JFP: page 1 of 32. © Cambridge University Press 2014
doi:10.1017/S0956796813000282

Proof-producing translation of higher-order logic into pure and stateful ML

MAGNUS O. MYREEN
Computer Laboratory, University of Cambridge, Cambridge, UK
(*e-mail:* magnus.myreen@cl.cam.ac.uk)

SCOTT OWENS
School of Computing, University of Kent, Canterbury, UK
(e-mail: S.A.Owens@kent.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 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 subset of Standard ML, and have applied the implementation to examples including functional data structures, a parser generator, cryptographic algorithms, a garbage collector and the 500-line kernel of the HOL light theorem prover. This paper extends our previous conference publication with new material that shows how functions defined in terms of a state-and-exception monad can be translated, with proofs, into stateful ML code. The HOL light example is also new.

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 Standard ML (SML), OCaml, Lisp or Haskell. This approach is now used even in very large verification efforts such as the CompCert verified compiler (Leroy, 2009) and several projects based on CompCert (McCreight *et al.*, 2010; Ševčík *et al.*, 2011; Barthe *et al.*, 2012); it has also been used in database verification (Malecha *et al.*, 2010).

However, the printing step is a potential weak link, as Harrison remarks in a survey on reflection (Harrison, 1995):

> [...] 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 (Myreen, 2012), i.e. we can synthesise verified Lisp from Lisp-like functions living in higher-order logic (HOL); and achieve *B* by running the generated programs on a verified Lisp implementation (Myreen & Davis, 2011) 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 HOL 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 SML language, for which we are constructing a verified compiler (Kumar *et al.*, 2014), so that *B* can be achieved. Our verified compiler and runtime are similar to Chlipala (2010), Dargaye (2009) and Myreen & Davis (2011). We call our ML subset CakeML 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 HOL but falls within the subset of the logic that corresponds directly to pure ML. As a result, we can translate sum into ML.[2]

```
val sum = foldl (fn x => fn 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`, with respect to 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,

[1] Throughout the paper, we will typeset HOL equations and definitions in sans-serif (constants) and *italic* (variables and types), and CakeML code in `typewriter`.

[2] Like Poly/ML (`http://www.polyml.org`), CakeML supports arbitrary precision integer arithmetic.

then it will return a result, a number, which 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 in logic, and deeply embedded values in the ML semantics, e.g. closures. Our solution involves refinement/coupling invariants and combinators over refinement invariants.

1.2 Contribution

This paper is an expanded version of our ICFP '12 paper 'Proof-producing synthesis of ML from higher-order logic' (Myreen & Owens, 2012). The main technical addition is the translation of HOL functions written in a state-and-exception monad into ML code that uses imperative features (i.e. `ref`, `raise` and `handle`). The HOL light example of Section 2 is also new, and the presentation has been reorganised.

The main contribution of the work overall is a new technique by which functions as defined in HOL can be translated, with proof, into ML equipped with an operational semantics. The ML-like subset of HOL we consider includes:

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

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 finite sets and finite maps.

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.

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 theorem prover's kernel.

Our translator, all of our examples, and our semantics for CakeML, are all part of the ongoing CakeML project, `https://cakeml.org/`.

2 Examples

Before explaining how our technique works, we first present a few examples which show what our proof-producing translation provides on simple examples. Section 6 lists several larger and more significant examples.

2.1 Quicksort—from algorithm proof to verified ML code

One can define quicksort for lists in HOL as follows.[3] Here ++ appends lists and partition splits a list into two lists: one with those elements that satisfy the given predicate and another with those that do not.

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

Given this definition of the algorithm, one can use a HOL theorem prover, such as HOL4, to prove the correctness of quicksort: given a transitive, total relation R and a list l, qsort returns a sorted permutation of list l.

$$\begin{array}{l}\forall R\ l\ l'. \\ \quad \mathsf{transitive}\ R \wedge \mathsf{total}\ R \wedge (l' = \mathsf{qsort}\ R\ l) \Longrightarrow \\ \quad \mathsf{perm}\ l\ l' \wedge \mathsf{sorted}\ R\ l'\end{array}$$

Such proofs are textbook exercises in program verification. Note that this definition and proof could have been (and indeed were) developed 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 abstract syntax tree (AST) for the following CakeML function. We write list cons as `::` and nil as `[]`.

```
fun qsort r = fn l => case l of
 | [] => []
 | (h::t) =>
    let val x = partition (fn y => r y h) t in
    case x of
     | Pair(l1,l2) =>
        append (append (qsort r l1) (h::[])) (qsort r l2)
    end
```

In the process of generating the above code, the translator also establishes a correspondence between CakeML values and HOL terms, and it automatically proves a theorem stating correctness of the translation. We will call such theorems *certificate theorems*. The following certificate theorem can informally be read as follows: when given an application of `qsort` to arguments corresponding to HOL terms, the CakeML operational semantics will terminate with a value that corresponds to the application of HOL function qsort to those terms. The formal statement of this theorem is shown below. Here bool is a refinement invariant, $\rightarrow$ is a refinement invariant combinator and Eval is a judgement about the CakeML semantics. These concepts (Eval, $\rightarrow$, bool, etc.) are defined in later sections.

$$\begin{array}{l}\mathsf{DeclAssum}\ \mathsf{qsort\_ml}\ env \Longrightarrow \\ \mathsf{Eval}\ env\ \lfloor \texttt{qsort} \rfloor\ (((a \rightarrow a \rightarrow \mathsf{bool}) \rightarrow \mathsf{list}\ a \rightarrow \mathsf{list}\ a)\ \mathsf{qsort})\end{array}$$

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

The beauty of these certificate theorems is that they allow the result of the algorithm verification to also apply to the generated CakeML code. By a trivial combination of the two theorems above, one can prove that the generated CakeML code always terminates and returns a sorted permutation of its input list whenever the generated `qsort` code has been loaded and the function value `r` corresponds to a transitive and total relation. The formal statement of this theorem is given below for reference.

$$
\begin{array}{l}
\forall \mathit{env}\ a\ \mathit{ord}\ R\ l\ \mathit{xs}. \\
\quad \mathsf{DeclAssum}\ \mathsf{qsort\_ml}\ \mathit{env} \wedge \\
\quad \mathsf{list}\ a\ l\ \mathit{xs} \wedge (\mathsf{lookup}\ \texttt{"xs"}\ \mathit{env} = \mathsf{some}\ \mathit{xs}) \wedge \\
\quad (a \to a \to \mathsf{bool})\ \mathit{ord}\ R \wedge (\mathsf{lookup}\ \texttt{"R"}\ \mathit{env} = \mathsf{some}\ R) \wedge \\
\quad \mathsf{transitive}\ \mathit{ord} \wedge \mathsf{total}\ \mathit{ord} \\
\quad \Longrightarrow \\
\quad \exists l'\ \mathit{xs}'. \\
\qquad \langle \mathsf{emp}, \mathit{env} \rangle \vdash \lfloor \texttt{qsort R xs} \rfloor \Downarrow \langle \mathsf{emp}, \mathsf{Rval}\ \mathit{xs}' \rangle \wedge \\
\qquad \mathsf{list}\ a\ l'\ \mathit{xs}' \wedge \mathsf{perm}\ l\ l' \wedge \mathsf{sorted}\ \mathit{ord}\ l'
\end{array}
$$

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

2.2 RedBlack trees—from algorithm proofs to more advanced translations

The translation technique described in this paper can translate ML-like functions' definitions from HOL into CakeML. However, the ML-like subset of HOL that is supported is not fixed and can easily be extended. The translation routine can be 'taught' how to translate new HOL types and terms.

The following example will show how operations over finite maps can be translated into operations over RedBlack trees in CakeML. First, we look at how the finite-map type, (α,β) *fmap*, is defined in HOL. In HOL, every non-primitive type must map into some previously defined type, its *representation type*. New types are defined based on a representation type, a characteristic predicate and a theorem that proves that the predicate can be satisfied. For finite maps, we can use representation type $\alpha \to \beta + \mathit{unit}$ and predicate $\mathsf{is\_fmap}\ m = \mathsf{finite}\ \{a \mid m\ a \neq \mathsf{inr}\ ()\}$, and prove $\exists m.\ \mathsf{is\_fmap}\ m$. Using these components, a type definition can be made. A type definition results in a new type, e.g. (α,β) *fmap*, and two functions: one that maps terms of the new type into the representation type and another that performs the reverse translation; e.g. fmap_rep and fmap_abs respectively. Using these new functions, it is a simple exercise to define finite-map update $\mapsto$:

$$m[x \mapsto y] \ = \ \mathsf{fmap\_abs}\ (\lambda a.\ \mathsf{if}\ a = x\ \mathsf{then}\ \mathsf{inl}\ y\ \mathsf{else}\ \mathsf{fmap\_rep}\ f\ a)$$

Most HOL provers come equipped with such type definitions, and a long list of lemmas about them, as part of their standard library.

As an example, consider that we want to translate the finite-map update operation, $\mapsto$, into a corresponding operation over RedBlack trees. The first step is to formalise RedBlack tree operations, such as lookup and update, as functions in HOL:

$$
\begin{array}{l}
\mathit{color} \;=\; \mathsf{Red} \mid \mathsf{Black} \\
(\alpha, \beta)\ \mathit{tree} \;=\; \mathsf{Empty} \mid \mathsf{Tree}\ \mathit{color}\ \mathit{tree}\ \alpha\ \beta\ \mathit{tree} \\
\\
\mathsf{lookup}\ R\ x\ \mathsf{Empty} \;=\; \mathsf{None} \\
\mathsf{lookup}\ R\ x\ (\mathsf{Tree}\ \mathit{color}\ a\ y\ z\ b) \;= \\
\quad \mathsf{if}\ R\ x\ y\ \mathsf{then}\ \mathsf{lookup}\ R\ x\ a\ \mathsf{else} \\
\quad \mathsf{if}\ R\ y\ x\ \mathsf{then}\ \mathsf{lookup}\ R\ x\ b\ \mathsf{else}\ \mathsf{Some}\ z \\
\\
\mathsf{update}\ R\ x\ y\ t \;= \\
\quad \mathsf{case}\ \mathsf{upd}\ R\ x\ y\ t\ \mathsf{of}\ \mathsf{Tree}\ \_\ a\ x\ y\ b \Rightarrow \mathsf{Tree}\ \mathsf{Black}\ a\ x\ y\ b \\
\\
\mathsf{upd}\ R\ x\ y\ t \;=\; \ldots
\end{array}
$$

One can then prove that the RedBlack trees correctly implement operations over finite maps if a RedBlack tree invariant is met. For example, for update one can prove the following, given an appropriate definition of a relationship between RedBlack trees and finite maps: redblack_represents_fmap.

$$
\begin{array}{l}
\forall R\ \mathit{tree}\ \mathit{fmap}\ x\ y. \\
\quad \mathsf{redblack\_represents\_fmap}\ R\ \mathit{tree}\ \mathit{fmap} \Longrightarrow \\
\quad \mathsf{redblack\_represents\_fmap}\ R\ (\mathsf{update}\ R\ x\ y\ \mathit{tree})\ (\mathit{fmap}[x \mapsto y])
\end{array}
\tag{1}
$$

By translating the ML-like functions, we can produce CakeML code that is proved to implement the RedBlack tree operations. The translation of update produces a certificate theorem of the following form:

$$
\begin{array}{l}
\mathsf{DeclAssum}\ \mathsf{redblack\_ml}\ \mathit{env} \Longrightarrow \\
\mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{update} \rfloor\ (((a \to a \to \mathsf{bool}) \to a \to b \to \mathsf{tree}\ a\ b \to \mathsf{tree}\ a\ b)\ \mathsf{update})
\end{array}
\tag{2}
$$

By combining Lemmas (1) and (2), we can prove a theorem expressed in terms of HOL's finite maps, rather than the *tree* type, in terms of which update is defined. For simplicity, in this example, we instantiate α to *int*, a to int and R to $\leqslant$.

$$
\begin{array}{l}
\mathsf{DeclAssum}\ \mathsf{redblack\_ml}\ \mathit{env} \Longrightarrow \\
\mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{update (<=)} \rfloor\ ((\mathsf{int} \to b \to \mathsf{fmap}\ b \to \mathsf{fmap}\ b)\ (\mapsto)) \\
\\
\text{where}\ \ \mathsf{fmap}\ b\ m\ v \;=\; \exists t.\ \mathsf{redblack\_represents\_fmap}\ (\leqslant)\ t\ m \wedge \mathsf{tree}\ \mathsf{int}\ b\ t\ v
\end{array}
\tag{3}
$$

By supplying altered certificate theorems, such as (3), back to the translation routine, future translations can translate more abstract opera operations over finite maps into CakeML. For example, the translation of HOL term $m[2 \mapsto x][y \mapsto z]$ produces the following CakeML using Lemma (3) whenever it encounters $\mapsto$ for finite maps:

```
update (<=) y z (update (<=) 2 x m)
```

The same also works for nested refinement invariants, e.g. term $m[2 \mapsto (t[y \mapsto z])]$ translates into the following with refinement invariant fmap (fmap a).

```
update (<=) 2 (update (<=) y z t) m)
```

This extension mechanism is discussed in Section 4.4.4.

2.3 HOL light—from state-and-exception monads to stateful CakeML code

Functions in HOL are pure and stateless. As a result, translation into pure CakeML is natural. However, as is shown in Section 5, if the HOL functions are written using a state-and-exception monad then the functions can be translated into CakeML code that uses imperative features such as references and exceptions. With only a few modifications to the core definitions for translation into pure CakeML (Section 4) we can perform exactly the same style of translation into stateful CakeML.

In the HOL light case study, we encountered functions that read from and write to references, raise exceptions, and pass monadic functions as first-class values, e.g. assoc can raise an exception

$$\begin{array}{l}\mathsf{assoc}\ s\ l\ = \\ \quad \mathsf{case}\ l\ \mathsf{of} \\ \quad\ [] \Rightarrow \mathsf{failwith}\ \texttt{"not in list"} \\ \quad \mid ((x,y) :: t) \Rightarrow \mathsf{if}\ s = x\ \mathsf{then}\ \mathsf{return}\ y\ \mathsf{else}\ \mathsf{assoc}\ s\ t\end{array}$$

and map takes a monadic function f as input

$$\begin{array}{l}\mathsf{map}\ f\ l\ = \\ \quad \mathsf{case}\ l\ \mathsf{of} \\ \quad\ [] \Rightarrow \mathsf{return}\ [] \\ \quad \mid (h :: t) \Rightarrow \mathsf{do}\ \ h' \leftarrow f\ h\ ;\ t' \leftarrow \mathsf{map}\ f\ t\ ;\ \mathsf{return}\ (h' :: t')\ \mathsf{od}\end{array}$$

The most complicated function in the HOL light case study uses exceptions to perform efficient backtracking in the implementation of type instantiation.

The generated code uses CakeML's `let` expressions for translation of the monadic bind operator. For example, the generated code for map is as follows:

```
fun map f l =
  case l of
    [] => []
  | (h::t) => let
                val h' = f h
                val t' = map f t
              in
                h'::t'
              end
```

The resulting certificate theorems are now stated in terms of slightly modified version of the predicates used above, e.g. instead of Eval we have EvalM.

$$\ldots \implies \mathsf{EvalM}\ env\ \lfloor \texttt{map} \rfloor\ (\mathsf{refinement\_invariant}\ \mathsf{map})$$

where refinement_invariant is $((\mathsf{pure}\ a \rightarrow^{\mathsf{M}} \mathsf{M}\ b) \rightarrow^{\mathsf{M}} \mathsf{pure}\ (\mathsf{list}\ a) \rightarrow^{\mathsf{M}} \mathsf{M}\ (\mathsf{list}\ b))$.

3 Target language: CakeML

In this paper, the target of translation is a subset of SML that we call CakeML. The subset supports:

- arbitrary precision integers,
- mutually recursive datatype definitions,
- higher-order, anonymous and mutually recursive functions,
- nested pattern matching,
- references, and
- handled exceptions.

Unsupported features include records, functors and `local` definitions. Figure 1 gives the source grammar for CakeML types t, literals l, patterns p, expressions e, type definitions td/c and top-level definitions d.

We define, in HOL4, a CakeML big-step, call-by-value operational semantics, with function values represented as closures. Figure 2 gives the auxiliary definitions needed to support the semantics: values v, environments env, locations loc, reference stores s and evaluation results r. The big-step semantics is deterministic:

$$\forall s_1\ env\ e\ r_1\ r_2.\\ \langle s, env\rangle \vdash e \Downarrow r_1\ \wedge\ \langle s, env\rangle \vdash e \Downarrow r_2\ \Longrightarrow\ (r_1 = r_2)$$

3.1 Design rationale

CakeML is designed to be the interface between the two tasks from Section 1: A—translating from logic to a programming language, and B—building a verified platform for running the language. In this paper, we solve task A, but we also take care to ensure that CakeML is a suitable language for task B (Kumar *et al.*, 2014). Here, we explain our design choices:

1. CakeML is a non-trivial subset of a real programming language. A simple λ-calculus-like programming language would satisfy the technical requirements of being expressive enough to map HOL functions to, and of being simple enough to support a fully verified implementation. However, that choice would limit the use of both the translator (part A) and compiler (part B). By making CakeML a subset of a real language, we generate code that can be used with existing industrial-strength optimising compilers for cases where verified compilation is not considered necessary (e.g. regression testing of an application to be verified). By making CakeML a large enough subset to actually program in, we ensure that the result of task B will be useful for functions that do not originate in a proof assistant.
2. CakeML is a subset of SML. Of existing programming languages, HOL's logic is most similar to ML, so it is a natural choice, and because we only translate terminating functions, call-by-value semantics are acceptable. SML has a minor advantage over OCaml in this setting: evaluation order of function arguments is well defined in SML (but not in Ocaml). However, adapting the translation process to such a setting is a simple exercise in just inserting the necessary `let`-expressions to force a left-to-right evaluation order.

t ::= α | `int` | `bool` | `unit` | t `*` t | t `->` t | t `list` | t `ref` | tc | `(`t(`,`t)*`)` tc | `(`t`)`

l ::= i | `true` | `false` | `()` | `[]`

p ::= `_` | x | l | C | C p | `(`p(`,`p)*`)` | p`::`p | `[`p(`,`p)*`]` | `ref` p

e	::=	l	literal constant
	|	x	variable reference
	|	C	constant constructor
	|	C e	constructor
	|	`(`e`,`e(`,`e)*`)`	tuple
	|	`[`e`,`e(`,`e)*`]`	list
	|	`raise` e	exception raising
	|	e `handle` p `=>` e (`|` p `=>` e)*	exception handling
	|	`fn` x `=>` e	function
	|	e e	function application
	|	uop e	unary operator
	|	e op e	binary operator
	|	`(`(e`;`)*e`)`	sequencing
	|	`if` e `then` e `else` e	conditional
	|	`case` e `of` p `=>` e (`|` p `=>` e)*	pattern matching
	|	`let` (ld`|``;`)$^*$ `in` ($e$ `;`)$^*$ $e$ `end`	let definitions
ld	::=	`val` x `=` e	value definition
	|	`fun` x y^+ `=` e (`and` x y^+ `=` e)*	function definition
uop	::=	`ref` | `!`	reference and dereference
	|	`~`	negation
op	::=	`:=`	assignment
	|	`+` | `-` | `*` | `div` | `mod`	arithmetic
	|	`=` | `<` | `<=` | `>` | `>=` | `<>`	comparison
	|	`::`	cons
	|	`before` | `andalso` | `orelse`	sequencing and logical
c	::=	C | C `of` t	
tyn	::=	`(`α(`,`α)*`)` x | α x | x	
tyd	::=	tyn `=` c (`|` c)*	
d	:=	`val` p `=` e	value declaration
	|	`fun` x y^+ `=` e (`and` x y^+ `=` e)*	function declaration
	|	`datatype` tyd (`and` tyd)*	type declaration
	|	`exception` c	exception declaration

where x and y range over identifiers (must not start with a capital letter), α over SML-style type variable (e.g. `'a`), C over constructor names (must start with a capital letter), tc over type constructor names and i over integers.

Fig. 1. CakeML source grammar (excluding modules).

3. CakeML has big-step operational semantics. Because the translator is inductive on the syntax of HOL functions, it is most convenient to have a semantics that is inductive on CakeML's syntax. However, as part of task B, different semantics are required for different purposes. Their equivalence needs to be proved.
4. CakeML's semantics are not taken directly from the definition of SML (Milner *et al.*, 1997). Although they are similar, CakeML is simpler because it omits some complexities. For example, CakeML does not include arbitrary declarations, including

$$
\begin{array}{lcll}
v & := & \mathsf{Lit}\ l & \text{literal constant} \\
 & | & \mathsf{Con}\ C\ [v_1,\ldots,v_n] & \text{constructor} \\
 & | & \mathsf{Closure}\ env\ x\ e & \text{closure} \\
 & | & \mathsf{Recclosure}\ env\ [(x_1,y_1,e_1),\ldots,(x_n,y_n,e_n)]]\ x & \text{recursive function closure} \\
 & | & \mathsf{Loc}\ loc & \text{a reference to the store} \\
r & := & \langle s, \mathsf{Rval}\ v\rangle & \\
 & | & \langle s, \mathsf{Rerr}\ ex\rangle &
\end{array}
$$

where *env* ranges over finite maps from x to v,
loc ranges over store locations, and
s ranges over finite maps from *loc* to v.

Fig. 2. Semantic auxiliaries for CakeML.

`datatype`, inside of `let`, and it enforces the OCaml-style restriction that constructors begin upper-case and other names begin lower case.

5. CakeML's native integers are arbitrary precision. This is most convenient for translation from HOL's logic which uses natural numbers; however, there is precedent in practical implementation: the Poly/ML compiler implements arbitrary precision integers natively; other ML implementations usually support them as a library. This decision requires that the part B compiler come equipped with a verified bignum library to implement arbitrary precision arithmetic (Myreen & Curello, 2013).

4 Synthesis of pure ML

The following sections explain our approach to proof-producing synthesis of CakeML from functions in HOL. This section explains our approach for producing *pure* ML functions. Section 5 describes an extension which can produce *stateful* ML.

4.1 Core definitions and concepts

Each run of the translation algorithm produces a proof with respect to the CakeML operational semantics (Section 3). The entire translation approach is developed to produce such proofs, and thus centred around the operational semantics. The synthesis algorithm does not make direct statements about the operational semantics; instead a predicate called Eval is used to express properties of the operational semantics.

We define Eval *env exp post* to be true if CakeML expression *exp* evaluates, in environment *env*, to some value *val* such that *post* is true for *val*, i.e. *post val*. The fact that it returns a value—as opposed to an error—tells us that evaluation terminates and that no error happened during evaluation, e.g. evaluation did not hit any missing cases while pattern matching. Below, $\Downarrow$ is the evaluation relation from the big-step semantics for CakeML, emp is the empty state and *post* has type $v \to bool$. Here Eval requires that the expression is pure: given an empty state, evaluation must return an empty state.

$$\mathsf{Eval}\ env\ exp\ post \quad = \quad \exists val.\ (\langle \mathsf{emp}, env\rangle \vdash exp \Downarrow \langle \mathsf{emp}, \mathsf{Rval}\ val\rangle)\ \wedge\ post\ val$$

The interesting part is how *post* gets instantiated and used. We instantiate *post* with predicates that relate terms in HOL with CakeML values from the semantics of CakeML, i.e. values of type v. These predicates are refinement invariants, sometimes called coupling

invariants. The most basic refinement invariants relate HOL representations of boolean and integers to the same concepts in the CakeML semantics. We define refinement invariants bool and int as follows:

$$\begin{array}{rcll} \mathsf{bool}\ \mathtt{true} & = & \lambda v.\ (v = \mathsf{Lit}\ \mathtt{true}) & \\ \mathsf{bool}\ \mathtt{false} & = & \lambda v.\ (v = \mathsf{Lit}\ \mathtt{false}) & \\ \mathsf{int}\ i & = & \lambda v.\ (v = \mathsf{Lit}\ i) & \text{where } i \text{ is an integer.} \end{array}$$

We make statements in terms of Eval and refinement invariants. For example, Eval and int can be used to state that the constant CakeML expression `5` always evaluates to an CakeML value that is related to integer 5 in HOL. Throughout, we write SML syntax enclosed within $\lfloor \cdot \rfloor$ as an abbreviation for the often verbose AST for CakeML:

$$\mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{5} \rfloor\ (\mathsf{int}\ 5)$$

Similarly, the Eval predicate can also be used to make statements about variables. Using Eval, we can, for example, state that CakeML expression `n`, i.e. CakeML variable `n`, evaluates to a value that corresponds to HOL integer n:

$$\mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{n} \rfloor\ (\mathsf{int}\ n)$$

4.2 Bottom-up translation of terms

Given a HOL term to translate, e.g. $n+5$, the goal of our proof-producing translation is to construct a CakeML expression, in this case $\lfloor \mathtt{n+5} \rfloor$, and prove a theorem that relates the HOL term with the evaluation of the CakeML expression. The resulting theorem is stated in terms of Eval. For $n+5$, this resulting theorem is to state that $n+5$ is Eval-related to $\lfloor \mathtt{n+5} \rfloor$, if `n` is related to HOL variable n.

$$\begin{array}{l} \mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{n} \rfloor\ (\mathsf{int}\ n) \implies \\ \mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{n+5} \rfloor\ (\mathsf{int}\ (n+5)) \end{array}$$

For a given HOL term t and generated CakeML expression *exp*, the shape of the resulting theorem is always the following, for some appropriate refinement invariant *inv*:

$$\mathit{assumptions} \implies \mathsf{Eval}\ \mathit{env}\ \mathit{exp}\ (\mathit{inv}\ t) \tag{4}$$

Translation of HOL terms is performed as a bottom-up traversal based on the syntax of the given HOL term. Each recursive call in this traversal returns a theorem of shape (4). For term $n+5$, the leaves of this bottom-up traversal, i.e. n and 5, prove the following theorems:

$$\mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{n} \rfloor\ (\mathsf{int}\ n) \implies \mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{n} \rfloor\ (\mathsf{int}\ n) \tag{5}$$

$$\mathsf{true} \implies \mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{5} \rfloor\ (\mathsf{int}\ 5) \tag{6}$$

Compound expressions are combined using lemmas that aid translation. For the running example, the HOL operation for integer addition is translated using the following lemma

relating integer addition in HOL (+) with integer addition in CakeML (+):

$$\begin{array}{l}\forall e_1\ e_2\ i\ j.\\ \quad \mathsf{Eval}\ env\ \lfloor e_1 \rfloor\ (\mathsf{int}\ i) \wedge\\ \quad \mathsf{Eval}\ env\ \lfloor e_2 \rfloor\ (\mathsf{int}\ j) \Longrightarrow\\ \quad \mathsf{Eval}\ env\ \lfloor e_1\ \texttt{+}\ e_2 \rfloor\ (\mathsf{int}\ (i+j))\end{array}$$

The lemma above is used to combine theorems (5) and (6), to prove the desired theorem, which is as follows:

$$\mathsf{Eval}\ env\ \lfloor \texttt{n} \rfloor\ (\mathsf{int}\ n) \implies \mathsf{Eval}\ env\ \lfloor \texttt{n+5} \rfloor\ (\mathsf{int}\ (n+5)) \tag{7}$$

The same lemma can, of course, be used to translate any combination of integer HOL variables, integer constants and integer addition, e.g.

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

Entire term translation—as opposed to translation of recursive functions (Section 4.3)—is performed in exactly this bottom-up manner. The following subsections detail how features such as type variables and λ-abstractions fit into this approach to term translation. Support for ML-like features such as pattern-matching and partial specifications is covered in Section 4.4.

4.2.1 Functions as first-class values

Both HOL and CakeML support the use of functions as first-class values. In order to allow for function values in the translation, we need a refinement invariant that relates function values in HOL with function values in CakeML, i.e. closures. For this purpose, we have a refinement combinator $\rightarrow$. Given refinement invariants, a and b, this refinement combinator $a \rightarrow b$ is a refinement invariant between function values in HOL and CakeML. We define $(a \rightarrow b)\ f\ cl$ to be true if cl is a closure such that, whenever the closure is applied to a value v satisfying refinement invariant input a, it returns a value u satisfying output invariant b; and furthermore, its input–output relation coincides with f with respect to a and b.

This combinator's formal 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 $a \rightarrow b$ to be true for function f and closure cl if and only if, for every input x and CakeML value v such that $a\ x\ v$, the closure cl applied to v evaluates to some value u such that refinement invariant b relates f applied to x with u.

$$(a \rightarrow b)\ f \quad = \quad \lambda cl.\ \forall x\ v.\ a\ x\ v \Longrightarrow \exists u.\ \mathsf{evaluate\_closure}\ v\ cl\ u \wedge b\ (f\ x)\ u$$

Here the type of f is $\alpha \rightarrow \beta$, and cl, v and u are CakeML values, i.e. have type v.

This refinement combinator allows us to make statements about function values. For example, the following states that `f` evaluates to a closure which corresponds to a HOL function f which maps integers to integers.

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

To aid translation we have a few lemmas for reasoning about function values: one for function application, and a few for translation of λ-abstractions. The lemma which enables translation of function application is the following:

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

With this lemma, it is easy to translate $f\ 5$ into CakeML using the bottom-up traversal technique outlined above. The result of such a translation is a theorem:

$$\mathsf{Eval}\ env\ \lfloor \texttt{f} \rfloor\ ((\mathsf{int} \to \mathsf{int})\ f) \implies \mathsf{Eval}\ env\ \lfloor \texttt{f 5} \rfloor\ (\mathsf{int}\ (f\ 5))$$

The simplest lemma for the translation of λ-abstractions is the following. This lemma requires that the abstract and concrete values, x and v, can be universally quantified. Here n is a name and $n \mapsto v$ extends the environment *env* with binding: name n maps to value v.[4]

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

As an example, this lemma allows for the translation of terms such as $\lambda n.\ n+5$. The proof essentially matches (7) with the left-hand side of the lemma above in order to reach the following:

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

The lemma above is sometimes not directly applicable. The reason is that the universal quantifier on x on the left-hand side of the lemma above is too restrictive. Consider, for example, translation of the term $\lambda n.\ 5\ \mathsf{div}\ n$. Translation of the body produces a theorem where n has a side condition other than just a binding to name `n`.

$$n \neq 0\ \wedge\ \mathsf{Eval}\ env\ \lfloor \texttt{n} \rfloor\ (\mathsf{int}\ n) \implies \mathsf{Eval}\ env\ \lfloor \texttt{5 div n} \rfloor\ (\mathsf{int}\ (5\ \mathsf{div}\ n))$$

In such cases, a less restrictive form of the lemma from above is used. The less restrictive lemma is the same except that the abstract variable is not universally quantified. The price one must pay is the introduction of an `eq` combinator that restricts the input to be exactly value x. Here and throughout $\mathsf{eq}\ a\ x\ y\ v = (x = y) \wedge a\ y\ v$.

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

With this lemma, the translation of $\lambda n.\ 5\ \mathsf{div}\ n$ yields

$$\forall n.\ n \neq 0 \implies \mathsf{Eval}\ env\ \lfloor \texttt{fn n => 5 div n} \rfloor\ ((\mathsf{eq}\ \mathsf{int}\ n \to \mathsf{int})\ (\lambda n.\ 5\ \mathsf{div}\ n))$$

A different but somewhat similar looking lemma is used for translation of HOL's let-expressions. Below, let is HOL's internal combinator which represents let-expressions. In

[4] The CakeML semantics represents names in a very direct manner: the names appear as strings in the deep embedding. Variable expressions are evaluated as look-ups in an environment that the semantics carries around. Note that our tool never needs to perform substitution or α-conversion on CakeML expressions (for HOL terms this is supported natively in the logic). The tool just constructs CakeML expressions bottom-up. It can therefore avoid the many technical difficulties (Aydemir *et al.*, 2005) associated with substitution and variable renaming in deeply embedded syntax.

HOL, let abbreviates $\lambda f\ x.\ f\ x$ 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 $a+1$.

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

Lemmas for translation of recursive functions are described in Section 4.3.

4.2.2 Type variables

The examples above 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 $\alpha \to v \to bool$ as the invariant. The HOL type of the int refinement invariant is $int \to v \to bool$, 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 CakeML 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 CakeML value is a closure such that HOL function f of type $\alpha \to \alpha$ is an accurate representation.

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

With this approach to translation of terms with free type variables, we can apply the lemmas mentioned above at a more abstract level. For example, we can derive CakeML code corresponding to a HOL function $\lambda f\ x.\ f\ (f\ x)$ which contains a type variable α.

$$\begin{array}{ll}\forall a. & \mathsf{Eval}\ env\ \lfloor \texttt{fn f => fn x => f (f x)} \rfloor \\ & (((a \to a) \to a \to a)\ (\lambda f\ x.\ f\ (f\ x)))\end{array} \tag{8}$$

4.3 Translation of (recursive) functions

The previous section presented how terms can be translated with certificate proofs from HOL to CakeML. The following subsections explain how we apply this term translation to translate top-level function definitions, and, in particular, recursive functions.

4.3.1 Translation of non-recursive functions

When a term is translated into CakeML, as above, the result is a CakeML expression. However, at the top-level, a CakeML program consists of a list of declarations. We therefore want top-level HOL terms to be translated into CakeML declararions, not expressions.

Before turning our attention to recursive functions, we first continue the example from Section 4.2.2 on translation of a non-recursive function. We show how CakeML terms are packaged up into lists of declarations.

The example from Section 4.2.2 translated $\lambda f\, x.\ f\ (f\ x)$ into a CakeML expression. The certificate theorem produced is as follows:

$$\begin{aligned}&\mathsf{Eval}\ env\ \lfloor\texttt{fn f => fn x => f (f x)}\rfloor\\ &\quad(((a \to a) \to a \to a)\ (\lambda f\, x.\ f\ (f\ x)))\end{aligned}$$

If twice is a HOL constant defined to be $\lambda f\, x.\ f\ (f\ x)$, then this can be rephrased as follows:

$$\begin{aligned}&\mathsf{Eval}\ env\ \lfloor\texttt{fn f => fn x => f (f x)}\rfloor\\ &\quad(((a \to a) \to a \to a)\ \mathsf{twice})\end{aligned}$$

To introduce a declaration giving a similar name to the generated CakeML expression, one applies a lemma, explained below, which introduces a declaration assumption DeclAssum (see Section 4.3.1). The result is a theorem which states: if the environment *env* is the result of evaluating the top-level declarations in the DeclAssum, then evaluation of the CakeML name `twice` corresponds to the behaviour of HOL function twice.

$$\begin{aligned}&\mathsf{DeclAssum}\ \lfloor\texttt{val twice = fn f => fn x => f (f x);}\rfloor\ env \implies\\ &\mathsf{Eval}\ env\ \lfloor\texttt{twice}\rfloor\ (((a \to a) \to a \to a)\ \mathsf{twice})\end{aligned}$$

The goal of a complete translation effort is to produce a CakeML program, i.e. a list of CakeML declararions.

The following lemma is used to introduce CakeML declarations for terms and non-recursive functions. This lemma appends a new declaration to the end of a DeclAssum list of declarations. In cases where there is no previous declaration assumption, one can simply add a new assumption DeclAssum with an empty declaration list.

$$\begin{aligned}&(\forall env.\ \mathsf{DeclAssum}\ \lfloor decs\rfloor\ env \implies \mathsf{Eval}\ env\ \lfloor exp\rfloor\ post) \implies\\ &(\forall env.\ \mathsf{DeclAssum}\ \lfloor decs\ \texttt{val}\ n\ \texttt{=}\ exp\texttt{;}\rfloor\ env \implies \mathsf{Eval}\ env\ \lfloor n\rfloor\ post)\end{aligned}$$

A full translation consists of translating HOL function- and constant-definitions one at a time. Each translation appends the new declaration to the list of generated declarations.

4.3.2 Algorithm for translation of (recursive) functions

The previous section provided an example which explained how a function constant from HOL can be translated into a CakeML declaration. In this section, we summarise the high-level steps that are involved in each function translation in general, whether non-recursive or not. For clarity, this description assumes that the function is not mutually recursive. The algorithm is, however, easily modified to also apply to mutually recursive functions. Our implementation supports mutual recursion.

Information retrieval. Given a function f to translate, the initial phase collects the necessary information about this 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 theory context. The recursive case will be explained in the next section.

Preprocessing. The next step prepares the definition for translation: the definition is collapsed to a single top-level clause, as mentioned in Section 4.4.1, 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 following 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 4.2. This theorem is derived through a bottom-up traversal of the HOL term. At each stage, a proof rule or lemma is applied to introduce the corresponding CakeML 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 CakeML counterpart.

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

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

Packaging. The next phase reduces the term *rhs* to the function constant f. To do this, lemmas are applied that introduce a λ-abstraction for each formal parameter, and then the following simplification on the right-hand side is performed: 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-abstractions on the right-hand side of the HOL expression introduces closures on the CakeML side. For recursive functions, the final closure lemma is a special rule for introducing a `fun`-declaration for recursive functions.

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, which will be presented in Section 4.4.2.

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. The resulting certificate theorem is always of the form, for some *inv* and some possible *precondition*:

$$\forall env.\ \mathsf{DeclAssum}\ \lfloor declarations \rfloor\ env \wedge precondition \implies \mathsf{Eval}\ env\ \lfloor \mathtt{f} \rfloor\ (inv\ \mathsf{f})$$

4.3.3 Translation of recursive functions

CakeML code for non-recursive functions can be derived as shown above. Recursive HOL functions require some additional effort. To illustrate why, consider the following definition of the gcd function:

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

If we were to do the **bottom-up traversal** step for the right-hand side of the definition of gcd in exactly the same way it is done for non-recursive functions, then we would get

stuck. The method described in bottom-up derivation described in Section 4.2 would not have an Eval-theorem describing the recursive call. At the stage where it gets stuck, one would like to have a theorem of the following form:

$$\ldots \Longrightarrow \mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{gcd} \rfloor\ ((\mathsf{int} \to \mathsf{int} \to \mathsf{int})\ \mathsf{gcd})$$

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

We use the combinator eq, mentioned earlier in Section 4.2.1, to 'record' what values we have assumed that the recursive call is applied to.

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

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

We state the assumption at call sites using the eq combinator, for some m and n:

$$\mathsf{Eval}\ \mathit{env}\ \lfloor \mathtt{gcd} \rfloor\ ((\mathsf{eq\ int}\ m \to \mathsf{eq\ int}\ n \to \mathsf{int})\ \mathsf{gcd}) \tag{9}$$

For the rest of this example we abbreviate (9) 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 for argument n,

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

and an Eval theorem for argument $m \bmod n$,

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

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

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

and then we apply to application lemma for $\to$ from Section 4.2.1 to get an Eval theorem for $\mathsf{gcd}\ n\ (m \bmod n)$:

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

By then continuing the bottom-up traversal as usual and packaging up the right-hand side into a declaration, we arrive at the following theorem where our abbreviation P appears both as an assumption and as the conclusion:

$$\begin{array}{l} \mathsf{DedclAssum}\ \lfloor \mathtt{fun\ gcd\ m\ =\ fn\ n\ =>\ ...} \rfloor\ \mathit{env} \Longrightarrow \\ \forall m\ n.\ (0 < n \Longrightarrow \mathsf{P}\ n\ (m \bmod n)) \Longrightarrow \mathsf{P}\ m\ n \end{array} \tag{10}$$

The shape of the right-hand side of the implication above matches the left-hand side of the following induction theorem:

$$\forall P.\ \ (\forall m\ n.\ (0 < n \implies P\ n\ (m \bmod n)) \implies P\ m\ n) \implies (\forall m\ n.\ P\ m\ n) \quad (11)$$

Such induction theorems come out as a side product of the conventional definition mechanisms that are built on top of HOL (Slind, 1999; Krauss, 2009). The most commonly used definition mechanism uses (or guesses) a well-founded measure in order to prove totality of recursive functions. Our translation algorithm relies on induction theorems that match the shape of the function that is to be translated. If no such induction theorem exists, i.e. the recursive function was not defined using the most common definition mechanism, then custom induction theorems can be used instead. For the running of gcd, one might use a custom induction theorem with an assumption gcd_terminates_for $m\ n$.

$$\begin{array}{l}\forall P.\ \ (\forall m\ n.\ (0 < n \implies P\ n\ (m \bmod n)) \implies P\ m\ n) \implies \\ \quad (\forall m\ n.\ \mathsf{gcd\_terminates\_for}\ m\ n \implies P\ m\ n)\end{array}$$

Note that this works for functions that do not 'terminate' for all input values.

By one application of modus ponens of (10) and (11), 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, when possible, eq (as will be explained in the next section), we arrive at the following desired certificate theorem for the gcd function:

$$\begin{array}{l}\mathsf{DeclAssum}\ \lfloor \texttt{fun gcd m = fn n => ...} \rfloor\ env \implies \\ \mathsf{Eval}\ env\ \lfloor \mathsf{gcd} \rfloor\ ((\mathsf{int} \to \mathsf{int} \to \mathsf{int})\ \mathsf{gcd})\end{array}$$

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 multiple 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 next.

4.3.4 Simplification of eq

The example above 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 the following 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 $\mathsf{A}\ (\lambda x.\ p\ x)$.

$$(\forall x.\ \mathsf{Eval}\ env\ code\ ((p\ x)\ f)) \ = \ \mathsf{Eval}\ env\ code\ ((\mathsf{A}x.\ p\ x)\ f) \tag{12}$$

Once we have introduced A, we can push it through $\rightarrow$ using the following two rewrites:

$$\mathsf{A}x.\ (a \rightarrow p\ x) \ = \ (a \rightarrow (\mathsf{A}x.\ p\ x)) \tag{13}$$

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

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:

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

The proofs of rewrites (12) and (13) 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.

$$(\forall x.\ \exists r.\ (\langle s, env\rangle \vdash code \Downarrow r) \wedge \ldots) \ = \ (\exists r.\ (\langle s, env\rangle \vdash code \Downarrow r) \wedge \forall x.\ \ldots)$$

Such equations can be proved if we assume that $\Downarrow$ is deterministic since then there is only one r that can be chosen by the existential quantifier. Note that the definition of Eval in Section 4.1 would not have had its intended meaning if the operational semantics had been genuinely non-deterministic.

4.4 Supporting ML-like features

The previous sections explained how terms and functions can be translated with proof into CakeML. All examples kept to simple types, such as *bool* and *int*, and functions among such types. The following subsections explain how we translate a richer ML-like subset of HOL: a subset with user-defined datatypes, partially specified functions and terms involving equality tests.

4.4.1 Datatypes and pattern matching

We start with a look at the datatypes. HOL provides ways of defining ML-like datatypes, e.g. the usual *list* datatype can be defined as follows:

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

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 $\alpha\ list$, a new refinement invariant list is defined which

takes a refinement invariant a as an argument. The definition of list can be automatically produced from the datatype definition. Here Conv is a constructor-value.

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

Based on this definition we automatically derive lemmas that aid translation of *list*-constructor applications in HOL.

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

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

We also derive lemmas which aid translation of pattern matching over these HOL constructors. HOL functions that have pattern matching at the top-level tend to be defined as using multiple equations. For example, the map function is typically defined as follows:

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

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

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

It is these single-line definitions that we translate into CakeML functions. By making sure translations are always performed only on these collapsed single-line definitions, it is sufficient to add support for translation 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 in terms of 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 translations of

let-expressions and if-statements.

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

Each time a new datatype is encountered our implementation (Section 6) automatically defines the refinement invariant and proves lemmas for translation of constructors and pattern matching. The implementation also attempts to prove lemmas that aid reasoning about the EqualityType predicate (see Section 4.4.3).

4.4.2 Partial functions and under-specification

The use of pattern matching leads to partiality.[5] 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

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

using the special ARB[6] constant in HOL, which cannot be related to any specific CakeML value because one cannot prove anything in HOL about ARB.

When translating a partially specified definition into CakeML, we can only prove a connection between CakeML and HOL for certain well-defined input values. For this purpose we use eq, which was defined in Section 4.2.1, to restrict the possible input values. The theorem that relates hd to its CakeML counterpart includes a side-condition $xs \neq []$ on the input, which is applied via eq:

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

The generated CakeML code includes `raise Error`[7] in the places where the translation is disconnected from the HOL function.

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

[5] All functions in HOL are total. However, their definitions can omit cases causing their equational specification to appear partial, i.e. they are actually only *partially specified*.

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

[7] We could equally well have generated any other CakeML expression at this point.

When translating ARB into CakeML, we use a trivially true lemma with false as the assumption on the left-hand side of the implication.

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

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

Translation of recursive, partially specified 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 (see Section 4.3.3).

4.4.3 Equality types

There is another source of partiality: equality tests. CakeML and HOL have different semantics regarding equality. In CakeML, 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, defined below, on the refinement invariant a for the values that are tested.

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

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.

A refinement invariant a supports equality if the corresponding CakeML value cannot contain a closure and testing for structural equality of CakeML values is equivalent to testing equality at the abstract level:

$$\begin{array}{l} \mathsf{EqualityType}\ a\ = \\ \quad (\forall x\ v.\ \ a\ x\ v \Longrightarrow \neg(\mathsf{contains\_closure}\ v)) \wedge \\ \quad (\forall x\ v\ y\ w.\ \ a\ x\ v \wedge a\ y\ w \Longrightarrow (v = w \iff x = y)) \end{array}$$

Previously mentioned refinement invariants bool and int satisfy EqualityType.

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

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

Such lemmas cannot always be proved, e.g. if the datatype contains a function type or is a more abstract user-defined extension as outlined in the next section.

4.4.4 User-defined extensions

Our approach to supporting user-defined datatypes in Section 4.4.1 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 CakeML:

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

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` respectively.

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

$$\mathsf{Eval}\ env\ \lfloor \texttt{x} \rfloor\ ((\mathsf{set}\ a)\ x) \wedge \mathsf{Eval}\ env\ \lfloor \texttt{y} \rfloor\ ((\mathsf{set}\ a)\ y) \Longrightarrow \mathsf{Eval}\ env\ \lfloor \texttt{append x y} \rfloor\ ((\mathsf{set}\ a)\ (x \cup y))$$

$$\mathsf{Eval}\ env\ \lfloor \texttt{r} \rfloor\ (a\ r) \wedge \mathsf{Eval}\ env\ \lfloor \texttt{x} \rfloor\ ((\mathsf{set}\ a)\ x) \wedge \mathsf{EqualityType}\ a \Longrightarrow \mathsf{Eval}\ env\ \lfloor \texttt{mem r x} \rfloor\ (\mathsf{bool}\ (r \in x))$$

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 (Okasaki, 1998).

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

5 Synthesis of stateful ML

The previous section explained how functions from HOL can be translated into pure functions in CakeML. Functions in HOL have no side effects and as such the translation into pure CakeML is the most natural target. However, ML code is sometimes significantly more efficient if references and exceptions are used.

This section presents an experiment where we translate HOL functions into *stateful* CakeML that makes use of references and exceptions. In order to translate into stateful CakeML, we require the HOL functions to be written in terms of a state-and-exception monad. Our experiment is motivated by a case study which aims to produce a verified CakeML implementation of the HOL light theorem prover (Myreen *et al.*, 2013).

As part of this case study, the entire HOL light kernel has been defined in HOL as monadic functions and we have successfully translated all of these functions into stateful CakeML using the technique outlined in the following subsections.

5.1 A state-and-exception monad

The HOL light case study uses a state-and-exception monad of the following type:

$$\alpha\ M\ =\ hol\_refs \rightarrow \alpha\ hol\_result \times hol\_refs$$

where *hol_refs* is a record type: the record holds the state of the HOL light kernel, i.e. lists of that keep track of defined types, constants, axioms etc.

$$\begin{array}{lcll} \mathit{hol\_refs} & = & \langle| & \mathsf{the\_type\_constants} : (\mathit{string} \times \mathit{num})\ \mathit{list}\ ; \\ & & & \mathsf{the\_term\_constants} : (\mathit{string} \times \mathit{hol\_type})\ \mathit{list}\ ; \\ & & & \mathsf{the\_axioms} : \mathit{thm}\ \mathit{list}\ ; \\ & & & \mathsf{the\_definitions} : \mathit{def}\ \mathit{list}\ ; \\ & & & \mathsf{the\_clash\_var} : \mathit{hol\_term} \quad |\rangle \end{array}$$

Using this state-and-exception monad, each computation results in something of type *hol_result*, i.e. either a result or an error message.

$$\alpha\ \mathit{hol\_result} = \mathsf{HolRes}\ \alpha \mid \mathsf{HolErr}\ \mathit{string}$$

Based on these types, we define the obvious functions for working with the monad. The definition of bind, return, failwith and an exception catching function otherwise are shown below. There are also functions for accessing and updating the state, e.g. get_the_axioms and set_the_axioms access one of the state components (accessing a field of the record).

$$\begin{array}{l} \mathsf{bind}\ (x : \alpha\ M)\ (f : \alpha \to \beta\ M)\ \mathit{state} = \\ \quad \mathsf{case}\ x\ \mathit{state}\ \mathsf{of} \\ \quad\quad (\mathsf{HolRes}\ y, \mathit{state}) \Rightarrow f\ y\ \mathit{state} \\ \quad \mid (\mathsf{HolErr}\ e, \mathit{state}) \Rightarrow (\mathsf{HolErr}\ e, \mathit{state}) \end{array}$$

$$\begin{array}{l} \mathsf{otherwise}\ (x : \alpha\ M)\ (y : \alpha\ M)\ \mathit{state} = \\ \quad \mathsf{case}\ x\ \mathit{state}\ \mathsf{of} \\ \quad\quad (\mathsf{HolRes}\ y, \mathit{state}) \Rightarrow (\mathsf{HolRes}\ y, \mathit{state}) \\ \quad \mid (\mathsf{HolErr}\ e, \mathit{state}) \Rightarrow y\ \mathit{state} \end{array}$$

$$\begin{array}{l} \mathsf{return}\ (x : \alpha)\ \mathit{state} = \\ \quad (\mathsf{HolRes}\ x, \mathit{state}) \end{array}$$

$$\begin{array}{l} \mathsf{get\_the\_axioms}\ \mathit{state} = \\ \quad (\mathsf{HolRes}\ (\mathit{state}.\mathsf{the\_axioms}), \mathit{state}) \end{array}$$

$$\begin{array}{l} \mathsf{failwith}\ (e : \mathit{string})\ \mathit{state} = \\ \quad (\mathsf{HolErr}\ e, \mathit{state}) \end{array}$$

$$\begin{array}{l} \mathsf{set\_the\_axioms}\ x\ \mathit{state} = \\ \quad (\mathsf{HolRes}\ (), \mathit{state}[\mathsf{the\_axioms} \mapsto x]) \end{array}$$

Using these operations and an appropriate do-notation, it is easy to define the HOL light kernel as functions in HOL. One of the simplest functions is the definition of map given in Section 2.3.

5.2 Core definitions for translation of monadic functions

When stateful ML is to be generated, the Eval predicate used for pure CakeML is insufficient since it cannot refer to the state. We define a version of Eval, called EvalM, that can make statements about the reference store and exceptions. We define EvalM using an invariant hol_store that specifies the relationship between CakeML's reference store and the HOL representation of HOL light's state, i.e. a record of type *hol_refs*.

$$\begin{array}{l} \mathsf{hol\_store}\ \mathit{refs}\ s = \\ \quad \mathsf{length}\ s \geqslant 5 \wedge \\ \quad \mathsf{list}\ (\mathsf{pair\ string\ num})\ \mathit{refs}.\mathsf{the\_type\_constants}\ (\mathsf{store\_lookup}\ 0\ s) \wedge \\ \quad \mathsf{list}\ (\mathsf{pair\ string\ hol\_type})\ \mathit{refs}.\mathsf{the\_term\_constants}\ (\mathsf{store\_lookup}\ 1\ s) \wedge \\ \quad \mathsf{list\ thm}\ \mathit{refs}.\mathsf{the\_axioms}\ (\mathsf{store\_lookup}\ 2\ s) \wedge \\ \quad \mathsf{list\ def}\ \mathit{refs}.\mathsf{the\_definitions}\ (\mathsf{store\_lookup}\ 3\ s) \wedge \\ \quad \mathsf{hol\_term}\ \mathit{refs}.\mathsf{the\_clash\_var}\ (\mathsf{store\_lookup}\ 4\ s) \end{array}$$

We define EvalM *env exp post* to say that any evaluation from a reference store s that satisfies the hol_store invariant must evaluate to a new store s_2 and result *res* such that *post* relates the initial store s and state *refs* to the result value *res*, store s_2 and state $refs_2$.

$$\begin{aligned}\mathsf{EvalM}\ env\ exp\ post\ =\ & \forall s\ refs.\\ & \quad \mathsf{hol\_store}\ refs\ s\implies\\ & \quad \exists s_2\ res\ refs_2.\ \langle s, env\rangle \vdash exp \Downarrow \langle s_2, res\rangle \wedge\\ & \qquad post\ (refs, s)\ (refs_2, s_2, res)\ \wedge\ \mathsf{hol\_store}\ refs_2\ s_2\end{aligned}$$

Again, the interesting part is how *post* is instantiated with refinement invariants. We define M to lift refinement invariants used for Eval to refinement invariants that fit EvalM. Given a refinement invariant a for use with Eval, M a is the equivalent refinement invariant for EvalM. At present, the error message from the HOL level is not represented in the generated CakeML, just to keep things simple.

$$\begin{aligned}&\mathsf{M}\ (a:\alpha\to v\to bool)\ (x:\alpha\ M)\ (refs_1, state_1)\ (refs_2, state_2, res)\ =\\ &\quad \mathsf{case}\ (x\ refs_1, res)\ \mathsf{of}\\ &\qquad ((\mathsf{HolRes}\ y, refs), \mathsf{Rval}\ v)\Rightarrow (refs = refs_2)\ \wedge\ a\ y\ v\\ &\quad \mid ((\mathsf{HolErr}\ e, refs), \mathsf{Rerr}\ s)\Rightarrow (refs = refs_2)\\ &\quad \mid \_\ \Rightarrow \mathsf{false}\end{aligned}$$

Using M, we can state that a monadic computation f computes a list and accesses the state in the same manner as evaluation of a CakeML expression *exp*:

$$\mathsf{EvalM}\ env\ exp\ (\mathsf{M}\ (\mathsf{list}\ a)\ f)$$

Just as there is the $\to$ refinement invariant for Eval, a similar refinement invariant combinator, $\to^{\mathsf{M}}$, for state-updating closures, is required for EvalM. We define this as follows. Here evaluate_stateful_closure $(s_2, v)\ c\ (s_3, res_3)$ refers to the CakeML evaluation of closures: it is true if closure c on input v in state s_2 evaluates to a new state s_3 and result res_3. Its formal definition is omitted.

$$\begin{aligned}&\mathsf{pure}\ a\ x\ (refs_1, s1)\ (refs_2, s_2, res)\ =\\ &\quad \exists v.\ (res = \mathsf{Rval}\ v)\ \wedge\ (refs_1 = refs_2)\ \wedge\ (s_1 = s_2)\ \wedge\ a\ x\ v\end{aligned}$$

$$\begin{aligned}&(a\to^{\mathsf{P}} b)\ f\ c\ =\\ &\quad \forall x\ refs_1\ s_1\ refs_2\ s_2\ res.\\ &\qquad a\ x\ (refs_1, s_1)\ (refs_2, s_2, res)\ \wedge\ \mathsf{hol\_store}\ refs_1\ s_1\implies\\ &\qquad (refs_2 = refs_1)\ \wedge\ (s_2 = s_1)\ \wedge\\ &\qquad \exists v\ s_3\ res_3\ refs_3.\\ &\qquad\quad (res = \mathsf{Rval}\ v)\ \wedge\ \mathsf{evaluate\_stateful\_closure}\ (s_2, v)\ c\ (s_3, res_3)\ \wedge\\ &\qquad\quad b\ (f\ x)\ (refs_1, s_1)\ (refs_3, s_3, res_3)\ \wedge\ \mathsf{hol\_store}\ refs_3\ s_3\end{aligned}$$

$$a\to^{\mathsf{M}} b\ =\ \mathsf{pure}\ (a\to^{\mathsf{P}} b)$$

Using $\to^{\mathsf{M}}$, we can now state the theorem that translation of the stateful map functions from above is to prove. Here map has type $(\alpha\to\beta\ M)\to\alpha\ list\to(\beta\ list)\ M$.

$$\ldots\implies \mathsf{EvalM}\ env\ \lfloor\texttt{map}\rfloor\ (((\mathsf{pure}\ a\to^{\mathsf{M}}\mathsf{M}\ b)\to^{\mathsf{M}}\mathsf{pure}\ (\mathsf{list}\ a)\to^{\mathsf{M}}\mathsf{M}\ (\mathsf{list}\ b))\ \mathsf{map})$$

5.3 Lemmas that aid translation of monadic functions

Translation of monadic functions into stateful CakeML is performed in very much the same way as the translation into pure CakeML, as described in Section 4. The difference is that theorems are expressed in terms of EvalM instead of Eval, $\rightarrow^{\mathsf{M}}$ instead of $\rightarrow$, etc. This section presents some of the key lemmas about EvalM that complement or replace the lemmas about Eval used in Section 4.

The most basic lemma is the following which allows use of pure terms as part of monadic functions (in this case using return):

$$\begin{array}{l}\mathsf{Eval}\ env\ \lfloor exp \rfloor\ (a\ x) \Longrightarrow \\ \mathsf{EvalM}\ env\ \lfloor exp \rfloor\ ((\mathsf{M}\ a)\ (\mathsf{return}\ x))\end{array}$$

We translate bind into CakeML's let expressions using the following lemma:

$$\begin{array}{l}\mathsf{EvalM}\ env\ \lfloor exp_1 \rfloor\ ((\mathsf{M}\ b)\ y) \wedge \\ (\forall x\ v.\ b\ x\ v \implies \mathsf{EvalM}\ (env[n \mapsto v])\ \lfloor exp_2 \rfloor((\mathsf{M}\ a)\ (f\ x))) \Longrightarrow \\ \mathsf{EvalM}\ env\ \lfloor \texttt{let val}\ n\ \texttt{=}\ exp_1\ \texttt{in}\ exp_2\ \texttt{end} \rfloor\ ((\mathsf{M}\ a)\ (\mathsf{bind}\ y\ f))\end{array}$$

Raising an exception is done with failwith and exceptions can be caught using otherwise:

$$\mathsf{EvalM}\ env\ \lfloor \texttt{raise 0} \rfloor\ ((\mathsf{M}\ a)\ (\mathsf{failwith}\ \ message))$$

$$\begin{array}{l}\mathsf{EvalM}\ env\ \lfloor exp_1 \rfloor\ ((\mathsf{M}\ a)\ x) \wedge \\ \mathsf{EvalM}\ env\ \lfloor exp_2 \rfloor\ ((\mathsf{M}\ a)\ y) \Longrightarrow \\ \mathsf{EvalM}\ env\ \lfloor exp_1\ \texttt{handle \_ =>}\ exp_2 \rfloor\ ((\mathsf{M}\ a)\ (\mathsf{otherwise}\ x\ y))\end{array}$$

We also have lemmas that can be used for reasoning about monadic function values. A monadic arrow $\rightarrow^{\mathsf{M}}$ can be applied to any expression with a matching refinement invariant:

$$\begin{array}{l}\mathsf{EvalM}\ env\ \lfloor \texttt{f} \rfloor\ ((a \rightarrow^{\mathsf{M}} b)\ f) \wedge \\ \mathsf{EvalM}\ env\ \lfloor \texttt{x} \rfloor\ (a\ x) \Longrightarrow \\ \mathsf{EvalM}\ env\ \lfloor \texttt{f x} \rfloor\ (b\ (f\ x))\end{array}$$

Similarly, $\rightarrow^{\mathsf{M}}$ can be introduced with either of the following lemmas:

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

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

Using the lemmas shown above and other similar results for EvalM, it is straightforward to perform the style of proofs explained in Section 4 for translations that produce stateful CakeML from monadic functions. The overall algorithm, as explained in Section 4.3.2, is essentially unchanged once lemmas about Eval have been swapped for corresponding lemmas about EvalM.

6 Implementation and case studies

We have implemented our translation procedure for the HOL4 theorem prover.[8] The implementation is an ML program which performs the proof steps outlined above. Concretely, the ML program constructs elements of type `thm` (theorem) using the logical kernel's primitives (which correspond to axioms and inference rules of HOL). Following the LCF-approach, this design ensures that all proved theorems are the result of the basic inference rules of HOL.

Our implementation produces a certificate of correctness *if* the translation attempt succeeds. It is therefore useful to test the robustness of our implementation. The following list describes examples that our tool successfully translates. The examples consist of a variety of ML-like functions living within the collection of sample theories that are bundled with the HOL4 prover. Our implementation has successfully translated a range of different functions into pure CakeML:

- Miller-Rabin primality test (Hurd, 2003):
  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 (Barthwal & Norrish, 2009):
  This is a 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 (Duan *et al.*, 2005):
  These algorithms operate on fixed-size word values, which we support through the technique for user-defined extensions (Section 4.4.4). We represent fixed-size words as integers in CakeML and use a refinement invariant to make sure the correspondence is maintained.
- McCarthy's 91 function, quicksort (Slind, 1999) and a regular expression matching function (Owens & Slind, 2008):
  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 (Myreen, 2010):
  This is a model of Cheney's algorithm for copying garbage collection—a verified algorithm used in constructing a verified Lisp runtime (Myreen & Davis, 2011). It models memory as a mapping from natural numbers to a datatype of abstract memory values.
- Functional data structures from Okasaki's book (Okasaki, 1998):
  1. heap datatypes: leftist, pairing, lazy, splay, binomial
  2. set datatypes: unbalanced, red-black
  3. sorting algorithms: merge sort
  4. list datatypes: binary random-access lists
  5. queues datatypes: batched, bankers, physicists, real-time, implicit, Hood-Melville

[8] Source code and examples are available at: `https://cakeml.org/`

- CakeML parser, elaborator, type inferencer and compiler (Kumar *et al.*, 2014): These functions constitute the core part of a verified implementation of CakeML (part *B*). This example is much larger than the rest: it consists of approximately 600 functions, some of which are very large.

All algorithms, except those from Okasaki's book and the last bullet point, 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 (2010) 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). Our verification proofs are similar in length to Charguéraud's. 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.

As mentioned in Section 5, we have used the translation into stateful CakeML on a sizeable case study: a definition of the HOL light theorem prover's logical kernel expressed as monadic functions in HOL. This has been translated into CakeML code that uses references and exceptions in the same manner as the original HOL light implementation. The definition of the HOL light kernel is approximately 500 lines long when expressed in terms of the state-exception monad that we use.

7 Discussion of related work

There is a long tradition in interactive theorem proving of using logics that look like functional programming languages: notable examples include LCF (Milner, 1972), the Boyer-Moore prover (Boyer & Moore, 1975), the Calculus of Constructions (Coquand & Huet, 1988) and TFL (Slind, 1999; Krauss, 2009). The logic of the Boyer–Moore prover (and its successor, ACL2 (**?**)) are actual programming languages with standard denotational or operational semantics. However, many other systems, including Coq (`http://coq.inria.fr/`) and various HOL systems (Norrish & Slind, 2002) (including Isabelle/HOL (`http://www.cl.cam.ac.uk/research/hvg/isabelle/`) and HOL4 (`http://hol.sourceforge.net/`)), 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 (Slind, 1999) and Krauss' extensions (Krauss, 2009) make HOL (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 simplify pattern matching expressions (Section 4.4.1).

Extraction from Coq (Letouzey, 2003) has two phases. First, purely logical content (e.g. proofs about the definitions) is removed from the definitions to be extracted, then the remaining computational content 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 (Davis & Myreen, 2012; Myreen, 2012).

Proof-producing synthesis has previously been used in HOL for various low-level targets including hardware (Slind *et al.*, 2007) and assembly-like languages (Li & Slind, 2007, 2008; Li *et al.*, 2007). 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 (CakeML'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 (Myreen *et al.*, 2009; Li, 2011) 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 (Charguéraud, 2010), and it can support imperative features (Charguéraud, 2011). 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, require the ability to handle partial functions and side effects. In contrast, this paper is about generating, from pure functional specifications, CakeML 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 ASTs. 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 CakeML; 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 4.3.4), but most languages that do not define evaluation order (e.g. Scheme, OCaml) should be able to support a deterministic semantics for, at least, the pure, total subset.

Our technique should also extend to other provers, including Isabelle/HOL and possibly even Coq. For function definitions that are in the ML-like fragment (i.e. they 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 implementation of the automation will vary as certain provers make it inconvenient to program extensions to the prover in the implementation language of the prover. The HOL4 theorem prover has a design which encourages programming such extensions.

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 HOL 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 automatic 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

The first author was funded by the Royal Society, UK. Many thanks to the JFP and ICFP reviewers for their detailed and helpful comments.

References

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. & Zdancewic, S. (2005) Mechanized metatheory for the masses: The PoplMark challenge. In *Theorem Proving in Higher Order Logics (TPHOLs)*, Hurd, J. & Melham, T. F. (eds). Berlin: Springer, pp. 50–65.

Barthe, G., Demange, D. & Pichardie, D. (2012) A formally verified SSA-based middle-end: Static single assignment meets CompCert. In *Proceedings of European Symposium on Programming (ESOP'12*), Seidl, H. (ed), vol. 7211. Berlin: Springer, pp. 47–66.

Barthwal, A. & Norrish, M. (2009) Verified, executable parsing. In *Proceedings of European Symposium on Programming (ESOP'09)*, Castagna, G. (ed), vol. 5502. Berlin: Springer, pp.160–174.

Boyer, R. S. & Moore, J. S. (1975) Proving theorems about LISP functions. *J. Assoc. Comput. Mach.* **22**(1), 129–144.

Charguéraud, A. (2010) Program verification through characteristic formulae. In *Proceedings of International Conference on Functional Programming (ICFP'10)*. New York: ACM, 321–332.

Charguéraud, A. (2011) Characteristic formulae for the verification of imperative programs. In *Proceedings of International Conference on Functional Programming (ICFP'11)*. New York: ACM, pp. 418–430.

Chlipala, A. (2010) A verified compiler for an impure functional language. In *Proceedings of Principles of Programming Languages (POPL'10)*. New York: ACM, pp. 93–106.

Coquand, T. & Huet, G. (1988) The calculus of constructions. *Inf. Comput.* **76**(2–3), 95–120.

Dargaye, Z. (2009) *Vèrification formelle d'un compilateur pour langages fonctionnels*. Paris: Universitè Paris 7 Diderot.

Davis, J. & Myreen, M. O. (2012) *The Self-Verifying Milawa Theorem Prover is Sound (Down to the Machine Code that Runs it)*. Available at: `http://www.cl.cam.ac.uk/~mom22/jitawa/` Accessed Nov 1, 2013.

Duan, J., Hurd, J., Li, G., Owens, S., Slind, K. & Zhang, J. (2005) Functional correctness proofs of encryption algorithms. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR),* Sutcliffe, G. & Voronkov, A. (eds). Berlin: Springer-Verlag, pp. 519–533.

Harrison, J. (1995) *Metatheory and Reflection in Theorem Proving: A Survey and Critique*, Technical Report CRC-053. Cambridge, UK: SRI Cambridge.

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

Krauss, A. (2009) *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. Munich: Technische Universitiät München.

Kumar, R., Myreen, M. O., Norrish, M. & Owens, S. (2014) CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, Sewell, P. (ed). ACM.

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

Letouzey, P. (2003) A new extraction for Coq. In *Types for Proofs and Programs (TYPES)*. Berlin: Springer, pp. 200–219.

Li, G. (2011) Validated compilation through logic. In *Formal Methods (FM)*, Butler, M. & Schulte, W. (eds), vol. 6664. Berlin: Springer, pp. 169–183.

Li, G., Owens, S. & Slind, K. (2007) Structure of a proof-producing compiler for a subset of higher order logic. In *European Symposium on Programming (ESOP)*, Nicola, R. De (ed). Berlin: Springer, pp. 205–219.

Li, G. & Slind, K. (2007) Compilation as rewriting in higher order logic. In *Automated Deduction (CADE)*, Pfenning, F. (ed), vol. 4603. Berlin: Springer, pp. 19–34.

Li, G. & Slind, K. (2008) Trusted source translation of a total function language. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Ramakrishnan, C. R. & Rehof, J. (eds), vol. 4963. Berlin: Springer, pp. 471–485.

Malecha, J. G., Morrisett, G., Shinnar, A. & Wisnesky, R. (2010) Toward a verified relational database management system. In *Proceedings of Principles of Programming Languages (POPL'10)*. New York: ACM, pp. 237–248.

McCreight, A., Chevalier, T. & Tolmach, A. P. (2010) A certified framework for compiling and executing garbage-collected languages. In *Proceedings of International Conference on Functional Programming (ICFP'10)*. New York: ACM, pp. 273–284.

Milner, R. (1972) *Logic for Computable Functions: Description of a Machine Implementation*, Technical Report STAN-CS-72-288, A.I. Memo 169. Stanford University.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML (Revised)*. Cambridge, MA: The MIT Press.

Myreen, M. O. (2010) Reusable verification of a copying collector. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, Leavens, G. T., O'Hearn, P. W. & Rajamani, S. K. (eds). Berlin: Springer, pp. 142–156.

Myreen, M. O. (2012) Functional programs: conversions between deep and shallow embeddings. In *Interactive Theorem Proving (ITP)*, Beringer, L. & Felty, A. (eds), vol. 7406. Berlin: Springer, pp. 412–417.

Myreen, M. O. & Curello, G. (2013) Proof pearl: A verified bignum implementation in x86-64 machine code. In *Certified Programs and Proofs (CPP)*. Cham, Switzerland: Springer, pp. 66–81.

Myreen, M. O. & Davis, J. (2011) A verified runtime for a verified theorem prover. In *Interactive Theorem Proving (ITP)*, van Eekelen, M. C. J. D., Geuvers, H., Schmaltz, J. & Wiedijk, F. (eds), vol. 6898. Berlin: Springer, pp. 265–280.

Myreen, M. O. & Owens, S. (2012) Proof-producing synthesis of ML from higher-order logic. In *Proceedings of International Conference on Functional Programming (ICFP'12)*. New York: ACM, pp. 115–126.

Myreen, M. O., Owens, S., & Kumar, R. (2013) Steps towards verified implementations of HOL Light. In *Interactive Theorem Proving (ITP)*, Blazy, S., Paulin-Mohring, C., & Pichardie, D. (eds). Berlin: Springer, pp. 490–495.

Myreen, M. O., Slind, K. & Gordon, M. J. C. (2009) Extensible proof-producing compilation. In *Compiler Construction (CC)*, de Moor, O. & Schwartzbach, M. I. (eds). Berlin: Springer, pp. 2–16.

Norrish, M. & Slind, K. (2002) A thread of HOL development. *Comput. J.* **45**(1), 37–45.

Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge, UK: Cambridge University.

Owens, S. & Slind, K. (2008) Adapting functional programs to higher-order logic. *Higher-order Symb. Comput.* **21**(4), 377–409.

Ševčík , J., Vafeiadis, V., Nardelli, F. Z., Jagannathan, S. & Sewell, P. (2011) Relaxed-memory concurrency and verified compilation. In *Proceedings of Principles of Programming Languages (POPL'11)*. New York: ACM, pp. 43–54.

Slind, K. (1999) *Reasoning about Terminating Functional Programs*, PhD Thesis. Technical University of Munich.

Slind, K., Owens, S., Iyoda, J. & Gordon, M. (2007) Proof producing synthesis of arithmetic and cryptographic hardware. *Form. Asp. Comput.* **19**(3), 343–362.