End-to-End Verification with CakeML

Hugo Férée[1], Johannes Åman Pohjola[2], Ramana Kumar[3], Scott Owens[1], Magnus O. Myreen[2], and Son Ho[4]

[1] School of Computing, Univerity of Kent, UK
[2] CSE Department, Chalmers University of Technology, Sweden
[3] Data61, CSIRO / UNSW, Australia
[4] École Polytechnique, France

Abstract. Software verification tools that build machine-checked proofs of functional correctness usually focus on the algorithmic content of the code. Their proofs are not grounded in a formal semantic model of the hardware and operating system that will ultimately run the program. As a result, several layers of translation and wrapper code must be trusted. In contrast, the CakeML project focusses on end-to-end verification to replace this trusted code with verified code in a cost-effective manner.
In this paper, we show how to prove functional correctness at the binary level for real executables that do I/O and file handling, with only a little more effort than an algorithmic-level proof. Specifically, we extend the CakeML ecosystem with a low-level model of file I/O, and use this ecosystem — including its verified compiler, proof-producing synthesis tool, and verified libraries — to verify several Un*x-style command-line utilities: echo, cat, sort, grep, and diff/patch. The workflow we present is built around the HOL4 theorem prover, and therefore all our results have machine-checked proofs.

1 Introduction

When explaining a formally verified software system, we must be clear about what exactly we are verifying. For example, we might be verifying an algorithm at an abstract level, apart from the semantics of any particular programming language or CPU that will run the algorithm, or we might be verifying some machine code that implements it. One way to understand what has been verified, and where there might still be bugs, is to closely inspect the top-level theorem that has been proved, and the assumptions that it makes about the environment that the verified system operates in. The hardware and software that must meet those assumptions forms the trusted computing base (TCB) along with the verification technology itself. We want fewer assumptions and a smaller TCB, but not at any cost: verification is still difficult enough that we need to prioritise verifying the parts of the system where the bugs are most likely to be.

In this paper, we show how to verify impure functional programs in CakeML[5] [16] at the algorithmic level, then, with very little additional effort, arrive at a small-TCB correctness theorem that makes assumptions only about the correctness

[5] `https://cakeml.org/`

of our machine code semantics, and the operating system's I/O system calls. Everything else is handled by our proof-producing translation [22], our verified compiler [30], and our new verified CakeML I/O routines. Thus, we maintain the convenience that one expects from a shallower verification effort.

Explicitly, our TCB is the HOL4 theorem proving system[6]; a simple Standard ML program that writes the compiled bytes of machine code into a file; the linker that produces the executable; the loading and I/O facilities provided by the operating system – *not* including libc – and our model of making I/O system calls over our foreign function interface (FFI); and our machine code semantics. Together, these constitute the whole formalisation gap. Notably, we do not need to trust any code extraction procedure standing between the verified model of each application and its code-level implementation, nor do we need to trust the compiler and runtime system that bridge between source code and machine code.

Step-by-step, this is our approach:

1. We write the pure (side-effect free) parts of our program as shallowly embedded functions in the HOL4 theorem prover. Using HOL4, we prove any properties of our functions that interest us. Our automatic synthesis tool then generates corresponding CakeML code [22], along with a refinement proof that allows us to lift properties of our HOL functions to properties about the generated code. At this point, development and verification is largely independent of CakeML-specific details, except that the programmer may benefit from building on our basis library of verified and already synthesised utility functions.
2. Impure parts of the program are written as deeply embedded CakeML code within HOL4. Properties of programs are stated as Hoare triples in the style of separation logic [27], and verified with respect to the *characteristic formula* (CF) [5] of the CakeML program. The soundness theorem of CF for CakeML [11] allows us to lift such Hoare triples to theorems about the evaluation of the original CakeML program. A library of specialised tactics alleviates some of the manual labour involved in CF proofs; particularly helpful is a tactic that generates intermediate postconditions for let-expressions.
3. The verified CakeML compiler [30], which is itself written as a HOL function, is evaluated on the program from the preceding steps, yielding concrete machine code for several mainstream architectures (x86-64, ARMv6, ARMv8, RISCV, MIPS). The compiler correctness theorem states (roughly) that the observable behaviour of the generated machine code program coincides with the observable behaviour of the source program, unless the target machine runs out of memory. By instantiating this theorem for our concrete run of the compiler, we can lift whatever properties we proved in the previous steps about our CakeML program to properties about its machine code implementation.

To demonstrate this approach, we will use a simple running example of a word-frequency counter: a program that takes a file name as input on the command

[6] `https://hol-theorem-prover.org/`

line, and prints to `stdout` the frequencies of the words occuring in the file. To illustrate that our approach scales to more interesting examples, we also present new verified implementations of some staple Un*x-style utilities: `cat`, `sort`, `grep`, `diff` and `patch`.

Contributions Our main contribution is fitting the various components of the CakeML ecosystem together into a framework for developing high-assurance verified software. While many of these components have been introduced in previous work [22, 30, 11], past presentations have focused on how the components work individually rather than how to use them in a larger context. To fit these systems together, and to target realistic programs, we also built several new components, to which we give special emphasis:

- a verified basis library with which to build applications (Section 4),
- a low-level file system and I/O model (Section 4.1), and
- better automation for CF proofs (Section 5).

Furthermore, the following specific contributions are geared towards validating and elucidating our method:

- We break down the final correctness theorem about the generated machine code, explicating what it covers and what it does not cover (Section 3)
- Several examples of verified programs, mainly Un*x utilities (Section 6)

All of our code and proofs are available at `https://github.com/CakeML/cakeml`. The running example is in the `tutorial` directory, the other example programs are in the `examples` directory, and the file system implementation is in the `basis` directory.

2 Overview

In this section, we will give a front-seat view of our approach to end-to-end verification from a user perspective. We assume the reader understands the general ideas behind interactive theorem proving in higher-order logic, but not the details of any particular system (e.g., Coq, Isabelle, HOL4).

2.1 Running example

Our running example is a simple program for counting word frequencies in files, and presenting the results in (case sensitive) alphabetical order. For example, given a file containing the text `As falls Wichita so falls Wichita falls`, the following should be printed to `stdout`:

```
As: 1
Wichita: 2
falls: 3
so: 1
```

Let us begin with the specification of the pure part, postponing for the moment any consideration of file handling. The HOL predicate valid_wordfreq_output *fc out* defines which output strings *out* are valid for which file content strings *fc*:

$$\begin{array}{l}\mathsf{valid\_wordfreq\_output}\ fc\ out \iff \\ \quad \exists\, ws. \\ \quad\quad \mathsf{set}\ ws = \mathsf{set}\ (\mathsf{splitwords}\ fc) \wedge \mathsf{sorted}\ (\lambda\, x\ y.\ x < y)\ ws \wedge \\ \quad\quad out = \mathsf{flat}\ (\mathsf{map}\ (\lambda\, w.\ \mathsf{explode}\ (\mathsf{format\_output}\ (w, \mathsf{frequency}\ fc\ w)))\ ws)\end{array}$$

In words: *out* is valid for *fc* iff there exists a list of words *ws* such that (a) the words that occur in *ws* are precisely those that occur in *fc*, (b) the words in *ws* are sorted, and (c) *out* contains each word in *ws* and its frequency presented in the format above. The strict ordering $<$ implies that the words in *ws* are pairwise distinct; hence no word is listed twice. frequency is defined as follows:

$$\mathsf{frequency}\ fc\ w = \mathsf{length}\ (\mathsf{filter}\ ((=)\ w)\ (\mathsf{splitwords}\ fc))$$

We elide the similarly straightforward definitions of format_output and splitwords. As a technical detail, we use explode and implode to convert between lists of characters and (packed) strings, using the latter type in functions that will be re-used in the implementation (see Remark 1 below).

Before we embark on implementation, we prove that the specification above is implementable, in the sense that there is a valid output for every input, and that this output is unique:

$$\begin{array}{l}\vdash \exists\, out.\ \mathsf{valid\_wordfreq\_output}\ (\mathsf{implode}\ fc)\ out \\ \vdash \mathsf{valid\_wordfreq\_output}\ s\ out_1 \wedge \mathsf{valid\_wordfreq\_output}\ s\ out_2 \Rightarrow out_1 = out_2\end{array}$$

Let wordfreq_output_spec *fc* denote the unique valid output which, given the existence theorem above, we can define by indefinite specification (Hilbert choice).

2.2 Pure parts and their synthesis

For the pure part of the word frequency program, our main task is to give a computable characterisation of wordfreq_output_spec. Note that wordfreq_output_spec itself cannot serve this purpose — it is not an executable specification, and thus falls outside the subset of HOL4 that our synthesis tool supports.

To compute word frequencies, we build a map from words to natural numbers representing their frequency. For each word encountered in the input file, we increment its frequency. The final result can then be computed by applying format_output to the in-order walk of the map.

First, we need an efficient implementation of maps. We do not need to start from scratch: we are free to build on anything written in HOL4 that is sufficiently ML-like. In this case, we will use a previously verified implementation of maps based on size-balanced binary trees [23, 13] from the HOL4 distribution, which supports standard operations like lookup, insert and toAscList (in-order walk). For every word w we increment its frequency in the map t as follows:

$$\mathsf{insert\_word}\ t\ w = \mathsf{insert}\ \mathsf{cmp}\ w\ (\mathsf{lookup0}\ w\ t + 1)\ t$$

where lookup0 is a wrapper around lookup that returns 0 for any key not in the map, and cmp is a total order on strings (used for binary tree key comparison). For each line of input, we obtain its words by splitting the line on whitespace, and call insert_word on each word:

$$\mathsf{insert\_line}\ t\ s = \mathsf{foldl}\ \mathsf{insert\_word}\ t\ (\mathsf{tokens}\ \mathsf{isSpace}\ s)$$

This stops somewhat short of a complete implementation of wordfreq_output_spec — we still need to walk our map and format the final output string. We will save these tasks for the next section, and proceed with verification and synthesis. For insert_line, we prove that the balancing invariant of the map is respected, that every word in the old map is incremented by its frequency on the line, and that the domain of the resulting map is the old domain with the words on the line added:

$$\begin{aligned}\vdash\ & \mathsf{invariant\ cmp}\ t \wedge \mathsf{insert\_line}\ t\ s = t' \Rightarrow \\ & \mathsf{invariant\ cmp}\ t' \wedge (\forall w.\ \mathsf{lookup0}\ w\ t' = \mathsf{lookup0}\ w\ t + \mathsf{frequency}\ s\ w) \wedge \\ & \mathsf{domain}\ t' = \mathsf{domain}\ t \cup \mathsf{set}\ (\mathsf{splitwords}\ s)\end{aligned}$$

To synthesise CakeML code from these HOL definitions, we invoke our proof-producing translation tool [22]. For most use cases, this is as simple as typing, e.g., `translate insert_line_def`. This does several things for us:

- Generates a CakeML AST corresponding to the definition of `insert_line`:

```
fun insert_line v2 = fn v1 =>
  List.foldl insert_word v2
    (String.tokens Char.isSpace v1)
```

- Appends the code above to the end of the current program, so further tranlation can call previously translated code — in this case, insert_word. Note also that foldl, isSpace and tokens were automatically mapped to pre-translated functions from the CakeML basis library.
- Proves a *refinement invariant* that relates the return value of the HOL function to the evaluation of its translation in the CakeML semantics. In this case, the refinement invariant states that if v_1 and v_2 are the (deeply embedded) CakeML values that encode the HOL terms s and t, then evaluation of `insert_line v2 v1` terminates with a return value u that is the CakeML encoding of $\mathsf{insert\_line}\ v_2\ v_1$. Using the refinement invariant, we can lift our correctness theorem about insert_line to a corresponding theorem about the CakeML code that runs it. For more details on refinement invariants and the encoding of HOL terms see [22].

Remark 1. For most use cases, synthesis is robust and easy to use. However, there are some tips, tricks and limitations that the user ought to be aware of:

- The translator supports underspecified HOL functions, but will generate preconditions for them. For example, `translate hd_def` yields a refinement invariant that is predicated upon the argument not being the empty list.[7] This

[7] hd takes the head of a list.

precondition will propagate to future translations of functions that mention hd, where the user is often expected to discharge them manually. The experienced CakeML developer avoids this tedium by not using underspecified functions unnecessarily, e.g., by preferring pattern matching over hd.
- The HOL type `string` is an alias for `char list` and will be encoded accordingly. To synthesise code that operates on a more efficient byte vector representation, the CakeML basis library provides the type `mlstring`.
- The proof automation for proving termination of the synthesised code depends on a close structural similarity between the synthesised code and the induction principle for the HOL function. In rare cases where this proof automation fails, gently massaging the induction principle often does the job.

2.3 Impure parts

For the remainder of our program, we shall write deeply embedded CakeML rather than shallowly embedded HOL. Here's how:

```
val _ = (append_prog o process_topdecs) ‘
  fun wordfreq u =
    case IO.inputLinesFrom (List.hd (Commandline.arguments()))
    of SOME lines =>
      IO.print_list
        (List.map format_output
          (toAscList
            (List.foldl insert_line empty lines)))‘;
```

Here `process_topdecs` is a tool that parses the quoted region as CakeML code and returns a CakeML AST, and `append_prog` appends this AST to the current program, i.e., after the synthesised code from the previous section. Note that everything after `print_list` is pure and consists entirely of calls to synthesised code.

To verify this code, we use our framework for verification condition generation based on characteristic formulae [11]. First, we give a specification of our program as a Hoare triple, where pre- and postconditions describe the state before and after execution as heap predicates in the style of separation logic. By "heap" we do not mean a literal memory heap, but a more abstract aggregation of the resources that our program can see and manipulate. The resources of interest here are the program's command line arguments, and the file system and standard I/O streams (STDIO). In these terms, we may state the top-level correctness theorem of our word frequency program as follows:

$$
\begin{aligned}
\vdash\ & \mathsf{hasFreeFD}\ fs \wedge \mathsf{inFS\_fname}\ fs\ (\mathsf{File}\ fname) \wedge \\
& cl = [\mathsf{explode}\ pname;\ \mathsf{explode}\ fname] \wedge \\
& fc = \mathsf{the}\ (\mathsf{assoc}\ fs.\mathsf{files}\ (\mathsf{File}\ fname)) \Rightarrow \\
& \mathsf{app}\ p\ \mathsf{wordfreq\_v}\ [uv]\ (\mathrm{CMDLN}\ cl * \mathrm{STDIO}\ fs) \\
& \quad (\mathsf{POSTv}\ uv. \\
& \qquad \&\mathrm{UNIT}\ ()\ uv * \mathrm{STDIO}\ (\mathsf{add\_stdout}\ fs\ (\mathsf{wordfreq\_output\_spec}\ fc)) * \\
& \qquad \mathrm{CMDLN}\ cl)
\end{aligned}
$$

This states that if our view of the file system *fs* has a free file descriptor and a file named *fname*, and *fname* is the first command-line argument then after calling `wordfreq` the following holds: the return value is unit, the command-line arguments are unchanged, and the I/O system is unchanged except for a string written to `stdout`: the program's intended output as defined by wordfreq_output_spec.

The workhorse for proving this kind of specification is our library of CF tactics. The tactic `xcf` reduces our goal to proving the same Hoare triple about the characteristic formula of the function body:

```
⊢ cf_let (Some "a") (cf_con None [])
    (cf_let (Some "b")
      (cf_app p (Var (Long "Commandline" (Short "arguments")))
        [Var (Short "a")])
      (cf_let (Some "c")
        (cf_app p (Var (Long "List" (Short "hd"))) [Var (Short "b")])
        (cf_let (Some "d")
          (cf_app p (Var (Long "IO" (Short "inputLinesFrom")))
            [Var (Short "c")]) ...))) st (CMDLN cl * STDIO fs)
    (POSTv uv. ...)
```

For each `cf_*` construct, we provide a corresponding `x*` tactic that consumes the outermost `cf_*`, propagates its weakest precondition forward and generates verification conditions that arise from this step. For example, the call `List.hd b` imposes the verification condition that `b` is non-empty. We apply these tactics until the entire CF is consumed, and all that remains is a separating implication: that the postcondition that results from propagating weakest preconditions through the CF implies the postcondition we stated originally. In our case, this follows immediately from the uniqueness and existence of outputs satisfying our specification of the pure part.

Finally, we expand away the separation logic definitions to arrive at the following theorem stated using the CakeML top-level semantics, semantics_prog.

CakeML-Level Theorem

```
⊢ hasFreeFD fs ∧ inFS_fname fs (File fname) ∧
  cl = [explode pname; explode fname] ∧
  fc = the (assoc fs.files (File fname)) ∧ wfcl cl ∧ wfFS fs ∧ stdFS fs ⇒
  ∃ io.
   semantics_prog (init_state (basis_ffi cl fs)) init_env wordfreq_prog
    (OK io) ∧
   ∃ ns.
    extract_fs fs io =
     Some (add_stdout fs (wordfreq_output_spec fc) with numchars := ns)
```

Here we see the CMDLN and STDIO preconditions have been reduced to some additional well-formedness assumptions, and the observable behaviour of the program is extracted from the FFI event trace (*io*) produced by the CakeML semantics. (numchars is an artefact of our low-level I/O model; see Section 4.2.)

2.4 Compilation

Now that we have a deeply embedded CakeML program and a correctness theorem in terms of the CakeML semantics, we can use the CakeML compiler [30] to obtain a machine code program, and a correctness theorem in terms of the machine semantics.

Since the CakeML compiler is written as a shallowly embedded HOL function, we use in-logic evaluation — unfolding definitions and simplifying until we reach a fully ground term — to obtain the machine code program. The code is wrapped in some boilerplate assembly that defines, e.g., entry points for the foreign function calls. But the user need not worry: this process is automated by simply calling

```
compile_x64 stack_size heap_size "wordfreq" wordfreq_prog_def
```

where desired stack and heap sizes of the binary are given in megabytes. Besides exporting the program to a `.S` file, which must be linked with implementations of the foreign functions to produce the final x86-64 binary, this command proves a theorem about a concrete run of the compiler:

$$\vdash \mathsf{compile\ x64\_conf\ wordfreq\_prog} = \mathsf{Some}\ (\mathsf{wordfreq\_code,wordfreq\_data,wordfreq\_conf})$$

Instantiating the compiler correctness theorem for this particular run of the compiler yields our final correctness theorem, which states that the machine code either terminates successfully and prints to `stdout` the same output as the CakeML program, or runs out of memory after printing some prefix of the intended output. See Section 3 for a detailed description of the final correctness theorem. Compiling our program and obtaining an end-to-end correctness theorem takes around 20 lines of boilerplate.

2.5 Summary

This concludes our verification of the word frequency program. The end result is a theorem about the generated machine code that actually runs, but at no point do we as users have to prove properties directly about machine code, or even directly about the CakeML semantics: all our specifications and proofs thereof are either in terms of HOL functions or in terms of the CF program logic.

All in all, this development amounts to around 250 lines of HOL, including definitions, proofs, specifications and boilerplate but excluding comments and blank lines. The extra effort of going end-to-end amounted to little, thanks to the combination of synthesis, verification-condition generation and verified compilation.

3 Trusted computing base

Unlike traditional software development, which is centered on source code in a programming language, our workflow puts definitions in logic in the spotlight, relegating source code to either an intermediate abstraction layer or the medium for expressing the program's impure I/O wrapper. Fortunately, writing definitions

in logic is a rather similar experience to traditional programming. However, one could ask whether this shift in focus is worthwhile. What have we gained?

The main dividend for working inside a theorem prover is the correctness theorem for our final executable. Here is what it looks like for the word frequency program:

Binary-Level Theorem

$$
\begin{array}{l}
\vdash \mathsf{hasFreeFD}\ fs \wedge \mathsf{inFS\_fname}\ fs\ (\mathsf{File}\ fname) \wedge \\
\quad cl = [\mathsf{explode}\ pname;\ \mathsf{explode}\ fname] \wedge \\
\quad fc = \mathsf{the}\ (\mathsf{assoc}\ fs.\mathsf{files}\ (\mathsf{File}\ fname)) \wedge \mathsf{wfcl}\ cl \wedge \mathsf{wfFS}\ fs \wedge \mathsf{stdFS}\ fs \Rightarrow \\
\quad (\mathsf{is\_x64\_mc}\ mc \wedge \\
\quad\ \mathsf{installed}\ \mathsf{wordfreq\_code}\ \mathsf{wordfreq\_data}\ \mathsf{wordfreq\_conf.ffis}\ \mathsf{x64\_regs}\ mc\ ms \Rightarrow \\
\quad\ \mathsf{machine\_sem}\ mc\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \mathsf{allow\_oom}\ \{\ \mathsf{OK}\ (\mathsf{wordfreq\_io}\ fs\ cl)\ \}\ ) \wedge \\
\quad \exists\, ns. \\
\quad\ \mathsf{extract\_fs}\ fs\ (\mathsf{wordfreq\_io}\ fs\ cl) = \\
\quad\quad \mathsf{Some}\ (\mathsf{add\_stdout}\ fs\ (\mathsf{wordfreq\_output\_spec}\ fc)\ \mathsf{with\ numchars}\ :=\ ns)
\end{array}
$$

The top-level assumptions of this theorem are unchanged from the CakeML-level theorem we have already seen: they are about the command line *cl* and I/O system *fs* being well-formed and the input file being present. The second conjunct of the conclusion, at the bottom, is also unchanged: it states that the trace of I/O events, here designated by wordfreq_io *fs cl*, amount to printing the correct string onto `stdout`. The remainder of the theorem concerns execution of the machine-code program wordfreq_code that came out of the compiler.

Binary-level semantics Our machine code semantics, machine_sem, is parametrised by a machine configuration, *mc*, a model of FFI calls, basis_ffi *cl fs*, and a model of the machine state, *ms*. The main conclusion of the binary-level theorem is that the semantics of executing a suitably-loaded machine state is within the behaviours prescribed by allow_oom { OK (wordfreq_io *fs cl*) }, which means following the wordfreq_io *fs cl* trace but possibly stopping early out-of-memory.

The definition of machine_sem is simple: it reads the next instruction from the machine state's program counter and executes it according to the underlying machine ISA semantics, or, if there is a foreign-function being called, defers to the FFI model. As explained in [9], in both cases the model accounts for non-deterministic interference to parts of the machine state not under the CakeML process's control. Our is_x64_mc *mc* assumption fixes the underlying ISA model to the one we use for x86-64: Fox's L3 model (`http://www.cl.cam.ac.uk/~acjf3/l3/`).

We assume that the machine behaves according to basis_ffi *cl fs* when it needs to make a foreign call. In practice, this means the system calls we link against via a small C wrapper file are correctly modelled by our FFI model. We describe both sides of this further in Section 4.

Loading the machine correctly Now let us turn to setting up the machine state. In our theorem, the installed predicate formalises our assumptions about the initial

machine state. It states that the code and data produced by the compiler is loaded into appropriate parts of the machine's memory, and the FFI names used by the program (which are also returned by the compiler) match the names in the machine configuration. What do we have to trust to satisfy this assumption? In practice, we print the verified wordfreq_code and wordfreq_data into a file, `wordfreq.S`, which also includes a small amount of boilerplate. We then link this against the system calls to produce the final executable. Thus, we must trust that our specification in installed correctly models the combined behaviour of the linker, our `.S` template, and the operating system's loader.

Trusting the verification framework We trust that HOL4 has no bugs that might cause it to present non-theorems as theorems. HOL4 has an LCF architecture, meaning that a small trusted kernel implements the inference rules of higher-order logic, and these constitute the only way to construct theorems. We trust that this kernel correctly implements the inference rules, and that the logic they implement is sound. Higher-order logic is well-established, and its soundness is well-studied [25, 12, 15].

Note that we do not address the specification gap, i.e., the gap between user intention and formal specification; however, since we work in a rich logic, this gap can plausibly be reduced by showing refinement from more intuitive abstractions.

4 Foreign functions in our file system library

The Foreign Function Interface allows us to call foreign — and thus potentially unverified — functions within CakeML programs. Each such function needs to be modelled by a function in HOL (the *FFI oracle*), and to establish trust, they should be carefully scrutinised for semantic equivalence. This is why we should define as few of them as possible, and their code must be kept as simple as can be.

We have implemented in C a small set of foreign functions for command-line arguments and file system operations, enough to write the examples which will be described in Section 6. In this section we present the file system FFI, and describe how we model the file system.

4.1 File system model

We want to be able to treat input and output operations in a uniform manner on both conventional files, which are identified by a filename, and streams, especially standard streams which we will call stdin, stdout and stderr. We define the file system datatype as follows.

```
Datatype inode = IOStream mlstring | File mlstring
```

Remark 2. This model could be made more accurate in many ways, e.g., by allowing several filenames per *inode* to represent hard links, or by adding other POSIX attributes to handle file permissions. Our model is meant to grow over

time. However, any feature that remains to be modelled can be seen as an implicit assumption on the correctness of CakeML programs using our file system FFI. For example, we offer no guarantees for programs manipulating several hard links to the same inode. This simple model is nonetheless sufficient to implement interesting examples (detailed in Section 6) and to show the feasibility of more involved features.

The file system model we use is defined by the following record datatype:

```
IO_fs =
 <| files : ((inode, char list) alist);
    infds : ((num, inode × num) alist);
    numchars : (num llist) |>
```

The first two fields are association lists which describe the file system's contents: *files* associates each existing *inode* with its contents. Then, *infds* maps each file descriptor (encoded as an integer) to an *inode* and an offset. The latter list could contain more attributes in a more accurate model, like the mode on which a file has been opened (read-only, append mode, etc.). The last field is a non-determinism oracle modelled as a lazy list of integers, whose purpose is explained below.

We have already said that we want our foreign functions to be as small, simple and easily inspectable as it is the main blind spot of our formal verification chain. Thus, the C implementation for the write operation — and respectively for read, open and close — will be a simple wrapper around C's `write` function as it is low-level and well specified (see POSIX standard [14, p. 2310]).

The main issue with this function (as well as for `read`) is that it is not deterministic: given a number of characters to write, it may not write all of them — and possibly none — depending on various parameters, some of which are external to our file system model (e.g. signal interruptions). Our solution is thus to use a global oracle numchars to model this non-determinism. More precisely, it is a lazy list of integers whose head is popped on each read or write operation to bound the number of read/written characters.

We can now specify basic operations on the file system, namely open, close, read and write. The latter being the most interesting one, we will mostly focus on it in the rest of the section. Its definition is provided by figure 1, and it works as follows. Given a file descriptor, a number of characters to write, and a list of such characters, it looks up the file name and offset associated with the file descriptor, fetches its contents, asserts that there are enough characters to write and that the lazy list is not empty. Its head is then used to decide how many characters at most will be written. Then, the number of written characters is returned, and the file system is updated using fsupdate, which drops one element of the lazy list, shifts the offset and updates the contents of the file accordingly.

In a similar manner, read also uses the head of the lazy list to know how many characters will be read at most. Note that in this case, according to the corresponding system call specification, at least one character will always be read except if an error is raised or the end of file is reached.

```
⊢ write fd n chars fs =
  do
   (fnm,off) ← assoc fs.infds fd;
   content ← assoc fs.files fnm;
   assert (n ≤ length chars);
   assert (fs.numchars ≠ [[]]);
   strm ← lhd fs.numchars;
   let k = min n strm
   in
    Some
     (k,
      fsupdate fs fd 1 (off + k)
       (take off content @ take k chars @ drop (off + k) content))
  od
```

Fig. 1. Write operation on files in HOL

4.2 File system FFI

We will now take a closer look at the C-side FFI implementation and its HOL oracle, focusing again on the write operation. Although there is some ongoing work to allow more flexibility in the type and number of arguments these functions are allowed to use, their basic C type is `void ffiF (unsigned char *a, long alen)` where `len` is the length of the array `a`. On the HOL side, this corresponds to an oracle function of type `word8 list` $\rightarrow$ $'state$ $\rightarrow$ (`word8 list` $\times$ $'state$) `option` where the argument of type $'state$ represents a resource on which the function has an effect — which in our case will be the file system — and the input of type `word8 list` encodes the state of the array `a` at the beginning of the call. The return type is an option type in order to handle malformed inputs, which returns the state of the array after the call, and the new state of the resource.

In the case of the write function, this corresponds to the following HOL implementation:

```
⊢ ffi_write bytes fs =
  option_choice
   do
    assert (length bytes ≥ 3);
    (nw,fs') ←
     write (w2n (hd bytes)) (w2n (hd (tl bytes)))
      (map (chr ∘ w2n) (drop (3 + w2n (hd (tl (tl bytes)))) bytes))
      fs;
    Some (update (n2w nw) 1 (update 0w 0 bytes),fs')
   od (Some (update 1w 0 bytes,fs))
```

It takes the file descriptor, number of characters to write as well as an offset from *bytes*, and calls the write operation (defined in figure 1) on the file system with these parameters. As write may fail to write all the requested bytes, it may be

necessary to call it several times successively on decreasing suffixes of the data, which is why we use an offset to avoid unnecessary copying. After this, the first byte of the array is updated with a return code (0 on success, 1 on failure), and the number of written bytes is encoded in the second byte of the array.

Note that the number of bytes to write and the number of bytes actually written are each encoded in a single byte, which prevents reading or writing more than 255 bytes at a time. This restriction could be later lifted by encoding them on two bytes, but for now it suffices to do rather efficient I/O operations. The performance of the `cat` program in Section 6.5 is a good measure of this as it mostly entirely consists in `read` and `write` FFI calls.

Note that together with the three bytes encoding the other arguments, this means that using an array of length 258 is sufficient (this will be reflected in the definition of IOFS_iobuff in Section 4.3).

Let's now see how this FFI call was implemented in C. First, we want to encode file descriptors as single bytes, so we store a mapping from C file descriptors to HOL ones in an array `infds[255]`. This implies a constraint of having at most 255 simultaneously open files, which is rarely limiting, and could also be lifted in the future. Open and close are thus implemented as wrappers around C's `open` and `close`, which serves to keep this mapping up-to-date.

Here is the implementation of write, which we trust to be correctly modelled by ffi_write, the three other file system FFI functions are handled in a quite similar way.

```
void ffiwrite (unsigned char *a, long alen){
  int nw = write(infds[a[0]], &a[3+a[2]], a[1]);
  if(nw < 0) a[0] = 1;
  else{ a[0] = 0; a[1] = nw; }
}
```

4.3 File system properties

We now want to reason about the file system, i.e., express separation logic properties about it in order to be able to write CF specifications.

First, as we have seen in Section 4.2, we need an array to handle all file system operations in order to use the FFI, i.e. where to write FFI arguments a read FFI outputs. The following property ensures that an array of length 258 is allocated at the address iobuff_loc.

$$\vdash \mathsf{IOFS\_iobuff} = \\ \mathsf{SEP\_EXISTS}\ v.\ \mathrm{W8ARRAY}\ \mathsf{iobuff\_loc}\ v * \&(\mathsf{length}\ v = 258)$$

Then, any program involving write will almost surely require the following property on the file system's non-determinism oracle:

$$\vdash \mathsf{liveFS}\ fs \iff \\ \mathsf{linfinite}\ fs.\mathsf{numchars} \wedge \\ \mathsf{always}\ (\mathsf{eventually}\ (\lambda\, ll.\ \exists\, k.\ \mathsf{lhd}\ ll = \mathsf{Some}\ k \wedge k \neq 0)) \\ fs.\mathsf{numchars}$$

Indeed, according to figure 1, something can only be written if the head of fs.numchars is non-zero. To write at least one character, one thus has to try writing until it is actually done. This will succeed if the non-determinism oracle list contains a non-zero integer, and is characterised by the following temporal logic property:

$$\mathsf{eventually}\ (\lambda\, ll.\ \exists\, k.\ \mathsf{lhd}\ ll = \mathsf{Some}\ k \wedge k \neq 0)\ fs.\mathsf{numchars}$$

Then, to ensure that this property still holds after an arbitrary number of read or write operations, we need to ensure that it always holds and that the lazy list is infinite, hence the definition of liveFS. Another way to put it is that the file system will never block a write operation forever, which is not a strong assumption to make.

We wrap the previous property with other checks on the file system — namely that its open file descriptors can be encoded into a byte, and that they are mapped to existing files — to state that the file system is well-formed.

$$\begin{array}{l}\vdash \mathsf{wfFS}\ fs \iff \\ \quad (\forall\, fd. \\ \quad\quad fd \in \mathsf{fdom}\ (\mathsf{alist\_to\_fmap}\ fs.\mathsf{infds}) \Rightarrow \\ \quad\quad fd \leq 255 \wedge \\ \quad\quad \exists\, fnm\ off. \\ \quad\quad\quad \mathsf{assoc}\ fs.\mathsf{infds}\ fd = \mathsf{Some}\ (fnm, off) \wedge \\ \quad\quad\quad fnm \in \mathsf{fdom}\ (\mathsf{alist\_to\_fmap}\ fs.\mathsf{files})) \wedge \mathsf{liveFS}\ fs\end{array}$$

Now here is the main property of file systems.

$$\vdash \mathrm{IOFS}\ fs = \mathsf{IOx}\ \mathsf{fs\_ffi\_part}\ fs * \mathsf{IOFS\_iobuff} * \&\mathsf{wfFS}\ fs$$

It states that we have a buffer for file system FFI calls, and that the well-formed file system fs is actually the current file system.

More precisely, IOx fs_ffi_part fs means that the heap contains an element encoding a list of FFI calls whose successive compositions produce the file system fs.

The latter property was heavily used when specifying various low-level I/O functions, but we need more convenient user-level properties. In particular, most programs using I/O will use the standard streams. Thus we need to ensure that they exist, are open on their respective file descriptors (i.e. 0, 1, and 2), and that standard output and error's offsets are at the end of the stream, all of which are ensured by the stdFS property.

The following property asserts that this is the case for the current file system and also abstracts away the value of fs.numchars.

$$\vdash \mathrm{STDIO}\ fs = (\mathsf{SEP\_EXISTS}\ ns.\ \mathrm{IOFS}\ (fs\ \mathsf{with\ numchars}\ :=\ ns)) * \&\mathsf{stdFS}\ fs$$

Indeed, the value of this additional field is not relevant, and we only need to know that it makes the file system "live". It would otherwise be cumbersome to specify it, as we would need to know how many read and write calls have been made during the execution of the program, which itself depends on fs.numchars (the smaller its elements are, the higher the number of calls).

We also define properties such as stdout *fs out*, which states that the content of the standard output stream is *out* (and similarly for the other two streams), as well as the function up_stdout *out fs* which updates the standard output of the file system *fs* to *out*. The specification of *wordfreq* in Section 2.3 provides a typical example of their usage.

5 Automatic postcondition generation

Some of the `x*` tactics for CF proofs that we discussed in Section 2.3 require some hints from the user for the automation to work. This is particularly onerous in the case of the `xlet` tactic handling the `cf_let` construct, introduced whenever there is a `let` expression in the CakeML AST: it requires an intermediate postcondition from the user. Such postconditions can be tedious to write and make the proofs harder to maintain, especially as `let` constructs are frequent — for example, the CF of the `wordfreq` function of Section 2.3 has seven `cf_let`s. This is because the CF of a program is based on A-normal form. Hence we implement the `xlet_auto` tactic, which automates the determination of the postcondition given to `xlet`. This tactic is designed to be used in an eager manner like the other `x*` tactics.

The difficult case is `let` expressions containing function calls (i.e.: of the form `let` $x = f$ ... `in`...). In that case, `xlet_auto` first looks for an appropriate specification for the function, then instantiates the specification, compares the precondition given by the specification with the one given by the context, extracts the frame and applies the frame rule to find the appropriate postcondition. Note that all the functions in the CakeML library are provided with specifications which can be automatically reused by `xlet_auto`. Such a specification has the following shape:

$$\vdash\ h_1\ \Rightarrow\ ...\ \Rightarrow\ h_k\ \Rightarrow\ \mathsf{app}\ p\ fv\ [xv_1;\ ...;\ xv_n]\ P_f\ Q_f$$

In order to instantiate correctly the variables in the specification, we use the fact that there often exists a unique possible instantiation. For example, the specification for the dereferencing function is given below.

$$\vdash\ \mathsf{app}\ p\ \mathsf{deref\_v}\ [rv]\ (rv\ \rightsquigarrow\ xv)\ (\mathsf{POSTv}\ yv.\ \&(xv\ =\ yv)\ *\ rv\ \rightsquigarrow\ xv)$$

Here the parameter given to the function fixes the unique possible instantiation for rv. Then, as a pointer can only point to a single value, we get the unique possible value for xv by looking at the current precondition given by the context. Finally, we apply the frame rule.

The `xlet_auto` tactic works in a similar manner, mainly by rewriting the hypotheses of the specification, so as to trigger the apparition of equalities between schematic and non-schematic variables, and by applying user-provided theorems to compare the preconditions and extract information such as the unicity of the value pointed to by a pointer, etc. The frame is extracted by comparing the preconditions given by the specification and by the context, as we expect the user to use injective heap predicates (which can be satisfied by only one heap, such as: $rv\ \rightsquigarrow\ xv$).

In practice, it turns out that when `xlet_auto` fails, it is because it cannot find a proper instantiation. To deal with this issue, the user can activate heuristics based on unification. We observe that `xlet_auto` manages to find the postcondition in many situations, and that turning on the heuristics often allows our algorithm to perform the last necessary steps whenever it fails.

6 Examples

We present a selection of Un*x commands —including `grep`, `sort`, `diff`, and `cat`— that we have developed end-to-end-verified implementations of, using the method described in preceding sections. In each case, the end product is a verified x86-64 binary, which is available for download[8]. We focus on implementing the commands' default behaviour. Hence they fall somewhat short of being drop-in replacements: we do not support the abundance of command-line options that full implementations of the POSIX specifications deliver.

In Section 6.5, we compare the performance of our binaries to MLton, PolyML and the GNU implementations.

6.1 Cat

A verified `cat` implementation was presented in our previous work on CF [11]. The `cat` implementation presented here differs in two respects: first, it is verified with respect to a significantly more low-level file system model (see Section 4.1). Second, it has significantly improved performance (see Section 6.5), since it is implemented in terms of more low-level I/O primitives. Hence this example demonstrates that reasonably performant I/O verified with respect to a low-level I/O model is feasible in our setting. Here is the code:

```
fun pipe_255 fd1 fd2 =
  let val nr = IO.read fd1 (Word8.fromInt 255) in
    if nr = 0 then 0 else (IO.write fd2 nr 0; nr) end

fun do_onefile fd =
  if pipe_255 fd (IO.stdout()) > 0 then do_onefile fd else ();

fun cat fnames =
  case fnames of
    [] => ()
  | f::fs => (let val fd = IO.openIn f in
                do_onefile fd; IO.close fd; cat fs end)
```

The difference over the previous implementation is `pipe_255`, which gains efficiency by requesting 255 characters at a time from the input stream, rather than single characters as previously. We elide its straightforward CF specification, which essentially states that the output produced on `stdout` is the concatenation

[8] `cakeml.org/esop18/x86_binaries.zip`

of the file contents of the filenames given as command line arguments. The `cat` implementation above does not handle exceptions thrown by `IO.openIn`; hence the specification assumes that all command line arguments are valid names of existing files.

6.2 Sort

The sort program reads all of the lines in from a list of files given on the command-line, puts the lines into an array, sorts them using Quicksort, and then prints out the contents of the array. The proof that the printed output contains all of the lines of the input files, and in sorted order, is tedious, but straightforward.

We do not use an existing Quicksort implementation, but write and verify one from scratch. Unlike the various list-based Quicksort algorithms found in HOL, Coq, and Isabelle, we want an efficient array-based implementation of pivoting. Hence we implement something more akin to Hoare's original algorithm. We sweep two pointers inward from the start and end of the array, swapping elements when they are on the wrong side of the pivot. We stop when the pointers pass each other. Note that we pass in a comparison function: our Quicksort is parametric in the type of array elements.

```
fun partition cmp a pivot lower upper =
let
  fun scan_lower lower =
  let val lower = lower + 1 in
    if cmp (Array.sub a lower) pivot
    then scan_lower lower
    else lower end

  fun scan_upper upper = ...

  fun part_loop lower upper =
  let
    val lower = scan_lower lower
    val upper = scan_upper upper in
    if lower < upper
    then let val v = Array.sub a lower in
        (Array.update a lower (Array.sub a upper);
         Array.update a upper v;
         part_loop lower upper)
      end
    else upper end in
  part_loop (lower - 1) (upper + 1) end;
```

Because this is intrinsically imperative code, we do not use the synthesis tool, but instead verify it with CF directly. The only tricky thing about the proof is working out the invariants for the various recursive functions, which are surprisingly subtle, for an algorithm so appealingly intuitive. We elide the details, since our focus is not on how to verify algorithms, but they are all spelled out in the CakeML examples directory.

Our approach to verifying the algorithm is to assume a correspondence between the CakeML values in the array, and HOL values that have an appropriate ordering on them. The Quicksort algorithm needs that ordering to be a *strict weak order*. This is a less restrictive assumption than requiring it to be a linear order (strict or otherwise). Roughly speaking, this will allow us to assume that unrelated elements are equivalent, even when they are not equal. Hence, we can sort arrays that hold various kinds of key/value pairs, where there are duplicate keys which might have different values.

$$\begin{array}{l}\mathsf{strict\_weak\_order}\ r \iff \\ \quad \mathsf{transitive}\ r \wedge (\forall x\ y.\ r\ x\ y \Rightarrow \neg r\ y\ x) \wedge \\ \quad \mathsf{transitive}\ (\lambda x\ y.\ \neg r\ x\ y \wedge \neg r\ y\ x)\end{array}$$

Even though we are not using the synthesis tool, we do use its refinement invariant combinators to maintain the CakeML/HOL correspondence. This enforces a mild restriction that our comparison function must be pure, but greatly simplifies the proof by allowing us to reason about ordering and permutation naturally in HOL.

The following is our correctness theorem for partition. We assume that there is a strick weak order cmp that corresponds to the CakeML value passed in as the comparison. We also assume some arbitrary refinement invariant a on the elements of the array. The _ → _ combinator lifts refinement invariants to functions.

$$\begin{array}{l}\vdash \mathsf{strict\_weak\_order}\ cmp \wedge (a \to a \to \textsc{Bool})\ cmp\ cmp\_v \wedge \\ \quad \mathsf{pairwise}\ a\ elems_2\ elem\_vs_2 \wedge elem\_vs_2 \neq [\,] \wedge \\ \quad \textsc{Int}\ (\&\mathsf{length}\ elem\_vs_1)\ lower\_v \wedge \\ \quad \textsc{Int}\ (\&(\mathsf{length}\ elem\_vs_1 + \mathsf{length}\ elem\_vs_2 - 1))\ upper\_v \wedge \\ \quad (pivot,pivot\_v) \in \mathsf{set}\ (\mathsf{front}\ (\mathsf{zip}\ (elems_2,elem\_vs_2))) \Rightarrow \\ \quad \mathsf{app}\ ffi\_p\ \mathsf{partition\_v}\ [cmp\_v;\ arr\_v;\ pivot\_v;\ lower\_v;\ upper\_v] \\ \quad\ (\textsc{ARRAY}\ arr\_v\ (elem\_vs_1\ @\ elem\_vs_2\ @\ elem\_vs_3)) \\ \quad\ (\mathsf{POST v}\ p\_v. \\ \quad\quad \mathsf{SEP\_EXISTS}\ part_1\ part_2. \\ \quad\quad\ \textsc{ARRAY}\ arr\_v\ (elem\_vs_1\ @\ part_1\ @\ part_2\ @\ elem\_vs_3) * \\ \quad\quad\ \&\mathsf{partition\_pred}\ cmp\ (\mathsf{length}\ elem\_vs_1)\ p\_v\ pivot\ elems_2\ elem\_vs_2 \\ \quad\quad\quad part_1\ part_2)\end{array}$$

We can read the above as follows, starting in the conclusion of the theorem. Partition takes 5 arguments cmp_v, arr_v, $pivot\_v$, $lower\_v$, and $upper\_v$, all of which are CakeML values. As a precondition, the array's contents can be split into 3 lists of CakeML values $elems\_vs_1$, $elems\_vs_2$, and $elems\_vs_3$.[9] Now looking at the assumptions, the length of $elem\_vs_1$ must be the integer value for the lower pointer. A similar relation must hold for the upper pointer, so that $elem\_vs_2$ is the list of elements in-between the pointers, inclusive. We also must assume that the pivot element is in segment to be partitioned (excluding the last element).

The postcondition states that the partition code will terminate, and that there exists two partitions. The array in the heap now contains the two partitions

[9] @ appends lists.

instead of $elem\_vs_2$. The partition_pred predicate (definition omitted), ensures that the two partitions are non-empty, permute $elem\_vs_2$, and that the elements of the first are not greater than the pivot, and the elements of the second are not less. These last two points use the shallowly embedded cmp and $elems_2$, rather than cmp_v and $elems\_vs_2$.

6.3 Diff and Patch

We have created verified implementations of `diff` and `patch`. For space reasons the presentation here will focus mostly on `diff`.

At the heart of `diff` lies the notion of *longest common subsequence* (LCS). A list s is a *subsequence* of t if by removing elements from t we can obtain s. s is a *common subsequence* of t and u if it is a subsequence of both, and an LCS if no other subsequence of t and u is shorter than it.

$$
\begin{aligned}
&\mathsf{lcs}\ s\ t\ u \iff \\
&\quad \mathsf{common\_subsequence}\ s\ t\ u \wedge \\
&\quad \forall\, s'.\ \mathsf{common\_subsequence}\ s'\ t\ u \Rightarrow \mathsf{length}\ s' \leq \mathsf{length}\ s
\end{aligned}
$$

`diff` first computes an LCS of the two input files' lines[10], and then presents any lines not present in the LCS as additions, deletions or changes as the case case might require.

We implement and verify shallow embeddings for a sequence of progressively more realistic LCS algorithms: a naive algorithm that runs in exponential time wrt. the number of lines; a dynamic programming programming version that runs in quadratic time; and a further optimisation that achieves linear best-case performance[11].

On top of the latter LCS algorithm, we write a shallow embedding $\mathsf{diff\_alg}\ l\ l'$ that given two lists of lines returns a list of lines corresponding to the verbatim output of `diff`. To give the flavour of the implementation, we show the main loop that diff_alg uses:

```
diff_with_lcs [] l n l' n' =
  if l = [] ∧ l' = [] then [] else diff_single l n l' n'
diff_with_lcs (f::r) l n l' n' =
  let (ll,lr) = split ((=) f) l; (l'l,l'r) = split ((=) f) l'
  in
    if ll = [] ∧ l'l = [] then
      diff_with_lcs r (tl lr) (n + 1) (tl l'r) (n + 1)
    else
      diff_single ll n l'l n' @
      diff_with_lcs r (tl lr) (n + length ll + 1) (tl l'r)
        (n' + length l'l + 1)
```

The first argument to `diff_with_lcs` is the LCS of l and l', and the numerical arguments are line numbers. If the LCS is empty, all remaining lines in l and l'

[10] The LCS is not always unique: both $[a, c]$ and $[b, c]$ are LCS:es of $[a, b, c]$ and $[b, a, c]$.
[11] Asymptotically faster algorithms exist [2]; we leave their verification for future work.

must be additions and deletions, respectively; the auxiliary function diff_single presents them accordingly. If the LCS is non-empty, partition l and l' around their first occurences of the first line in the LCS. Anything to the left is presented as additions or deletions, and anything to the right is recursed over using the remainder of the LCS.

We take our specification of `diff` directly from its POSIX standard description [14, p. 2658]:

> The diff utility shall compare the contents of *file1* and *file2* and write to standard output a list of changes necessary to convert *file1* into *file2*. This list should be minimal. No output shall be produced if the files are identical.

For each sentence in the above quote, we prove a corresponding theorem about our `diff` algorithm:

$$\begin{array}{l}
\vdash \mathsf{patch_alg}\ (\mathsf{diff_alg}\ l\ r)\ l = \mathsf{Some}\ r \\
\vdash \mathsf{lcs}\ l\ r\ r' \Rightarrow \\
\quad \mathsf{length}\ (\mathsf{filter}\ \mathsf{is_patch_line}\ (\mathsf{diff_alg}\ r\ r')) = \\
\quad\quad \mathsf{length}\ r + \mathsf{length}\ r' - 2 \times \mathsf{length}\ l \\
\vdash \mathsf{diff_alg}\ l\ l = [\,]
\end{array}$$

The convertibility we formalise as the property that `patch` cancels `diff`. The minimality theorem states that the number of change lines printed is precisely the number of lines that deviate from the files' LCS[12].

We apply our synthesis tool to diff_alg, and write a CakeML I/O wrapper around it:

```
fun diff' fname1 fname2 =
  case FileIO.inputLinesFrom fname1 of
      NONE => print_err (notfound_string fname1)
    | SOME lines1 =>
      case FileIO.inputLinesFrom fname2 of
          NONE => print_err (notfound_string fname2)
        | SOME lines2 => List.app print (diff_alg lines1 lines2)

fun diff u =
  case Commandline.arguments () of
      (f1::f2::[]) => diff' f1 f2
    | _ => print_err usage_string
```

We prove a CF specification (here elided) stating that: if there are two command-line arguments that are both valid filenames, the return value of diff_alg is printed to `stdout`; otherwise, an appropriate error message is printed to `stderr`.

[12] Note that this differs from the default behaviour of the GNU implementation of `diff`, which uses heuristics that do not compute the minimal list if doing so would be prohibitively expensive.

6.4 grep

`grep <regex> <file> <file>...` prints to `stdout` every line from the files that matches the regular expression `<regex>`. Unlike `sort`, `diff` and `patch` which need to see the full file contents before producing output, `grep` can process lines one at a time and produce output after each line. The main loop of `grep` reads a line, and prints it if it satisfies the predicate `m`:

```
fun print_matching_lines m prefix fd =
  case FileIO.inputLine fd of NONE => ()
  | SOME ln => (if m ln then (print prefix; print ln) else ();
                print_matching_lines m prefix fd)
```

For each filename, we run the above loop if the file can be opened, and print an appropriate error message to `stderr` otherwise:

```
fun print_matching_lines_in_file m file =
  let val fd = FileIO.openIn file
  in (print_matching_lines m (String.concat[file,":"]) fd;
      FileIO.close fd)
  end handle FileIO.BadFileName =>
      print_err (notfound_string file)
```

The latter function satisfies the following CF specification (eliding `stderr` output):

$$
\begin{array}{l}
\vdash \textsc{Filename}\ f\ fv \wedge \mathsf{hasFreeFD}\ fs \wedge (\textsc{String} \to \textsc{Bool})\ m\ mv \Rightarrow \\
\quad \mathsf{app}\ p\ \mathsf{print\_matching\_lines\_in\_file\_v}\ [mv;\ fv]\ \ldots \\
\quad\ (\mathsf{POSTv}\ uv. \\
\quad\quad \ldots * \\
\quad\quad \mathsf{STDOUT} \\
\quad\quad\ (out\ \texttt{@} \\
\quad\quad\ \mathsf{if\ inFS\_fname}\ fs\ f\ \mathsf{then} \\
\quad\quad\quad \mathsf{flat} \\
\quad\quad\quad\ (\mathsf{map}\ ((++)\ (\mathsf{explode}\ f\ \texttt{@}\ \text{“:”})) \\
\quad\quad\quad\quad (\mathsf{filter}\ (m \circ \mathsf{implode}) \\
\quad\quad\quad\quad\quad (\mathsf{map}\ (\mathsf{flip}\ (++)\ \text{“\textbackslash n”}) \\
\quad\quad\quad\quad\quad\quad (\mathsf{splitlines}\ (\mathsf{the}\ (\mathsf{assoc}\ fs.\mathsf{files}\ f)))))) \\
\quad\quad\ \mathsf{else}\ \text{“”}) * \ldots)
\end{array}
$$

The postcondition states that the output to `stdout` is precisely those lines in f that satisfy m, with f and a colon prepended to each line. The three assumptions mean, respectively: that f is a string without null characters, and fv is its corresponding deeply embedded CakeML value; that our view of the file system has a free file descriptor; and that m is a fully specified (i.e., lacking preconditions) function of type `char lang` and mv is the corresponding CakeML closure value.

The main function of `grep` is as follows:

```
fun grep u =
  case Commandline.arguments ()
  of [] => print_err usage_string
```

```
| [_] => print_err usage_string
| (regexp::files) =>
  case parse_regexp (String.explode regexp) of
    NONE => print_err (parse_failure_string regexp)
  | SOME r =>
      List.app (fn file => print_matching_lines_in_file
                             (build_matcher r) file) files
```

`parse_regexp` and `build_matcher` are synthesised from a previous formalisation of regular expressions by Slind [28], based on Brzozowski derivatives [26].

The semantics of `grep` is given by the function grep_sem, which returns a tuple of output for `stdout` and `stderr`, respectively.

```
grep_sem (v0::regexp::filenames) fs =
 if null filenames then ("",explode usage_string)
 else
  case parse_regexp regexp of
    None ⇒ ("",explode (parse_failure_string (implode regexp)))
  | Some r ⇒
     let l =
          map (grep_sem_file (regexp_lang r) fs)
            (map implode filenames)
      in (flat (map fst l),flat (map snd l))
grep_sem _ v2 = ("",explode usage_string)
```

`regexp_lang` is a specification of `build_matcher` due to Slind, and `grep_sem_file` is a semantics definition for `print_matching_lines_in_file`. The final CF specification states that the output to the `std*` streams are as in grep_sem, and has two premises: that there is an unused file descriptor, and that Brzozowski derivation terminates on the given regular expression[13].

6.5 Performance

In this section, we will examine how the end-products of following our approach — the verified binaries — measure up in terms of performance. Table 1 compares our binaries with the same code[14] compiled with MLton 20100608 and PolyML 5.7, and with the GNU coreutils 8.25 implementations. We run GNU `diff` with the `--minimal` option, since the behaviour of our `diff` is to always return a minimal change set.

For `sort` we test already-sorted input and unsorted input. For `grep`, we search for occurrences of a word both at the start of a line, and anywhere on the line. For `diff` we test cases where differences are scattered all over the files, and where differences are concentrated in a small part of the file.

[13] Finding a termination proof for the kind of Brzozowski derivation we use is an open problem that is not addressed by Slind's work nor by the present paper. See, e.g., Nipkow and Traytel [24] for a discussion.

[14] The code has minor differences in, e.g., argument order of library functions.

	`cat`	`sort`		`grep`		`diff`	
	to file	sorted	unsorted	start	all	scattered	concentrated
CakeML	1	21.15	0.22	7.47	72.05	0.77	8.63
PolyML	1.35	4.57	0.14	3.92	39.34	0.47	17.25
MLton	0.61	3.87	0.13	2.30	32.2	0.36	0.86
GNU	0.13	0.25	0.14	0.00	0.00	0.00	0.06
CakeML (old)	18.73						

Table 1. Performance comparison. All times are in seconds of wall-time. Benchmarks were run with a 2.6GHz Intel(R) Core(tm) i7-6600U CPU and 16GB of memory, running Ubuntu 16.04.2 LTS.

An instructive way to read the table is as follows: the differences between MLton and GNU implementations show the performance overhead imposed by running our verified algorithms rather than the unverified but highly-tuned, matured-over-decades C code of the GNU utilities. The differences between MLton and CakeML show the overhead imposed by running our verified compiler, as opposed to running a state-of-the-art unverified ML compiler.

7 Related work

There are numerous impressive systems for verifying algorithms, including Why3 [8], Dafny [18], and F* [29] that focus on effective verification, but at the algorithmic level only. Here we focus on projects whose goal includes generating code with a relatively small TCB.

One commonly used route to building verified systems is to use the unverified code extraction mechanisms that all modern interactive theorem provers have. The idea is that users verify properties of functions inside the theorem prover and then call routines that print the in-logic functions into source code for some mainstream functional programming language outside the theorem prover's logic. This is an effective way of working, as can be seen in CompCert [19] where the verified compile function is printed to OCaml before running. The printing step leaves a hole in the correctness argument: there is no theorem relating user-proved properties with how the extracted functions compile or run outside the logic. There has been work on verifying parts of the extraction mechanisms [20, 10], but none of these close the hole completely. The CakeML toolchain is the first to provide a proof-producing code extraction mechanism that gives formal guarantees about the execution of the extracted code outside of the logic. In a slightly different way, ACL2 can efficiently execute code with no trusted printing step, since their logic is just pure, first-order Common Lisp. However, the Common Lisp compiler must then be trusted in a direct way, rather than only indirectly as part of the soundness of the proof assistant.

The above code extraction mechanisms treat functions in logic as if they were pure functional programs. This means that specifications can only make statements relating input values to output values; imperative features are not directly

supported. The Imperative HOL [4] project addresses this issue by defining an extensible state monad in Isabelle/HOL and augmenting Isabelle/HOL's code extraction to map functions written in this monadic style to the corresponding imperative features of the external programming languages. This adds support for imperative features, but does not close the printing gap.

The above approaches expect users to write their algorithms in the normal style of writing functions in theorem provers. However, if users are happy to adapt to a style supported by a refinement framework, e.g., the Isabelle Refinement Framework [17] or Fiat [7], then significant imperative features can be introduced through proved or proof-producing refinements within the logic. The Isabelle Refinement Framework lets users derive fast imperative code by stepwise refinement from high-level abstract descriptions of algorithms. It targets Imperative HOL, which again relies on unverified code extraction. Fiat aims to be a mostly automatic refinement engine that derives efficient code from high-level specifications. The original version of Fiat required use of the Coq's unverified code extraction. However, more recent versions seem to perform refinement all the way down to assembly code [6]. The most recent versions amount to proof-producing compilation inside the logic of Coq. Instead of proving that the compiler will always produce semantically compatible code, in the proof-producing setting, each run of the tools produces a certificate theorem explaining that this compilation produced a semantically compatible result.

The Verified Software Toolchain VST [3] shares many of the goals of our effort here, and provides some of the same end-to-end guarantees. VST builds a toolchain based on the CompCert compiler, in particular they place a C dialect, which they call Verifiable C, on top of CompCert C minor and provide a powerful separation logic-style program logic for this verification-friendly version of C. VST can deal with input and output and, of course, with highly imperative code. VST can provide end-to-end theorems about executable code since verified programs can be compiled through CompCert, and CompCert's correctness theorem transfers properties proved at the Verifiable C level down to the executable. The major difference wrt. the CakeML toolchain is that in VST one is always proving properties of imperative C code. In contrast, with CakeML, the pure functional parts can be developed as conventional logic functions in a shallow embedding, i.e. no complicated separation logic gets in the way, while imperative features and I/O are supported by characteristic formulae. We offer similar end-to-end guarantees by composing seamlessly with the verified CakeML compiler.

The on-going CertiCoq project [1] aims to do for Coq what CakeML has done for HOL4. CertiCoq is constructing a verified compiler from a deeply embedded version of Gallina, the language of function definitions in the Coq logic, to the C minor intermediate language in CompCert and from there via CompCert to executable code. This would provide verified code extraction for Coq, that is similar to CakeML's partly proof-producing and partly verified code extraction. In their short abstract [1], the developers state that this will only produce pure functional programs. However, they aim for interoperability with C and thus

might produce a framework where pure functions are produced from CertiCoq, and the imperative parts and I/O parts are verified in VST.

We note that there appears to be a small gap, from the user's perspective, in the verification story of CertiCoq, at least as described in Anand et al. [1]. Users want to prove properties of shallowly embedded functions in Gallina, but the verified CertiCoq compiler starts from a deep embedding of the OCaml datatype used by Coq to represent Gallina functions. In other words, the correctness theorem of CertiCoq does not seem to relate exactly what the user has, i.e. shallow embeddings in Coq's Gallina language, with the generated executable. The on-going Œuf [21] project, which is also for Coq and shares many goals with CertiCoq, claims to not have this small gap in the verification story.

8 Conclusion

We have demonstrated that the CakeML approach can be used to develop imperative programs with I/O for which we have true end-to-end correctness theorems. The applications are verified down to the concrete machine code that runs on the CPU, subject to reasonable, and documented, assumptions about the underlying operating system. Verifying these applications demonstrate how it is possible to separate the high-level proof task, such as proofs about sorting and regular expression matching, from the details of interacting with files and processing command-line arguments. In this way, the proof task naturally mimics the modular construction of the code.

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: Coq for Programming Languages (CoqPL) (2017)
2. Apostolico, A., Galil, Z. (eds.): Pattern Matching Algorithms. Oxford University Press, Oxford, UK (1997)
3. Appel, A.W.: Verified software toolchain - (invited talk). In: Barthe, G. (ed.) European Symposium on Programming (ESOP). pp. 1–17. Springer (2011), https://doi.org/10.1007/978-3-642-19718-5_1
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.) Theorem Proving in Higher Order Logics (TPHOLs). Springer (2008)
5. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011. pp. 418–430 (2011), http://doi.acm.org/10.1145/2034773.2034828
6. Chlipala, A., Delaware, B., Duchovni, S., Gross, J., Pit-Claudel, C., Suriyakarn, S., Wang, P., Ye, K.: The end of history? Using a proof assistant to replace language design with library design. In: Summit on Advances in Programming Languages (SNAPL). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017), https://doi.org/10.4230/LIPIcs.SNAPL.2017.3

7. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: Principles of Programming Languages (POPL). pp. 689–700. ACM (2015), http://doi.acm.org/10.1145/2676726.2677006
8. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
9. Fox, A.C.J., Myreen, M.O., Tan, Y.K., Kumar, R.: Verified compilation of CakeML to multiple machine-code targets. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017. pp. 125–137 (2017), http://doi.acm.org/10.1145/3018610.3018621
10. Glondu, S.: Vers une certification de lextraction de Coq. Ph.D. thesis, Universit Paris Diderot (2012)
11. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. In: Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017. pp. 584–610 (2017), https://doi.org/10.1007/978-3-662-54434-1_22
12. Harrison, J.: Towards self-verification of HOL light. In: Automated Reasoning, Third International Joint Conference, IJCAR 2006. Lecture Notes in Computer Science, vol. 4130, pp. 177–191. Springer (2006), https://doi.org/10.1007/11814771_17
13. Hirai, Y., Yamamoto, K.: Balancing weight-balanced trees. Journal of Functional Programming 21(3), 287307 (2011)
14. IEEE Computer Society, The Open Group: The open group base specifications issue 7. IEEE Std 1003.1, 2016 Edition (2016)
15. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: Self-formalisation of higher-order logic - semantics, soundness, and a verified implementation. J. Autom. Reasoning 56(3), 221–259 (2016), https://doi.org/10.1007/s10817-015-9357-x
16. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. In: POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 179–191. ACM Press (Jan 2014)
17. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving (ITP). Springer (2015), https://doi.org/10.1007/978-3-319-22102-1_17
18. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). pp. 348–370 (2010), https://doi.org/10.1007/978-3-642-17511-4_20
19. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning 43(4), 363–446 (2009)
20. Letouzey, P.: Extraction in Coq: An overview. In: Computability in Europe (CiE). Springer (2008), https://doi.org/10.1007/978-3-540-69407-6_39
21. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: Verified Coq Extraction in Coq, http://oeuf.uwplse.org/ and https://courses.cs.washington.edu/courses/cse599w/16sp/projects/oeuf.pdf, retrieved 2017
22. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. Journal of Functional Programming 24(2-3), 284–315 (May 2014)
23. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. In: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing. pp.

137–142. STOC '72, ACM, New York, NY, USA (1972), http://doi.acm.org/10.1145/800152.804906
24. Nipkow, T., Traytel, D.: Unified decision procedures for regular expression equivalence. In: Interactive Theorem Proving - 5th International Conference, ITP 2014. LNCS, vol. 8558, pp. 450–466. Springer (2014), https://doi.org/10.1007/978-3-319-08970-6_29
25. Norrish, M., Slind, K., et al.: The HOL System: Description, 3rd edn., http://hol.sourceforge.net/documentation.html
26. Owens, S., Reppy, J.H., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. 19(2), 173–190 (2009), https://doi.org/10.1017/S0956796808007090
27. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002). pp. 55–74 (2002), https://doi.org/10.1109/LICS.2002.1029817
28. Slind, K.L.: High performance regular expression processing for cross-domain systems with high assurance requirements. Presented at the Third Workshop on Formal Methods And Tools for Security (FMATS3) (2014)
29. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoué, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 256–270. ACM (Jan 2016), https://www.fstar-lang.org/papers/mumon/
30. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A new verified compiler backend for CakeML. In: ICFP '16: Proceedings of the 21th ACM SIGPLAN International Conference on Functional Programming. pp. 60–73. ACM Press (Sep 2016)