Verified Characteristic Formulae for CakeML

Armaël Guéneau, Magnus Myreen, Ramana Kumar, Michael Norrish

Verified Characteristic Formulae for CakeML

Armaël Guéneau[1], Magnus O. Myreen[2], Ramana Kumar[3], and Michael Norrish[4]

[1] ENS de Lyon and Inria, France
[2] CSE Department, Chalmers University of Technology, Sweden
[3] Data61, CSIRO / UNSW, Australia
[4] Data61, CSIRO / ANU, Australia

Abstract. Characteristic Formulae (CF) offer a productive, principled approach to generating verification conditions for higher-order imperative programs, but so far the soundness of CF has only been considered with respect to an informal specification of a programming language (OCaml). This leaves a gap between what is established by the verification framework and the program that actually runs. We present a fully-fledged CF framework for the formally specified CakeML programming language. Our framework extends the existing CF approach to support exceptions and I/O, thereby covering the full feature set of CakeML, and comes with a formally verified soundness theorem. Furthermore, it integrates with existing proof techniques for verifying CakeML programs. This validates the CF approach, and allows users to prove end-to-end theorems for higher-order imperative programs, from specification to language semantics, within a single theorem prover.

1 Introduction

In previous work, Charguéraud introduced a framework for the verification of imperative higher-order programs, based on characteristic formulae (CF). Given a source-level program, the approach allows the user to state a specification for it, in the style of Separation Logic [22], and prove the specification using the full power of a proof assistant. It has proved successful in verifying robust and modular specifications for non-trivial programs [6], and even establishing complexity results [7].

The key component of such a framework is a function that produces, from a source-level program e, its characteristic formula $\mathsf{cf}\ e$. Applying the logical predicate $\mathsf{cf}\ e$ to an environment env, a pre-condition H and a post-condition Q yields the proposition $\mathsf{cf}\ e\ env\ H\ Q$, which implies program e admits H as a pre-condition and Q as a post-condition, in environment env. The user is left with the task of proving the goal $\mathsf{cf}\ e\ env\ H\ Q$ using specialised CF tactics alongside general-purpose tactics in an interactive theorem prover.

Charguéraud's work is realised in a tool named CFML, where (a subset of) OCaml is the language of the certified programs, and Coq is the proof assistant that hosts the characteristic formulae. Only part of the soundness theorem for CFML has been proved in Charguéraud's Coq formalisation.

In this paper, we describe how a CF framework has been constructed and proved sound for the entire CakeML language [26], including its exception mechanism and I/O features. CakeML is a substantial subset of Standard ML, with the notable feature that its compiler has been verified (in the HOL4 proof assistant). In addition to capturing language features not modeled in CFML, we give this framework a fully verified soundness theorem. The entire development is formalised in HOL4, which also plays the role of the proof assistant hosting the characteristic formulae. Though tactic details are not the main topic of this paper, we also provide HOL4 tactic support for our CF framework, just as CFML provides Coq tactics to support the proof of $\mathsf{cf}\ e\ H\ Q$ theorems.

This paper's material goes beyond previous work on characteristic formulae and CFML in the following ways:

- We give a mechanised proof of soundness of characteristic formulae with respect to CakeML's formal semantics (Section 2). By way of contrast, CFML's soundness proof is mostly performed outside of Coq.
- We support additional language features, such as I/O (Section 3) and exceptions (Section 3.2). This makes our framework go beyond CFML, and thus able to handle all features of the CakeML programming language.
- We implement technology to make proofs using characteristic formulae interoperate with the existing synthesis tool for CakeML, namely the proof-producing translator from HOL functions to CakeML (Section 4).

As an appetiser, in Figure 1 we show the code for a simple implementation of the Unix `cat` program, that we are able to verify using our framework. The specification for `cat`, proven correct in our framework, and thus a HOL4 theorem, is given in Figure 2. The main steps of the `cat` proof are described in Section 5.

1.1 Background on CF

This subsection and the next one provide background on CF and CakeML. Readers familiar with these topics can skip ahead to Section 1.3.

Characteristic formulae, as introduced in Charguéraud's PhD thesis [4], are essentially total correctness Hoare triples for ML-style functional programs. The key component of any CF framework is a function cf that produces, from a source-level expression e, the expression's characteristic formula $\mathsf{cf}\ e$. Applying $\mathsf{cf}\ e$ to an environment env, pre-condition H and post-condition Q yields a proposition $\mathsf{cf}\ e\ env\ H\ Q$, which implies program expression e can have H as a pre-condition and Q as a post-condition, in environment env.

While the cf function is the main workhorse behind any CF framework, most user-proved specifications are stated in terms of a Hoare-triple-like judgement for functional applications, app, written with Hoare-triple notation. The intuition is that $\{\!|H|\!\}\ f\ \cdot\ args\ \{\!|Q|\!\}$ is true if the application of function-value f to curried arguments $args$ admits H as a pre-condition and Q as a post-condition. An example of a specification stated in terms of app is shown in Figure 2.

```
fun cat1 fname =
  let
    val fd = CharIO.openIn fname
    fun recurse () =
      case CharIO.read1 fd of
          NONE ⇒ ()
        | SOME c ⇒ (CharIO.write c; recurse ())
  in
    recurse () ;
    CharIO.close fd
  end

fun cat fnames =
  case fnames of
    [] ⇒ ()
  | f::fs ⇒ (cat1 f ; cat fs)
```

Fig. 1: Code implementing concatenation of files to standard out. The `CharIO` module is our verified implementation of an FFI interface to a rudimentary file-system model (see Section 5 for more details).

$$\mathsf{FILENAME}\ s\ sv \iff \mathsf{STRING}\ s\ sv \wedge \mathsf{noNullBytes}\ s \wedge \mathsf{strlen}\ s < 256$$

$$\begin{array}{l}
\vdash \mathsf{LIST\ FILENAME}\ fns\ fnsv \wedge \mathsf{every}\ (\lambda\, fnm.\ \mathsf{inFS_fname}\ fnm\ fs)\ fns \wedge \\
\quad \mathsf{numOpenFDs}\ fs < 255 \Rightarrow \\
\quad\ \{|\mathsf{CATFS}\ fs * \mathsf{STDOUT}\ out|\} \\
\quad\ \ \mathsf{cat_v} \cdot [fnsv] \\
\quad\ \{|\mathsf{POSTv}\ u. \\
\quad\ \ \langle \mathsf{UNIT}\ ()\ u\rangle * \mathsf{CATFS}\ fs * \\
\quad\ \ \mathsf{STDOUT}\ (out \mathbin{@} \mathsf{catfiles_string}\ fs\ fns)|\}
\end{array}$$

Fig. 2: A CF specification of the `cat` function from Figure 1. The `app` predicate underlies the $\{|H|\}\ fv \cdot args\ \{|Q|\}$ notation, giving a CF Hoare triple for a function, indicating that if fv is applied to $args$ in a state satisfying H, the result satisfies Q. The $(*)$ operator (defined on page 14) corresponds to the separating conjunction of separation logic. Parts of specifications occurring within angle brackets (here, only in the post-condition) are conditions that do not depend on the state of the heap. Above, the implication's assumptions require that no file name contains a null byte or has 256 characters or more (enforced by the FILENAME predicate), that every file name corresponds to a real file in the system, and that fewer than 255 files are open. These various requirements naturally fall out of the way the interactions with the file-system are mediated by the FFI interface. The post-condition states that `cat` returns a unit value, that the CATFS component (the "`cat` file-system") of the state is unchanged, and that the standard output stream has been extended with the contents of all the files.

Charguéraud's initial version of CF [5] only applied to pure ML programs. Charguéraud has since extended his approach to support reasoning about imperative stateful ML programs in a style inspired by separation logic and its frame rule [6]. More recently, Charguéraud and Pottier have verified amortized complexity results using CFML [7]. The version we have ported to CakeML is based on Charguéraud's framework for imperative stateful ML programs, but without support for proofs about complexity results.

In Charguéraud's implementation of CF, called CFML, the mechanism for generating characteristic formulae from OCaml programs, i.e., the `cf` function, is external to the proof assistant (Coq), and the translation from OCaml to Coq is not completely transparent, e.g., it translates the OCaml's fixed-size `int` type to the mathematical integers in Coq. The soundness theorem for CFML has been proved on paper using an idealised semantics for a subset of OCaml. In contrast, our CakeML formalisation of CF models all formal entities in the logic of the proof assistant (HOL4 in our case) and the key theorem, i.e., soundness, is proved as a theorem inside the proof assistant.

1.2 Background on CakeML

The original goal of the CakeML project, as outlined in the first CakeML paper [18], was to provide a fully proof-producing code generation tool (code extraction tool) that given ML-like functions in higher-order logic (HOL) automatically produces equivalent executable machine code. The CakeML *translator* [18] is a proof-producing tool which generates CakeML code from functions in HOL. The output of the translator can then be input into a verified compiler [15,26] that transforms CakeML programs to observationally compatible machine code. The verified CakeML compiler function was bootstrapped in logic using the fully proof-producing work-flow mentioned above [15].

As the compiler is maturing, the focus of CakeML project is shifting to the task of developing a general ecosystem of tools around the CakeML language. This is where CF technology comes into the picture. Our CF formalisation provides a verification framework that enables users to prove correctness theorems for imperative CakeML programs that use any of CakeML's language features, e.g., references, arrays, exceptions and I/O. One can, of course, prove correctness theorems directly over the CakeML semantics. However, such direct proofs would be incredibly tedious for anything but very simple programs.

The formal semantics of the CakeML language is central to its CF framework and the CF framework's soundness proof. Figures 3 and 5 provide some detail of CakeML's operational semantics, which we write in the functional big-step style [20]. Figure 5 shows the definitions of the datatype for the deeply embedded CakeML values that the semantics operates over. Figure 3 shows a few cases of the expression evaluation function evaluate. The figure includes the case of function application App Opapp $[f;\ v]$, i.e., application of expression f to expression v, and shows the semantics, using the helper function do_opapp, of applying a non-recursive Closure value to an argument. For this application, the environment *env* from the Closure value is extended to map the variable

```
evaluate st env [Lit l] = (st,Rval [Litv l])
evaluate st env [Var n] =
  case lookup_var_id n env of
    None ⇒ (st,Rerr (Rabort Rtype_error))
  | Some v ⇒ (st,Rval [v])
evaluate st env [Fun x e] = (st,Rval [Closure env x e])
evaluate st env [App Opapp [f; v]] =
  case evaluate st env [v; f] of
    (st',Rval [v; f]) ⇒
        case do_opapp [f; v] of
          None ⇒ (st',Rerr (Rabort Rtype_error))
        | Some (env',e) ⇒
            if st'.clock = 0 then
              (st',Rerr (Rabort Rtimeout_error))
            else evaluate (dec_clock st') env' [e]
  | res ⇒ res
do_opapp vs =
  case vs of
    [Closure env n e; v] ⇒ Some ((n,v)::env,e)
  | [Recclosure env funs n; v] ⇒ ...
  | _ ⇒ None
```

Fig. 3: An extract of the CakeML semantics.

n to value v. Before evaluation enters the expression from the Closure a clock is checked and decremented, following the style of functional big-step semantics [20]. In the semantics, each function is only applied to one argument at a time.

1.3 A tour of the material

The remainder of this section provides a brief tour of the contributions of this paper: the soundness theorem, our extensions for I/O and exceptions, and our integration of the CakeML CF technology with our existing CakeML proof tools.

We formalise the theorem of soundness of CF with respect to the CakeML semantics. In CFML, the soundness proof is only captured on paper, using idealised semantics for a subset of ML, and the Coq library uses axioms in the places where it would relate to the language semantics. In contrast, we were able to implement an axiom-free CF library for the whole CakeML language, and perform a mechanical proof of soundness, using CakeML's pre-existing semantics.

This not only validates the CF approach introduced by Charguéraud, but also shows that it is flexible as well as extensible. Although CakeML's semantics were not designed with CF in mind, we could directly reuse the CakeML language without any modification, and we were able to carry out the proofs without any particular issue (although some technical details differ from the paper proof).

Moreover, as detailed in Section 3, we could extend the approach to handle new language features that are not supported by CFML.

The soundness theorem, which justifies proving properties about a characteristic formula to give equivalent properties about the program itself, is stated as follows. If the characteristic formula for the deeply embedded expression e (and environment env) holds for some shallowly embedded pre-condition H and shallowly embedded post-condition Q, i.e., $\mathsf{cf}\ e\ env\ H\ Q$, then, starting from a state satisfying H, e is guaranteed to successfully evaluate in CakeML's functional big-step semantics [20], and reach a new state st' and value v satisfying Q. Here state_to_set converts a CakeML state into a representation to which one can apply separation logic connectives, and split asserts disjoint union: $\mathsf{split}\ s\ (s_1,s_2) \iff s_1 \cup s_2 = s \wedge s_1 \cap s_2 = \emptyset$.

$$
\begin{array}{l}
\vdash \mathsf{cf}\ e\ env\ H\ Q \Rightarrow \\
\quad \forall\, st. \\
\quad\quad H\ (\mathsf{state\_to\_set}\ st) \Rightarrow \\
\quad\quad \exists\, st'\ h_f\ h_g\ v\ ck. \\
\quad\quad\quad \mathsf{evaluate}\ (st\ \mathsf{with\ clock}\ :=\ ck)\ env\ [e] = (st', \mathsf{Rval}\ [v]) \wedge \\
\quad\quad\quad \mathsf{split}\ (\mathsf{state\_to\_set}\ st')\ (h_f,h_g) \wedge Q\ v\ h_f
\end{array}
$$

This mechanised proof eliminates the last bits of paper proof that need to be trusted in CFML. Section 2 details the main steps leading to the proof.

We extend the CF framework introduced in CFML to handle two new language features: exceptions, and I/O through CakeML's foreign-function interface (FFI). These extensions are proved sound with respect to the CakeML semantics, and neatly make our framework able to handle all features of the CakeML programming language.[1]

The extension which adds support for I/O is implemented by carefully modifying the state_to_set function, shown in the soundness theorem above. We modified the state_to_set function so that it makes visible the state of the FFI in the pre- and post-conditions. There were numerous tricky details to get right in the definition of state_to_set because the design goal was to *make I/O reasoning local* in the style of separation logic. Our support for I/O is local in that the proof for a piece of code which only uses, say, the print-to-stdout FFI ports does not impose any assumptions on the behaviour, state, or even existence of other FFI ports, e.g., ports for reading-from-stdin. In the spirit of separation logic, our framework allows combining different assertions about the FFI using CF's equivalent to the separation logic frame rule. Section 3 provides details on how we modified state_to_set to make the FFI available in CF proofs.

Support for exceptions is implemented by making the post-conditions differentiate whether the result is a normal return with a value or a value raised as an exception. The new framework is able to reason about exception handling

[1] CakeML's module system is also supported in our CF framework, but supporting modules did not require extending the original ideas of CFML.

code. Section 3.2 explains how exceptions are supported and the effect their introduction had on the proofs.

With these extensions our framework covers all of CakeML's language features and makes it possible to develop a verified standard library for CakeML with complete specifications for library functions that perform I/O or must raise exceptions in certain circumstances. For example, our cat implementation has a routine for opening files, called openIn (whose specification is shown in Figure 4). A call to the CakeML function for openIn raises an exception if the file could not be opened, e.g., if there is no file at the given path. More precisely, inFS_fname *fname fs* describes whether a file exists in *fs* with name *fname*, and the BadFileName exception is raised when no file could be found.

$$
\begin{array}{l}
\vdash \mathsf{FILENAME}\ s\ sv \wedge \mathsf{numOpenFDs}\ fs < 255 \Rightarrow \\
\quad \{|\mathsf{CATFS}\ fs|\} \\
\quad\ \mathsf{openIn\_v} \cdot [sv] \\
\quad \{|\mathsf{POST} \\
\quad\quad (\lambda\, wv. \\
\quad\quad\quad \langle \mathsf{WORD}\ (\mathsf{n2w}\ (\mathsf{nextFD}\ fs))\ wv \wedge \mathsf{validFD}\ (\mathsf{nextFD}\ fs)\ (\mathsf{openFileFS}\ s\ fs) \wedge \\
\quad\quad\quad\ \mathsf{inFS\_fname}\ s\ fs\rangle * \mathsf{CATFS}\ (\mathsf{openFileFS}\ s\ fs)) \\
\quad\quad (\lambda\, e.\ \langle \mathsf{BadFileName\_exn}\ e \wedge \neg \mathsf{inFS\_fname}\ s\ fs\rangle * \mathsf{CATFS}\ fs)|\}
\end{array}
$$

Fig. 4: A specification of the openIn function.

In compiled CakeML code, the actual system call for opening a file is handled by a short stub of C code that is attached to the external side of CakeML's FFI. If an error occurs, the C code signals failure via the return value for the FFI call and, on the CakeML side, the library routine raises the relevant exception on receiving the error code from the C stub. At present, the external C code is unverified and we just make assumptions about its effect on the rest of the world. In the future, we aim to provide verified external assembly stubs that can replace the current unverified C code.

We integrate the CF framework into the CakeML ecosystem by making it interoperate with an existing synthesis tool, namely the automatic translation from HOL functions into CakeML. This tool [18] essentially implements a proof-producing extraction mechanism: given a function in higher-order logic (HOL), the tool generates CakeML code along with a proof that the produced code correctly implements the HOL function with respect to CakeML's semantics. As HOL functions are pure, the translator is essentially limited to producing purely functional CakeML code.[2]

[2] To be precise: Myreen and Owens [18] show that the tool can also be used for production of stateful CakeML code that maintains a hard-coded invariant over a hard-coded number of references. CF allows for much more flexibility.

At present, the most important use of this translation tool is in bootstrapping the verified CakeML compiler, where we now benefit from CF. The translation tool is used to generate CakeML code for the CakeML compiler's implementation. The compiler is defined as functions in HOL, so before we can run the compiler on itself, we need to transform the compiler definition into the source language of the compiler, i.e., CakeML abstract syntax. CF comes into the picture because the translation tool can only produce pure functions. Previously we had to manually verify low-level I/O code that reads the input and passes it to the compiler function, and separate code that prints the result of running the CakeML's compile function. By making the CF and translation tools able to build on each other's results, we have replaced the difficult manual I/O code proofs by understandable CF proofs about I/O.

The bootstrap has thus far benefited from automatic conversion of translator produced results to CF theorems. The bridge between them also works in the other direction: proved results from CF can be used in the translator. Since the translator essentially only deals in pure functions, the CF-verified programs have to implement a pure interface in order to fit the translator. Such programs are not necessarily pure themselves: they can allocate memory, and use imperative structures and algorithms. We plan to make use of the CF-to-translator direction in the future to provide more efficient drop-in replacements for parts of the bootstrapped compiler. These replacement parts would be verified using CF, and replace the code produced by the translator. The register allocator is a particular example that we believe would benefit from using an imperative-style implementation instead of the current automatically generated pure functional implementation.

Section 4 provides details on how we have connected the translation tool and the CakeML CF framework.

All our developments were carried out in the HOL4 theorem prover, and have been integrated into the main CakeML repository. They are available online at `https://cakeml.org` and `https://code.cakeml.org` under the `characteristic` sub-directory.

2 A formal proof of soundness for characteristic formulae

In this section, we explain how CakeML CF differs from CFML, how we avoided axioms in our formalisation, and how we proved soundness of CF for CakeML.

2.1 Adapting CFML to CakeML

A first necessary step towards a proof of soundness was reimplementing the CFML definitions, lemmas and tactics in the CakeML setting. Most of them worked similarly to CFML – in particular the CF definitions and the various tactics (although they are implemented differently). There are however some technical differences worth noting.

```
v =
  Litv lit
| Conv ((string × tid_or_exn) option) (v list)
| Closure (v environment) string exp
| Recclosure (v environment) ((string × string × exp) list) string
| Loc num
| Vectorv (v list)
```

Fig. 5: The CakeML semantic value datatype.

CakeML's semantics uses environments, whereas CMFL assumes substitution semantics. As a consequence, CakeML environments (which map names to semantic values) are threaded through CakeML's characteristic formulae as a new parameter. Environments are accessed in the generated formulae, e.g., the CF for Var x, shown below, returns the value for x in the given environment. Here $\rhd$ is the entailment relation on heap predicates, i.e., $H_1 \rhd H_2$ is true if any heap satisfying H_1 also satisfies H_2, and defined by $p \rhd q \iff \forall s.\ p\ s \Rightarrow q\ s$. The local predicate adds the frame rule of separation logic to the formula.

$$\begin{aligned}&\mathsf{cf}\ (\mathsf{Var}\ \mathit{name})\ \mathit{env} = \\ &\quad \mathsf{local}\ (\lambda H\ Q.\ \exists v.\ \mathsf{lookup\_var\_id}\ \mathit{name}\ \mathit{env} = \mathsf{Some}\ v \wedge H \rhd Q\ v)\end{aligned}$$

In practice, environments are never manipulated explicitly by the user. The user states top-level specifications of the form "$\forall x_i.\ \{\!|H|\!\}\ f \cdot [x_1;\ \ldots;\ x_n]\ \{\!|Q|\!\}$", specifying the behavior of the application of the function value f to some arguments $x_1, \ldots, x_n$. The value f can be fetched given its name as a CakeML function, thanks to a small library that keeps track of top-level definitions.

As f is in fact a closure, the following lemma applies. This lemma, which is a consequence of the CF soundness theorem, turns the goal into proving the CF of the body of f, for the environment that was packed in the closure. Here naryClosure creates a Closure value that takes several curried arguments.

$$\begin{aligned}&\vdash \mathit{ns} \neq [] \Rightarrow \\ &\quad \mathsf{length}\ \mathit{xvs} = \mathsf{length}\ \mathit{ns} \Rightarrow \\ &\quad\quad \mathsf{cf}\ \mathit{body}\ (\mathsf{extend\_env}\ \mathit{ns}\ \mathit{xvs}\ \mathit{env})\ H\ Q \Rightarrow \\ &\quad\quad\quad \{\!|H|\!\}\ \mathsf{naryClosure}\ \mathit{env}\ \mathit{ns}\ \mathit{body} \cdot \mathit{xvs}\ \{\!|Q|\!\}\end{aligned}$$

A custom pretty printer hides the contents of environments from the user. Sub-goals of the form "lookup_var_env x env = v" are always automatically proved by unfolding env.

CF for CakeML uses a deep embedding of CakeML values, while CFML translates ML values to corresponding Coq values. CakeML values are described by the HOL type v (shown in Figure 5), which is defined as part of the semantics.

To relate CakeML values of type v to logical values (such as int, bool, ...), we re-use the *refinement invariants* presented by Myreen and Owens [18] in the

```
if x < 0 then
  print_int (~ x)
else
  print_int x
```

(a) Original program

```
let val _x1 = x < 0 in
  if _x1 then
    (let val _x2 = ~ x in
       print_int _x2
     end)
  else
    print_int x
end
```

(b) Normalised program

Fig. 6: An example of the normalisation pass.

context of a proof-producing translation from HOL functions to CakeML programs. These refinement invariants are a collection of composable predicates that relate HOL types and data structures to the same concepts as deeply embedded CakeML values. The INT and BOOL refinement invariants are defined as follows:

$$\begin{aligned}
&\mathsf{INT}\ i = (\lambda\, v.\ v = \mathsf{Litv}\ (\mathsf{IntLit}\ i)) \\
&\mathsf{BOOL}\ \mathsf{T} = (\lambda\, v.\ v = \mathsf{Conv}\ (\mathsf{Some}\ (\text{“true”}, \mathsf{TypeId}\ (\mathsf{Short}\ \text{“bool”}))))\ []) \\
&\mathsf{BOOL}\ \mathsf{F} = (\lambda\, v.\ v = \mathsf{Conv}\ (\mathsf{Some}\ (\text{“false”}, \mathsf{TypeId}\ (\mathsf{Short}\ \text{“bool”}))))\ [])
\end{aligned}$$

A specification for the CakeML addition function can then be written as follows. Here the angle brackets turn a pure proposition into a heap predicate for heaps represented as sets: $\langle c \rangle = (\lambda\, s.\ s = \emptyset \wedge c)$; and emp is $\langle \mathsf{T} \rangle$.

$$\begin{aligned}
&\vdash \mathsf{INT}\ x_0\ v_0 \wedge \mathsf{INT}\ x_1\ v_1 \Rightarrow \\
&\quad \{|\mathsf{emp}|\}\ \mathsf{plus_v} \cdot [v_0;\ v_1]\ \{|(\lambda\, v.\ \langle \mathsf{INT}\ (x_0 + x_1)\ v \rangle)|\}
\end{aligned}$$

This is somewhat heavier than CFML specifications, where Coq integers would simply be used in place of semantic values. We believe it is hardly an issue for more involved data structures, for which it is common to define such predicates anyway in CFML in order to keep track of additional invariants.

Normalisation of input programs and CF generation are performed in the logic, whereas in CFML they are performed by an external tool. Before being fed to the cf function, programs are normalised in a process similar to A-normalisation. The motivation is that it significantly simplifies formally reasoning about programs, while preserving their semantics. Figure 6 displays an example of the normalisation process. Due to the fact that cf is implemented as a total function in the logic, assumptions about the program being in normal form are made explicit in characteristic formulae. In CFML, the external CF generator simply fails on unhandled input programs.

The cf function assumes that the input program is in normal form. This assumption is reflected by the use of the exp_is_val predicate in characteristic formulae. This predicate, of type `v environment → exp → v option`, checks

whether an expression is in fact a value or a name bound to a value. It is used in characteristic formulae to assert that some expression must be trivial, because of the normalisation pass. For example, the CF for If, below, uses exp_is_val to assert that evaluation of the condition must be dealt with beforehand, by introducing a let-binding, which the normalisation step does. If for some reason the program appears not to be in normal form, the corresponding CF reduces to F.

$$
\begin{array}{l}
\mathsf{cf}\ p\ (\mathsf{If}\ cond\ e_1\ e_2)\ env = \\
\ \mathsf{local} \\
\ \ (\lambda H\ Q. \\
\ \ \ \exists\, condv\ b. \\
\ \ \ \ \mathsf{exp\_is\_val}\ env\ cond = \mathsf{Some}\ condv \wedge \mathsf{BOOL}\ b\ condv \wedge \\
\ \ \ \ ((b \iff \mathsf{T}) \Rightarrow \mathsf{cf}\ p\ e_1\ env\ H\ Q) \wedge \\
\ \ \ \ ((b \iff \mathsf{F}) \Rightarrow \mathsf{cf}\ p\ e_2\ env\ H\ Q))
\end{array}
$$

The sub-goals related to exp_is_val in characteristic formulae are always automatically proved by our CF tactics, and are thus kept hidden from the user.

2.2 Realising CFML axioms

Using CakeML's semantics, we are able to give an implementation of the app predicate, which was axiomatised in CFML.

Let us first consider the semantics of a Hoare triple for an expression e in environment env, denoted $env \vdash \{\!|H|\!\}\ e\ \{\!|Q|\!\}$. We define validity for such a Hoare triple, which we then use to define app. The Hoare triple $env \vdash \{\!|H|\!\}\ e\ \{\!|Q|\!\}$ holds if and only if evaluation of the expression e, in a heap that satisfies the heap predicate H, terminates and produces a value v and a heap satisfying $Q\ v$.

$$
\begin{array}{l}
env \vdash \{\!|H|\!\}\ e\ \{\!|Q|\!\} \iff \\
\ \forall\, st\ h_i\ h_k. \\
\ \ \mathsf{split}\ (\mathsf{state\_to\_set}\ st)\ (h_i,h_k) \Rightarrow \\
\ \ \ H\ h_i \Rightarrow \\
\ \ \ \exists\, v\ st'\ h_f\ h_g\ ck. \\
\ \ \ \ \mathsf{split3}\ (\mathsf{state\_to\_set}\ st')\ (h_f,h_k,h_g) \wedge \\
\ \ \ \ \mathsf{evaluate}\ (st\ \mathsf{with\ clock}\ :=\ ck)\ env\ [e] = (st',\mathsf{Rval}\ [v]) \wedge Q\ v\ h_f
\end{array}
$$

In this definition, split and split3 are used to split a state represented as a set of state elements into disjoint subsets: $\mathsf{split}\ s\ (s_1,s_2) \iff s_1 \cup s_2 = s \wedge s_1 \cap s_2 = \emptyset$; similarly $\mathsf{split3}\ s\ (s_1,s_2,s_3)$ splits a set s into three disjoint subsets s_1, s_2, s_3. This state splitting is here in order to make the frame rule available, as explained further down.

We now define a simple version of app, called app_basic, which characterises the application of a closure to a single argument. When provided a valid function application, where do_opapp can extract the body of the closure and the extended environment, app_basic simply asserts the general Hoare triple defined above. When do_opapp fails, app_basic asserts that the pre-condition H cannot hold of

any state (because otherwise the function application would need to succeed).

$$\begin{array}{l}\{\!|H|\!\}\ f\cdot x\ \{\!|Q|\!\}\ \Longleftrightarrow\\ \ \mathsf{case\ do\_opapp}\ [f;\ x]\ \mathsf{of}\\ \ \ \mathsf{None}\ \Rightarrow\ \forall st\ h_1\ h_2.\ \mathsf{split}\ (\mathsf{state\_to\_set}\ p\ st)\ (h_1,h_2)\Rightarrow \neg H\ h_1\\ \ |\ \mathsf{Some}\ (env,exp)\ \Rightarrow\ env\vdash\ \{\!|H|\!\}\ exp\ \{\!|Q|\!\}\end{array}$$

Finally we define the app predicate, which characterises the application of a closure to multiple arguments, by iterating app_basic.

$$\begin{array}{l}\{\!|H|\!\}\ f\cdot []\ \{\!|Q|\!\}\ \Longleftrightarrow\ \mathsf{F}\\ \{\!|H|\!\}\ f\cdot [x]\ \{\!|Q|\!\}\ \Longleftrightarrow\ \{\!|H|\!\}\ f\cdot x\ \{\!|Q|\!\}\\ \{\!|H|\!\}\ f\cdot x{::}x'{::}xs\ \{\!|Q|\!\}\ \Longleftrightarrow\\ \ \{\!|H|\!\}\ f\cdot x\ \{\!|(\lambda\, g.\ \exists\ H'.\ H'\ast\langle\{\!|H'|\!\}\ g\cdot x'{::}xs\ \{\!|Q|\!\}\rangle)|\!\}\end{array}$$

It is worth noting that our Hoare triple validity integrates the frame rule in its definition. The split predicate (respectively split3) expresses that some heap can be split into two (resp. three) disjoint parts. Therefore, the function application may involve only some subpart of the heap h_i, while the rest h_k is preserved. The function is also allowed to produce some garbage h_g, which is left unconstrained. This is necessary for top-level specifications to be modular, as they are formulated in terms of app.

The built-in frame rule also means that when carrying proofs using the framework, the definition of app is kept abstract and never unfolded. When faced with a "$\{\!|\ldots|\!\}\ f\cdot\ldots\ \{\!|\ldots|\!\}$" goal, a specification for f, also of the form "$\{\!|\ldots|\!\}\ f\cdot\ldots\ \{\!|\ldots|\!\}$" will be fetched and used to prove the goal, either directly or using the frame rule.

2.3 Proving CF soundness

Soundness of characteristic formulae means that, for every expression e, if cf $e\ env\ H\ Q$ holds, then the Hoare triple $env\vdash\ \{\!|H|\!\}\ e\ \{\!|Q|\!\}$ is valid. We define soundness for arbitrary formulae as follows.

$$\mathsf{sound}\ e\ R\ \Longleftrightarrow\ \forall\, env\ H\ Q.\ R\ env\ H\ Q\Rightarrow env\vdash\ \{\!|H|\!\}\ e\ \{\!|Q|\!\}$$

The main result of this section can now be stated. We prove soundness of cf as the following HOL theorem:

Theorem 1 (CF are sound wrt. CakeML semantics).

$$\vdash \mathsf{sound}\ e\ (\mathsf{cf}\ e)$$

Proof. By induction on the size of e.

This proof is most tricky for CakeML language constructs for which characteristic formulae differ significantly from the semantics. The reason is typically

to abstract away from specifics of the semantics, and have proof-friendly characteristic formulae. Two instances of this are closures and pattern matching.

CakeML semantics has closure values. Functions evaluate to closures, and function application is defined in terms of applying a closure to values. The CF for function declaration introduces an abstract value fv, and a specification $\mathcal{H}$ for it. Our formulation differs from that in CFML [6] due to CakeML's use environment semantics instead of CFML's substitution semantics.

$$\begin{array}{l}\mathsf{cf}\ (\mathsf{Let}\ (\mathsf{Some}\ f)\ (\mathsf{Fun}\ x\ e_1)\ e_2)\ env = \\ \quad \mathsf{local}\ (\lambda H\ Q.\ \forall fv.\ \mathcal{H} \Rightarrow \mathsf{cf}\ e_2\ ((f,fv)::env)\ H\ Q) \\ \text{where}\ \mathcal{H} \iff \forall xv\ H'\ Q'.\ \mathsf{cf}\ e_1\ ((x,xv)::env)\ H'\ Q' \Rightarrow \{\!|H'|\!\}\ fv \cdot [xv]\ \{\!|Q'|\!\}\end{array}$$

In the soundness proof, fv is instantiated by a function closure, and one has to prove that $\mathcal{H}$ characterises it.

Proving the soundness of CF for pattern-matching also requires some amount of proof engineering. CakeML semantics provides a logical function that implements a pattern-matching algorithm, and returns whether the match succeeded or not. Characteristic formulae for pattern-matching are instead formulated as nested ifs, which test the equality between the matched value and values produced from the successive patterns.

3 Sound extensions of CF for I/O and exceptions

This section explains how our CF framework has extended the original CFML framework to enable reasoning about I/O and exceptions.

3.1 Support for I/O

As mentioned earlier, the goal of our extension for I/O was to enable convenient *local reasoning* about I/O operations without unreasonable restrictions on the kind of I/O one can verify.

We start with a quick explanation of how I/O is supported in the CakeML language, then show how we made CF pre- and post-conditions able to make assertions about parts of the I/O state, what I/O looks like in the cf function's output, and finally how we have used these techniques in the bootstrapping of the latest CakeML compiler.

The CakeML language supports I/O through a byte-array-based foreign-function interface (FFI). The abstract syntax for CakeML includes an FFI expression. The semantics of executing an FFI expression is to update the state of the FFI which is threaded through the operational semantics together with the state of the CakeML references. The intuition is that CakeML's FFI state component models the state of the outside world and how the outside world will react to any calls made from the CakeML program to the external world.

The formal definition of the FFI state is shown in Figure 7. When designing the CakeML semantics we wanted to make the FFI state as flexible as possible,

so we left the type of the rest of the world as a type variable θ, and we only require that the user provide some oracle function s.oracle that describes how the outside world will react to any FFI call. The FFI state has a s.final_event field that indicates whether the outside world has stopped the process (e.g., due to a call to `exit`). The FFI state also keeps a list of all calls to the FFI (s.io_events): each event records the name of the FFI port[3] that was called, and a list of byte pairs, where map fst of that list is the input to the FFI call and map snd of the list is the state of the array on return from the FFI call.

```
'θ ffi_state =
 <| oracle : (string → 'θ → byte list → 'θ oracle_result);
    ffi_state : 'θ;
    final_event : (final_event option);
    io_events : (io_event list) |>

final_event = Final_event string (byte list) ffi_outcome
ffi_outcome = FFI_diverged | FFI_failed
io_event = IO_event string ((byte × byte) list)
'θ oracle_result = Oracle_return 'θ (byte list) | Oracle_diverge | Oracle_fail
```

Fig. 7: The type for an FFI state in the CakeML operational semantics.

We enable reasoning about I/O in CF by modifying the state_to_set function to expose an image of the FFI state as part of the set representation that the separation logic connectives operate over.

The role of the state_to_set function is to split the state into parts that can be separated using separating conjunction ($*$). For example, a CakeML state s_1 with references at locations 0, 1 and 2 becomes the following. Note that state_to_set can only produce one Mem l _ for each location l in the store.

$$\text{state\_to\_set } s_1 = \{ \text{ Mem } 0\ val_0;\ \text{Mem } 1\ val_1;\ \text{Mem } 2\ val_2;\ \ldots \ \}$$

We can use $p * q = (\lambda\, s.\ \exists\, u\ v.\ \text{split}\ s\ (u,v) \wedge p\ u \wedge q\ v)$ to separate between assertions such as the following. Here Loc l is the value of a reference in the CakeML semantics (Figure 5), and Refv, Varray, and W8array are constructors of the value type for store values.

$$r \rightsquigarrow v = (\lambda\, s.\ \exists\, loc.\ r = \text{Loc } loc \wedge s = \{ \text{ Mem } loc\ (\text{Refv } v) \ \})$$
$$\text{array } r\ vs = (\lambda\, s.\ \exists\, loc.\ r = \text{Loc } loc \wedge s = \{ \text{ Mem } loc\ (\text{Varray } vs) \ \})$$
$$\text{byte\_array } r\ bs = (\lambda\, s.\ \exists\, loc.\ r = \text{Loc } loc \wedge s = \{ \text{ Mem } loc\ (\text{W8array } bs) \ \})$$

With these definitions it follows from $(r_1 \rightsquigarrow v_1 * r_2 \rightsquigarrow v_2 * \ldots)$ (state_to_set s) that $r_1 \neq r_2$ and that updates to reference r_1 do not affect $r_2 \rightsquigarrow v_2 * \ldots$.

[3] We have recently switched to using strings for port names, while numbers were used previously [26] for FFI port names. Johannes Åman Pohjola made this improvement.

The simplest way to make it possible to reason about FFI using CF would be to just make state_to_set produce sets that contain an element that contains the entire current state of the CakeML FFI, i.e., s.ffi_state. However, such a simplistic approach would mean that there can only be one assertion about the state of the FFI in any pre- or post-condition since the assertion could not be split by separating conjunction ($*$). We need to make state_to_set split the FFI state into multiple elements of the state component sets so that we can use the separating conjunction in reasoning about FFI states.

The splitting of the FFI state is non-trivial since we want to keep the FFI state as abstract as possible in the CakeML semantics. The FFI state is modelled by a type variable θ, and thus we know nothing about its structure. Our solution is to parametrise the state_to_set function with information on how to partition an FFI state. The information is a pair consisting of:

- *proj*: a projection function of type $\theta \rightarrow (\texttt{string} \mapsto \texttt{ffi})$, which given an FFI state of type θ returns a finite map from strings to a new type called `ffi`. Here `ffi` is a datatype that is meant to be convenient for modelling projected FFI states in general.[4]

```
ffi =
  Str string
| Num num
| Cons ffi ffi
| List (ffi list)
| Stream (num stream)
```

- *parts*: a list of partitions which are pairs: each pair contains a list of FFI port names (of type `string list`) and a behaviour modelling next-state function, i.e., a representation of part of the oracle function in CakeML's FFI state. The type of the behaviour modelling function is:

```
string → ffi → byte list → (byte list × ffi) option
```

The partitioning information (*proj*,*parts*) is considered well-formed and applicable to an FFI state *st* if:

- the FFI state *st* has not hit a stopping state, i.e., *st*.final_event = None
- no partition has names that overlap with other partitions
- every I/O event has an index that belongs to one of the partitions
- for each partition, *proj* maps all states to the same `ffi` value

[4] We would have liked to use a type variable instead of `ffi`, but a type variable would have been shared between all partitions. Such sharing would need to be done across FFI partitions which goes against the design goal of local specifications and local reasoning. This is a restriction imposed by the HOL type system, which we could have avoided by using a variant of HOL with quantifiers for type variables [13].

– the update function u in each partition respects the FFI's oracle function[5].

The FFI-enabled definition of state_to_set maps CakeML states to the union of the parts of the state that describe the references and the partitioned parts of the FFI state. If the partition for the FFI state is well-defined, then the FFI state is split into a set of FFI_part elements, where each such element carries:

– s, the projected view of the state of this partition
– u, the update function for the partition
– ns, the FFI port names associated with the partition
– ts, a list of all previous I/O events for these names.

We can now make assertions about I/O in CF using state_to_set and separation logic connectives. We define a generic IO assertion as follows.

$$\mathsf{IO}\ st\ u\ ns = (\lambda\, s.\ \exists\, ts.\ s = \{\ \mathsf{FFI\_part}\ st\ u\ ns\ ts\ \})$$

With these we can make assertions about I/O. For example, the following asserts that the projected FFI state must have a part that is described by FFI_part $s_1\ u_1\ [n]$, and a disjoint part that is described by FFI_part $s_2\ u_2\ ns$.

$$(\mathsf{IO}\ s_1\ u_1\ [n] * \mathsf{IO}\ s_2\ u_2\ ns * \dots)\ (\mathsf{state\_to\_set}\ pp\ st)$$

Using such statements in their pre- and post-conditions, the user may express strong specifications concisely.

The following proof obligation is generated every time the cf function is applied to the abstract syntax for an FFI expression. This proof obligation can be read as follows: pre-condition H must imply that there is a byte array and I/O partition in the state. The I/O partition must include the *name* of the called FFI entry point. Furthermore, the result of running the next-state function from the FFI partition, i.e., u, must successfully return a new state s' and this state and the updated byte array must imply the desired post-condition Q. FFI calls return unit_value.

$$\begin{array}{l}
\mathsf{cf}\ pp\ (\mathsf{App}\ (\mathsf{FFI}\ name)\ [array])\ env = \\
\ \mathsf{local} \\
\ \ (\lambda\, H\ Q. \\
\ \ \ \exists\, rv. \\
\ \ \ \ \mathsf{exp_is_val}\ env\ array = \mathsf{Some}\ rv \wedge \\
\ \ \ \ \exists\, bs\ F\ bs'\ s\ s'\ u\ ns. \\
\ \ \ \ \ u\ name\ bs\ s = \mathsf{Some}\ (bs',s') \wedge \mathsf{mem}\ name\ ns \wedge \\
\ \ \ \ \ H \rhd F * \mathsf{byte_array}\ rv\ bs * \mathsf{IO}\ s\ u\ ns \wedge \\
\ \ \ \ \ F * \mathsf{byte_array}\ rv\ bs' * \mathsf{IO}\ s'\ u\ ns \rhd Q\ \mathsf{unit_value})
\end{array}$$

[5] Our use of projection functions and updates at both a concrete and abstract view of the state bears some resemblance to lenses [21]. Note however that lenses must have *get* and *putback* functions. Our set up lacks the *putback* functions, i.e., we only project in one direction. Our initial formalisation had a *putback* function, but we decided to simplify the definitions and arrived at the current solution with only a *get* function, which we call *proj*.

The proof goal produced by cf mentions IO, which from the user's perspective is the primitive I/O assertion in CakeML CF. Users define their own specialisations of IO for each application, see Section 5.

This support for I/O has, together with the connection between CF and the CakeML translator (Section 4), been used to verify the I/O code required for giving input and producing output from the bootstrapped CakeML compiler. The I/O code is a little snippet of code that wraps around the translator-generated pure CakeML code which implements the logical CakeML compile function.

3.2 Support for exceptions

We implement complete support for specifying CakeML programs that use exceptions. Up to this point, we required expressions to evaluate and reduce to a value: post-conditions were of type `v → heap → bool`, taking the returned value as an argument. We now allow expressions to raise an exception instead: we define a `res` datatype `res = Val v | Exn v`, and change the type of post-conditions to be `res → heap → bool`. We define some wrappers for writing post-conditions, in particular for the cases where the expression never (resp. always) raises an exception. POST handles both cases by taking one post-condition for each case.

$$\begin{array}{l}
(\mathsf{POSTv})\ Q_v = (\lambda\, r.\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{Val}\ v\ \Rightarrow\ Q_v\ v \mid \mathsf{Exn}\ e\ \Rightarrow\ \langle \mathsf{F} \rangle) \\
(\mathsf{POSTe})\ Q_e = (\lambda\, r.\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{Val}\ v\ \Rightarrow\ \langle \mathsf{F} \rangle \mid \mathsf{Exn}\ e\ \Rightarrow\ Q_e\ e) \\
\mathsf{POST}\ Q_v\ Q_e = (\lambda\, r.\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{Val}\ v\ \Rightarrow\ Q_v\ v \mid \mathsf{Exn}\ e\ \Rightarrow\ Q_e\ e)
\end{array}$$

We update the definitions that relate CF to CakeML semantics. For example, the definition of Hoare triple validity we presented earlier contains:

$$\ldots \wedge \mathsf{evaluate}\ (st\ \mathsf{with\ clock}\ :=\ ck)\ env\ [exp] = (st', \mathsf{Rval}\ [v])$$

The second component returned by evaluate, of which Rval is a constructor, is of type `(v list, v) result`, where:

$$\begin{array}{l}
(\alpha,\ \beta)\ \texttt{result}\ =\ \mathsf{Rval}\ \alpha \mid \mathsf{Rerr}\ (\beta\ \texttt{error_result}) \\
\alpha\ \texttt{error_result}\ =\ \mathsf{Rraise}\ \alpha \mid \mathsf{Rabort}\ \texttt{abort}
\end{array}$$

This gives us two other cases: Rerr (Rraise *exn*) for expressions that raise an exception, and Rerr (Rabort *cause*) for expressions that fail to evaluate. We still rule out the latter, but add support for the former: the definition of Hoare triple validity becomes:

$$\begin{array}{l}
env \vdash \{\!|H|\!\}\ e\ \{\!|Q|\!\}\ \Longleftrightarrow \\
\quad \forall\, st\ h_i\ h_k. \\
\quad\ \mathsf{split}\ (\mathsf{state_to_set}\ p\ st)\ (h_i,h_k)\ \Rightarrow \\
\quad\quad H\ h_i\ \Rightarrow \\
\quad\quad\ \exists\, r\ st'\ h_f\ h_g\ ck. \\
\quad\quad\quad \mathsf{split3}\ (\mathsf{state_to_set}\ p\ st')\ (h_f,h_k,h_g) \wedge Q\ r\ h_f\ \wedge \\
\quad\quad\quad \mathsf{case}\ r\ \mathsf{of} \\
\quad\quad\quad\quad \mathsf{Val}\ v\ \Rightarrow\ \mathsf{evaluate}\ (st\ \mathsf{with\ clock}\ :=\ ck)\ env\ [e] = (st', \mathsf{Rval}\ [v]) \\
\quad\quad\quad\quad \mid \mathsf{Exn}\ v\ \Rightarrow\ \mathsf{evaluate}\ (st\ \mathsf{with\ clock}\ :=\ ck)\ env\ [e] = (st', \mathsf{Rerr}\ (\mathsf{Rraise}\ v))
\end{array}$$

We update the existing CF definitions as well. We add side-conditions to deal with exceptions; for example the CF for Let handles the case where an exception is raised by the first expression.

$$
\begin{array}{l}
\mathsf{cf}\ p\ (\mathsf{Let}\ (\mathsf{Some}\ x)\ e_1\ e_2)\ env = \\
\ \ \mathsf{local} \\
\ \ \ \ (\lambda H\ Q. \\
\ \ \ \ \ \ \exists Q'. \\
\ \ \ \ \ \ \ \ \mathsf{cf}\ p\ e_1\ env\ H\ Q' \wedge Q' \blacktriangleright_{\mathrm{e}} Q \wedge \\
\ \ \ \ \ \ \ \ \forall xv.\ \mathsf{cf}\ p\ e_2\ ((x,xv)::env)\ (Q'\ (\mathsf{Val}\ xv))\ Q)
\end{array}
$$

This uses the entailment relation on post-conditions for the exception case, written $Q_1 \blacktriangleright_{\mathrm{e}} Q_2$, and defined as $\forall e.\ Q_1\ (\mathsf{Exn}\ e) \rhd Q_2\ (\mathsf{Exn}\ e)$. On exceptions, the post-condition for e_1 (Q') has to directly entail validity of the post-condition for the whole formula (Q), since e_2 does not get executed in case e_1 raises an exception.

Some other side-conditions are not needed for establishing the soundness theorem, but are added to enforce a "no garbage" property on post-conditions. For example, the CF for Var becomes as follows, where **F** is a post-condition false for any value and any heap:

$$
\begin{array}{l}
\mathsf{cf}\ p\ (\mathsf{Var}\ name)\ env = \\
\ \ \mathsf{local} \\
\ \ \ \ (\lambda H\ Q. \\
\ \ \ \ \ \ (\exists v.\ \mathsf{lookup\_var\_id}\ name\ env = \mathsf{Some}\ v \wedge H \rhd Q\ (\mathsf{Val}\ v)) \wedge \\
\ \ \ \ \ \ Q \blacktriangleright_{\mathrm{e}} \mathbf{F})
\end{array}
$$

This requires Q to be false on exceptions, as evaluating a Var x always produces a value on well scoped code. We believe having such side-conditions make the following proposition true (and plan to prove it as future work): if the CF for e is true for pre-condition H and post-condition Q, then $Q \blacktriangleright_{\mathrm{e}} \mathbf{F}$ if and only if e does not raise exceptions.

We update the existing tactics, so that easy side-conditions are automatically proved. We rely on the following lemma:

$$\vdash (\mathsf{POSTv})\ Q_v \blacktriangleright_{\mathrm{e}} Q$$

This is trivially true, as (POSTv) Q_v (Exn e) unfolds to $\langle \mathsf{F} \rangle$. Thanks to this lemma, carrying out proofs about programs that do not involve exceptions requires no additional effort. The only modification necessary is changing the "$\lambda\, v. \ldots$" to "POSTv $v. \ldots$" in post-conditions.

Finally, we handle CakeML's primitives for exception handling, Raise and Handle, whose semantics match SML's. Here $(f \mathbin{\#\#} g)\ (x,y) = (f\ x, g\ y)$.

$$\mathsf{cf}\ p\ (\mathsf{Raise}\ e)\ env = \mathsf{local}\ (\lambda H\ Q.\ \exists v.\ \mathsf{exp\_is\_val}\ env\ e = \mathsf{Some}\ v \wedge H \rhd Q\ (\mathsf{Exn}\ v) \wedge Q \blacktriangleright_{\mathrm{v}} \mathbf{F})$$

$$\begin{array}{l}\mathsf{cf}\ p\ (\mathsf{Handle}\ e\ rows)\ env = \\ \quad \mathsf{local} \\ \quad\quad (\lambda H\ Q. \\ \quad\quad\quad \exists Q'. \\ \quad\quad\quad\quad \mathsf{cf}\ p\ e\ env\ H\ Q' \wedge Q' \blacktriangleright_{\mathrm{v}} Q \wedge \\ \quad\quad\quad\quad \forall ev. \\ \quad\quad\quad\quad\quad \mathsf{cf\_cases}\ ev\ ev\ (\mathsf{map}\ (\mathsf{I} \mathbin{\#\#} \mathsf{cf}\ p)\ rows)\ env\ (Q'\ (\mathsf{Exn}\ ev))\ Q)\end{array}$$

The entailment relation on post-conditions for the value case, written $Q \blacktriangleright_{\mathrm{v}} Q_2$, is without surprise defined as $\forall v.\ Q_1\ (\mathsf{Val}\ v) \rhd Q_2\ (\mathsf{Val}\ v)$. The CFs for Raise and Handle resemble the CFs for Var and Let respectively, but with the respectives roles of exceptions and values swapped. The cf_cases auxiliary definition corresponds to the CF for pattern-matching.

Let us present an illustrative example. The cat program presented earlier in Figure 1 doesn't do any exception handling, and for simplicity its specification (Figure 2) requires that all input filenames represent existing files. In this way, our specification above only specifies the non-exceptional behaviour. Nonetheless, the various I/O primitives can be modeled so as to allow the possibility that they might raise various exceptions, and when they are, we can prove more detailed post-conditions capturing those behaviours.

We define a simple cat1exn program that handles invalid filenames. It is implemented as shown in Figure 8, by calling cat1 and handling the CharIO.BadFileName exception that may be raised.

```
fun cat1exn fname =
  cat1 fname handle CharIO.BadFileName ⇒ ()
```

Fig. 8: Code displaying the contents of a single file.

Figure 9 shows the specification of cat1, and Figure 10 shows the specification we prove for cat1exn. It relies on the catfile_string function, which corresponds to the text displayed by cat1exn, and is defined as:

$$\mathsf{catfile\_string}\ fs\ fnm = \mathsf{if}\ \mathsf{inFS\_fname}\ fnm\ fs\ \mathsf{then}\ \mathsf{file\_contents}\ fnm\ fs\ \mathsf{else}\ []$$

Proving the specification for cat1exn boils down to proving three subgoals, corresponding to the three conjunctions appearing in the Handle case of cf. The

```
⊢ FILENAME fnm fnv ∧ numOpenFDs fs < 255 ⇒
  {|CATFS fs ∗ STDOUT out|}
   cat1_v · [fnv]
  {|POST
    (λ u.
      ∃ content.
       ⟨UNIT () u⟩ ∗ ⟨alist_lookup fs.files fnm = Some content⟩ ∗
       CATFS fs ∗ STDOUT (out @ content))
    (λ e.
      ⟨BadFileName_exn e⟩ ∗ ⟨¬inFS_fname fnm fs⟩ ∗ CATFS fs ∗
      STDOUT out)|}
```

Fig. 9: A specification for cat1, which outputs the contents of a file on standard out, or raises an exception if the file could not be found.

```
⊢ FILENAME fnm fnmv ∧ numOpenFDs fs < 255 ⇒
  {|CATFS fs ∗ STDOUT out|}
   cat1_v · [fnmv]
  {|POSTv u.
    ⟨UNIT () u⟩ ∗ CATFS fs ∗
    STDOUT (out @ catfile_string fs fnm)|}
```

Fig. 10: A specification for cat1exn, which will not raise the BadFileName exception.

first one is trivially solved using the appropriate tactic. The second one requires proving that the post-condition of cat1 entails the post-condition of cat1exn, for the value case. This is true, using a lemma proving that inFS_fname *fnm fs* holds if the file could be found with some content in the file system. The last goal finally requires proving that the file system *fs* is unchanged in the exception case. Knowing ¬inFS_fname *fnm fs*, this is proved by unfolding catfile_string.

4 Interoperating with the CakeML translator

We prove an equivalence result between the theorems produced by the translator, and a particular shape of CF specifications.

Called on a function succ of type `int → int`, the translator will produce a CakeML program succ_ml, and the following theorems. The theorems state that: running the succ_ml program results in an environment, succ_env, in which looking up the variable "succ" yields a value succ_v, and finally that this value

implements the function succ.

$$\begin{array}{l}\vdash \mathsf{run\_prog\ succ\_ml\ succ\_env}\\ \vdash \mathsf{lookup\_var}\ \text{“succ”}\ \mathsf{succ\_env} = \mathsf{Some\ succ\_v}\\ \vdash (\mathsf{INT} \longrightarrow \mathsf{INT})\ \mathsf{succ\ succ\_v}\end{array}$$

We are here mostly interested in the last theorem, expressed using the “arrow” predicate, “$(a \longrightarrow b)\ f\ fv$”, which relates the HOL function f to the closure fv. It states that for any argument xv satisfying $a\ x$, evaluating the closure produces a value u satisfying $b\ (f\ x)$. Formally:

$$\begin{array}{l}(a \longrightarrow b)\ f\ fv \iff\\ \quad \forall x\ xv\ refs.\\ \quad\quad a\ x\ xv \Rightarrow\\ \quad\quad\quad \exists env\ exp\ refs'\ u\ c.\\ \quad\quad\quad\quad \mathsf{do\_opapp}\ [fv;\ xv] = \mathsf{Some}\ (env,exp) \wedge\\ \quad\quad\quad\quad \mathsf{evaluate}\ (\mathsf{empty\_state\ with\ <|clock}\ :=\ c;\ \mathsf{refs}\ :=\ refs\ \mathsf{|>})\ env\ [exp] =\\ \quad\quad\quad\quad\quad (\mathsf{empty\_state\ with\ <|clock}\ :=\ 0;\ \mathsf{refs}\ :=\ refs\ @\ refs'\ \mathsf{|>,Rval}\ [u]) \wedge\\ \quad\quad\quad\quad b\ (f\ x)\ u\end{array}$$

This is reminiscent of the app_basic predicate used in CF, and indeed we prove that “arrow” is a special case of app_basic.

The CF specifications we prove equivalent to “arrow” are of the form $\{\!|\mathsf{emp}|\!\}\ f\ \cdot\ x\ \{\!|\mathsf{POSTv}\ v.\ \langle P\ v\rangle|\!\}$, where P is some logical predicate of type `v → bool`. A pure function does not raise exceptions, hence the post-condition is false for exceptions. Both the pre- and post-condition assert emptiness of the heap.

A function f satisfying such a spec can still be called on any heap, thanks to the frame rule built into the CF framework. The specification simply means that the function cannot assume anything about the heap, or access it. Less obviously, this kind of specification allows the function to allocate heap objects (references, arrays, ...) for internal use. This becomes apparent after unfolding the definition of Hoare triple validity that underlies app_basic (which we recall below).

$$\begin{array}{l}env \vdash\ \{\!|H|\!\}\ exp\ \{\!|Q|\!\} \iff\\ \quad \forall st\ h_i\ h_k.\\ \quad\quad \mathsf{split}\ (\mathsf{state\_to\_set}\ p\ st)\ (h_i,h_k) \Rightarrow\\ \quad\quad\quad H\ h_i \Rightarrow\\ \quad\quad\quad\quad \exists r\ st'\ h_f\ h_g\ ck.\\ \quad\quad\quad\quad\quad \mathsf{split3}\ (\mathsf{state\_to\_set}\ p\ st')\ (h_f,h_k,h_g) \wedge Q\ r\ h_f \wedge \ldots\end{array}$$

The final heap “state_to_set $p\ st'$” is split in three sub-heaps: h_f, h_k and h_g. The post-condition must be true on h_f, and h_k was present in the initial heap and is unchanged. There remains h_g, which represents heap objects that may have been allocated by the function and now need to be garbage collected. Consequently, even though such specifications require the function f to offer a pure interface, it

is not necessarily pure itself: it can be implemented using imperative structures and algorithms.

The exact equivalence theorem we prove is as follows:

$$\vdash (a \longrightarrow b)\ f\ fv \iff \forall x\ xv.\ a\ x\ xv \Rightarrow \{|\mathsf{emp}|\}\ fv \cdot xv\ \{|\mathsf{POSTv}\ v.\ \langle b\ (f\ x)\ v\rangle|\}$$

The arrow-to-app_basic direction is the easiest to prove. With the right automation, it allows programs certified using CF to use programs produced by the translator, and automatically retrieve their specification. The app_basic-to-arrow direction is significantly trickier. It required changing the definition of "arrow" to allow heap allocation (represented by *refs′* earlier), and subsequent updating of the translator. Moreover, the proof itself involved careful reasoning about the state of the FFI. This direction makes it possible to provide programs certified using CF as drop-in replacements for translated functions.

5 Case study: a verified `cat` implementation

Our case study builds a simple model coupling a read-only file-system with one standard output stream. The type of the read-only file-system is

```
RO_fs =
  <| files : ((mlstring × byte list) list);
     infds : ((num × mlstring × num) list) |>
```

The files and infds fields are association lists. The files field maps file names to file contents. The infds field maps file descriptors (numbers) to pairs of file names and offsets within that file. File names are of type `mlstring`; in CakeML, these map to vectors of characters occupying contiguous blocks of memory. This model supports multiple descriptors reading from a common file at different positions, and is also subject to realistic problems such as the possibility of file descriptors becoming stale.

The four file-system operations needed for our example are openFile, eof, read1, and closeFD. At this initial stage, we can define the type and its operations in a natural style, concerning ourselves only with the logical model, and not needing to worry about its realisation in the CF framework. (One exception is the use of association lists; it would be more natural to use finite maps, but we must ultimately encode our values into the `ffi` type presented on page 15.)

Making a model of this sort visible within the CF framework then requires us to cast the operations as messages being sent using single, fixed-size buffers (a mutable array of bytes, to be precise). For example, when accessed from CakeML, the read1 operation must begin by writing the file descriptor value into such a buffer. The same buffer is then used to store the return value. If the file descriptor passed to read1 is not valid, or if the file descriptor has come to the end of file, the error-condition must be returned using the same buffer.

We choose to use a one-byte buffer in the case of read1, partly because it is simple, but also because it naturally leads to realistic "misfeatures": bad inputs

cause a −1 return code, which must be returned "in-band". To know whether or not this is genuine, the client has to call the eof test first.

The final part of the process requires us to write CakeML wrappers that make calls through the FFI. The wrapper code for read1, using the one-byte buffer onechar, is presented in Figure 11.

```
fun read1 fd =
  let val eofp = eof fd in
    if eofp then NONE
    else
      let val _ = Word8Array.update onechar 0 fd
          val _ = FFI "read1" onechar
          val c = Word8Array.sub onechar 0
      in
          SOME c
      end
  end
```

Fig. 11: CakeML code implementing read1. For the purposes of simplicity this does not catch the error possible when the argument fd is not valid; rather the specification we use imposes "fd-validity" as a pre-condition. By using the eof function, the code *does* allow for the successful return of any character, including character 255 (−1).

We now have a piece of CakeML abstract syntax given the name read1, as well as a logical function of the same name operating over values of type RO_fs. We make the logical RO_fs values visible to the CF framework by lifting them into the language of assertions over I/O-extended heaps, using the IO function defined on page 16. The CATFS predicate is of type RO_fs → hprop. A proposition CATFS *fs* asserts that the state of the external file-system is as given by the logical value *fs*.

Our specification for read1 is given in Figure 12. When this, and the specifications for the other entry-points have been proved, the verification of cat1 and then cat (see Figures 1 and 2) proceeds quite straightforwardly. In particular, the low-level specifications ensure that the proofs are oblivious to the fact that I/O through the FFI is involved; instead, they proceed just as if the state of the file-system was a part of memory. The proof of cat1 is by induction on the length of the file still to be read; that of cat by induction on the list of arguments.

6 Discussion of related work

The CakeML projects aims to build an extensive ecosystem of verification tools around the CakeML programming language. By adapting CF techniques to the

$$
\begin{array}{l}
\vdash \mathsf{WORD}\ fdw\ fdv \wedge \mathsf{validFD}\ (\mathsf{w2n}\ fdw)\ fs \Rightarrow \\
\quad \{|\mathsf{CATFS}\ fs|\} \\
\quad\ \mathsf{read1\_v} \cdot [fdv] \\
\quad \{|\mathsf{POSTv}\ coptv. \\
\quad\ \langle \mathsf{OPTION\ WORD}\ (\mathsf{FDchar}\ (\mathsf{w2n}\ fdw)\ fs)\ coptv \rangle * \\
\quad\ \mathsf{CATFS}\ (\mathsf{bumpFD}\ (\mathsf{w2n}\ fdw)\ fs)|\}
\end{array}
$$

Fig. 12: The specification of read1. The read1_v value is the closure defined by the abstract-syntax for read1. The function FDchar returns the current character designated by the given file descriptor, if any; the function bumpFD increments the position of the file descriptor within its file. At the ML level, file descriptors are encoded as bytes, but the underlying model for file-system uses natural numbers. This is why the logic of the specification coerces from one to the other with w2n. This is also what causes the pre-condition in openFile's specification (Figure 4) requiring that not too many files be open already.

setting of CakeML, this paper has extended the toolset and, at the same time, validated some of the pen-and-paper proofs of prior word on CF. Prior work on CF and CakeML is discussed in Section 1.1 and 1.2.

In this section we discuss other verification projects that build ecosystems of verification tools around and within theorem provers such as HOL4, Isabelle/HOL, Coq, Nqthm and ACL2.

In the Isabelle/HOL theorem prover, a substantial ecosystem of verification technology has been developed around the Simpl framework by Schirmer [23], which is an extensible framework for Hoare logic over imperative programs. Simpl played a central role in the seL4 micro-kernel verification [14], where the C code was verified using Simpl. Later, a tool called AutoCorres by Greenaway [12] was developed for automatically lifting C programs written in Simpl into more-convenient-to-verify monadic functions in the logic. The AutoCorres tool and Simpl were subsequently used in the recent proof-producing Cogent compiler [19] for its translation validation step. The Cogent compiler compiles a by-design restrictive functional language to C and produces a correctness theorem in Isabelle/HOL for each compiler run.

The Isabelle Refinement Framework by Lammich [17,16] is a recent set of tools for producing verified code using the Isabelle/HOL prover. In this work, the Sepref tool can synthesise concrete code from high-level descriptions of imperative algorithms and data structures. Lammich's work takes a top-down path, in contrast to CF and AutoCorres, and the final translation from code in Isabelle/HOL to code running outside the prover is not proved correct w.r.t. any formal semantics of the target programming language.

In the context of Coq, the Bedrock project [9] lead by Chlipala has developed an impressive ecosystem around a separation-logic-inspired Hoare logic for low-level code. Bedrock connects to FIAT [11], which is a set of tools for performing refinement from high-level declarative specifications to concrete im-

plementations. This technology has been applied to complicated examples such as a web server, database applications and even file systems [8].

The Verified Software Toolchain (VST) [2] from Princeton is another substantial verification framework in Coq. VST defines a C-like language, provides a separation logic on top of this C-like language and maps it into the CompCert C compiler, with proof in Coq relating properties proved at the top to the assembly that CompCert produces.

The CertiCoq project [1] also from Princeton aims to build a proof-producing code extraction mechanism for Coq, which will essentially do for Coq what CakeML's translator and CakeML compiler already does for HOL.

The Nqthm theorem prover hosted a project in this area that was two or three decades ahead of the field: the "CLI stack" project [3] developed a substantial verification toolchain with a verification-friendly programming language supported by a verified compiler, which targetted a machine language for which the project developed a verified hardware implementation. The logic of the Nqthm prover is a pure first-order functional language but the input language of the verified compiler is not functional.

The recent F* project [25] develops a new dependently-typed monadic language with refinement types. One can use F*'s expressive types to verify programs written in F*. Users can have extra confidence in the results since the typechecker for F* has been verified using Coq [24]. Programs developed in F* can be extracted to OCaml for compilation and execution.

There are many other functional languages with type-systems that allow verification using types. Ynot is one that has been re-implemented in Coq [10].

There are numerous verification ecosystem without connections to the above mentioned theorem provers. Most of these other ecosystems only consider imperative programs. HALO is one such system that applies to functional programs [28]. HALO enables verification of contracts for Haskell programs and uses first-order provers in its implementation.

7 Summary

In this paper, we have explained how to build a fully verified CF framework for the entirety of the CakeML language. We have shown how to add support for I/O and exceptions, as well as interoperability with the CakeML tool used for bootstrapping the verified CakeML compiler.

At a higher level, one can read this paper as a validation that Charguéraud's original work on CF is flexible as well as extensible.

Acknowledgements. We thank Arthur Charguéraud for advice on characteristic formulae. We thank Mike Gordon and Thomas Sewell for commenting on drafts of this paper. The second author was partially supported by the Swedish Research Council.

References

1. A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017. URL: `http://conf.researchr.org/event/CoqPL-2017/main-certicoq-a-verified-compiler-for-coq`.
2. A. W. Appel. Verified software toolchain. In G. Barthe, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2011. `doi:10.1007/978-3-642-19718-5_1`.
3. W. R. Bevier, W. A. H. Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4), 1989. `doi:10.1007/BF00243131`.
4. A. Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris-Diderot, 2010. URL: `http://arthur.chargueraud.org/research/2010/thesis/`.
5. A. Charguéraud. Program verification through characteristic formulae. In P. Hudak and S. Weirich, editors, *International Conference on Functional programming (ICFP)*. ACM, 2010.
6. A. Charguéraud. Characteristic formulae for the verification of imperative programs. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *International Conference on Functional Programming (ICFP)*. ACM, 2011. `doi:10.1145/2034773.2034828`.
7. A. Charguéraud and F. Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In Urban and Zhang [27]. `doi:10.1007/978-3-319-22102-1_9`.
8. H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In A. Gulati and H. Weatherspoon, editors, *USENIX Annual Technical Conference*. USENIX Association, 2016.
9. A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In G. Morrisett and T. Uustalu, editors, *International Conference on Functional Programming (ICFP)*. ACM, 2013. URL: `http://doi.acm.org/10.1145/2500365.2500592`, `doi:10.1145/2500365.2500592`.
10. A. Chlipala, J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In G. Hutton and A. P. Tolmach, editors, *International conference on Functional programming (ICFP)*. ACM, 2009. URL: `http://doi.acm.org/10.1145/1596550.1596565`, `doi:10.1145/1596550.1596565`.
11. B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In S. K. Rajamani and D. Walker, editors, *Principles of Programming Languages (POPL)*. ACM, 2015. URL: `http://doi.acm.org/10.1145/2676726.2677006`, `doi:10.1145/2676726.2677006`.
12. D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 429–439, Edinburgh, UK, June 2014. ACM. `doi:10.1145/2594291.2594296`.
13. P. V. Homeier. The HOL-Omega logic. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS. Springer, 2009.

14. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct. 2009. ACM.
15. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *Principles of Programming Languages (POPL)*, 2014. doi:10.1145/2535838.2535841.
16. P. Lammich. Refinement to Imperative/HOL. In Urban and Zhang [27]. doi:10.1007/978-3-319-22102-1_17.
17. P. Lammich. Refinement based verification of imperative data structures. In J. Avigad and A. Chlipala, editors, *Conference on Certified Programs (CPP)*. ACM, 2016. URL: http://dl.acm.org/citation.cfm?id=2854065, doi:10.1145/2854065.2854067.
18. M. O. Myreen and S. Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3), 2014. doi:10.1017/S0956796813000282.
19. L. O'Connor, Z. Chen, C. Rizkallah, S. Amani, J. Lim, T. C. Murray, Y. Nagashima, T. Sewell, and G. Klein. Refinement through restraint: bringing down the cost of verification. In J. Garrigue, G. Keller, and E. Sumii, editors, *International Conference on Functional Programming (ICFP)*. ACM, 2016. URL: http://doi.acm.org/10.1145/2951913.2951940, doi:10.1145/2951913.2951940.
20. S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In P. Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2016. doi:10.1007/978-3-662-49498-1_23.
21. B. C. Pierce. The weird world of bi-directional programming, Mar. 2006. ETAPS invited talk.
22. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*. IEEE Computer Society, 2002.
23. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universitat Munchen, 2006. URL: http://arthur.charguéraud.org/research/2010/thesis/.
24. P. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: bootstrapping certified typecheckers in F* with Coq. In J. Field and M. Hicks, editors, *Principles of Programming Languages (POPL)*. ACM, 2012. URL: http://doi.acm.org/10.1145/2103656.2103723, doi:10.1145/2103656.2103723.
25. N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F. In R. Bodík and R. Majumdar, editors, *Principles of Programming Languages (POPL)*. ACM, 2016. URL: http://doi.acm.org/10.1145/2837614.2837655, doi:10.1145/2837614.2837655.
26. Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A new verified compiler backend for CakeML. In *International Conference on Functional Programming (ICFP)*. ACM Press, 2016.
27. C. Urban and X. Zhang, editors. *Interactive Theorem Proving (ITP)*, volume 9236 of *LNCS*. Springer, 2015.
28. D. Vytiniotis, S. L. P. Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. In R. Giacobazzi and R. Cousot, editors, *Principles of Programming Languages (POPL)*. ACM, 2013. URL: http://doi.acm.org/10.1145/2429069.2429121, doi:10.1145/2429069.2429121.