Program Verification in the Presence of I/O

Semantics, verified library routines, and verified applications

Hugo Férée[1], Johannes Åman Pohjola[2,3], Ramana Kumar[3,5], 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
[5] (now at DeepMind, UK)

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 environment that the program runs in, or the program's interaction with that environment. As a result, several layers of translation and wrapper code must be trusted. In contrast, the CakeML project focuses on end-to-end verification to replace this trusted code with verified code in a cost-effective manner.

In this paper, we present infrastructure for developing and verifying impure functional programs with I/O and imperative file handling. Specifically, we extend CakeML with a low-level model of file I/O, and verify a high-level file I/O library in terms of the model. We use this library to develop and verify several Unix-style command-line utilities: cat, sort, grep, diff and patch. The workflow we present is built around the HOL4 theorem prover, and therefore all our results have machine-checked proofs.

1 Introduction

Program verification using interactive theorem provers is at its most pleasant when one reasons about shallow embeddings of the program's core algorithms in the theorem prover's native logic. For a simple example, consider this shallow embedding in the HOL4 theorem prover[6] of a program that given two lists returns the longest:

$$\mathsf{longest}\ l\ l' = \mathsf{if}\ \mathsf{length}\ l \geq \mathsf{length}\ l'\ \mathsf{then}\ l\ \mathsf{else}\ l'$$

Reasoning about such a shallow embedding is a breeze. The definition above is an equation in the HOL4 logic, so in a proof we can always replace the left-hand side with the right-hand side. The numbers and lists it uses are those of the HOL4 library, so all pre-existing theorems and proof procedures for them are

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

directly applicable to our development. There is no need for the indirection and tedium of explicitly invoking the semantic rules of some calculus, program logic or programming language semantics.

This approach, while convenient, leaves two gaps in the verification story:

1. Any properties we prove are about a mathematical function in the HOL4 logic, and do not apply to the real program that runs outside of the logic, other than by a questionable informal analogy between functions in logic and procedures in a programming language.
2. Software must interact with its environment in order to be useful, but our toy verification example above is a pure functional program, i.e., it is unable to interact with its environment.

An overarching goal of the CakeML project[7] [20] is to create a verification framework that plugs both of these gaps, so as to maintain a small trusted computing base (TCB) without sacrificing the convenience of working with shallow embeddings. Our focus in this paper is how to plug the second gap. In particular, we are concerned with verifying impure functional programs in CakeML that interact with their environment in ways typically required by console applications: programs that receive input via command-line arguments and `stdin`, read from and write to the file system, and produce output via `stdout` and `stderr`.

The other components of this overarching story have largely been established in previous work. Our proof-producing translation [26] allows us to generate executable code from shallow embeddings for the pure parts of our code, so that shallow verifications done at the algorithm level in HOL4 can be automatically transferred to CakeML programs that implement the algorithm. Our program logic [13] based on characteristic formulae (CF) [7] supports the verification of the impure parts of CakeML programs. Finally, our verified compiler [33] allows us to transport whatever properties we verified using translation and CF to properties about concrete machine code for several mainstream architectures (x86-64, ARMv6, ARMv8, RISC-V, MIPS).

Our specific contributions in this paper are:

- We enrich CakeML with a low-level programmer's model of file I/O, which goes far beyond our previous toy read-only file I/O model [13]. The new model of read and write operations covers the non-determinism that is inherent in the fact that e.g. writing n bytes to a stream may sometimes fail to write all n bytes, or indeed any bytes at all.
- On top of this file I/O model, we write a verified `TextIO` library in CakeML that abstracts away from the low-level details. Instead it exposes an interface of familiar high-level functions for file handling, such as `inputLine`. These functions do not expose the aforementioned non-determinism to the user, e.g. `inputLine` is verified to always return the first line of the stream, provided the file system satisfies a natural liveness property.

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

– We present a case study of a verified implementation of the `diff` and `patch` command-line utilities.

The case study serves two main purposes. First, it shows that our approach can be used to verify interesting programs. Second, it illustrates how our specific contributions fit into the bigger picture of our verification story. The bulk of the verification effort is cast in terms of a shallow embedding of the core algorithms, such as the auxiliary function longest above. Yet our file system model and `TextIO` library, together with our proof-producing translation, CF program logic, and verified compilation, allow us to transfer our theorems about the core algorithm to theorems about the environmental interactions of the machine code that implements the algorithm.

The end-result is a theorem with a remarkably small TCB: the HOL4 theorem proving system[8]; 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 as wrapped by the `read`, `write`, `open` and `close` functions of the C standard library; 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.

All of our code and proofs are contained in CakeML's 2.1 release, available at `https://code.cakeml.org/`. The example programs are in the `examples` directory, and the file system model and library is in the `basis` directory. The `examples` directory also contains verified implementations of `cat`, `grep`, and `sort` that we have developed using the techniques and tools presented in this paper. For lack of space we will not discuss these other examples further.

The CF-verified functions of the `TextIO` library, which is the topic of this paper, have been used as opaque building blocks in a recent paper [15] on synthesis of impure CakeML code.

2 Overview

In this section, we present an overview of how we achieved our results: we first give background on how CakeML handles interaction with the outside world; then explain how we instantiate the mechanism to a model of file I/O; and how we build a verified `TextIO` library on top; and finally present a verification case study that uses the new `TextIO` library.

CakeML supports interaction with the environment via a foreign function interface (FFI) based on byte arrays that is very open-ended: the precise implementation of the FFI is an external — and thus potentially unverified — program that must be linked with the output of the CakeML compiler.

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

At the source code level, CakeML programs may contain FFI calls, written `#(p) s ba`, where `p` is the FFI port name, `s` is an (immutable) string argument, and `ba` is a (mutable) byte array argument. The FFI call may read the contents of `s` and `ba`, affect the state of the external environment, and relay information back to the caller by writing to `ba`. After compilation this becomes a subroutine call to, e.g., the label `ffiwrite` if the port name happens to be `write`. This subroutine must be present in the FFI implementation we link with.

The semantics of the CakeML language is parameterised on an *FFI oracle* that describes the effect of FFI calls on the outside environment. For each port name used by the program under consideration, the FFI oracle provides an *oracle function* of type:

$$\texttt{byte list} \rightarrow \texttt{byte list} \rightarrow {}'state \rightarrow (\texttt{byte list} \times {}'state)\ \texttt{option}$$

The semantics of the aforementioned FFI call `#(p) s ba` is then given by $p\_oracle\ st\ s\ ba$, where st is the current state of the external environment. If $p\_oracle\ st\ s\ ba = \mathsf{Some}\ (st', ba')$, the result is that the state of the environment is updated to st' and the contents of ba' are written to ba. If the oracle returns None, the FFI call fails.

The design described above allows us to enrich CakeML with our file system model by instantiating, rather than modifying, its semantics: our file system model is simply an FFI oracle. Specifically, the $'state$ type variable above is instantiated to a concrete type that models the file system. It describes which files are present and their contents, the set of file descriptors currently in use, and a *non-determinism oracle* for modelling the possibility that reading and writing may process fewer characters than expected. We define oracle functions for standard file system operations — `write`, `read`, `open_in`, `open_out`, and `close` — that describe their expected behaviour in terms of state updates to the file system model. (Section 3.2)

For each of the file system operations described above, we supply an implementation of the corresponding FFI call. These are simple C functions that are responsible for unmarshalling the byte array it receives from CakeML into the format that, e.g., the `write` standard library function expects. For the purposes of our verification story, we trust that the behaviour of these C functions is correctly modelled by the oracle functions described above. Hence we have strived to keep their implementations simple enough so that it is reasonable to assess their correctness by inspection. (Section 3.1)

We implement and verify a `TextIO` library for CakeML and integrate it into CakeML's basis library. The library is written entirely in the CakeML language and verified with respect to our file system model using the CF program logic. We handle low-level details such as non-deterministic write failures and marshalling of parameters to byte arrays in a way that does not expose them to the user during programming and verification. For example, the `TextIO.inputLine` function takes a file descriptor as argument, and returns the first unread line in the file as a string provided one exists and `NONE` otherwise. Its CF specification is kept at the same level of abstraction as the preceding sentence. (Section 4)

As a case study, we develop a verified implementation of the `diff` command-line tool. The core algorithm, i.e., computing a longest common subsequence of two sequences and presenting their deviations from this subsequence in the `diff` format, is developed and verified as a shallow embedding in HOL4. We verify the correctness of this algorithm against a specification taken directly from the POSIX standard description of `diff`. Thanks to the `TextIO` library described above, and the CakeML translator and CF program logic described in previous work, with minimal effort we can lift our theorems about a pure, shallowly embedded HOL function on sequences, to theorems about the I/O behaviour of a command-line tool. The main theorem says that the output produced on `stdout` is the same as the diff computed by the shallow embedding, when given as arguments two sequences corresponding to the contents of the files whose names are given as command-line arguments, and that appropriate error or usage messages are printed to `stderr` when called for. Thanks to the CakeML compiler correctness theorem, we can further transfer this result to a theorem about the I/O behaviour of the resulting binary. (Section 5)

3 File System Interaction

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 simple.

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 5 and Appendix A. In this section we present the file system FFI, and describe how we model the file system itself.

3.1 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 the standard streams stdin, stdout and stderr. The datatype *inode* models a file system object as being either a file with an associated path, or a stream with an associated name.

```
Datatype inode = UStream mlstring | File mlstring
```

We model the state of the file system using the following record datatype:

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

The first two fields are association lists which describe the file system's contents: files maps each filename with its inode identifier (meant to be an argument of the File constructor); and inode_tbl associates each existing inode with its contents. Then, infds maps each file descriptor (encoded as a natural number) to an inode and an offset. The latter list could easily be extended to contain more detailed attributes, such as 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 natural numbers, whose purpose will be explained shortly.

Remark 1. This model could be made more detailed in many ways and is meant to grow over time. Its limitations can be understood as implicit assumptions on the correctness of CakeML programs using our file system FFI. For example, we assume that the program has exclusive access to the file system (i.e. no concurrent program writes to the same files as ours); file permissions are ignored (inode_tbl will need to be extended to take this into account); file contents are assumed to be finite (infinite contents could have been used to model pipes fed by another program running concurrently); streams are assumed to be distinct from regular files, although in practice standard streams also correspond to named files (e.g./dev/std* on Linux). There is also no representation of the directory structure of the file system: the files field can be seen as a unique directory listing all the existing files. This simple model is nonetheless sufficient to reason about interesting examples (detailed in Section 5) and to show the feasibility of more involved features.

Foreign function implementations form part of our trusted computing base, so we want them to be small, simple and easily inspectable. 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. We choose write because it is well specified (see POSIX standard [17, p. 2310]), and because it is the most low-level entry point available to us that does not commit us to the particulars of any one operating system.

The main issue with write 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 factors. Some factors are included in our model (e.g. whether the end of the file is reached) but others like signal interruptions are not. This is the non-determinism that we model with the numchars oracle. More precisely, numchars 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, in terms of our file system model. Here write is the most interesting one and the rest of the section will mostly focus on it. We give the definition of write in Figure 1, which informally can be read: given a file descriptor, a number of characters to write, a list of characters to write, and a file system state, write looks up the inode 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

```
⊢ write fd n chars fs =
    do
      (ino,off) ← assoc fs.infds fd;
      content ← assoc fs.inode_tbl ino;
      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

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.

3.2 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. The C type of such a function is

```
void ffiwrite (unsigned char *c, long clen, unsigned char *a, long alen)
```

where `clen` and `alen` are the respective lengths of the arrays `c` of immutable arguments and `a` of mutable arguments/outputs. On the HOL side, this corresponds to an oracle function of type

$$\texttt{byte list} \rightarrow \texttt{byte list} \rightarrow {}'\mathit{state} \rightarrow (\texttt{byte list} \times {}'\mathit{state})\ \texttt{option}$$

where the argument of type $'\mathit{state}$ represents a resource on which the function has an effect — which in our case will be the file system state — and the inputs of type `byte list` encode respectively the immutable argument `c` and 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 `a` after the call, and the new state of the resource.

In the case of the write function, this corresponds to the HOL specification shown in Figure 2. It takes the file descriptor (encoded as eight bytes in c), the number of characters to write as well as an offset from a (both encoded on two bytes in the second array), 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.

```
$\vdash$ ffi_write $c$ $(a_0::a_1::a_2::a_3::a)$ $fs$ =
  do
    assert (length $c = 8$);
    $fd$ $\leftarrow$ Some (byte8_to_int $c$);
    $n$ $\leftarrow$ Some (byte2_to_int $[a_0;\ a_1]$);
    $off$ $\leftarrow$ Some (byte2_to_int $[a_2;\ a_3]$);
    assert (length $a \geq n + off$);
    do
      $(nw,fs')$ $\leftarrow$ write $fd$ $n$ (implode (drop $off$ $a$)) $fs$;
      Some ($0w$::int_to_byte2 $nw$ @ $a_3::a,fs'$)
    od $+\!\!+$ Some ($1w::a_1::a_2::a_3::a,fs$)
  od
```

Fig. 2. Oracle function for `write`. The $+\!\!+$ operator returns the first argument unless it is None, and the second argument otherwise.

After this, the first byte of the array is updated with a return code (0 on success, 1 on failure) followed by the number of written bytes, encoded on two bytes.

Note that the arbitrary, and fixed size of the inputs and outputs allow to address 2^{64} file descriptors and read/write 2^{16} bytes at once, which has not been a restriction in practice so far.

Now let's see how this FFI call is implemented in C. The other file system FFI functions are handled similarly. Note that we trust this implementation to behave according to its specification, namely ffi_write.

```c
void ffiwrite (unsigned char *c, long clen, unsigned char *a, long alen){
  assert(clen = 8);
  int fd = byte8_to_int(c);
  int n = byte2_to_int(a);
  int off = byte2_to_int(&a[2]);
  assert(alen >= n + off + 4);
  int nw = write(fd, &a[4 + off], n);
  if(nw < 0){ a[0] = 1; }
  else{ a[0] = 0; int_to_byte2(nw,&a[1]); }
}
```

All it does is the corresponding system call, and marshalling its inputs and output between integers and fixed-sized sets of bytes using some easily-verifiable marshalling functions (`bytes*_to_int` and `int_to_bytes*`).

4 A Verified TextIO Library

In this section, we illustrate how we built a standard library of high-level input-output functions on top of the previously described foreign functions as well as their specification. For this, we first need to reason about the file system, i.e., express separation logic properties about it. We are then able to write and prove correctness properties about the file system operations in the CF program logic.

4.1 File System Properties

First, as we have seen in Section 3.2, when we make FFI calls from CakeML we use a mutable byte array for carrying input and output. The following property asserts that an array of length 2052 (i.e. 2048 plus 4 bytes to encode the two two-byte arguments) is allocated at the address $\mathsf{iobuff\_loc}$.

$$\vdash \mathsf{IOFS\_iobuff} = \mathsf{SEP\_EXISTS}\ v.\ \mathrm{W8ARRAY}\ \mathsf{iobuff\_loc}\ v * \&(\mathsf{length}\ v \geq 2052)$$

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

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

Indeed, according to Figure 1, something can only be written if the head of $\mathit{fs}.\mathsf{numchars}$ 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:

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

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 eight bytes, and that they (as well as all valid filenames) are mapped to existing inodes — to state that the file system is well-formed.

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

Now here is the main property of file systems.

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

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, $\mathsf{IOx}\ \mathsf{fs\_ffi\_part}\ \mathit{fs}$ means that there is a ghost state encoding a list of FFI calls whose successive compositions (like `ffi_write` from Figure 2) 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 $\mathit{fs}.\mathsf{numchars}$.

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

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 $\mathit{fs}.\mathsf{numchars}$ (the smaller its elements are, the higher the number of calls).

We also define convenient properties such as $\mathsf{stdout}\ \mathit{fs}\ \mathit{out}$ (and respectively for standard input and error), which states that the content of the standard output stream is *out* (and similarly for the other two streams), as well as the function $\mathsf{add\_stdout}\ \mathit{fs}\ \mathit{out}$ which appends the string *out* at the end of the standard output of the file system *fs* to *out*. The specifications of `TextIO.output` and `TextIO.print` in Figure 3 and of `diff` in Figure 4 provide typical examples of their usage.

4.2 Library Implementation and Specifications

In the same way that a typical standard library is supposed to expose high-level functions to the user and hide their possibly intricate implementation, one of the main challenges of a verified standard library is to provide simple and reusable *specifications* for these functions so that users can build high-level verified programs on top of it. Once again, we take the `write` FFI call as a running example and build a user-level function `TextIO.output` which will be used in most of our examples in Section 5.

Now that we have an FFI call for `write`, we define (in CakeML's concrete syntax) a function `writei` which on file descriptor `fd` and integers `n` and `i`, encodes these inputs properly for the `write` FFI call, and keeps trying to write `n` bytes from the array `iobuff` from the offset `i` until it actually succeeds to write at least one byte.

As it is a quite low-level function, its specification won't be reproduced here, but the key point is that it requires the file system to be well-formed, and thus to verify the liveFS property. Its correctness, and especially termination, relies on the fact that, according to the latter property, the file system will always eventually write at least one byte. Its proof is mainly based on the following derived induction principle over lazy lists:

$$\begin{array}{l}\vdash (\forall\, ll.\ P\ ll \lor \neg P\ ll \land Q\ (\mathsf{the}\ (\mathsf{LTL}\ ll)) \Rightarrow Q\ ll) \Rightarrow \\ \quad \forall\, ll.\ ll \neq [||] \Rightarrow \mathsf{always}\ (\mathsf{eventually}\ P)\ ll \Rightarrow Q\ ll\end{array}$$

In words: in order to prove that Q holds for a non-empty lazy list such that P always eventually holds, it suffices to prove a) that whenever P holds of a lazy list, so does Q, and b) whenever P does not hold and Q holds of the list's tail, Q holds of the entire list. In the proof these get instantiated so that P is a predicate stating that the next write operation will write at least one byte, and Q is the CF Hoare triple for `writei`.

The `writei` function takes care of some part of the non-determinism induced by the `write` system call. We can then use it to define a function `write` which will actually write all the required bytes and whose outcome is thus fully deterministic. But this is yet another intermediate function whose specification has a fair number of hypotheses and whose Hoare triple is quite involved. We thus define SML-like user-level functions like `TextIO.output` and `TextIO.print` whose specifications involve the high-level property STDIO defined in Section 4.1. The latter are given in Figure 3, in the form $\mathsf{app}\ p\ \mathsf{f\_v}\ args\ P\ (\mathsf{POSTv}\ uv.\ Q)$ essentially meaning that whenever the separation logic precondition P is satisfied, the function named f, on arguments $args$ (related to HOL values with relations like FD, String or Unit) terminates on a value uv which satisfies the postcondition Q.

$$\begin{array}{l}\vdash \mathsf{FD}\ fd\ fdv \land \mathsf{get\_file\_content}\ fs\ fd = \mathsf{Some}\ (content,pos) \land \textsc{String}\ s\ sv \Rightarrow \\ \quad \mathsf{app}\ p\ \mathsf{TextIO\_output\_v}\ [fdv;\ sv]\ (\mathrm{STDIO}\ fs) \\ \quad\quad (\mathsf{POSTv}\ uv. \\ \quad\quad\quad\quad \&\textsc{Unit}\ ()\ uv\ * \\ \quad\quad\quad\quad \mathrm{STDIO}\ (\mathsf{fsupdate}\ fs\ fd\ 0\ (pos + \mathsf{strlen}\ s)\ (\mathsf{insert\_atl}\ (\mathsf{explode}\ s)\ pos\ content))) \\ \vdash \textsc{String}\ s\ sv \Rightarrow \\ \quad \mathsf{app}\ p\ \mathsf{TextIO\_print\_v}\ [sv]\ (\mathrm{STDIO}\ fs) \\ \quad\quad (\mathsf{POSTv}\ uv.\ \&\textsc{Unit}\ ()\ uv\ *\ \mathrm{STDIO}\ (\mathsf{add\_stdout}\ fs\ s))\end{array}$$

Fig. 3. Specifications for `TextIO.output` and `TextIO.print`

From a user's perspective, these theorems simply state that on a standard file system, the return type of these functions is `unit` and they produce a standard file system, modified as expected.

5 Case Study: A Verified Diff

In this section, we present verified implementations of `diff` and `patch`, using the method described in preceding sections. For space reasons the presentation here will focus mostly on `diff`. The end product is a verified x86-64 binary, which is available for download[9]. We focus on implementing the default behaviour. Hence it falls somewhat short of being a drop-in replacement for, e.g., GNU `diff`: we do not support the abundance of command-line options that full implementations of the POSIX specification deliver.

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 longer than it.

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

`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 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 with respect to the number of lines; a dynamic 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 $\mathsf{diff\_alg}$ 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' must be additions and deletions, respectively; the auxiliary function $\mathsf{diff\_single}$ presents them accordingly. If the LCS is non-empty, partition l and l' around their first occurrences 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.

[9] `https://cakeml.org/vstte18/x86_binaries.zip`

[10] The LCS is not always unique: both $[a, c]$ and $[b, c]$ are LCSes of $[a, b, c]$ and $[b, a, c]$.

[11] There are algorithms that do better than quadratic time for practically interesting special cases [3]; we leave their verification for future work.

We take our specification of `diff` directly from its POSIX standard description [17, 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:

$$
\begin{array}{l}
\vdash \mathsf{patch\_alg}\ (\mathsf{diff\_alg}\ l\ r)\ l = \mathsf{Some}\ r \\
\vdash \mathsf{lcs}\ l\ r\ r' \Rightarrow \\
\quad \mathsf{length}\ (\mathsf{filter}\ \mathsf{is\_patch\_line}\ (\mathsf{diff\_alg}\ r\ r')) = \\
\quad\ \ \mathsf{length}\ r + \mathsf{length}\ r' - 2 \times \mathsf{length}\ l \\
\vdash \mathsf{diff\_alg}\ l\ l = [\,]
\end{array}
$$

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 TextIO.inputLinesFrom fname1 of
      NONE => TextIO.print_err (notfound_string fname1)
    | SOME lines1 =>
      case TextIO.inputLinesFrom fname2 of
          NONE => TextIO.print_err (notfound_string fname2)
        | SOME lines2 => TextIO.print_list (diff_alg lines1 lines2)
fun diff u =
  case CommandLine.arguments () of
      (f1::f2::[]) => diff' f1 f2
    | _ => TextIO.print_err usage_string
```

We prove a CF specification shown in Figure 4 stating that: if an unused file descriptor is available, and 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`. Note that we have a separating conjunction between the file system and command-line, despite the fact that both conjuncts describe the FFI state. This is sound since they are about two disjoint, non-interfering parts of the FFI state; for details we refer the reader to [13].

For an indication of where the effort went in this case study, we can compare the size of the source files dedicated to each part of the development. Definitions and proofs for LCS algorithms are 1098 lines of HOL script, and definitions

[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.

```
diff_sem cl fs =
  if length cl = 3 then
    if inFS_fname fs (EL 1 cl) then
      if inFS_fname fs (EL 2 cl) then
        add_stdout fs
          (concat
             (diff_alg (all_lines fs (EL 1 cl))
                (all_lines fs (EL 2 cl))))
      else add_stderr fs (notfound_string (EL 2 cl))
    else add_stderr fs (notfound_string (EL 1 cl))
  else add_stderr fs usage_string
```

```
⊢ hasFreeFD fs ⇒
   app p diff_v [Conv None [ ]]
     (STDIO fs ∗ CMDLN cl)
     (POSTv uv.
          &Unit () uv ∗
          STDIO (diff_sem cl fs) ∗
          CMDLN cl)
```

Fig. 4. Semantics for `diff` (left) showing how it changes the file system state, and its specification (right) as a CF Hoare triple.

and proofs for the `diff` and `patch` algorithms is 1270 lines. Translation of these algorithms to CakeML, and definition and verification of the CakeML I/O wrapper comprises 200 lines of proofs in total. Of these, 59 lines are tactic proofs for proving the CF specification from Figure 4. These proofs are fairly routine and consist mostly of tactic invocations for unfolding the next step in the weakest precondition computation; in particular, none of it involves reasoning about file system internals. We conclude that our contributions in previous sections do indeed deliver on their promise: almost all our proof effort was cast in terms of shallow embeddings, yet our end product is a theorem about the I/O behaviour of the binary code that actually runs, and at no point did we have to sweat the small stuff with respect to the details of file system interaction.

6 Related work

There are numerous impressive systems for verifying algorithms, including Why3 [11], Dafny [22], and F* [32] that focus on effective verification, but at the algorithmic level only. Here we focus on projects whose goal includes either generating code with a relatively small TCB, reasoning about file systems, or verification of Unix-style utilities.

Small-TCB verification 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 [23] 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 [24, 12], 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 [6] 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 [21] or Fiat [9], 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 Coq's unverified code extraction. However, more recent versions seem to perform refinement all the way down to assembly code [8]. 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 [4] 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. Much like CakeML, VST supports using an oracle for predicting the meaning of instructions that interact with the outside world [16], though to the best of our knowledge this feature has not been used to reason about file system interaction. 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 [2] 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 [2], 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.

File systems and Unix-style utilities There is a rather substantial literature on file system modelling and verification [14, 5, 10, 1, 30], but comparatively little work on reasoning about user programs on top of file systems. An exception is Ntzik and Gardner [28], who define a program logic for reasoning about client programs of the POSIX file system. Their emphasis is on directory structure and pathname traversal, which we do not consider on our model, but apart from this, the two models are equivalent (our `files` field behaves as a single directory containing all file names). The programs they consider are written in a simple while language enriched with file system operations; this is sufficient for their aims since their aim is to study the correctness of file system algorithms in the abstract, not binary correctness of implementations as in the present paper. As a case study they consider the `rm -r` algorithm, in which they expose bugs in several known implementations.

Kosmatovet al. mention a verification of the `Get_Line` function in Spark ADA [25].[13] The file system is modelled by ghost variables that represent the file contents and current position of the file under consideration. The `fgets` function from `libc` is annotated with a contract that describes its behaviour in terms of updates on the ghost variables, and is thus part of the TCB in the same way as the system calls that we model by the FFI oracle is part of our TCB. This effort uncovered several long-standing bugs in the implementation of `Get_Line`.

In terms of investigating `diff` from a formal methods point of view, Khanna et al. [19] study the three-way `diff` algorithm and attempt to determine what its specification is; the surprising conclusion is that it satisfies few, if any, of the properties one might expect it to. It does not attempt to verify two-way `diff`,

[13] The verification is described in more detail in a blog post by Yannick Moy: `https://blog.adacore.com/formal-verification-of-legacy-code`

which is the topic of the present paper; instead, it takes the properties of `diff` that we prove in Section 5 as given.

Recently, Jeannerod et al. [18] verified an interpreter for a shell-like language called CoLiS using Why3. The model of the underlying file system and the behaviour of external commands is kept abstract, since the paper's main focus is on the CoLiS language itself. Verification of shell scripts that invoke verified external commands such as our `diff` in, e.g., the setting of Jeannerod et al. extended with a file system model, would be an interesting direction for future work.

7 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 demonstrates how it is possible to separate the high-level proof task, such as proofs about longest common subsequence algorithms, 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.

Acknowledgments

The first and fourth authors were supported by EPSRC Grant JEP/N028759/1, UK. The second and fifth authors were partly supported by the Swedish Research Council.

References

1. Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T.C., Klein, G., Heiser, G.: Cogent: Verifying high-assurance file system implementations. In: Conte, T., Zhou, Y. (eds.) Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016. pp. 175–188. ACM (2016), http://doi.acm.org/10.1145/2872362.2872404
2. 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)
3. Apostolico, A., Galil, Z. (eds.): Pattern Matching Algorithms. Oxford University Press, Oxford, UK (1997)
4. 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

5. Arkoudas, K., Zee, K., Kuncak, V., Rinard, M.C.: Verifying a file system implementation. In: Davies, J., Schulte, W., Barnett, M. (eds.) Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3308, pp. 373–390. Springer (2004), https://doi.org/10.1007/978-3-540-30482-1_32
6. 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)
7. 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
8. 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
9. 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
10. Ernst, G., Schellhorn, G., Haneberg, D., Pfähler, J., Reif, W.: Verification of a virtual filesystem switch. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8164, pp. 242–261. Springer (2013), https://doi.org/10.1007/978-3-642-54108-7_13
11. 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)
12. Glondu, S.: Vers une certification de l'extraction de Coq. Ph.D. thesis, Université Paris Diderot (2012)
13. 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
14. Heisel, M.: Specification of the Unix file system: A comparative case study. In: Alagar, V.S., Nivat, M. (eds.) Algebraic Methodology and Software Technology. pp. 475–488. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
15. Ho, S., Abrahamsson, O., Kumar, R., Myreen, M.O., Tan, Y.K., Norrish, M.: Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In: International Joint Conference on Automated Reasoning (IJCAR) (2018), to appear
16. Hobor, A.: Oracle semantics. Princeton University (2008)
17. IEEE Computer Society, The Open Group: The open group base specifications issue 7. IEEE Std 1003.1, 2016 Edition (2016)
18. Jeannerod, N., Marché, C., Treinen, R.: A formally verified interpreter for a shell-like programming language. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017,

Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10712, pp. 1–18. Springer (2017), https://doi.org/10.1007/978-3-319-72308-2_1

19. Khanna, S., Kunal, K., Pierce, B.C.: A formal investigation of 3diff. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4855, pp. 485–496. Springer (2007), https://doi.org/10.1007/978-3-540-77050-3_40
20. 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)
21. 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
22. 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
23. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning 43(4), 363–446 (2009)
24. 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
25. McCormick, J.W.: Building High Integrity Applications with Spark ADA. Cambridge University Press, Cambridge, UK (2015)
26. 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)
27. 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
28. Ntzik, G., Gardner, P.: Reasoning about the POSIX file system: local update and global pathnames. In: Aldrich, J., Eugster, P. (eds.) Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015. pp. 201–220. ACM (2015), http://doi.acm.org/10.1145/2814270.2814306
29. 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
30. Ridge, T., Sheets, D., Tuerk, T., Giugliano, A., Madhavapeddy, A., Sewell, P.: Sibylfs: formal specification and oracle-based testing for POSIX and real-world file systems. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015. pp. 38–53. ACM (2015), http://doi.acm.org/10.1145/2815400.2815411
31. 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)

32. 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/`

33. 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)

A Appendix: further example programs

For the benefit of readers, we describe our verified implementations of the `grep`, `sort`, and `cat` command-line utilities.

A.1 Cat

A verified `cat` implementation was presented in our previous work on CF [13]. 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 3.1). Second, it has significantly improved performance, 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_2048 fd1 fd2 =
  let val nr = TextIO.read fd1 2048 in
    if nr = 0 then 0 else (TextIO.write fd2 nr 0; nr) end

fun do_onefile fd =
  if pipe_2048 fd TextIO.stdOut > 0 then do_onefile fd else ();

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

The difference over the previous implementation is `pipe_2048`, which gains efficiency by requesting 2048 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 of the file contents of the filenames given as command line arguments. The `cat` implementation above does not handle exceptions thrown by `TextIO.openIn`; hence the specification assumes that all command line arguments are valid names of existing files.

A.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.

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 $\_ \to \_$ combinator lifts refinement invariants to functions.

$$
\begin{array}{l}
\vdash \mathsf{strict\_weak\_order}\ \mathit{cmp} \wedge (a \to a \to \textsc{Bool})\ \mathit{cmp}\ \mathit{cmp\_v} \wedge \\
\quad \mathsf{pairwise}\ a\ \mathit{elems}_2\ \mathit{elem\_vs}_2 \wedge \mathit{elem\_vs}_2 \neq [\,] \wedge \\
\quad \textsc{Int}\ (\&\mathsf{length}\ \mathit{elem\_vs}_1)\ \mathit{lower\_v} \wedge \\
\quad \textsc{Int}\ (\&(\mathsf{length}\ \mathit{elem\_vs}_1 + \mathsf{length}\ \mathit{elem\_vs}_2 - 1))\ \mathit{upper\_v} \wedge \\
\quad (\mathit{pivot},\mathit{pivot\_v}) \in \mathsf{set}\ (\mathsf{front}\ (\mathsf{zip}\ (\mathit{elems}_2,\mathit{elem\_vs}_2))) \Rightarrow \\
\quad \mathsf{app}\ \mathit{ffi\_p}\ \mathsf{partition\_v}\ [\mathit{cmp\_v};\ \mathit{arr\_v};\ \mathit{pivot\_v};\ \mathit{lower\_v};\ \mathit{upper\_v}] \\
\quad\ \ (\mathrm{ARRAY}\ \mathit{arr\_v}\ (\mathit{elem\_vs}_1\ @\ \mathit{elem\_vs}_2\ @\ \mathit{elem\_vs}_3)) \\
\quad\ \ (\mathsf{POSTv}\ \mathit{p\_v}. \\
\qquad\qquad \mathsf{SEP\_EXISTS}\ \mathit{part}_1\ \mathit{part}_2. \\
\qquad\qquad\qquad \mathrm{ARRAY}\ \mathit{arr\_v} \\
\qquad\qquad\qquad\ \ (\mathit{elem\_vs}_1\ @\ \mathit{part}_1\ @\ \mathit{part}_2\ @\ \mathit{elem\_vs}_3) * \\
\qquad\qquad\qquad \&\mathsf{partition\_pred}\ \mathit{cmp}\ (\mathsf{length}\ \mathit{elem\_vs}_1)\ \mathit{p\_v}\ \mathit{pivot} \\
\qquad\qquad\qquad\ \ \mathit{elems}_2\ \mathit{elem\_vs}_2\ \mathit{part}_1\ \mathit{part}_2)
\end{array}
$$

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 $elems\_vs_3$.[14] 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 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 $elems\_vs_2$.

A.3 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 TextIO.inputLine fd of NONE => ()
```

[14] @ appends lists.

```
  | SOME ln => (if m ln then (TextIO.print prefix; TextIO.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 = TextIO.openIn file
  in (print_matching_lines m (String.concat[file,":"]) fd;
      TextIO.close fd)
  end handle TextIO.BadFileName =>
      TextIO.print_err (notfound_string file)
```

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

```
⊢ 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. ...)
```

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
    [] => TextIO.print_err usage_string
  | [_] => TextIO.print_err usage_string
  | (regexp::files) =>
    case parse_regexp (String.explode regexp) of
      NONE => TextIO.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 [31], based on Brzozowski derivatives [29].

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

```
grep_sem ($v_0$::$regexp$::$filenames$) $fs$ =
 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 _ $v_2$ = (“”,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[15].

[15] 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 [27] for a discussion.