Figure 5 gives the shapes of the various semantic relations.

$$
\begin{array}{lcl}
t & := & \alpha \mid x \mid \texttt{(}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{)}x \mid t_1\ \texttt{->}\ t_2 \mid \texttt{int} \mid \texttt{bool} \\
v & := & C \mid \texttt{true} \mid \texttt{false} \mid \mathbb{Z} \\
p & := & x \mid v \mid C\texttt{(}p_1\texttt{,}\ldots\texttt{,}p_n\texttt{)} \\
e & := & \texttt{raise}\ ex \\
 & \mid & x \mid v \mid C\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \\
 & \mid & \texttt{fn}\ x\ \texttt{=>}\ e \\
 & \mid & e_1\ e_2 \mid e_1\ op\ e_2 \mid e_1\ \texttt{andalso}\ e_2 \mid e_1\ \texttt{orelse}\ e_2 \\
 & \mid & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ p_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{=>}\ e_n \\
 & \mid & \texttt{let val}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\
 & \mid & \texttt{let fun}\ x_1\ y_1\ \texttt{=}\ e_1\ \texttt{and}\ldots\texttt{and}\ x_n\ y_n\ \texttt{=}\ e_n\ \texttt{in}\ e\ \texttt{end} \\
c & := & C \mid C\ \texttt{of}\ t_1\ \texttt{*}\ldots\texttt{*}\ t_n \\
td & := & \texttt{(}\alpha_1\texttt{,}\ldots\texttt{,}\alpha_m\texttt{)}\ x\ \texttt{=}\ c_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n \\
 & \mid & x\ \texttt{=}\ c_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n \\
d & := & \texttt{val}\ p\ \texttt{=}\ e \\
 & \mid & \texttt{fun}\ x_1\ y_1\ \texttt{=}\ e_1\ \texttt{and}\ldots\texttt{and}\ x_n\ y_n\ \texttt{=}\ e_n \\
 & \mid & \texttt{datatype}\ td_1\ \texttt{and}\ldots\texttt{and}\ td_n \\
ex & := & \texttt{Bind} \mid \texttt{Div} \\
op & := & \texttt{=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{div} \mid \texttt{mod} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=}
\end{array}
$$

where x and y range over identifiers and C over constructor names

Figure 3. MiniML source grammar

$$
\begin{array}{lcl}
v & := & C\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)} \\
 & \mid & \langle env, x, e\rangle \\
 & \mid & \langle env, (\texttt{fun}\ x_1\ y_1\ \texttt{=}\ e_1\ \texttt{and}\ldots\texttt{and}\ x_n\ y_n\ \texttt{=}\ e_n), x\rangle \\
 & \mid & C \mid \texttt{true} \mid \texttt{false} \mid \mathbb{Z} \\
F & := & \texttt{[]}\ e \mid v\ \texttt{[]} \mid \texttt{[]}\ op\ e \mid v\ op\ \texttt{[]} \\
 & \mid & \texttt{[]}\ \texttt{andalso}\ e \mid \texttt{[]}\ \texttt{orelse}\ e \\
 & \mid & \texttt{if []}\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
 & \mid & \texttt{case []}\ \texttt{of}\ p_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{=>}\ e_n \\
 & \mid & \texttt{let val}\ x\ \texttt{=}\ \texttt{[]}\ \texttt{in}\ e\ \texttt{end} \\
 & \mid & C\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{,[],}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \\
S & := & \langle Cenv, env, e, \langle F_1, env_1\rangle \ldots \langle F_n, env_n\rangle\rangle \\
R_{match} & := & env \mid \texttt{no\_match} \mid \texttt{type\_error} \\
R_{step} & := & S \mid \texttt{type\_error} \mid \texttt{stuck} \\
R_{eval} & := & v \mid \texttt{raise}\ ex \mid \texttt{type\_error}
\end{array}
$$

where env ranges over finite maps from x to v,
$Cenv$ ranges over finite maps from C to $\langle \mathbb{N}, 2^C\rangle$.
$envT$ ranges over finite maps from x to $(\alpha_1, \ldots, \alpha_n)\ t$, *and*
$CenvT$ ranges over finite maps from C to
$\langle(\alpha_1, \ldots, \alpha_m), t_1 \ldots t_n, x\rangle$

Figure 4. Semantic auxiliaries for MiniML

The small-step semantics is a CEK-like machine [12] (see [13] for a textbook treatment) with states S (Figure 4) using a continuation stack built from frames F and environments env. Values are extended with constructed values (e.g., `Some(1)`), with closures pairing a function's environment, parameter, and body, and with recursive closures pairing an environment with a mutually recursive nest of functions. A single reduction step either gives a new state, signals a "type error", e.g., due to a misapplied primitive, or gets stuck (R_{step}). We use the small-step semantics to support a type soundness proof via preservation and progress [41], and to ensure a satisfactory treatment of divergence. Small-step evaluation and divergence are defined in terms of the transitive closure of the reduction relation.

Our technique for translating from HOL to MiniML uses a bottom-up, syntax-directed pass, and so requires a syntax-directed big-step semantics. The big-step semantics returns the same kind of things as small-step evaluation: values, exceptions and "type errors" (R_{eval}). We ensure that it gives type errors in enough cases so

Pattern matching:	$\langle Cenv, p, v, env\rangle \Downarrow R_{match}$
Small-step reduction:	$S \longrightarrow R_{step}$
Small-step evaluation:	$S \downarrow R_{eval}$
Small-step divergence:	$\langle Cenv, env, e\rangle \uparrow$
Big-step evaluation:	$\langle Cenv, env, e\rangle \Downarrow R_{eval}$
Alternate big-step evaluation:	$\langle env, e\rangle \Downarrow R_{eval}$
Typing:	$\langle CenvT, envT\rangle \vdash e : t$
Typing for environments:	$Cenv \vdash env : envT$

Figure 5. MiniML semantic relations

that only diverging expressions are not related to any result. This allows us to use (in our non-concurrent, deterministic setting) an inductive relation, instead of following a co-inductive approach [21]. Theorems 4 and 6 guarantee this property.

6.3 MiniML metatheory

Theorem 4 (Small-step/big-step equivalence)**.** $\langle Cenv, env, e, \epsilon\rangle \downarrow R_{eval}$ *iff* $\langle Cenv, env, e\rangle \Downarrow R_{eval}$.

Proof. In HOL4.

- Forward implication:
  We first extend the big-step relation with context stack inputs, $Fs := \langle F_1, env_1\rangle \ldots \langle F_n, env_n\rangle$. We then show that if $\langle Cenv_1, env_1, e_1, Fs_1\rangle \longrightarrow \langle Cenv_2, env_2, e_2, Fs_2\rangle$ and $\langle Cenv_2, env_2, e_2, Fs_2\rangle \Downarrow R_{eval}$ then $\langle Cenv_1, env_1, e_1, Fs_1\rangle \Downarrow R_{eval}$ by cases on the small-step relation. We then finish the proof by induction on the transitive closure of $\longrightarrow$. Note that unlike type soundness, we go backwards along the small-step trace; this is necessary to properly handle non-termination.
- Reverse implication:
  By induction on the big-step relation, with pervasive reasoning about adding context frames to the frame stacks of many-step small-step reduction sequences.

□

Theorem 5 (Big-step determinism)**.** *If* $\langle Cenv, env, e\rangle \Downarrow R_{eval1}$ *and* $\langle Cenv, env, e\rangle \Downarrow R_{eval2}$ *then* $R_{eval1} = R_{eval2}$.

Proof. In HOL4, by induction on the big-step evaluation relation. □

Corollary 1 (Small-step determinism)**.** *If* $\langle Cenv, env, e, \epsilon\rangle \downarrow R_{eval1}$ *and* $\langle Cenv, env, e, \epsilon\rangle \downarrow R_{eval2}$ *then* $R_{eval1} = R_{eval2}$.

Theorem 6 (Untyped safety)**.** $\langle Cenv, env, e, \epsilon\rangle \downarrow R_{eval}$ *iff it is not the case that* $\langle Cenv, env, e\rangle \uparrow$.

Proof. In HOL4, by cases on the small-step relation. □

Theorem 7 (Type soundness)**.** *If*

- $Cenv$ *and* $CenvT$ *are well-formed and consistent,*
- $CenvT \vdash env : envT$*, and*
- $\langle CenvT, envT\rangle \vdash e : t$

then either

- $\langle Cenv, env, e\rangle \uparrow$*, or*
- $\langle Cenv, env, e, \epsilon\rangle \downarrow R_{eval}$ *and* $R_{eval} \neq$ `type_error`.

Proof. In HOL4, a typical preservation and progress proof about the small-step semantics. □

The small- and big-step semantics are given a $Cenv$ which allows them to return `type_error` when an undefined data constructor (i.e., one not defined in a `datatype` definition) is applied, or when a data constructor is applied to the wrong number of arguments. However, we can simplify the translation from HOL by using an alternate big-step semantics that omits this argument. This alternate big-step semantics differs only in that mis-applied constructors are accepted and do not result in an error. However, they coincide on well-typed programs.

Theorem 8 (Alternate big step equivalence)**.** *If $Cenv$ and $CenvT$ are well-formed and consistent, and $CenvT \vdash env : tenv$ and $\langle CenvT, envT\rangle \vdash e : t$ then $\langle Cenv, env, e\rangle \Downarrow R_{eval}$ iff $\langle env, e\rangle \Downarrow R_{eval}$.*

Proof. In HOL4, by induction on the big-step relation, and Theorems 6 and 7 and Corollary 1. □

6.4 Key definitions

As described in earlier sections, our translation makes statements about the semantics in terms of a predicate called Eval. We define this predicate as follows using the alternate big-step semantics evaluation relation $\Downarrow$. We define Eval *env exp post* to be true if *exp* evaluates, in environment *env*, to some value v such that *post* v. The fact that it returns a value — as opposed to an error, `raise` *ex* — tells us that no error happened during evaluation, e.g. evaluation did not hit any missing cases while pattern matching.

$$\mathsf{Eval}\ env\ exp\ post\ =\ \exists v.\ \langle env, exp\rangle \Downarrow v \wedge post\ v$$

Here *post* has type $ml\_value \to bool$.

The interesting part is what we instantiate *post* with, i.e. the refinement invariants. The basic refinement invariants have the following definitions. Boolean and integer values relate to corresponding literal values in the MiniML semantics:

$$\begin{array}{rcll} \mathsf{bool\ true} & = & \lambda v.\ (v = \texttt{true}) & \\ \mathsf{bool\ false} & = & \lambda v.\ (v = \texttt{false}) & \\ \mathsf{int}\ i & = & \lambda v.\ (v = i) & \text{where } i \in \mathbb{Z} \end{array}$$

We also have combinators for refinement invariants. The definition of the eq combinator was given in Section 3.4. We now turn to the $\to$ combinator which lifts refinement invariant to closures. The $\to$ combinator's definition is based on an evaluation relation for application of closures, evaluate_closure (which is defined in terms of $\Downarrow$, and applies to non-recursive and recursive closures). Read evaluate_closure $v\ cl\ u$ as saying: application of closure cl to argument v returns value u. We define a total-correctness Hoare-triple-like Spec for closure evaluation on top of this:

$$\begin{array}{l} \mathsf{Spec}\ p\ cl\ q\ = \\ \quad \forall v.\ p\ v\ \Longrightarrow\ \exists u.\ \mathsf{evaluate\_closure}\ v\ cl\ u \wedge q\ u \end{array}$$

The definition of the $\to$ combinator is an instance of Spec, where an abstract value x is universally quantified:

$$(a \to b)\ f\ =\ \lambda v.\ \forall x.\ \mathsf{Spec}\ (a\ x)\ v\ (b\ (f\ x))$$

Here the type of f is $\alpha \to \beta$ and the type of v is simply the type of a MiniML value in our MiniML semantics, i.e. ml_value.

The remaining definition is that of EqualityType a. A refinement invariant a supports equality if the corresponding MiniML value cannot be a closure, not_contains_closure, and testing for structural equality of MiniML values is equivalent to testing equality at the abstract level:

$$\begin{array}{l} \mathsf{EqualityType}\ a\ = \\ \quad (\forall x\ v.\ a\ x\ v\ \Longrightarrow\ \mathsf{not\_contains\_closure}\ v) \wedge \\ \quad (\forall x\ v\ y\ w.\ a\ x\ v \wedge a\ y\ w\ \Longrightarrow\ (v = w\ \Longleftrightarrow\ x = y)) \end{array}$$

For example, bool and int, defined above, satisfy EqualityType.

6.5 Lemmas used in translations

In this section we present the lemmas about Eval that are used to perform the translations. All variables in these theorems are implicitly universally quantified at the top-level. The proof of these lemmas follow almost directly from the underlying definitions: none of the proofs required more than ten lines of script in HOL4.

Closure application. We start with the rule for applying a closure. A closure $a \to b$ can always be applied to an Eval-theorem with a matching refinement invariant a.

$$\begin{array}{l} \mathsf{Eval}\ env\ \lfloor\texttt{f}\rfloor\ ((a \to b)\ f) \wedge \\ \mathsf{Eval}\ env\ \lfloor\texttt{x}\rfloor\ (a\ x)\ \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor\texttt{f x}\rfloor\ (b\ (f\ x)) \end{array}$$

Closure introduction. Closures can be created with the following rule if the abstract and concrete values, x and v, which the body depends on can be universally quantified. Here $n \mapsto v$ extends the environment env with binding: name n maps to value v.

$$\begin{array}{l} (\forall x\ v.\ a\ x\ v \Longrightarrow \mathsf{Eval}\ (env[n \mapsto v])\ \lfloor\texttt{body}\rfloor\ (b\ (f\ x))) \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor\texttt{fn}\ n\ \texttt{=>}\ \texttt{body}\rfloor\ ((a \to b)\ f) \end{array}$$

Alternative closure introduction. The rule above is not always applicable because side conditions restrict the variable x, i.e. the universal quantification cannot be introduced. This is an alternative rule which achieves the same without universal quantification of x — at the cost of introducing the eq combinator.

$$\begin{array}{l} (\forall v.\ a\ x\ v \Longrightarrow \mathsf{Eval}\ (env[n \mapsto v])\ \lfloor\texttt{body}\rfloor\ (b\ (f\ x))) \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor\texttt{fn}\ n\ \texttt{=>}\ \texttt{body}\rfloor\ ((\mathsf{eq}\ a\ x \to b)\ f) \end{array}$$

Closure evaluation. The translator always returns theorems where the code is described by an assumption stating that the function name refers to the relevant code in the environment, i.e. an assumption of the form $env\ name = \mathsf{closure}\ \ldots$. The following rule is used for deriving theorems with such assumptions for non-recursive closures:

$$\begin{array}{l} \mathsf{Eval}\ cl\_env\ \lfloor\texttt{fn}\ n\ \texttt{=>}\ \texttt{body}\rfloor\ p\ \Longrightarrow \\ env\ name = \mathsf{Closure}\ cl\_env\ n\ \lfloor\texttt{body}\rfloor\ \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor name\rfloor\ p \end{array}$$

Introduction of recursive closure. Our rule for introducing recursive closures, i.e. closures where the environment can refer to itself and hence perform recursive function calls to itself, is more verbose. Introduction of recursive closures is done using the following lemma. For this lemma to be applicable some name $name$ must refer to a recursive closure where $name$ is given. Let Recclosure $cl\_env\ [(name, n, \lfloor\texttt{body}\rfloor)]\ name$ be abbreviated by Rec below.

$$\begin{array}{l} (\forall v.\ a\ x\ v \Longrightarrow \\ \quad \mathsf{Eval}\ (env[n \mapsto v, name \mapsto \mathsf{Rec}])\ \lfloor\texttt{body}\rfloor\ (b\ (f\ x))) \\ \Longrightarrow \\ env\ name = \mathsf{Rec}\ \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor name\rfloor\ ((\mathsf{eq}\ a\ x \to b)\ f) \end{array}$$

Let introduction. Let-statements are constructed using the following lemma. Here let is HOL's internal combinator which represents let expressions. In HOL, let $f\ x\ =\ f\ x$ and the HOL printer knows to treat let as special, e.g. let $(\lambda a.\ a + 1)\ x$ is printed on the screen as let $a = x$ in $a + 1$.

$$\begin{array}{l} \mathsf{Eval}\ env\ \lfloor\texttt{x}\rfloor\ (a\ x) \wedge \\ (\forall v.\ a\ x\ v \Longrightarrow \mathsf{Eval}\ (env[n \mapsto v])\ \lfloor\texttt{body}\rfloor\ (b\ (f\ x))) \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor\texttt{let val}\ n\ \texttt{=}\ \texttt{x in body end}\rfloor\ (b\ (\mathsf{let}\ f\ x)) \end{array}$$

Variable simplification. During translation, the intermediate theorems typically contain assumptions specifying which HOL val-

ues relate to which MiniML values. It's convenient to state these as $\mathsf{Eval}\ env\ \lfloor m \rfloor\ (inv\ n)$, for some inv and some fixed variable name m. When variables get bound, e.g. as a result of introducing a closure, env is specialised and these assumptions can be simplified. We use the following lemma to simplify the assumptions when env gets specialised.

$$\begin{array}{l}\mathsf{Eval}\ (env[name \mapsto v])\ \lfloor m \rfloor\ p\ = \\ \quad \mathsf{if}\ m = name\ \mathsf{then}\ p\ v\ \mathsf{else}\ \mathsf{Eval}\ env\ \lfloor m \rfloor\ p\end{array}$$

If statements. The translation of HOL's if statements is done using the following rule. Note that the assumptions h_2 and h_3 get prefixed by the guard expression x_1.

$$\begin{array}{l}(h_1 \Longrightarrow \mathsf{Eval}\ env\ \lfloor \texttt{x1} \rfloor\ (\mathsf{bool}\ x_1)) \wedge \\ (h_2 \Longrightarrow \mathsf{Eval}\ env\ \lfloor \texttt{x2} \rfloor\ (inv\ x_2)) \wedge \\ (h_3 \Longrightarrow \mathsf{Eval}\ env\ \lfloor \texttt{x3} \rfloor\ (inv\ x_3)) \Longrightarrow \\ (h_1 \wedge (x_1 \Longrightarrow h_2) \wedge (\neg x_1 \Longrightarrow h_3)) \Longrightarrow \\ \quad \mathsf{Eval}\ env\ \lfloor \texttt{if x1 then x2 else x3} \rfloor \\ \qquad (inv\ (\mathsf{if}\ x_1\ \mathsf{then}\ x_2\ \mathsf{else}\ x_3))\end{array}$$

Literal values. MiniML has boolean and integer literals. The relevant lemmas for such literals:

$$\mathsf{Eval}\ env\ \lfloor \texttt{true} \rfloor\ (\mathsf{bool}\ \mathsf{true})$$

$$\mathsf{Eval}\ env\ \lfloor \texttt{false} \rfloor\ (\mathsf{bool}\ \mathsf{false})$$

$$\mathsf{Eval}\ env\ \lfloor i \rfloor\ (\mathsf{int}\ i) \qquad \text{where } i \in \mathbb{Z}$$

Binary operations. Each of the operations over the integers and booleans have separate lemmas. A few examples are listed below. Division and modulo have a side condition.

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor \texttt{i} \rfloor\ (\mathsf{int}\ i) \wedge \\ \mathsf{Eval}\ env\ \lfloor \texttt{j} \rfloor\ (\mathsf{int}\ j) \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \texttt{i + j} \rfloor\ (\mathsf{int}\ (i + j))\end{array}$$

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor \texttt{i} \rfloor\ (\mathsf{int}\ i) \wedge \\ \mathsf{Eval}\ env\ \lfloor \texttt{j} \rfloor\ (\mathsf{int}\ j) \Longrightarrow \\ j \neq 0 \Longrightarrow \mathsf{Eval}\ env\ \lfloor \texttt{i div j} \rfloor\ (\mathsf{int}\ (i\ \mathsf{div}\ j))\end{array}$$

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor \texttt{i} \rfloor\ (\mathsf{int}\ i) \wedge \\ \mathsf{Eval}\ env\ \lfloor \texttt{j} \rfloor\ (\mathsf{int}\ j) \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \texttt{i < j} \rfloor\ (\mathsf{bool}\ (i < j))\end{array}$$

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor \texttt{a} \rfloor\ (\mathsf{bool}\ a) \wedge \\ \mathsf{Eval}\ env\ \lfloor \texttt{b} \rfloor\ (\mathsf{bool}\ b) \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \texttt{a andalso b} \rfloor\ (\mathsf{bool}\ (a \wedge b))\end{array}$$

There are also dynamically derived lemmas, e.g. each translation results in a new lemma that can be used in subsequent translations and datatype definitions result in a few lemmas (as described in the next section). Users can also manually provide additional lemmas.

6.6 Lemmas automatically proved for datatypes

For each datatype, we define a refinement invariant that relates it to ML values. Type variables cause these definitions to take refinement invariants as input. For example, for the $list$ datatype from Section 3.5 we define a refinement invariant, called list, as the following map into constructor, Conv, applications in MiniML. We write application of list in post-fix notation, i.e. a list, to make it look like a type.

$$\begin{array}{rcl}(a\ \mathsf{list})\ []\ v & = & (v = \mathsf{Conv}\ \texttt{"Nil"}\ []) \\ (a\ \mathsf{list})\ (x :: xs)\ v & = & \exists v_1\ v_2.\ (v = \mathsf{Conv}\ \texttt{"Cons"}\ [v_1, v_2]) \\ & & \quad a\ x\ v_1 \wedge (a\ \mathsf{list})\ xs\ v_2\end{array}$$

Based on this definition we can derive lemmas that aid translation of constructor applications in HOL.

$$\mathsf{Eval}\ env\ \lfloor \texttt{Nil} \rfloor\ ((a\ \mathsf{list})\ [])$$

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor \texttt{x} \rfloor\ (a\ x) \wedge \\ \mathsf{Eval}\ env\ \lfloor \texttt{xs} \rfloor\ ((a\ \mathsf{list})\ xs) \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \texttt{Cons(x,xs)} \rfloor\ ((a\ \mathsf{list})\ (x :: xs))\end{array}$$

We also derive lemmas which aid in translating pattern matching over these HOL constructors. As mentioned in Section 3.5, multi-line pattern matches, i.e. HOL definitions that are defined as multiple equations, are merged into a single line definition with a case statement by the definition mechanism. By making sure translations are always performed only on these collapsed single line definitions, it is sufficient to add support for translations of case statements for the new datatype:

$$\mathsf{case}\ l\ \mathsf{of}\ [] \Rightarrow \ldots \mid (x :: xs) \Rightarrow \ldots$$

In HOL, case statements (including complicated-looking nested case statements) are internally represented as primitive 'case functions'. The case function for the $list$ datatype is defined using the following two equations:

$$\begin{array}{rcl}\mathsf{list\_case}\ []\ f_1\ f_2 & = & f_1 \\ \mathsf{list\_case}\ (x :: xs)\ f_1\ f_2 & = & f_2\ x\ xs\end{array}$$

Thus, in order to translate case statements for the $list$ datatype, it is sufficient to be able to translate any instantiation of $\mathsf{list\_case}\ l\ f_1\ f_2$. The lemma which we use for this is shown below. This lemma can be read as a generalisation of the lemma for translating closure introduction and if statements.

$$\begin{array}{l}(h_0 \Longrightarrow \mathsf{Eval}\ env\ \lfloor \texttt{l} \rfloor\ ((a\ \mathsf{list})\ l)) \wedge \\ (h_1 \Longrightarrow \mathsf{Eval}\ env\ \lfloor \texttt{y} \rfloor\ (b\ f_1)) \wedge \\ (\forall x\ xs\ v\ vs. \\ \quad a\ x\ v \wedge (a\ \mathsf{list})\ xs\ vs \wedge h_2\ x\ xs \Longrightarrow \\ \quad \mathsf{Eval}\ (env[n \mapsto v][m \mapsto vs])\ \lfloor \texttt{z} \rfloor\ (b\ (f_2\ x\ xs))) \Longrightarrow \\ (\forall x\ xs. \\ \quad h_0 \wedge ((l = []) \Longrightarrow h_1) \wedge \\ \quad ((l = x :: xs) \Longrightarrow h_2\ x\ xs)) \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \texttt{case l of Nil => y | Cons(}n\texttt{,}m\texttt{) => z} \rfloor \\ \qquad (b\ (\mathsf{list\_case}\ l\ f_1\ f_2))\end{array}$$

6.7 Translation of recursive functions

The most technical part of our approach is the details of how recursive functions are translated. In what follows, we expand on the gcd example given in Section 3.4 and explain our use of induction and eq in more detail.

$$\mathsf{gcd}\ m\ n\ =\ \mathsf{if}\ 0 < n\ \mathsf{then}\ \mathsf{gcd}\ n\ (m\ \mathsf{mod}\ n)\ \mathsf{else}\ m$$

As was already mentioned, when such a function is to be translated, we perform the bottom-up traversal (Section 5) for the right-hand side of the definition. When doing so we encounter the recursive call to gcd for which we need an Eval theorem. In this theorem we need to make explicit with what values we make the recursive call. For this purpose we use the eq combinator

$$\mathsf{eq}\ a\ x = \lambda y\ v.\ (x = y) \wedge a\ y\ v$$

which when used together with $\rightarrow$ restricts the universal quantifier that is hidden inside the $\rightarrow$ function combinator. One can informally read, refinement invariant $\mathsf{int} \rightarrow \ldots$ as saying "for any int input, ...". Similarly, $\mathsf{eq\ int}\ i \rightarrow \ldots$ can be read as "for any int input equal to i, ...", which is the same as "for int input i, ...".

We state the assumption we make at call sites as follows:

$$\mathsf{Eval}\ env\ \lfloor \texttt{gcd} \rfloor\ ((\mathsf{eq\ int}\ m \rightarrow \mathsf{eq\ int}\ n \rightarrow \mathsf{int})\ \mathsf{gcd}) \qquad (10)$$

For the rest of this example we abbreviate (10) as $\mathsf{P}\ m\ n$. In order to derive an Eval theorem for the expression $\mathsf{gcd}\ n\ (m \bmod n)$, we first derive an Eval theorem argument n

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor\mathtt{n}\rfloor\ (\mathsf{int}\ n)\implies\\ \mathsf{Eval}\ env\ \lfloor\mathtt{n}\rfloor\ (\mathsf{int}\ n)\end{array}$$

and an Eval theorem argument $m \bmod n$

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor\mathtt{m}\rfloor\ (\mathsf{int}\ m)\wedge\\ \mathsf{Eval}\ env\ \lfloor\mathtt{n}\rfloor\ (\mathsf{int}\ n)\wedge n\neq 0\implies\\ \mathsf{Eval}\ env\ \lfloor\mathtt{m\ mod\ n}\rfloor\ (\mathsf{int}\ (m \bmod n))\end{array}$$

Next, we use the following rule to introduce eq combinators to the above theorems

$$\forall a\ x\ m.\ \mathsf{Eval}\ env\ m\ (a\ x)\implies\mathsf{Eval}\ env\ m\ ((\mathsf{eq}\ a\ x)\ x)$$

and then we apply to Closure application rule from Section 6.5 to get an Eval theorem for $\mathsf{gcd}\ n\ (m \bmod n)$.

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor\mathtt{m}\rfloor\ (\mathsf{int}\ m)\wedge\mathsf{P}\ n\ (m \bmod n)\wedge\\ \mathsf{Eval}\ env\ \lfloor\mathtt{n}\rfloor\ (\mathsf{int}\ n)\wedge n\neq 0\implies\\ \mathsf{Eval}\ env\ \lfloor\mathtt{gcd\ n\ (m\ mod\ n)}\rfloor\ (\mathsf{int}\ (\mathsf{gcd}\ \ n\ (m \bmod n)))\end{array}$$

By then continuing the bottom-up traversal as usual and packaging up the right-hand side following the description in Section 5, we arrive at the following theorem where our abbreviation P appears both as an assumption and as the conclusion.

$$\begin{array}{l}env\ \texttt{"gcd"} = \mathsf{Recclosure}\ \ldots\ \lfloor\mathtt{if\ 0\ <\ n}\ \ldots\rfloor\implies\\ \forall m\ n.\ (0<n\implies\mathsf{P}\ n\ (m \bmod n))\implies\mathsf{P}\ m\ n\end{array}\tag{11}$$

Note that the shape of the right-hand side of the implication matches the left-hand side of the following induction which HOL provides as a side product of proving totality of the gcd function.

$$\begin{array}{l}\forall P.\ (\forall m\ n.\ (0<n\implies P\ n\ (m \bmod n))\implies P\ m\ n)\\ \quad\implies(\forall m\ n.\ P\ m\ n)\end{array}\tag{12}$$

By one application of modus ponens of (11) and (12), we arrive at a theorem with a right-hand side: $\forall m\ n.\ \mathsf{P}\ m\ n$. By expanding the abbreviation P and some simplification to remove eq (explained in the next section), we arrive at the desired certificate theorem for the gcd function:

$$\begin{array}{l}env\ \texttt{"gcd"} = \mathsf{Recclosure}\ \ldots\ \lfloor\mathtt{if\ 0\ <\ n}\ \ldots\rfloor\implies\\ \mathsf{Eval}\ env\ \lfloor\mathtt{gcd}\rfloor\ ((\mathsf{int}\to\mathsf{int}\to\mathsf{int})\ \mathsf{gcd})\end{array}$$

The gcd function is a very simple function. However, the technique above is exactly the same even for functions with nested recursion (e.g. as in McCarthy's 91 function) and mutual recursion (in such cases the induction has two conclusions). We always use the eq combinator to record input values, then apply the induction arising from the function's totality proof to discharge these assumptions and finally rewrite away the remaining eq combinators as described in the next section.

6.8 Simplification of eq

Our gcd example in Section 3.4 glossed over how eq combinators are removed. In this section, we expand on that detail.

When translating recursive functions, we use the eq combinator to 'record' what values we instantiate the inductive hypothesis with. Once the induction has been applied, we are left with an Eval-theorem which is cluttered with these eq combinators. The theorems have this shape:

$$\begin{array}{l}\forall x_1\ x_2\ \ldots\ x_n.\\ \quad\mathsf{Eval}\ env\ code\\ \quad\quad((\mathsf{eq}\ a_1\ x_1\to\mathsf{eq}\ a_2\ x_2\to\ldots\to\mathsf{eq}\ a_n\ x_n\to b)\ func)\end{array}$$

Next, we show how these eq combinators can be removed by rewriting. First, we need two new combinators. The examples below will illustrate their use.

$$\begin{array}{lcl}\mathsf{A}\ a\ y\ v&=&\forall x.\ a\ x\ y\ v\\ \mathsf{E}\ a\ y\ v&=&\exists x.\ a\ x\ y\ v\end{array}$$

We use these combinators to push the external $\forall$ inwards. The following rewrite theorem shows how we can turn an external $\forall$ into an application of the A combinator. Here $(\mathsf{A}x.\ p\ x)$ is an abbreviation for $\mathsf{A}\ (\lambda x.\ p\ x)$.

$$\begin{array}{l}(\forall x.\ \mathsf{Eval}\ env\ code\ ((p\ x)\ f))\ =\\ \mathsf{Eval}\ env\ code\ ((\mathsf{A}x.\ p\ x)\ f)\end{array}\tag{13}$$

Once we have introduced A, we can push it through $\to$ using the following two rewrite theorems.

$$\mathsf{A}x.\ (a\to p\ x)\ =\ (a\to(\mathsf{A}x.\ p\ x))\tag{14}$$

$$\mathsf{A}x.\ (p\ x\to a)\ =\ ((\mathsf{E}x.\ p\ x)\to a)\tag{15}$$

These rewrites push the quantifiers all the way to the eq combinators. We arrive at a situation where each eq combinator has an E quantifier surrounding it. Such occurrences of E and eq cancel out

$$\mathsf{E}x.\ \mathsf{eq}\ a\ x\ =\ a$$

leaving us with a theorem where all of the eq, A and E combinators have been removed:

$$\mathsf{Eval}\ env\ code\ \ ((a_1\to a_2\to\ldots\to a_n\to b)\ func)$$

The proofs of (13) and (14) require that the underlying big-step operational semantics is deterministic. This requirement arises from the fact that these lemmas boil down to an equation where an existential quantifier is moved across a universal quantifier.

$$\begin{array}{l}\forall x.\ \exists v.\ \langle env, code\rangle\Downarrow v\wedge\ldots\ =\\ \exists v.\ \langle env, code\rangle\Downarrow v\wedge\forall x.\ \ldots\end{array}$$

Such equations can be proved if we assume that $\Downarrow$ is deterministic since then there is only one v that can be chosen by the existential quantifier. Note that the definition of Eval in Section 6.4 would not have had its intended meaning if the operational semantics had been genuinely non-deterministic.

7. Related work

There is a long tradition in interactive theorem proving of using logics that look like functional programming languages: notable examples include LCF [30], the Boyer-Moore prover [3], the Calculus of Constructions [8], and TFL [19, 39]. The logic of the Boyer-Moore prover (and it successor, ACL2 [18]) are actual programming languages with standard denotational or operational semantics. However, many other systems, including Coq [7] and various HOL systems [35] (including Isabelle/HOL [17] and HOL4 [15]), use a more mathematical logic with model-theoretic or proof-theoretic semantics that differ from standard programming languages, e.g. the logics of HOL systems include non-computational elements. However, because these logics are based on various λ-calculi, they still resemble functional languages. A contribution of our work is to make this resemblance concrete by showing how (computable) functions in these logics can be moved to a language with a straight-forward operational semantics while provably preserving their meaning.

Slind's TFL library for HOL [39] and Krauss' extensions [19] make HOL's logic (which is roughly Church's simple theory of types) look like a functional language with support for well-founded general recursive definitions and nested pattern matching. We rely on TFL to collapse multi-clause definitions and to simplify pattern matching expressions (Sections 6.6 and 3.5).

Extraction from Coq [22] has two phases. First, purely logical content (e.g., proofs about the definitions) are removed from the

definitions to be extracted, then the remaining, computational context is printed to a programming language. The first step is theoretically well-justified; the second operates much as in HOL provers and is what we address in this paper.

ACL2 uses a first-order pure subset of Common Lisp as its logic, thus there is no semantic mismatch or need to perform extraction; logical terms are directly executable in the theorem prover. However, a translation technique similar to the one described in this paper can be of use when verifying the correctness of such theorem provers (including the correctness of their reflection mechanisms), as we did in previous work [10] using [32].

Proof producing synthesis has previously been used in HOL for various low-level targets including hardware [40] and assembly-like languages [24, 25, 26]. These systems implement verified compilers by term rewriting in the HOL4 logic. They apply a series of rewriting theorems to a HOL function yielding a proof that it is equivalent to a second HOL function that uses only features that have counterparts in the low-level language. Only then do they take a step relating these "low-level" HOL functions to the low-level language's operational semantics. This approach makes it easy to implement trustworthy compiler front-ends and optimisations, but significantly complicates the step that moves to the operational setting. In contrast, we move to (MiniML's) operational semantics immediately, which means that any preconditions we need to generate are understandable in terms of the original function, and not phrased in terms of a low-level intermediate language. This is why we can easily re-use the HOL-generated induction theorems to automatically prove termination.

In the other direction, proof producing decompilation techniques [23, 34] have addressed the problem of reasoning about low-level machine code by translating such code into equivalent HOL functions; however, these functions retain the low-level flavour of the machine language.

Charguéraud's characteristic formulae approach also addresses translation in the other direction, from OCaml to Coq [4], and it can support imperative features [5]. With his technique, an OCaml program is converted into a Coq formula that describes the program's behaviour, and verification is then carried out on this formula. His approach tackles the problem of verifying existing OCaml programs, which in particular requires the ability to handle partial functions and side effects. In contrast, this paper is about generating, from pure functional specifications, MiniML programs that are correct by construction. Part of our approach was inspired by Charguéraud's work, in particular our Eval predicate was inspired by his AppReturns predicate.

8. Future work

In this paper, we show how to create a verified path from the theorem prover to an operational semantics that operates on abstract syntax trees. We have not attempted to solve the problem of verified parsing or pretty printing. Ultimately, we want a verified compiler that will be able to accept abstract syntax as input, avoiding the problem altogether. However, it would still be useful to verify a translation from ASTs to concrete syntax strings for use with other compilers.

We have implemented our technique in HOL4 for translation to MiniML; however, we believe it would work for other target languages, so long as they both support ML-like features and can be given big-step semantics. Haskell support should be straightforward; laziness poses no problems because we are already proving termination under a strict semantics. We do rely on determinism of the big-step semantics for the quantifier shifting used in eq combinator removal (Section 6.8), but most languages that do not define evaluation order (e.g., Scheme, OCaml) should be able to support a deterministic semantics for the pure, total subset.

Our technique should also extend to other provers, including Isabelle/HOL and Coq. For function definitions that are in the ML-like fragment (i.e., that do not use sophisticated type classes or dependent types), including most of those in CompCert, it should be straightforward to implement our technique, although the details of the automation will vary.

Lastly, because MiniML also has a small step semantics, we hope to be able to verify complexity theoretic results about, e.g., our functional data structure case studies.

9. Conclusion

This paper's contribution is a step towards making proof assistants into trustworthy and practical program development platforms. We have shown how to give automated, verified translations of functions in higher-order logic to programs in functional languages. This increases the trustworthiness of programs that have been verified by shallowly embedding them in an interactive theorem prover, which has become a common verification strategy. We believe this is the first mechanically verified connection between HOL functions and the operational semantics of a high-level programming language. Our case studies include sophisticated data structures and algorithms, and validate the usefulness and scalability of our technique.

Acknowledgments

We thank Arthur Charguéraud, Anthony Fox, Mike Gordon, Kathy Gray, Ramana Kumar and Tom Sewell for commenting on drafts of this paper. This work was partially supported by EPSRC Research Grants EP/G007411/1, EP/F036345 and EP/H005633.

References

[1] G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end – Single Static Assignment meets CompCert. In H. Seidl, editor, *21st European Symposium on Programming, ESOP 2012*, volume 7211 of *LNCS*, pages 47–66. Springer, 2012.

[2] A. Barthwal and M. Norrish. Verified, executable parsing. In G. Castagna, editor, *18th European Symposium on Programming, ESOP 2009*, volume 5502 of *LNCS*, pages 160–174. Springer, 2009.

[3] R. S. Boyer and J. S. Moore. Proving theorems about LISP Functions. *Journal of the Association for Computing Machinery*, 22(1):129–144, 1975.

[4] A. Charguéraud. Program verification through characteristic formulae. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010*, pages 321–332. ACM, 2010.

[5] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011*, pages 418–430. ACM, 2011.

[6] A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 93–106. ACM, 2010.

[7] Coq. The Coq home page, 2012. `http://coq.inria.fr/`.

[8] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2–3):95–120, Feb. 1988.

[9] Z. Dargaye. *Vèrification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Universitè Paris 7 Diderot, July 2009.

[10] J. Davis and M. O. Myreen. The self-verifying Milawa theorem prover is sound (down to the machine code that runs it), 2012. `http://www.cl.cam.ac.uk/~mom22/jitawa/`.

[11] J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang. Functional correctness proofs of encryption algorithms. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence,*

and Reasoning: 12th International Conference, LPAR 2005, volume 3835 of *LNAI*, pages 519–533. Springer-Verlag, 2005.

[12] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, Aug. 1986.

[13] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[14] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Cambridge, UK, 1995.

[15] Hol. The HOL4 home page, 2012. `http://hol.sourceforge.net/`.

[16] J. Hurd. Verification of the Miller-Rabin probabilistic primality test. *J. Log. Algebr. Program.*, 56(1-2):3–21, 2003.

[17] Isabelle. The Isabelle home page, 2012. `http://www.cl.cam.ac.uk/research/hvg/isabelle/`.

[18] M. Kaufmann and J. S. Moore. The ACL2 home page, 2011. `http://www.cs.utexas.edu/users/moore/acl2/`.

[19] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universitiät München, 2009.

[20] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[21] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

[22] P. Letouzey. A new extraction for Coq. In *Proceedings of the 2002 International Conference on Types for Proofs and Programs*, TYPES'02, pages 200–219. Springer-Verlag, 2003.

[23] G. Li. Validated compilation through logic. In M. Butler and W. Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods*, volume 6664 of *LNCS*, pages 169–183. Springer, 2011.

[24] G. Li and K. Slind. Compilation as rewriting in higher order logic. In F. Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction*, volume 4603 of *LNCS*, pages 19–34. Springer, 2007.

[25] G. Li and K. Slind. Trusted source translation of a total function language. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 471–485. Springer, 2008.

[26] G. Li, S. Owens, and K. Slind. Structure of a proof-producing compiler for a subset of higher order logic. In R. D. Nicola, editor, *Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007*, volume 4421 of *LNCS*, pages 205–219. Springer, 2007.

[27] J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 237–248. ACM, 2010.

[28] D. Matthews. Poly/ML home page, 2012. `http://www.polyml.org`.

[29] A. McCreight, T. Chevalier, and A. P. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010*, pages 273–284, 2010.

[30] R. Milner. Logic for computable functions; description of a machine implementation. Technical Report STAN-CS-72-288, A.I. Memo 169, Stanford University, 1972.

[31] M. O. Myreen. Reusable verification of a copying collector. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010*, volume 6217 of *LNCS*, pages 142–156. Springer, 2010.

[32] M. O. Myreen. Functional programs: conversions between deep and shallow embeddings. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 412–418. Springer, 2012.

[33] M. O. Myreen and J. Davis. A verified runtime for a verified theorem prover. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving (ITP)*, volume 6898 of *LNCS*, pages 265–280. Springer, 2011.

[34] M. O. Myreen, K. Slind, and M. J. C. Gordon. Extensible proof-producing compilation. In O. de Moor and M. I. Schwartzbach, editors, *Compiler Construction, 18th International Conference, CC 2009*, volume 5501 of *LNCS*, pages 2–16. Springer, 2009.

[35] M. Norrish and K. Slind. A thread of HOL development. *Comput. J.*, 45(1):37–45, 2002.

[36] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[37] S. Owens and K. Slind. Adapting functional programs to higher-order logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, Dec. 2008.

[38] J. Ševčík , V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pages 43–54. ACM, 2011.

[39] K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, TU Munich, 1999.

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

[41] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.