Proof-Producing Synthesis of CakeML from Monadic HOL Functions

Journal of Automated Reasoning
https://doi.org/10.1007/s10817-020-09559-8

Proof-Producing Synthesis of CakeML from Monadic HOL Functions

Oskar Abrahamsson[1] · Son Ho[2] · Hrutvik Kanabar[3] · Ramana Kumar[4] · Magnus O. Myreen[1] · Michael Norrish[5] · Yong Kiam Tan[6]

Received: 21 April 2020 / Accepted: 24 April 2020

Abstract
We introduce an automatic method for producing stateful ML programs together with proofs of correctness from monadic functions in HOL. Our mechanism supports references, exceptions, and I/O operations, and can generate functions manipulating local state, which can then be encapsulated for use in a pure context. We apply this approach to several non-trivial examples, including the instruction encoder and register allocator of the otherwise pure CakeML compiler, which now benefits from better runtime performance. This development has been carried out in the HOL4 theorem prover.

Keywords Interactive theorem proving · Program synthesis · ML · Higher-order logic

1 Introduction

This paper is about bridging the gap between programs verified in logic and verified implementations of those programs in a programming language (and ultimately machine code). As a toy example, consider computing the nth Fibonacci number. The following is a recursion equation for a function, fib, in higher-order logic (HOL) that does the job:

$$\mathsf{fib}\ n \stackrel{\mathsf{def}}{=} \mathsf{if}\ n < 2\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{fib}\ (n-1) + \mathsf{fib}\ (n-2)$$

A hand-written implementation (shown here in CakeML [10], which has similar syntax and semantics to Standard ML) would look something like this:

✉ Oskar Abrahamsson
aboskar@chalmers.se

1 Chalmers University of Technology, Göteborg, Sweden
2 MINES ParisTech, PSL Research University, Paris, France
3 University of Kent, Canterbury, UK
4 DeepMind, London, UK
5 Data61, CSIRO/ANU, Canberra, Australia
6 Carnegie Mellon University, Pittsburgh, USA

Published online: 06 June 2020

```
fun fiba i j n = if n = 0 then i else fiba j (i+j) (n-1);
(print (n2s (fiba 0 1 (s2n (hd (CommandLine.arguments())))));
 print "\n")
handle _ => print_err ("usage: " ^ CommandLine.name() ^ " <n>\n");
```

In moving from mathematics to a real implementation, some issues are apparent:

(i) We use a tail-recursive linear-time algorithm, rather than the exponential-time recursion equation.
(ii) The whole program is not a pure function: it does I/O, reading its argument from the command line and printing the answer to standard output.
(iii) We use exception handling to deal with malformed inputs (if the arguments do not start with a string representing a natural number, `hd` or `s2n` may raise an exception).

The first of these issues (i) can easily be handled in the realm of logical functions. We define a tail-recursive version in logic:

$$\mathsf{fiba}\ i\ j\ n \stackrel{\mathsf{def}}{=} \mathsf{if}\ n = 0\ \mathsf{then}\ i\ \mathsf{else}\ \mathsf{fiba}\ j\ (i + j)\ (n - 1)$$

then produce a correctness theorem, $\vdash \forall n.\ \mathsf{fiba}\ 0\ 1\ n = \mathsf{fib}\ n$, with a simple inductive proof (a 5-line tactic proof in HOL4, not shown).

Now, because fiba is a logical function with an obvious computational counterpart, we can use proof-producing synthesis techniques [14] to automatically synthesise code verified to compute it. We thereby produce something like the first line of the CakeML code above, along with a theorem relating the semantics of the synthesised code back to the function in logic.

But when it comes to handling the other two issues, (ii) and (iii), and producing and verifying the remaining three lines of CakeML code, our options are less straightforward. The first issue was easy because we were working with a *shallow embedding*, where one writes the program as a function in logic and proves properties about that function directly. Shallow embeddings rely on an analogy between mathematical functions and procedures in a pure functional programming language. However, effects like state, I/O, and exceptions, can stretch this analogy too far. The alternative is a *deep embedding*: one writes the program as an input to a formal semantics, which can accurately model computational effects, and proves properties about its execution under those semantics.

Proofs about shallow embeddings are relatively easy since they are in the native language of the theorem prover, whereas proofs about deep embeddings are filled with tedious details because of the indirection through an explicit semantics. Still, the explicit semantics make deep embeddings more realistic. An intermediate option that is suitable for the effects we are interested in—state/references, exceptions, and I/O—is to use *monadic functions*: one writes (shallow) functions that represent computations, aided by a composition operator (monadic bind) for stitching together effects. The monadic approach to writing effectful code in a pure language may be familiar from the Haskell language which made it popular.

For our nth Fibonacci example, we can model the effects of the whole program with a monadic function, fibm, that calls the pure function fiba to do the calculation. Figure 1 shows how fibm can be written using do-notation familiar from Haskell. This is as close as we can get to capturing the effectful behaviour of the desired CakeML program while remaining in a shallow embedding. Now how can we produce real code along with a proof that it has the correct semantics? If we use the proof-producing synthesis techniques mentioned above [14], we produce *pure* CakeML code that exposes the monadic plumbing in an explicit state-passing style. But we would prefer verified *effectful* code that uses native features of the target language (CakeML) to implement the monadic effects.

Fig. 1 The Fibonacci program written using do-notation in logic

```
fibm () ≝
 do
  args ← commandline (arguments ());
  a ← hd args;
  n ← s2n a;
  stdio (print (n2s (fiba 0 1 n)));
  stdio (print "\n")
 od otherwise
 do
  name ← commandline (name ());
  stdio (print_err ("usage: " ^ name ^ " <n>\n"))
 od
```

In this paper, we present an automated technique for producing verified effectful code that handles I/O, exceptions, and other issues arising in the move from mathematics to real implementations. Our technique systematically establishes a connection between shallowly embedded functions in HOL with monadic effects and deeply embedded programs in the impure functional language CakeML. The synthesised code is efficient insofar as it uses the native effects of the target language and is close to what a real implementer would write. For example, given the monadic fibm function above, our technique produces essentially the same CakeML program as on the first page (but with a `let` for every monad bind), together with a proof that the synthesised program is a refinement.

Contributions. Our technique for producing verified effectful code from monadic functions builds on a previous limited approach [14]. The new generalised method adds support for the following features:

- global references and exceptions (as before, but generalised),
- mutable arrays (both fixed and variable size),
- input/output (I/O) effects,
- local mutable arrays and references, which can be integrated seamlessly with code synthesis for otherwise pure functions,
- composable effects, whereby different state and exception monads can be combined using a lifting operator, and,
- support for recursive programs where termination depends on monadic state.

As a result, we can now write *whole programs* as shallow embeddings and obtain real verified code via synthesis. Prior to this work, whole program verification in CakeML involved manual deep embedding proofs for (at the very least) the I/O wrapper. To exercise our toolchain, we apply it to several examples:

- the nth Fibonacci example already seen (exceptions, I/O)
- the Floyd Warshall algorithm for finding shortest paths (arrays)
- an in-place quicksort algorithm (polymorphic local arrays, exceptions)
- the instruction encoder in the CakeML compiler's assembler (local arrays)
- the CakeML compiler's register allocator (local refs, arrays)
- the Candle theorem prover's kernel [9] (global refs, exceptions)
- an OpenTheory [8] article checker (global refs, exceptions, I/O)

In Sect. 6, we compare runtimes with the previous non-stateful versions of CakeML's register allocator and instruction encoder; and for the OpenTheory reader we compare the amount of code/proof required before and after using our technique.

The HOL4 development is at https://code.cakeml.org; our new synthesis tool is at https://code.cakeml.org/tree/master/translator/monadic.
Additions. This paper is an extended version of our earlier conference paper [6]. The following contributions are new to this work: a brief discussion of how *polymorphic* functions that use type variables in their local state can be synthesized (Sect. 4), a section on synthesis of recursive programs where termination depends on the monadic state (Sect. 5), and new case studies using our tool, e.g., quicksort with polymorphic local arrays (Sect. 4), and the CakeML compiler's instruction encoder (Sect. 6).

2 High-Level Ideas

This paper combines the following three concepts in order to deliver the contributions listed above. The main ideas will be described briefly in this section, while subsequent sections will provide details. The three concepts are:

(i) synthesis of stateful ML code as described in our previous work [14],
(ii) separation logic [16] as used by characteristic formulae for CakeML [5], and
(iii) a new abstract synthesis mode for the CakeML synthesis tools [14].

Our previous work on proof-producing synthesis of stateful ML (i) was severely limited by the requirement to have a hard-coded invariant on the program's state. There was no support for I/O and all references had to be declared globally. At the time of its development, we did not have a satisfactory way of generalising the hard-coded state invariant.

In this paper we show (in Sect. 3) that the separation logic of CF (ii) can be used to neatly generalise the hard-coded state invariant of our prior work (i). CF-style separation logic easily supports references and arrays, including resizable arrays, and, supports I/O too because it allows us to treat I/O components as if they are heap components. Furthermore, by carefully designing the integration of (i) and (ii), we retain the frame rule from the separation logic. In the context of code synthesis, this frame rule allows us to implement a lifting feature for changing the type of the state-and-exception monads. Being able to change types in the monads allows us to develop *reusable* libraries—e.g. verified file I/O functions—that users can lift into the monad that is appropriate for their application.

The combination of (i) and (ii) does not by itself support synthesis of code with local state due to inherited limitations of (i), wherein the generated code must be produced as a concrete list of global declarations. For example, if monadic functions, say foo and bar, refer to a common reference, say `r`, then `r` must be defined globally:

```
val r = ref 0;
fun foo n = ...; (* code that uses r *)
fun bar n = ...; (* code that uses r and calls foo *)
```

In this paper (in Sect. 4), we introduce a new *abstract* synthesis mode (iii) which removes the requirement of generating code that only consists of a list of global declarations, and, as a result, we are now able to synthesise code such as the following, where the reference `r` is a local variable:

```
fun pure_bar k n =
  let
    val r = ref k
    fun foo n = ... (* code that uses r *)
    fun bar n = ... (* code that uses r and calls foo *)
  in Success (bar n) end
  handle e => Failure e;
```

In the input to the synthesis tool, this declaration and initialisation of local state corresponds to applying the state-and-exception monad. Expressions that fully apply the state-and-exception monad can subsequently be used in the synthesis of *pure* CakeML code: the monadic synthesis tool can prove a pure specification for such expressions, thereby encapsulating the monadic features.

3 Generalised Approach to Synthesis of Stateful ML Code

This section describes how our previous approach to proof-producing synthesis of stateful ML code [14] has been generalised. In particular, we explain how the separation logic from our previous work on characteristic formulae [5] has been used for the generalisation (Sect. 3.3); and how this new approach adds support for user-defined references, fixed- and variable-length arrays, I/O functions (Sect. 3.4), and a handy feature for reusing state-and-exception monads (Sect. 3.5).

In order to make this paper as self-contained as possible, we start with a brief look at how the semantics of CakeML is defined (Sect. 3.1) and how our previous work on synthesis of pure CakeML code works (Sect. 3.2), since the new synthesis method for stateful code is an evolution of the original approach for pure code.

3.1 Preliminaries: CakeML Semantics

The semantics of the CakeML language is defined in the *functional big-step* style [15], which means that the semantics is an interpreter defined as a functional program in the logic of a theorem prover.

The definition of the semantics is layered. At the top-level the semantics function defines what the observable I/O events are for a given whole program. However, more relevant to the presentation in this paper is the next layer down: a function called evaluate that describes exactly how expressions evaluate. The type of the evaluate function is shown below. This function takes as arguments a state (with a type variable for the I/O environment), a value environment, and a list of expressions to evaluate. It returns a new state and a value result.

```
evaluate : δ state →
      v sem_env → exp list → δ state * (v list, v) result
```

The semantics state is defined as the record type below. The fields relevant for this presentation are: refs, clock and ffi. The refs field is a list of store values that acts as a mapping from reference names (list index) to reference and array values (list element). The clock is a logical clock for the functional big-step style. The clock allows us to prove termination of evaluate and is, at the same time, used for reasoning about divergence. Lastly, ffi is the parametrised oracle model of the foreign function interface, i.e. I/O environment.

```
δ state = <| clock : num ; refs : store_v list ; ffi : δ ffi_state ; ... |>
where store_v = Refv v | W8array (word8 list) | Varray (v list)
```

A call to the function evaluate returns one of two results: Rval *res* for successfully terminating computations, and Rerr *err* for stuck computations.

Successful computations, Rval *res*, return a list *res* of CakeML values. CakeML values are modelled in the semantics using a datatype called v. This datatype includes (among other

things) constructors for (mutually recursive) closures (Closure and Recclosure), datatype constructor values (Conv), and literal values (Litv) such as integers, strings, characters etc. These will be explained when needed in the rest of the paper.

Stuck computations, Rerr *err*, carry an error value *err* that is one of the following. For this paper, Rraise *exc* is the most relevant case.

- Rraise *exc* indicates that evaluation results in an uncaught exception *exc*. These exceptions can be caught with a `handle` in CakeML.
- Rabort Rtimeout_error indicates that evaluation of the expression consumes all of the logical clock. Programs that hit this error for all initial values of the clock are considered diverging.
- Rabort Rtype_error, for other kinds of errors, e.g. when evaluating ill-typed expressions, or attempting to access unbound variables.

3.2 Preliminaries: Synthesis of Pure ML Code

Our previous work [14] describes a *proof-producing* algorithm for synthesising CakeML functions from functions in higher-order logic. Here proof-producing means that each execution proves a theorem (called a certificate theorem) guaranteeing correctness of that execution of the algorithm. In our setting, these theorems relate the CakeML semantics of the synthesised code with the given HOL function.

The whole approach is centred around a systematic way of proving theorems relating HOL functions (i.e. HOL terms) with CakeML expressions. In order for us to state relations between HOL terms and CakeML expressions, we need a way to state relations between HOL terms and CakeML values. For this we use relations (int, list $\cdot$, $\cdot \longrightarrow \cdot$, etc.) which we call refinement invariants. The definition of the simple int refinement invariant is shown below: int i v is true if CakeML value v of type v represents the HOL integer i of type `int`.

$$\mathsf{int}\ i \stackrel{\text{def}}{=} (\lambda v.\ v = \mathsf{Litv}\ (\mathsf{IntLit}\ i))$$

Most refinement invariants are more complicated, e.g. list (list int) xs v states that CakeML value v represents lists of int lists xs of HOL type `int list list`.

We now turn to CakeML expressions: we define a predicate called Eval in order to conveniently state relationships between HOL terms and CakeML expressions. The intuition is that Eval env exp P is true if exp evaluates (in environment env) to some result res (of HOL type v) such that P holds for res, i.e. P res. The formal definition below is cluttered by details regarding the clock and references: there must be a large enough clock and exp may allocate new references, $refs'$, but must not modify any existing references, $refs$. We express this restriction on the references using list append $+\!\!+$. Note that any list index that can be looked up in $refs$ has the same look up in $refs +\!\!+ refs'$.

$$\begin{array}{l}\mathsf{Eval}\ env\ exp\ P \stackrel{\text{def}}{=} \\ \quad \forall\, refs. \\ \quad\quad \exists\, res\ refs'. \\ \quad\quad\quad \mathsf{eval\_rel}\ (\mathsf{empty}\ with\ \mathsf{refs}\ :=\ refs)\ env\ exp \\ \quad\quad\quad\quad (\mathsf{empty}\ with\ \mathsf{refs}\ :=\ refs +\!\!+ refs')\ res \wedge P\ res\end{array}$$

The use of Eval and the main idea behind the synthesis algorithm is most conveniently described using an example. The example we consider here is the following HOL function:

$$\mathsf{add1} \stackrel{\mathsf{def}}{=} (\lambda x.\ x + 1)$$

The main part of the synthesis algorithm proceeds as a syntactic bottom-up pass over the given HOL term. In this case, the bottom-up pass traverses HOL term $\lambda x.\ x + 1$. The result of each stage of the pass is a theorem stated in terms of Eval in the format shown below. Such theorems state a connection between a HOL term t and some generated *code* w.r.t. a refinement invariant *ref_inv* that is appropriate for the type of t.

$$\text{general format:} \quad \mathit{assumptions} \Rightarrow \mathsf{Eval}\ \mathit{env}\ \mathit{code}\ (\mathit{ref\_inv}\ t)$$

For our little example, the algorithm derives the following theorems for the subterms x and 1, which are the leaves of the HOL term. Here and elsewhere in this paper, we display CakeML abstract syntax as concrete syntax inside $\lfloor \cdots \rfloor$, i.e. $\lfloor$`1`$\rfloor$ is actually the CakeML expression Lit (IntLit 1) in the theorem prover HOL4; similarly $\lfloor$`x`$\rfloor$ is actually displayed as Var (Short "x") in HOL4. Note that both theorems below are of the required general format.

$$\begin{array}{l} \vdash \mathsf{T} \Rightarrow \mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{1} \rfloor\ (\mathsf{int}\ 1) \\ \vdash \mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{x} \rfloor\ (\mathsf{int}\ x) \Rightarrow \mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{x} \rfloor\ (\mathsf{int}\ x) \end{array} \tag{1}$$

The algorithm uses theorems (1) when proving a theorem for the compound expression $x + 1$. The process is aided by an auxiliary lemma for integer addition, shown below. The synthesis algorithm is supported by several such pre-proved lemmas for various common operations.

$$\begin{array}{l} \vdash \mathsf{Eval}\ \mathit{env}\ x_1\ (\mathsf{int}\ n_1) \Rightarrow \\ \quad \mathsf{Eval}\ \mathit{env}\ x_2\ (\mathsf{int}\ n_2) \Rightarrow \\ \quad \mathsf{Eval}\ \mathit{env}\ \lfloor x_1 + x_2 \rfloor\ (\mathsf{int}\ (n_1 + n_2)) \end{array}$$

By choosing the right specialisations for the variables, x_1, x_2, n_1, n_2, the algorithm derives the following theorem for the body of the running example. Here the assumption on evaluation of $\lfloor$`x`$\rfloor$ was inherited from (1).

$$\vdash \mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{x} \rfloor\ (\mathsf{int}\ x) \Rightarrow \mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{x + 1} \rfloor\ (\mathsf{int}\ (x + 1)) \tag{2}$$

Next, the algorithm needs to introduce the λ-binder in $\lambda x.\ x + 1$. This can be done by instantiation of the following pre-proved lemma. Note that the lemma below introduces a refinement invariant for function types, $\longrightarrow$, which combines refinement invariants for the input and output types of the function [14].

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

An appropriate instantiation and combination with (2) produces the following:

$$\vdash \mathsf{T} \Rightarrow \mathsf{Eval}\ \mathit{env}\ \lfloor \texttt{fn x => x + 1} \rfloor\ ((\mathsf{int} \longrightarrow \mathsf{int})\ (\lambda x.\ x + 1))$$

which, after only minor reformulation, becomes a certificate theorem for the given HOL function add1:

$$\vdash \mathsf{Eval}\ env\ \lfloor \texttt{fn x => x + 1} \rfloor\ ((\mathsf{int} \longrightarrow \mathsf{int})\ \mathsf{add1})$$

Additional notes. The main part of the synthesis algorithm is always a bottom-up traversal as described above. However, synthesis of recursive functions requires an additional post-processing phase which involves an automatic induction proof. We omit a detailed description of such induction proofs since we have described our solution previously [14]. However, we discuss our solution at a high level in Sect. 5.3 where we explain how the previously published approach has been modified to tackle monadic programs in which termination depends on the monadic state.

3.3 Synthesis of Stateful ML Code

Our algorithm for synthesis of stateful ML is very similar to the algorithm described above for synthesis of pure CakeML code. The main differences are:

- the input HOL terms must be written in a state-and-exception monad, and
- instead of Eval and $\cdot \longrightarrow \cdot$, the derived theorems use EvalM and $\cdot \longrightarrow^{M} \cdot$,

where EvalM and $\cdot \longrightarrow^{M} \cdot$ relate the monad's state to the references and foreign function interface of the underlying CakeML state (fields refs and ffi). These concepts will be described below.

Generic state-and-exception monad. The new generalised synthesis work-flow uses the following state-and-exception monad $(\alpha,\ \beta,\ \gamma)\ \texttt{M}$, where α is the state type, β is the return type, and γ is the exception type.

$$\begin{aligned}&(\alpha,\ \beta,\ \gamma)\ \texttt{M}\ =\ \alpha\ \rightarrow\ (\beta,\ \gamma)\ \texttt{exc}\ *\ \alpha\\ &\text{where}\ \ (\beta,\ \gamma)\ \texttt{exc}\ =\ \mathsf{Success}\ \beta\ |\ \mathsf{Failure}\ \gamma\end{aligned}$$

We define the following interface for this monad type. Note that syntactic sugar is often used: in our case, we write do $n\ \leftarrow\ foo$; return $(bar\ n)$ od (as was done in Sect. 1) when we mean bind foo $(\lambda\, n.$ return $(bar\ n))$.

$$\begin{aligned}&\mathsf{return}\ x\ \stackrel{\mathsf{def}}{=}\ (\lambda\, s.\ (\mathsf{Success}\ x,s))\\ &\mathsf{bind}\ x\ f\ \stackrel{\mathsf{def}}{=}\\ &\quad (\lambda\, s.\ \mathsf{case}\ x\ s\ \mathsf{of}\ (\mathsf{Success}\ y,s)\ \Rightarrow\ f\ y\ s\ |\ (\mathsf{Failure}\ x,s)\ \Rightarrow\ (\mathsf{Failure}\ x,s))\\ &x\ \mathsf{otherwise}\ y\ \stackrel{\mathsf{def}}{=}\\ &\quad (\lambda\, s.\ \mathsf{case}\ x\ s\ \mathsf{of}\ (\mathsf{Success}\ v,s)\ \Rightarrow\ (\mathsf{Success}\ v,s)\ |\ (\mathsf{Failure}\ e,s)\ \Rightarrow\ y\ s)\end{aligned}$$

Functions that update the content of state can only be defined once the state type is instantiated. A function for changing a monad M to have a different state type is introduced in Sect. 3.5.

Definitions and lemmas for synthesis. We define EvalM as follows. A CakeML source expression *exp* is considered to satisfy an execution relation P if for any CakeML state s, which is related by state_rel to the state monad state st and state assertion H, the CakeML expression *exp* evaluates to a result *res* such that the relation P accepts the transition and state_rel_frame holds for state assertion H. The auxiliary functions state_rel and state_rel_frame will be described below. The first argument *ro* can be used to restrict effects to *references only*, as described a few paragraphs further down.

$$
\begin{array}{l}
\mathsf{EvalM}\ ro\ env\ st\ exp\ P\ H \stackrel{\mathsf{def}}{=} \\
\quad \forall s. \\
\qquad \mathsf{state\_rel}\ H\ st\ s \Rightarrow \\
\qquad \exists s_2\ res\ st_2\ ck. \\
\qquad\quad (\mathsf{evaluate}\ (s\ with\ \mathsf{clock}\ :=\ ck)\ env\ [exp] = (s_2,res)) \wedge \\
\qquad\quad P\ st\ (st_2,res) \wedge \mathsf{state\_rel\_frame}\ ro\ H\ (st,s)\ (st_2,s_2)
\end{array}
$$

In the definition above, state_rel and state_rel_frame are used to check that the user-specified state assertion H relates the CakeML states and the monad states. Furthermore, state_rel_frame ensures that the separation logic frame rule is true. Both use the separation logic set-up from our previous work on characteristic formulae for CakeML [5], where we define a function st2heap which, given a projection p and CakeML state s, turns the CakeML state into a set representation of the reference store and foreign-function interface (used for I/O).

The H in the definition above is a pair (h,p) containing a heap assertion h and the projection p. We define state_rel (h,p) st s to state that the heap assertion produced by applying h to the current monad state st must be true for some subset produced by st2heap when applied to the CakeML state s. Here $*$ is the separating conjunction and T is true for any heap.

$$
\mathsf{state\_rel}\ (h,p)\ st\ s \stackrel{\mathsf{def}}{=} (h\ st\ *\ \mathsf{T})\ (\mathsf{st2heap}\ p\ s)
$$

The relation state_rel_frame states: any frame F that is true separately from $h\ st_1$ for the initial state is also true for the final state; and if the references-only ro configuration is set, then the only difference in the states must be in the references and clock, i.e. no I/O operations are permitted. The ro flag is instantiated to true when a pure specification (Eval) is proved for local state (Sect. 4).

$$
\begin{array}{l}
\mathsf{state\_rel\_frame}\ ro\ (h,p)\ (st_1,s_1)\ (st_2,s_2) \stackrel{\mathsf{def}}{=} \\
\quad (ro \Rightarrow \exists refs.\ s_2 = s_1\ with\ \mathsf{refs}\ :=\ refs) \wedge \\
\quad \forall F. \\
\qquad (h\ st_1\ *\ F)\ (\mathsf{st2heap}\ p\ s_1) \Rightarrow \\
\qquad (h\ st_2\ *\ F\ *\ \mathsf{T})\ (\mathsf{st2heap}\ p\ s_2)
\end{array}
$$

We prove lemmas to aid the synthesis algorithm in construction of proofs. The lemmas shown in this paper use the following definition of monad.

$$
\begin{array}{l}
\mathsf{monad}\ a\ b\ x\ st_1\ (st_2,res) \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{case}\ (x\ st_1,res)\ \mathsf{of} \\
\quad\quad ((\mathsf{Success}\ y,st),\mathsf{Rval}\ [v])\ \Rightarrow\ (st = st_2) \wedge a\ y\ v \\
\quad | \ ((\mathsf{Failure}\ e,st),\mathsf{Rerr}\ (\mathsf{Rraise}\ v))\ \Rightarrow\ (st = st_2) \wedge b\ e\ v \\
\quad | \ \_\ \Rightarrow\ \mathsf{F}
\end{array}
$$

Synthesis makes use of the following two lemmas in proofs involving monadic return and bind. For return x, synthesis proves an Eval-theorem for x. For bind, it proves a theorem that fits the shape of the first four lines of the lemma and returns a theorem consisting of the last two lines, appropriately instantiated.

$$\vdash \mathsf{Eval}\ env\ exp\ (a\ x) \Rightarrow \mathsf{EvalM}\ ro\ env\ st\ exp\ (\mathsf{monad}\ a\ b\ (\mathsf{return}\ x))\ H$$

$$\begin{array}{l}\vdash ((assums_1 \Rightarrow \mathsf{EvalM}\ ro\ env\ st\ e_1\ (\mathsf{monad}\ b\ c\ x)\ H) \wedge \\ \quad \forall z\ v. \\ \quad\quad b\ z\ v \wedge assums_2\ z \Rightarrow \\ \quad\quad \mathsf{EvalM}\ ro\ (env\ [n \mapsto v])\ (\mathsf{snd}\ (x\ st))\ e_2\ (\mathsf{monad}\ a\ c\ (f\ z))\ H) \Rightarrow \\ \quad assums_1 \wedge (\forall z.\ (\mathsf{fst}\ (x\ st) = \mathsf{Success}\ z) \Rightarrow assums_2\ z) \Rightarrow \\ \quad \mathsf{EvalM}\ ro\ env\ st\ \lfloor \texttt{let n =}\ e_1\ \texttt{in}\ e_2 \rfloor\ (\mathsf{monad}\ a\ c\ (\mathsf{bind}\ x\ f))\ H\end{array}$$

3.4 References, Arrays and I/O

The synthesis algorithm uses specialised lemmas when the generic state-and-exception monad has been instantiated. Consider the following instantiation of the monad's state type to a record type. The programmer's intention is that the lists are to be synthesised to arrays in CakeML and the I/O component `IO_fs` is a model of a file system (taken from a library).

```
example_state =
 <| ref1 : int; farray1 : int list; rarray1 : int list; stdio : IO_fs |>
```

With the help of getter- and setter-functions and library functions for file I/O, users can conveniently write monadic functions that operate over this state type.

When it comes to synthesis, the automation instantiates H with an appropriate heap assertion, in this instance: ASSERT. The user has informed the synthesis tool that farray1 is to be a fixed-size array and rarray1 is to be a resizable-size array. A resizable-array is implemented as a reference that contains an array, since CakeML (like SML) does not directly support resizing arrays. Below, REF_REL int ref1_loc st.ref1 asserts that int relates the value held in a reference at a fixed store location ref1_loc to the integer in st.ref1. Similarly, ARRAY_REL and RARRAY_REL specify a connection for the array fields. Lastly, STDIO is a heap assertion for the file I/O taken from a library.

$$\begin{array}{l}\mathsf{ASSERT}\ st \stackrel{\text{def}}{=} \\ \quad \mathsf{REF\_REL\ int\ ref1\_loc}\ st.\mathsf{ref1}\ *\ \mathsf{RARRAY\_REL\ int}\ rarray1\_loc\ st.\mathsf{rarray1}\ * \\ \quad \mathsf{ARRAY\_REL\ int\ farray1\_loc}\ st.\mathsf{farray1}\ *\ \mathsf{STDIO}\ st.\mathsf{stdio}\end{array}$$

Automation specialises pre-proved EvalM lemmas for each term that might be encountered in the monadic functions. As an example, a monadic function might contain an automatically defined function update_farray1 for updating array farray1. Anticipating this, synthesis automation can, at set-up time, automatically derive the following lemma which it can use when it encounters update_farray1.

$$\begin{array}{l}\vdash \mathsf{Eval}\ env\ e_1\ (\mathsf{num}\ n) \wedge \mathsf{Eval}\ env\ e_2\ (\mathsf{int}\ x) \wedge \\ \quad (\mathsf{lookup\_var}\ \lfloor \texttt{farray1} \rfloor\ env = \mathsf{Some\ farray1\_loc}) \Rightarrow \\ \quad \mathsf{EvalM}\ ro\ env\ st\ \lfloor \texttt{Array.update (farray1,}\ e_1\texttt{,}\ e_2\texttt{)} \rfloor \\ \quad\quad (\mathsf{monad\ unit\ exc\ (update\_farray1}\ n\ x))\ (\mathsf{ASSERT}, p)\end{array}$$

3.5 Combining Monad State Types

Previously developed monadic functions (e.g. from an existing library) can be used as part of a larger context, by combining state-and-exception monads with different state types. Consider the case of the file I/O in the example from above. The following EvalM theorem has been proved in the CakeML basis library.

$$\begin{array}{l}\vdash \mathsf{Eval}\ env\ e\ (\mathsf{string}\ x) \wedge \\ \quad (\mathsf{lookup\_var}\ \lfloor \texttt{print} \rfloor\ env = \mathsf{Some\ print\_v}) \Rightarrow \\ \quad \mathsf{EvalM\ F}\ env\ st\ \lfloor \texttt{print}\ \ e \rfloor\ (\mathsf{monad\ unit}\ b\ (\mathsf{print}\ x))\ (\mathsf{STDIO}, p)\end{array}$$

This can be used directly if the state type of the monad is the `IO_fs` type. However, our example above uses `example_state` as the state type.

To overcome such type mismatches, we define a function liftM which can bring a monadic operation defined in libraries into the required context. The type of liftM $r\ w$ is $(\alpha,\ \beta,\ \gamma)\ \texttt{M} \rightarrow (\epsilon,\ \beta,\ \gamma)\ \texttt{M}$, for appropriate r and w.

$$\mathsf{liftM}\ r\ w\ op \stackrel{\text{def}}{=} (\lambda s.\ (\mathsf{let}\ (ret, new) = op\ (r\ s)\ \mathsf{in}\ (ret, w\ (\mathsf{K}\ new)\ s)))$$

Our liftM function changes the state type. A simpler lifting operation can be used to change the exception type.

For our example, we define stdio f as a function that performs f on the `IO_fs`-part of a `example_state`. (The fib example in Sect. 1 used a similar stdio.)

$$\mathsf{stdio} \stackrel{\text{def}}{=} \mathsf{liftM}\ (\lambda s.\ s.\mathsf{stdio})\ (\lambda f\ s.\ s\ with\ \mathsf{stdio}\ updated\_by\ f)$$

Our synthesis mechanism automatically derives a lemma that can transfer any EvalM result for the file I/O model to a similar EvalM result wrapped in the stdio function. Such lemmas are possible because of the separation logic frame rule that is part of EvalM. The generic lemma is the following:

$$\begin{array}{l}\vdash (\forall st.\ \mathsf{EvalM}\ ro\ env\ st\ exp\ (\mathsf{monad}\ a\ b\ op)\ (\mathsf{STDIO}, p)) \Rightarrow \\ \quad \forall st.\ \mathsf{EvalM}\ ro\ env\ st\ exp\ (\mathsf{monad}\ a\ b\ (\mathsf{stdio}\ op))\ (\mathsf{ASSERT}, p)\end{array}$$

And the following is the transferred lemma, which enables synthesis of HOL terms of the form stdio (print x) for Eval-synthesisable x.

$$\begin{array}{l}\vdash \mathsf{Eval}\ env\ e\ (\mathsf{string}\ x) \wedge \\ \quad (\mathsf{lookup\_var}\ \lfloor \texttt{print} \rfloor\ env = \mathsf{Some\ print\_v}) \Rightarrow \\ \quad \mathsf{EvalM\ F}\ env\ st\ \lfloor \texttt{print}\ \ e \rfloor\ (\mathsf{monad\ unit\ exc}\ (\mathsf{stdio}\ (\mathsf{print}\ x)))\ (\mathsf{ASSERT}, p)\end{array}$$

Changing the monad state type comes at no additional cost to the user; our tool is able to derive both the generic and transferred EvalM lemmas, when provided with the original EvalM result.

4 Local State and the Abstract Synthesis Mode

This section explains how we have adapted the method described above to also support generation of code that uses local state and local exceptions. These features enable use of stateful code (EvalM) in a pure context (Eval). We used these features to significantly speed up parts of the CakeML compiler (see Sect. 6).

In the monadic functions, users indicate that they want local state to be generated by using the following run function. In the logic, the run function essentially just applies a monadic function *m* to an explicitly provided state *st*.

$$\begin{aligned}&\mathsf{run} : (\alpha,\ \beta,\ \gamma)\,\mathtt{M} \rightarrow \alpha \rightarrow (\beta,\ \gamma)\,\mathtt{exc}\\&\mathsf{run}\ m\ st \stackrel{\text{def}}{=} \mathsf{fst}\ (m\ st)\end{aligned}$$

In the generated code, an application of run to a concrete monadic function, say bar, results in code of the following form:

```
fun run_bar k n =
  let
    val r = ref ... (* allocate, initialise, let-bind all local state *)
    fun foo n = ... (* all auxiliary funs that depend on local state *)
    fun bar n = ... (* define the main monadic function *)
  in Success (bar n) end (* wrap normal result in Success constructor *)
  handle e => Failure e; (* wrap any exception in Failure constructor *)
```

Synthesis of locally effectful code is made complicated in our setting for two reasons: (i) there are no fixed locations where the references and arrays are stored, e.g. we cannot define ref1_loc as used in the definition of ASSERT in Sect. 3.4; and (ii) the local names of state components must be in scope for all of the function definitions that depend on local state.

Our solution to challenge (i) is to leave the location values as variables (loc_1, loc_2, loc_3) in the heap assertion when synthesising local state. To illustrate, we will adapt the `example_state` from Sect. 3.4: we omit `IO_fs` in the state because I/O cannot be made local. The local-state enabled heap assertion is:

$$\begin{aligned}&\mathsf{LOCAL\_ASSERT}\ loc_1\ loc_2\ loc_3\ st \stackrel{\text{def}}{=}\\&\ \ \mathsf{REF\_REL\ int}\ loc_1\ st.\mathsf{ref1}\ *\ \mathsf{RARRAY\_REL\ int}\ loc_2\ st.\mathsf{rarray1}\ *\\&\ \ \mathsf{ARRAY\_REL\ int}\ loc_3\ st.\mathsf{farray1}\end{aligned}$$

The lemmas referring to local state now assume they can find the right variable locations with variable look-ups.

$$\begin{aligned}&\vdash \mathsf{Eval}\ env\ e_1\ (\mathsf{num}\ n) \wedge \mathsf{Eval}\ env\ e_2\ (\mathsf{int}\ x) \wedge\\&\ \ (\mathsf{lookup\_var}\ \lfloor\mathtt{farray1}\rfloor)\ env = \mathsf{Some}\ loc_3) \Rightarrow\\&\ \ \mathsf{EvalM}\ ro\ env\ st\ \lfloor\mathtt{Array.update\ (farray1,}e_1\mathtt{,}e_2\mathtt{)}\rfloor\\&\ \ \ \ (\mathsf{monad\ unit\ exc}\ (\mathsf{update\_farray1}\ n\ x))\ (\mathsf{LOCAL\_ASSERT}\ loc_1\ loc_2\ loc_3,p)\end{aligned}$$

Challenge (ii) was caused by technical details of our previous synthesis methods. The previous version was set up to only produce top-level declarations, which is incompatible with the requirement to have local (not globally fixed) state declarations shared between several functions. The requirement to only have top-level declarations arose from our desire to keep things simple: each synthesised function is attached to the end of a concrete linear program that is being built. It is beneficial to be concrete because then each assumption on the lexical environment where the function is defined can be proved immediately on definition. We will call this old approach the *concrete mode* of synthesis, since it eagerly builds a concrete program.

In order to support having functions access local state, we implement a new *abstract mode* of synthesis. In the abstract mode, each assumption on the lexical environment is left as an unproved side condition as long as possible. This allows us to define functions in a dynamic environment.

To prove a pure specification (Eval) from the EvalM theorems, the automation first proves that the generated state-allocation and -initialisation code establishes the relevant heap assertion (e.g. LOCAL_ASSERT); it then composes the abstractly synthesised code while proving the environment-related side conditions (e.g. presence of loc_3). The final proof of an Eval theorem requires instantiating the references-only *ro* flag to true, in order to know that no I/O occurs (Sect. 3.3).

Type Variables in Local Monadic State

Our previous approach [14] allowed synthesis of (pure) polymorphic functions. Our new mechanism is able to support the same level of generality by permitting type variables in the type of monadic state that is used locally. As an example, consider a monadic implementation of an in-place quicksort algorithm, quicksort, with the following type signature:

quicksort : α `list` $\rightarrow$ ($\alpha \rightarrow \alpha \rightarrow$ `bool`) $\rightarrow$ (α `state`, α `list`, `exn`) `M`
where α `state` = `<|` arr : α `list` `|>`

The function quicksort takes a list of values of type α and an ordering on α as input, producing a sorted list as output. However, internally it copies the input list into a mutable array in order to perform fast in-place random accesses.

The heap assertion for α `state` is called POLY_ASSERT, and is defined below:

$$\text{POLY\_ASSERT } A \; loc \; st \stackrel{\text{def}}{=} \text{RARRAY\_REL } A \; loc \; st.\text{arr}$$

Here, A is a refinement invariant for logical values of type α. This parametrisation over state type variables is similar to the way in which location values were parametrised to solve challenge (i) above.

Applying run to quicksort, and synthesising CakeML from the result gives the following certificate theorem which makes the stateful quicksort callable from pure translations.

$$\begin{aligned} &\vdash (\text{list } a \longrightarrow (a \longrightarrow a \longrightarrow \text{bool}) \longrightarrow \text{exc\_type } (\text{list } a) \text{ exn}) \\ &\quad \text{run\_quicksort } \lfloor \texttt{run\_quicksort} \rfloor \end{aligned}$$

Here exc_type (list a) exn is the refinement invariant for type (α `list`, `exn`) `exc`.

For the quicksort example, we have manually proved that quicksort will always return a Success value, provided the comparison function orders values of type α. The result of this effort is CakeML code for quicksort that uses state internally, but can be used as if it is a completely pure function without any use of state or exceptions.

5 Termination That Depends on Monadic State

In this section, we describe how the proof-producing synthesis method in Sect. 3 has been extended to deal with a class of recursive monadic functions whose termination depends on the state hidden in the monad. This class of functions creates new difficulties, as (i) the HOL4 function definition system is unable to prove termination of these functions; and, (ii) our synthesis method relies on induction theorems produced by the definition system to discharge preconditions during synthesis.

We address issue (i) by extending the HOL4 definition system with a set of congruence rewrites for the monadic bind operation, bind (Sect. 5.2). We then explain, at a high level,

how the proof-producing synthesis in Sect. 3 is extended to deal with the preconditions that arise when synthesising code from recursive monadic functions (Sect. 5.3).

We begin with a brief overview of how recursive function definitions are handled by the HOL4 function definition system (Sect. 5.1).

5.1 Preliminaries: Function Definitions in HOL4

In order to accept recursive function definitions, the HOL4 system requires a well-founded relation to be found between the arguments of the function, and those of recursive applications. The system automatically extracts conditions that this relation must satisfy, attempts to guess a well-founded relation based on these conditions, and then uses this relation to solve the termination goal.

Function definitions involving higher-order functions (e.g. bind) sometimes causes the system to derive unprovable termination conditions, if it cannot extract enough information about recursive applications. When this occurs, the user must provide a congruence theorem that specifies the context of the higher-order function. The system uses this theorem to derive correct termination conditions, by rewriting recursive applications.

5.2 Termination of Recursive Monadic Functions

By default, the HOL4 system is unable to automatically prove termination of recursive monadic functions involving bind. To aid the system in extracting provable termination conditions, we introduce the following congruence theorem for bind:

$$\begin{array}{l} \vdash (x = x') \wedge (s = s') \wedge \\ \quad (\forall\, y\ s''.\ (x'\ s' = (\mathsf{Success}\ y, s'')) \Rightarrow (f\ y\ s'' = f'\ y\ s'')) \Rightarrow \\ \quad (\mathsf{bind}\ x\ f\ s = \mathsf{bind}\ x'\ f'\ s') \end{array} \tag{3}$$

Theorem (3) expresses a rewrite of the term bind $x\ f\ s$ in terms of rewrites involving its component subterms (x, f, and s), but allows for the assumption that $x'\ s'$ (the rewritten effect) must execute successfully.

However, rewriting definitions with (3) is not always sufficient: in addition to ensuring that the effect x in bind $x\ f$ executed successfully, the HOL4 system must also know the value and state resulting from its execution. This problem arises because the monadic state argument to bind is left implicit in user definitions. We address this issue by rewriting the defining equations of monadic functions using η-expansion before passing them to the definition system, making all partial bind applications syntactically fully applied. The whole process is automated so that it is opaque to the user, allowing definition of recursive monadic functions with no additional effort.

5.3 Synthesising ML from Recursive Monadic Functions

The proof-producing synthesis method described in Sect. 3.2 is syntax-directed and proceeds in a bottom-up manner. For recursive functions, a tweak to this strategy is required, as bottom-up traversal would require any recursive calls to be treated before the calling function (this is clearly cyclic).

We begin with a brief explanation of how our previous (pure) synthesis tool [14] tackles recursive functions, before outlining how our new approach builds on this.

Pure recursive functions. As an example, consider the function gcd that computes the greatest common divisor of two positive integers:

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

Before traversing the function body of gcd in a bottom-up manner, we simply assume the desired Eval result to hold for all recursive applications in the function definition, and record their arguments during synthesis. This results in the following Eval theorem for gcd (where Eq is defined as $\mathsf{Eq}\ a\ x \stackrel{\mathsf{def}}{=} (\lambda\, y\ v.\ (x = y) \wedge a\ y\ v)$, and is used to record arguments for recursive applications):

$$\begin{array}{l} \vdash (n > 0 \Rightarrow \\ \quad \mathsf{Eval}\ env\ \lfloor \texttt{gcd} \rfloor\ ((\mathsf{Eq\ int}\ n \longrightarrow \mathsf{Eq\ int}\ (m\ \mathsf{mod}\ n) \longrightarrow \mathsf{int})\ \mathsf{gcd})) \Rightarrow \\ \quad \mathsf{Eval}\ env\ \lfloor \texttt{gcd} \rfloor\ ((\mathsf{Eq\ int}\ m \longrightarrow \mathsf{Eq\ int}\ n \longrightarrow \mathsf{int})\ \mathsf{gcd}) \end{array} \tag{4}$$

and below is the desired Eval result for gcd:

$$\vdash \mathsf{Eval}\ env\ \lfloor \texttt{gcd} \rfloor\ ((\mathsf{Eq\ int}\ m \longrightarrow \mathsf{Eq\ int}\ n \longrightarrow \mathsf{int})\ \mathsf{gcd}) \tag{5}$$

Theorems (4) and (5) match the shape of the hypothesis and conclusion (respectively) of the induction theorem for gcd:

$$\vdash (\forall m\ n.\ (n > 0 \Rightarrow P\ n\ (m\ \mathsf{mod}\ n)) \Rightarrow P\ m\ n) \Rightarrow \forall m\ n.\ P\ m\ n$$

By instantiating this induction theorem appropriately, the preconditions in (4) can be discharged (and if automatic proof fails, the goal is left for the user to prove).

Monadic recursive functions. Function definitions whose termination depends on the monad give rise to induction theorems which also depend on the monad. This creates issues, as the monad argument is left implicit in the definition. As an example, here is a function linear_search that searches through an array for a value:

```
linear_search val idx =
 do
  len ← arr_length;
  if idx ≥ len then return None else
   do
    elem ← arr_sub idx;
    if elem = val then return (Some idx) else linear_search val (idx + 1)
   od
 od
```

When given the above definition, the HOL4 system automatically derives the following induction theorem:

$$\begin{array}{l} \vdash (\forall\, val\ idx\ s. \\ \quad (\forall\, len\ s'\ elem\ s''. \\ \quad\quad (\mathsf{arr\_length}\ s = (\mathsf{Success}\ len,s')) \wedge \neg(idx \geq len) \wedge \\ \quad\quad (\mathsf{arr\_sub}\ idx\ s' = (\mathsf{Success}\ elem,s'')) \wedge elem \neq val \Rightarrow \\ \quad\quad P\ val\ (idx + 1)\ s'') \Rightarrow \\ \quad P\ val\ idx\ s) \Rightarrow \\ \forall\, val\ idx\ s.\ P\ val\ idx\ s \end{array} \tag{6}$$

Table 1 Compilation and run times (in s) for various CakeML benchmarks

Timing	Benchmark					
	Knuth–Bendix	Smith-normal-form	Tailfib	Pidigits	Life	Logic
Compile (old)	18.15	16.34	8.86	9.16	9.51	12.31
Run (old)	19.58	23.53	16.60	15.47	25.59	23.33
Compile (new)	1.21	1.46	0.99	1.02	1.05	1.62
Run (new)	19.90	22.91	16.70	15.64	24.17	22.33

These compare a version of the CakeML compiler where the register allocator is purely functional (old) against a version which uses local state and arrays (new)

The context of recursive applications (arr_length and arr_sub) has been extracted correctly by HOL4, using the congruence theorem (3) and automated η-expansion for bind (see Sect. 5.2).

However, there is now a mismatch between the desired form of the EvalM result and the conclusion of the induction theorem: the latter depends explictly on the state, but the function depends on it only implicitly. We have modified our synthesis tool to account for this, in order to correctly discharge the necessary preconditions as above. When preconditions cannot be automatically discharged, they are left as proof obligations to the user, and the partial results derived are saved in the HOL4 theorem database.

6 Case Studies and Experiments

In this section, we present the runtime and proof size results of applying our method to some case studies.

Register Allocation. The CakeML compiler's register allocator is written with a state (and exception) monad but it was previously synthesized to pure CakeML code. We updated it to use the new synthesis tool, resulting in the automatic generation of stateful CakeML code. The allocator benefits significantly from this change because it can now make use of CakeML arrays via the synthesis tool. It was previously confined to using tree-like functional arrays for its internal state, leading to logarithmic access overheads. This is not a specific issue for the CakeML compiler; a verified register allocator for CompCert [3] also reported log-factor overheads due to (functional) array accesses.

Tests were carried out using versions of the bootstrapped CakeML compiler. We ran each test 50 times on the same input program, recording time elapsed in each compiler phase. For each test, we also compared the resulting executables 10 times, to confirm that both compilers generated code of comparable quality (i.e. runtime performance). Performance experiments were carried out on an Intel i7-2600 running at 3.4GHz with 16 GB of RAM. The results are summarized in Table 1. Full data is available at https://cakeml.org/ijcar18.zip. [1]

In the largest program (`Knuth–Bendix`), the new register allocator ran 15 times faster (with a wide 95% CI of 11.76–20.93 due in turn to a high standard deviation on the runtimes for the old code). In the smaller `pidigits` benchmark, the new register allocator ran 9.01 times faster (95% CI of 9.01–9.02). Across 6 example input programs, we saw ratios of

[1] These tests were performed for the earlier conference version of this paper [6] comparing two earlier versions of the CakeML compiler. The compiler has changed significantly since then but we have we kept these experiments because they provide a fairer comparison of register allocation performance with/without using the synthesis tool to generate stateful code.

Table 2 CakeML compiler cross-compile bootstrap time (in s) spent in the assembly phase for its various compilation targets

Timing	Cross-compilation target				
	ARMv6	ARMv8 (†)	MIPS	RISC-V	x64
Assembly (old)	8.86	–	8.69	9.21	8.27
Assembly (new)	6.43	–	6.94	6.7	5.04

† For the ARMv8 target, the cross-compile bootstrap does not run to completion at the point of writing. This is for reasons unrelated to the changes in this paper

runtimes between 7.58 and 15.06. Register allocation was previously such a significant part of the compiler runtime that this improvement results in runtime improvements for the whole compiler (on these benchmark programs) of factors between 2 and 9 times.

Speeding up the CakeML compiler. The register allocator exemplifies one way the synthesis tool can be used to improve existing, verified CakeML programs and in particular, the CakeML compiler itself. Briefly, the steps are: (i) re-implement slow parts of the compiler with, e.g., an appropriate state monad, (ii) verify that this new implementation produces the same result as the existing (verified) implementation, (iii) swap in the new implementation, which synthesizes to stateful code, during the bootstrap of the CakeML compiler. (iv) The preceeding steps can be repeated as desired, relying on the automated synthesis tool for quick iteration.

As another example, we used the synthesis tool to improve the assembly phase of the compiler. A major part of time spent in this phase is running the instruction encoder, which performs several word arithmetic operations when it computes the byte-level representation of each instruction. However, duplicate instructions appear very frequently, so we implemented a cache of the byte-level representations backed by a hash table represented as a state monad (i). This caching implementation is then verified (ii), before a verified implementation is synthesized where the hash table is implemented as an array (iii). We also iterated through several candidate hash functions (iv). Overall, this change took about 1-person week to implement, verify, and integrate in the CakeML compiler. We benchmarked the cross-compile bootstrap times of the CakeML compiler after this change to measure its impact across different CakeML compilation targets. Results are summarized in Table 2. Across compilation targets, the assembly phase is between 1.25 to 1.64 times faster.

OpenTheory Article Checker. The type changing feature from Sect. 3.5 enabled us to produce an OpenTheory [8] article checker with our new synthesis approach, and reduce the amount of manual proof required in a previous version. The checker reads articles from the file system, and performs each logical inference in the OpenTheory framework using the verified Candle kernel [9]. Previously, the I/O code for the checker was implemented in stateful CakeML, and verified manually using characteristic formulae. By replacing the manually verified I/O wrapper by monadic code we removed 400 lines of tedious manual proof.

7 Related Work

Effectful code using monads. Our work on encapsulating stateful computations (Sect. 4) in pure programs is similar in purpose to that of the ST monad [12]. The main difference is how this encapsulation is performed: the ST monad relies on parametric polymorphism to prevent

references from escaping their scope, whereas we utilise lexical scoping in synthesised code to achieve a similar effect.

Imperative HOL by Bulwahn et al. [4] is a framework for implementing and reasoning about effectful programs in Isabelle/HOL. Monadic functions are used to describe stateful computations which act on the heap, in a similar way as Sect. 3 but with some important differences. Instead of using a state monad, the authors introduce a polymorphic *heap monad*—similar in spirit to the ST monad, but without encapsulation—where polymorphism is achieved by mapping HOL types to the natural numbers. Contrary to our approach, this allows for heap elements (e.g. references) to be declared on-the-fly and used as first-class values. The drawback, however, is that only countable types can be stored on the heap; in particular, the heap monad does not admit function-typed values, which our work supports.

More recently, Lammich [11] has built a framework for the refinement of pure data structures into imperative counterparts, in Imperative HOL. The refinement process is automated, and refinements are verified using a program logic based on separation logic, which comes with proof-tools to aid the user in verification.

Both developments [4,11] differ from ours in that they lack a verified mechanism for extracting executable code from shallow embeddings. Although stateful computations are implemented and verified within the confines of higher-order logic, Imperative HOL relies on the unverified code-generation mechanisms of Isabelle/HOL. Moreover, neither work presents a way to deal with I/O effects.

Verified Compilation. Mechanisms for synthesising programs from shallow embeddings defined in the logics of interactive theorem provers exist as components of several verified compiler projects [1,7,13,14]. Although the main contribution of our work is proof-producing synthesis, comparisons are relevant as our synthesis tool plays an important part in the CakeML compiler [10]. To the best of our knowledge, ours is the first work combining effectful computations with proof-producing synthesis and fully verified compilation.

CertiCoq by Anand et al. [1] strives to be a fully verified optimising compiler for functional programs implemented in Coq. The compiler front-end supports the full syntax of the dependently typed logic Gallina, which is reified into a deep embedding and compiled to Cminor through a series of verified compilation steps [1]. Contrary to the approach we have taken [14] (see Sect. 3.2), this reification is neither verified nor proof-producing, and the resulting embedding has no formal semantics (although there are attempts to resolve this issue [2]). Moreover, as of yet, no support exists for expressing effectful computations (such as in Sect. 3.4) in the logic. Instead, effects are deferred to wrapper code from which the compiled functions can be called, and this wrapper code must be manually verified.

The Œuf compiler by Mullen et al. [13] is similar in spirit to CertiCoq in that it compiles pure Coq functions to Cminor through a verified process. Similarly, compiled functions are pure, and effects must be performed by wrapper code. Unlike CertiCoq, Œuf supports only a limited subset of Gallina, from which it synthesises deeply embedded functions in the Œuf-language. The Œuf language has both denotational and operational semantics, and the resulting syntax is automatically proven equivalent with the corresponding logical functions through a process of computational denotation [13].

Hupel and Nipkow [7] have developed a compiler from Isabelle/HOL to CakeML AST. The compiler satisfies a partial correctness guarantee: if the generated CakeML code terminates, then the result of execution is guaranteed to relate to an equality in HOL. Our approach proves termination of the code.

8 Conclusion

This paper describes a technique that makes it possible to synthesise whole programs from monadic functions in HOL, with automatic proofs relating the generated effectful code to the original functions. Using the separation logic from characteristic formulae for CakeML, the synthesis mechanism supports references, exceptions, I/O, reusable library developments, encapsulation of locally stateful computations inside pure functions, and code generation for functions where termination depends on state. To our knowledge, this is the first proof-producing synthesis technique with the aforementioned features.

We hope that the techniques developed in this paper will allow users of the CakeML tools to develop verified code using only shallow embeddings. We hope that only expert users, who develop libraries, will need to delve into manual reasoning in CF or direct reasoning about deeply embedded CakeML programs.

Acknowledgements Open access funding provided by Chalmers University of Technology. The first and fifth authors were partly supported by the Swedish Foundation for Strategic Research. The seventh author was supported by an A*STAR National Science Scholarship (Ph.D.), Singapore. The third author was supported by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS).

References

1. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: a verified compiler for Coq. In: CoqPL (2017)
2. Anand, A., Boulier, S., Tabareau, N., Sozeau, M.: Typed template Coq—certified meta-programming in Coq. In: CoqPL (2018)
3. Blazy, S., Robillard, B., Appel, A.W.: Formal verification of coalescing graph-coloring register allocation. In: ESOP, Volume 6012 of LNCS (2010)
4. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs, Volume 5170 of LNCS, pp. 134–149 (2008)
5. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.L.: Verified characteristic formulae for CakeML. In: Yang, H. (ed.) ESOP, Volume 10201 of LNCS, pp. 584–610 (2017)
6. Ho, S., Abrahamsson, O., Kumar, R., Myreen, M.O., Tan, Y.K., Norrish, M.: Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR, pp. 646–662 (2018)
7. Hupel, L., Nipkow, T.: A verified compiler from Isabelle/HOL to CakeML. In: Ahmed, A. (ed.) European Symposium on Programming (ESOP). Springer, Berlin (2018)
8. Hurd, J.: The OpenTheory standard theory library. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM, Volume 6617 of LNCS, pp. 177–191 (2011)
9. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: Self-formalisation of higher-order logic-semantics, soundness, and a verified implementation. J. Autom. Reason. **56**(3), 221–259 (2016)
10. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Suresh, J., Sewell, P. (eds.) POPL, pp. 179–192 (2014)
11. Lammich, P.: Refinement to Imperative/HOL. In: ITP, Volume 9236 of LNCS (2015)
12. Launchbury, J., Peyton Jones, S.L.: Lazy functional state threads. In: Sarkar, V., Ryder, B.G., Soffa, M.L. (eds.) PLDI, pp. 24–35 (1994)

13. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: minimizing the Coq extraction TCB. In: CPP (2018)
14. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. J. Funct. Program. **24**(2–3), 284–315 (2014)
15. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: Thiemann, P. (ed.) ESOP, Volume 9632 of LNCS, pp. 589–615 (2016)
16. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS, pp. 55–74 (2002)