

Towards Machine-Verified Proofs for I/O

M. Dowse, A. Butterfield, M. van Eekelen , M. de Mol , R. Plasmeijer

Nijmegen Institute for Computing and Information Sciences/

NIII-R0415 April 2004

Nijmegen Institute for Computing and Information Sciences
Faculty of Science
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Towards Machine-Verified Proofs for I/O

Malcolm Dowse^{**}, Andrew Butterfield,

dowsem@cs.tcd.ie, butrfeld@cs.tcd.ie,

Trinity College, Dublin University, Ireland,

Marko van Eekelen, Maarten de Mol and Rinus Plasmeijer,

marko@cs.kun.nl, maartenm@cs.kun.nl, rinus@cs.kun.nl,

Department of Software Technology, University of Nijmegen, The Netherlands

Abstract. In functional languages, the shape of the external world affects both our understanding of I/O and how we would wish to have I/O expressed. This paper takes the first tentative steps at examining the consequences of using an explicit model of the external world-state when reasoning (using tool-support) about the behaviour of lazy functional programs. We construct a file-system model and develop a monadic language which lets us structure I/O. Two proofs are then performed regarding the observable effect of real-world functional I/O, and it is shown how properties of the file-system lend themselves naturally to the modelling of concurrent behaviour on a single, global state. All proofs in this paper were machine-verified using the Sparkle proof-assistant.

keywords Monads, Functional Programming, Theorem Proving, File-Systems, Localised State.

1 Introduction

Pure functional languages are often praised for their elegant semantics which are ideal for formal reasoning. Even if in practice the full rigour of formal proof is seldom applied to actual functional programs, especially large ones, it is still considered that an elegant formal semantics pays dividends towards the ease with which programs may be *informally* understood.

In the last 10 to 15 years, numerous solutions have also been proposed to the problem of performing I/O in pure functional languages. Two of the most successful solutions to appear in this time were monads [17] and uniqueness types [3] – two languages, Haskell [15] and Clean [18] respectively, have been developed in which these solutions are used as the sole means of structuring I/O. Other proposed solutions to specific I/O related problems have been implemented in the form of user-defined libraries for these languages. [1, 6, 7]

This leads to an important question. How does functional programming's initial *raison d'être* of aiding reasoning via an elegant formal semantics hold

^{**} Supported by Enterprise Ireland Basic Research Grant SC-2002-283 and, in part, by the University of Nijmegen during a five month visit as a guest researcher in 2003.

up when we consider functional I/O? The problem can be approached in the following manner: what is the semantics of I/O in functional languages, and what does it say about the effect of I/O on the external world?

In [10], Gordon uses CCS to give an operational semantics to I/O in a lazy functional language, and similar techniques are applied to give a semantics to I/O in Concurrent Haskell [16]. This approach models I/O actions as reactions within a process algebra, and two programs are considered equal if, when substituted for one another in a larger program, both perform the same actions in the same order (contextual equivalence). This type of semantics directly (and successfully) tackles the most serious issue facing any general solution to non-strict functional I/O: ensuring that one can interact consistently and predictably with the external world in a language where the order in which sub-expressions are evaluated depends on how those sub-expressions are used.

However, the semantics doesn't discuss the meaning of actions, so we must conclude that the answer to the second question is "not much". Semantics for I/O tend to be minimal, focusing only on the sequencing of actions since this is the most immediate practical concern. Programmers, on the other hand, are interested also in the observable effect of I/O actions. Ideally we would like both to be able to reason formally about the effects of actions, and for functional language design to be sympathetic towards these observable effects so that we can express I/O more succinctly.

As an example of the latter, certain solutions to specific I/O related problems already make use of expected properties of the external world. For example, stream-processors are used to structure the ordering of graphical I/O events (the order in which two on-screen windows are re-drawn should be irrelevant) and Clean's uniqueness types allow one to sequence actions over individual files, not just the whole file-system [2].

1.1 This Paper

In this paper we construct a model of the external-world and show how it can be used not only to prove properties of functional programs that perform I/O but also to give insight into the way I/O can be expressed in functional languages.

Proofs in this paper were performed using the Sparkle proof-assistant [8]. Sparkle is used to reason about programs written in the lazy functional language Clean, and supports a large subset of Clean's functionality, including Hindley-Milner typing and strictness annotation [14]. The model was constructed in Clean (using a functional language as a makeshift modelling language) and I/O is expressed through the use of a monadic language modelled using an algebraic type.

In Section 3, an explicit model of one specific part of the external world, the file-system, is constructed. The file-system itself is modelled with the aid of a general location-based state-transformer which itself has certain elegant properties as described in Section 2. In Section 4, the file-system is embedded within a monadic language, and a theory of program evaluation and equivalence is developed. In Section 5, two "real world" I/O proofs are performed which involve

both loops and flow-control. In Section 6, a crude model of non-determinism is developed which shows how the ordering of actions on the file-system can be loosened somewhat without affecting referential transparency.

The idea of formal I/O proofs is often treated with some scepticism. A full justification is in [9], but for the moment we shall emphasise two important points:

- Our file-system model is not intended to adhere to an industrial-strength specification. Its purpose is instead to capture some properties we might reasonably expect from a file-system for certain programs.
- If one is to reason formally about I/O in any computer science context, pure functional languages ought to be the best place to begin.

To our knowledge, this paper contains the first machine-verified model of the external-behaviour of monadic I/O and the first machine-assisted proofs of non-trivial I/O programs in a lazy functional language.

1.2 Related Work

Gordon’s two monadic semantics [10, 11] give an operational model for I/O using labelled transition systems and streams respectively.

Our work on monads is rather different. Firstly, we use an explicit world state to model I/O. Secondly, the proofs in this paper are machine verified with a tool tailor-made for reasoning about a lazy language. The fact that they are machine-verified is a result in itself, but it also yields a notably different theory. Since there is no need to construct our own PCF-like language, monadic I/O is just modelled as an algebraic type within the language. This is a little clumsy since the explicit monadic language is mixed up with Sparkle’s built-in functional language. However, the proof-assistant lets us concentrate solely on the monadic language which is quite small, so a more compact theory seems to be the result.

Formal I/O proofs using explicit state are quite unusual. Two preliminary case-studies [5, 9] by two of the authors use this technique in performing “by-hand” I/O proofs for Haskell and Clean.

Dealing with local-state state is a common area of research in functional languages. It is mostly directed towards the creation, deletion and efficient update of memory (lazy functional state-threads [12]; type and effect systems [13]). This paper is only concerned with local state that forms a specific, fixed part of a single global state.

1.3 Sparkle/Clean Issues

All Sparkle proofs are numbered throughout the paper and their associated Sparkle theorem names are listed in Appendix A along with instructions on how their machine-readable form can be obtained.

Although Sparkle’s logic is sound, as an application it is still under development and handles certain aspects of Clean more smoothly than others. As

a result, the theorems have been re-arranged very slightly. In the actual Clean files: lambda abstractions are replaced by named functions; `Chars` are replaced with `Ints`; functions are usually not used in a curried fashion.

For those unaccustomed to Clean's syntax, it is mostly similar to Haskell, but there are a few exceptions. The Clean expressions `flatten`, `o`, `[x:xs]` and `seq` are, in Haskell, `concat`, `(.)`, `(x:xs)` and `foldl (flip (.)) id` respectively. A Haskell type-signature of the form `a -> b -> c` is written `a b -> c` in Clean.

2 Location-Based State

In this section we define a state-transformer for location-based state. The global state is split into a potentially infinite number of locations, and any action which modifies the global state must, in fact, only ever read from and modify one single location. This state model is used in Section 3 to model a simple file-system.

The implementation of this state-transformer is ultimately driven by the desire for it to obey three properties:

- **Non-interference** Given two actions on two different locations, the order in which they are performed should be irrelevant.
- **Strictness** An action should be performed even if its return value is ignored.
- **Global Failure** If an action fails, no subsequent action should be capable of either ignoring or reversing that failure.

The first property is the most important, and is used in Section 6 when we develop a crude model of non-determinism. The second is a common-sense property of any implementation of I/O in a non-strict language. The third is a consequence of our overall model of failure, which is a simple and powerful one. If the result of the state-transformer is undefined (\perp) then something, somewhere went wrong and the problem is irreversible. If the result is not \perp then the action was entirely successful. This is powerful since it allows us to lump all forms of undesirable behaviour together so that we can prove properties of only those programs which behave correctly.¹

2.1 Implementation

The world-state is of type `MapN d`. That is: a map from `Nam` to a type `d`. The map itself is implemented as a function, where map look-up, `lkp`, is function application, and map update, `upd`, modifies the function by preceding it with a comparison. The type `Nam`, which uniquely names locations, is in fact just a type-synonym for `Int`.

¹ In some sense, of course, this is a cop-out. Being able to reason about when and how a program fails is essential. The file-system model in Section 3 makes some attempt at dealing with this.

```

:: MapN d ::= Nam -> d
:: Nam    ::= Int

lkp :: Nam (MapN d) -> d
lkp n m = m n

upd :: Nam d (MapN d) -> MapN d
upd n0 d m = \n1 -> if (n0==n1) d (lkp n1 m)

```

The `channel` function defines the abstract state-transformer. It is called “channel” since, when used sequentially on many locations, the resultant state will be as if actions on the same location were in fact isolated in their own separate channel. Except in the event of failure, there is no interference.

```

channel :: !Nam (d -> STup d r) (MapN d) -> (MapN d, r)
channel n f w = case (Force (lkp n w)) of
  Force d -> case (f d) of
    STup d r -> (upd n d w, r)

```

`channel n f` is a state-transformer function of type `(MapN d) -> (MapN d, r)`. The name `n` indicates which location is to be modified, and the function `f` describes both the effect of the state-transformer on that particular location and the resultant (global) return value. One can think of `f` as a small state-transformer of type `d -> (d,r)` which acts on a single arbitrary location, and `channel n f` as a state-transformer on all locations, where `n` is the location modified by `f`.

The use of the `Force` and `STup` constructors ensures strictness in an otherwise non-strict language. `Force` is used when we wish to force the evaluation of an expression. `STup` is a strict 2-tuple: if either the left or the right element is undefined, the entire structure is undefined. Both types make use of Clean’s strictness annotation, indicated by a `!`.

```

:: Force a    = Force !a
:: STup a b = STup !a !b

```

The state-transformer defined here is highly strict in its behaviour. The value of `channel n f` is undefined if the data residing initially in location `n` is undefined or the new data/return-value pair returned by `f` is undefined. It is also strict in `n` through the use of strictness annotation in `channel`’s type. Throughout this paper we omit the side condition that values of type `Nam` be defined. These conditions exist in some lemmas, but not in any important theorems.

2.2 Properties of the Model

Firstly, let us define some abbreviations for functions.

$$\begin{aligned}
 w \xrightarrow{n} f &\equiv \text{channel } n \text{ f } w \\
 w \xrightarrow{\rightarrow} f &\equiv \text{fst } (\text{channel } n \text{ f } w) \\
 w \xrightarrow{\rightsquigarrow} f &\equiv \text{snd } (\text{channel } n \text{ f } w) \\
 [w]_n &\equiv \text{lkp } n \text{ w}
 \end{aligned}$$

The first three are the same operators as were developed in [9]. The first, $w \xrightarrow{n} f$, is just notation for the `channel` function and is a tuple containing the world-state and return-value that results from having f modify location n ; $w \xrightarrow{\rightarrow} f$ is the world-state on its own; $w \xrightarrow{\rightsquigarrow} f$ is the resultant return value on its own.

The main result of this section is the non-interference property, which states that the order in which actions on different locations are performed is irrelevant:

$$w \xrightarrow{n_1} f_1 \xrightarrow{n_2} f_2 = w \xrightarrow{n_2} f_2 \xrightarrow{n_1} f_1 \quad (n_1 \neq n_2) \quad (1)$$

To prove the above, one requires some properties of failure. These take the form of another important result. This result states both that the failure of one of two actions results in the failure of both, and, crucially, that if an action f_2 fails on location n_2 then it will also fail on the world-state that results from having performed another action prior to that on a different location n_1 . The initial reason that f_2 failed cannot have gone away.

$$\frac{w \xrightarrow{n_1} f_1 \xrightarrow{n_2} f_2 = \perp}{w \xrightarrow{n_1} f_1 = \perp \vee w \xrightarrow{n_2} f_2 = \perp} \quad (n_1 \neq n_2) \quad (2)$$

$$\frac{w \xrightarrow{n} f = \perp \vee w \xrightarrow{\rightsquigarrow} f = \perp}{w \xrightarrow{\rightarrow} f = \perp} \quad (3)$$

$$\frac{w \xrightarrow{\rightarrow} f = \perp}{[w]_n = \perp \vee f[w]_n = \perp} \quad (4)$$

$$\frac{w \xrightarrow{n} f \neq \perp}{[w \xrightarrow{n} f]_n = \text{fst}(f[w]_n)} \quad (5)$$

$$\frac{w \xrightarrow{n_1} f_1 \neq \perp}{[w \xrightarrow{n_1} f_1]_{n_2} = [w]_{n_2}} \quad (n_1 \neq n_2) \quad (6)$$

$$\frac{w \xrightarrow{n_1} f_1 \neq \perp}{w \xrightarrow{n_1} f_1 \xrightarrow{n_2} f_2 = w \xrightarrow{n_2} f_2} \quad (n_1 \neq n_2) \quad (7)$$

Fig. 1. Auxiliary Location Model Lemmas

The proof of the two theorems above requires the use of some auxiliary lemmas shown in Figure 1. Lemma 3 and 4 sum up the the effect of strictness-annotation: the resultant tuple is strict; the state-transformer will result in \perp if

and only if the input to or the output from the function f is \perp . Lemma 5 states that the value at location n after changing location n with function f is exactly that which f gave it, as long as it didn't fail. Lemma 6 states that after updating location n_1 , if the update was successful then the value at a different location n_2 will not have changed. Lemma 7 is similar to Lemma 6 except it indicates that return values will not have changed.

The proof of the second important property, Lemma 2, is centered around the use of Lemma 4 to ascertain why an action on the world-state would fail. The proof of the main non-interference property Lemma 1 requires case analysis on whether failure occurred. If it did, it affects both action orderings equally. If it didn't, nothing will have failed. In this case extensionality is used (w is a function) and it must be proven that the order of f_1 and f_2 does not affect the resultant value at any arbitrary location n_0 . Since there was no failure, Lemmas 5 and 6 are used to determine the effect of either ordering on location n_0 (which may be n_1 , n_2 or neither.)

3 File-System Model

In this section a small model of a file-system is developed using the more general location-based model outlined in the previous section.

3.1 File-System Criteria

For the purposes of this paper, we model the following file-system properties: arbitrary numbers of files; arbitrary finite quantities of data within each file; file pointers; open/closed files; the creation and deletion of files.

The following more complex issues are ignored: shared reads; symbolic links; permissions/security; directories.

3.2 Implementation

Our overall approach is to design an API which is small enough to be manageable but powerful enough that by combining API calls, complex operations can be created. One upshot of this design decision is to make each API call as logically distinct from one another as possible. Instead of "overloading" the meaning of one action with that of another (for example, the idea that opening a non-existent file for writing should also create that file), each API serves one specific purpose.

Errors are indicated by simple non-termination – either explicitly by Clean's `undef` function whose meaning is \perp or by pattern-matching. No API call ever returns a value which somehow denotes the failure of that action – it simply fails. To make up for this short-coming, additional API calls are provided whose purpose is to (try to) predict if other API calls will fail. One reason for this apparently needless proliferation of \perp s within our model is that we shall be universally quantifying over all possible file-systems in our proofs. Since our

model is grounded in the semantics of a lazy language, for better or for worse undefinedness plays a central part, and there is no obvious advantage to having two separate ways of expressing failure.

File-System Data The FS type models our file-system:

```

:: FS          = MapN (Maybe FData)

:: OpenStatus = Open Int | Closed
:: Data       = [Char]
:: FData      = (Data,OpenStatus)
:: Maybe a    = Just a | Nothing

```

A file-system is a mapping from names to `Maybe FData` - a file either doesn't exist or has a `FData` associated with it. File data itself consists of a list of characters and an `OpenStatus` which indicates whether the file is open or closed, giving a read/write-pointer if the former.

File-System Actions There are nine primitive file-system actions in our API, which we encode as an algebraic type.²

```

:: FSAction = FRead | FOpen | FClose
             | FWrite !Char | FCreate
             | FDel | FExists | FEOF
             | FOpened

```

A specific API-call is an action associated with a particular file-name. Return values are modelled using algebraic types.

```

:: FSCall ::= (FSAction, Nam)

:: RV = RInt !Int | RChar !Char
      | RBool !Bool | RNull

```

All nine API-calls are modelled as state-transformers on individual files in accordance with the interface defined in the previous section.

```

fOpen (Just (d,Closed)) =
    STup (Just (d, Open 0)) RNull

fRead (Just (d,Open p)) =
    if (p<0 || p>=length d) undef
    STup (Just (d,Open (p+1))) (RChar (d!p))

fWrite c (Just (d,Open p)) =

```

² The use of an algebraic type isn't actually essential for our proofs, but it does help to clarify what we're trying to model.

```

if (p<0 || p>length d) undef
  STup (Just (take p d ++ [c] ++
            drop (p+1) d, Open (p+1))) RNull

fClose (Just (d,Open _)) =
  STup (Just (d,Closed)) RNull

fCreate Nothing = STup (Just ([],Closed)) RNull

fDel (Just (d,Closed)) = STup Nothing RNull

fExists d = STup d (RBool (isJust d))

fEOF f = STup f (RBool (fEOF1 f))
fEOF1 (Just (d,Open p)) =
  if (p<0 || p>=length d) undef (p==length d)

fOpened f = STup f (RBool (fOpened1 f))
fOpened1 (Just (d,Closed)) = False
fOpened1 (Just (d,Open p)) = True

```

The behaviour of these actions is roughly as follows:

- **FOpen** If the file exists and is closed, open it setting the pointer to the start of the file. Otherwise fail.
- **FRead** If the file exists, is open, and the pointer is pointing to a character, read the next character and increment the pointer. Otherwise fail.
- **FWrite** If the file exists, is open, and the pointer is valid then overwrite the data being pointed to (or append, if pointing directly at the end of the data). Otherwise fail.
- **FClose** Close the file. If it doesn't exist or is already closed, fail.
- **FCreate** Create the file if it doesn't exist, failing if it does.
- **FDel** Delete the file if it does exist, failing if it doesn't.
- **FExists** Return whether the file exists.
- **FEOF** Return whether a file's pointer is at the end of the data, failing if the file doesn't exist, is closed, or the pointer is invalid.
- **FOpened** Return whether a file is opened or not, failing if it doesn't exist.

State-Transformer Interface The state-transformer interface to the file-system – `explain` – is constructed using the `channel` function defined previously.

```

explain :: FSCall FS -> (FS,RV)
explain (a,n) w = channel (explain1 a) w
  where
    explain1 FOpen      = fOpen
    explain1 FRead     = fRead

```

```
explain1 (FWrite c) = fWrite c
explain1 FClose     = fClose
explain1 FEOF       = fEOF
explain1 FExists    = fExists
explain1 FCreate    = fCreate
explain1 FDel       = fDel
explain1 FOpened    = fOpened
```

3.3 Capturing the Informal Model

Does the above implementation capture our informal specification? Not entirely. Certain “reasonable” properties cannot be guaranteed by the implementation given. Most notably:

- File-pointers may be negative or point far beyond the end of the file-data.
- File-data may be entirely or partially undefined.
- Files may contain an infinite amount of data.

These are awkward properties to enforce and result from our use of a lazy functional language to model the file-system. One solution would be to form an invariant stating formally the expected properties of the file-system, prove that all actions preserve this invariant, and only reason about file-systems for which the invariant holds.

Our approach, instead, is to guarantee as many of these properties as possible by having individual API-calls check them at “run-time”. As an example, the `FRead`, `FWrite` and `FEOF` calls fail if either the data length is infinite or the file-pointer is invalid. If a file’s data is `(d, Open p)`, then all three API calls check that `p` does not point beyond the end of the file, and, in doing so, forces the evaluation of `length d`. If `d` is infinite then this results in non-termination (i.e. failure).

4 Monadic Language

In a pure functional language the external world can only ever be updated in a single-threaded way. If one thinks of the world-state as a value, it must either be hidden altogether (a monadic approach), or its use must be heavily restricted (unique-types). Although our world-state really *is* just a simple value, in order for us to sensibly model I/O we must also impose the same restrictions on its use.

The single-threadedness property is enforced through the use of a small monadic language. In this section, an theory of evaluation and equivalence is developed for the language. Initially we explain single-step reduction, multiple-step reduction and evaluation. This culminates in a model of program equivalence in which the monad-laws, among other things, are proven.

The Clean language uses unique typing to perform I/O. Since Sparkle is designed to reason about Clean programs and unique-types let us work with an

explicit world-value, why do we instead take a monadic approach? The answer is that there are still some outstanding issues with regard to how easily unique types can be integrated with theorem-proving. At present Sparkle just throws away all unique types. This is fine for specific programs that have been compiled in Clean because they definitely represent real single-threaded programs, but it would still be possible for us to state and prove theorems about programs which violate uniqueness. Using monads, however, we are able to guarantee that all properties proven relate to real, executable programs.

4.1 Definition

The monadic language is defined in a similar way as to that in Haskell with the exception that we use an algebraic type, not a type constructor. A program is either a primitive action, a return value (or “value”, for short), or the binding of a program with a function from a value to another program.

```

:: Monad = MBind Monad (RV -> Monad)
           | MRet   RV
           | MAct   FSCall

```

4.2 Single-Step Reduction

The execution of a monadic program is modelled with the use of a single-step reduction function `mnext`:

```

mnext :: (Monad,FS) -> (Monad,FS)
mnext (MBind (MRet v1) mf2, w) = (mf2 v1, w)
mnext (MBind m1 mf2, w) = case (mnext (m1,w)) of
  (m2,w2) -> (MBind m2 mf2, w2)
mnext (MAct c, w) = case (explain c w) of
  (m2,v2) -> (MRet v2, w2)

```

When given a monadic program and a world-state, `mnext` performs one single reduction step somewhere within the structure, returning the modified program and the new world-state. This task is traditionally known as redex selection and reduction. However, using this terminology is probably unwise since with types such as `RV -> Monad` within the `Monad` definition, the meaning of any program is not explicit as a normal piece of syntax ought to be.

Let us assume the following notation for single-step expression reduction and monadic bind, (= denotes expression equality in Sparkle)

$$\begin{aligned}
\langle m, w \rangle &\longrightarrow \langle m_1, w_1 \rangle \equiv \text{mnext } (m, w) = (m_1, w_1) \\
m \triangleright f &\equiv \text{MBind } m \ f
\end{aligned}$$

Single-step reduction obeys three simple, easy to prove rules:

If an action successfully updates the world-state returning a value, it single-step reduces to that value and world-state.³

$$\frac{\text{explain } c \ w = (w_1, v_1)}{\langle \mathbf{MAct} \ c, w \rangle \longrightarrow \langle \mathbf{MRet} \ v_1, w_1 \rangle} \quad (8)$$

Binding a value with a function single-steps to the program that results from applying the function to the value. The world-state is not changed.

$$\langle \mathbf{MRet} \ v \triangleright f, w \rangle \longrightarrow \langle f \ v, w \rangle \quad (9)$$

If a program m single-step reduces to a program m_1 , then program $m \triangleright f$ single-step reduces to $m_1 \triangleright f$ with the same effect on world-state.

$$\frac{\langle m, w \rangle \longrightarrow \langle m_1, w_1 \rangle}{\langle m \triangleright f, w \rangle \longrightarrow \langle m_1 \triangleright f, w_1 \rangle} \quad (10)$$

It is worth noting that reducing a value $\mathbf{MRet} \ v$ will result in \perp , and if an action $\mathbf{MAct} \ c$ fails it will also reduce to \perp .

The single-step “reverse” lemmas in Figure 2 describe the possible values of the left-hand-side of a single-step reduction given the right-hand-side.

Most are relatively straight-forward to understand, and all were easy to prove using case-analysis. First we state four properties of single-step reduction: an action reduces to a value; A value can’t reduce to anything; any possible program can result from the reduction of a program of the form $\mathbf{MRet} \ v \triangleright f$; if a program m reduces successfully to m_1 (which means m can’t be a value), then $m \triangleright f$ will reduce to $m_1 \triangleright f$.

The justification for the lemmas is obtained by scanning the properties in the above paragraph for each of the three **Monad** data-constructors to determine what possible left-hand-side could have single-step reduced to that particular constructor on the right-hand-side. This yields the four lemmas.

Lemma 11: only an action and a program of the form $\mathbf{MRet} \ v \triangleright f$ can single-step reduce to a value; Lemma 12: only a program of the form $\mathbf{MRet} \ v \triangleright f$ can single-step reduce to an action; Lemmas 13 and 14: only a program of the form $m \triangleright f$ can single-step reduce to a program of the form $m_1 \triangleright f_1$, and if m is not a value, then $f = f_1$.

4.3 Evaluation

Next we have to implement an evaluation function which will continually single-step reduce a program until it becomes a value.

One would hope to implement this as follows:

```
meval :: (Monad, FS) -> (FS, RV)
meval (MRet v, w) = (w, v)
meval (m, w)      = meval (mnext (m, w))
```

³ This does not require us to reason about what **explain** actually does.

$$\frac{\langle m, w \rangle \longrightarrow \langle \mathbf{MRet} \ v, w_1 \rangle}{(\exists c. m = \mathbf{MAct} \ c) \vee (\exists v_1. \exists f. m = \mathbf{MRet} \ v_1 \triangleright f)} \quad (11)$$

$$\frac{\langle m, w \rangle \longrightarrow \langle \mathbf{MAct} \ c, w_1 \rangle}{\exists v_1. \exists f. m = \mathbf{MRet} \ v_1 \triangleright f} \quad (12)$$

$$\frac{\langle m, w \rangle \longrightarrow \langle m_1 \triangleright f_1, w_1 \rangle}{\exists m_0. \exists f_0. m = m_0 \triangleright f_0} \quad (13)$$

$$\frac{\langle m \triangleright f, w \rangle \longrightarrow \langle m_1 \triangleright f_1, w_1 \rangle}{(\exists v. m = \mathbf{MRet} \ v) \vee (\langle m, w \rangle \longrightarrow \langle m_1, w_1 \rangle \wedge f = f_1)} \quad (14)$$

Fig. 2. Single-Step “Reverse” Lemmas

Unfortunately the above definition is difficult to work with by itself since there is no obvious structure on which to perform induction. Inducting over the `Monad` type is of no use, since apart from the left-hand spine, everything is hidden within the `RV -> Monad` function space. Instead we re-write the above idealistic evaluation function in a more basic way.

```

miter :: Int (Monad,FS) -> (Monad,FS)
miter i (m,w)
  | i==0 = (m,w)
  | i>0  = mnext (miter (i-1) (m,w))

```

$$\text{miter } 0 = \text{id} \quad (15)$$

$$\text{miter } 1 = \text{mnext} \quad (16)$$

$$\text{miter } (i+1) = \text{mnext} \circ \text{miter } i \quad (17)$$

$$\text{miter } (i1+i2) = \text{miter } i1 \circ \text{miter } i2 \quad (18)$$

Fig. 3. Multiple-Step Integer Lemmas

`miter` iterates the single-step reduction function a specific (i) number of times, obeying the properties in Figure 3. If i is negative, it will result in \perp (side-conditions that i be non-negative do exist but are not shown). Unlike the `meval` function, `miter` does not “recognise” return values – it will blindly repeat single-step reduction precisely as many times as was specified. If there exists a value of i such that exactly i single-steps yield a return-statement, we take this instead as the meaning of evaluation. We adopt mostly standard notation, where \xrightarrow{i} means “reduces i steps to”, \Downarrow means “evaluates to” and \Uparrow means “never evaluates”, or “diverges”.

$$\begin{aligned}
\langle m, w \rangle \xrightarrow{i} \langle m_1, w_1 \rangle &\equiv \mathbf{miter} \ i \ (\mathbf{m}, \mathbf{w}) = (\mathbf{m1}, \mathbf{w1}) \\
\langle m, w \rangle \Downarrow \langle w_1, v \rangle &\equiv \exists i. \langle m, w \rangle \xrightarrow{i} \langle \mathbf{MRet} \ v, w_1 \rangle \\
\langle m, w \rangle \Uparrow &\equiv \neg \exists v. \exists w_1. \exists i. \langle m, w \rangle \xrightarrow{i} \langle \mathbf{MRet} \ v, w_1 \rangle
\end{aligned}$$

Intuitively, $\langle m, w \rangle \Downarrow \langle w_1, v \rangle$ means the exact same thing as $\mathbf{meval} \ (\mathbf{m}, \mathbf{w}) = (\mathbf{w1}, \mathbf{v})$, but only part of this correspondence can be proven using Sparkle. The proven relationship is as follows:

$$\frac{\langle m, w \rangle \Downarrow \langle w_1, v \rangle}{\mathbf{meval} \ (\mathbf{m}, \mathbf{w}) = (\mathbf{w1}, \mathbf{v})} \quad (19)$$

This states that if after a specific, i , number of reductions \mathbf{miter} yields a return-statement, its behaviour will then correspond with that of \mathbf{meval} (proven by inducting over i). There is every reason to believe the remainder of the correspondence to also be true:

$$\frac{\langle m, w \rangle \Uparrow}{\mathbf{meval} \ (\mathbf{m}, \mathbf{w}) = \perp}$$

To our knowledge, though, there is nothing we can induct over in Sparkle to prove it.

4.4 Multiple-Step Reduction Proofs

The definition of evaluation (\Downarrow) is that there exists an integer i such that single-step reducing a program precisely i times yields a value. To reason about evaluation we must first prove properties of multiple-step reduction to show whether these integers do indeed exist and, if so, what values they take.

Figure 4 contains four simpler auxiliary laws. Lemma 20 is Lemma 10 applied multiple times, proven by inducting over i . Lemma 21 states a value can only multiple-step to another value, if, in fact, it was never reduced at all. This is true since if $i \neq 0$ the value would fail to reduce. Lemma 22 shows that reduction to a value is deterministic by proving that if, say, $i_1 < i_2$ (or symmetrically $i_2 < i_1$) then reducing i_2 steps would require reducing i_1 steps, then $i_2 - i_1$ steps (lemma 18), but reducing the value yielded after i_1 steps would have to result in failure, not another value. Lemma 23 is Lemma 13 applied i times, proven by inducting over i .

The first important rule to be proven is as follows:

$$\frac{\langle m, w \rangle \xrightarrow{i_1} \langle \mathbf{MRet} \ v_1, w_1 \rangle \quad \langle f \ v_1, w_1 \rangle \xrightarrow{i_2} \langle \mathbf{MRet} \ v_2 w_2, \rangle}{\langle m \triangleright f, w \rangle \xrightarrow{i_1 + i_2 + 1} \langle \mathbf{MRet} \ v_2, w_2 \rangle} \quad (24)$$

This states that performing $i_1 + i_2 + 1$ reduction steps is the same as doing i_1 steps (evaluate m), then a single-step reduction (applying v_1 to f), then doing

$i + 1$ steps m_0 is still being evaluated and the left-hand-side of the disjunction is true.

Once the above lemma has been proven, the following can then be tackled:

$$\begin{aligned} \langle m_0 \triangleright f_0, w_0 \rangle &\xrightarrow{i} \langle \mathbf{MRet} v_2, w_2 \rangle \\ &\implies \\ \exists v_1. \exists w_1. \exists i_1. &\left(\begin{array}{c} i_1 < i \\ \wedge \\ \langle m_0, w_0 \rangle \xrightarrow{i_1} \langle \mathbf{MRet} v_1, w_1 \rangle \\ \wedge \\ \langle f_0 v_1, w_1 \rangle \xrightarrow{i-i_1-1} \langle \mathbf{MRet} v_2, w_2 \rangle \end{array} \right) \end{aligned} \quad (26)$$

This states that if a program of the form $m \triangleright f$ reduces after i steps to a value, then there exists a number $i_1 < i$ such that m reduces i_1 steps to an intermediate value v_0 and world-state w_0 , and $f v_0$ reduces $i - i_1 - 1$ steps to the final value. It is proven by using Lemma 25 in one of two different ways depending on the nature of the final single-step reductions that lead to $\mathbf{MRet} v_2$. The trick is finding the highest number i_1 such that after i_1 steps the program is still in the form $m_3 \triangleright f_3$. Either i_1 is $i - 1$, and the last reduction step reduces $m_3 \triangleright f_3$ into $\mathbf{MRet} v_2$ or i_1 is $i - 2$ and the second last reduction step reduces $m_3 \triangleright f_3$ into an action (which in turn becomes $\mathbf{MRet} v_2$). Either way, Lemma 25 is applied to the first i_1 reductions.

4.5 Evaluation Proofs

$$\langle \mathbf{MRet} v, w \rangle \Downarrow \langle w, v \rangle \quad (27)$$

$$\frac{\langle m_0, w_0 \rangle \Downarrow \langle w_1, v_1 \rangle \quad \langle f_0 v_1, w_1 \rangle \Downarrow \langle w_2, v_2 \rangle}{\langle m_0 \triangleright f_0, w_0 \rangle \Downarrow \langle w_2, v_2 \rangle} \quad (28)$$

$$\frac{\langle m_0 \triangleright f_0, w_0 \rangle \Downarrow \langle w_2, v_2 \rangle}{\exists v_1. \exists w_1. \langle m_0, w_0 \rangle \Downarrow \langle w_1, v_1 \rangle \wedge \langle f_0 v_1, w_1 \rangle \Downarrow \langle w_2, v_2 \rangle} \quad (29)$$

$$\frac{\langle m, w \rangle \Uparrow}{\langle m \triangleright f, w \rangle \Uparrow} \quad (30)$$

$$\frac{\langle m, w \rangle \Downarrow \langle w_1, v_1 \rangle \quad \langle f v_1, w_1 \rangle \Uparrow}{\langle m \triangleright f, w \rangle \Uparrow} \quad (31)$$

$$\frac{\langle m, w \rangle \Downarrow \langle w_1, v_1 \rangle \quad \langle m, w \rangle \Downarrow \langle w_2, v_2 \rangle}{w_1 = w_2 \wedge v_1 = v_2} \quad (32)$$

Fig. 5. Evaluation Rules

Proving properties of evaluation is now easy, and the rules are summed up in in Figure 5.

Lemma 27: A value evaluates to itself; Lemma 28: If m_0 evaluates to v_1 , and $f_0 v_1$ evaluates to v_2 , then $m_0 \triangleright f_0$ evaluates to v_2 (proven directly using Lemma 24); Lemma 29: If $m_0 \triangleright f_0$ evaluates to v_2 , then there exists a v_1 such that m_0 evaluates to v_1 and $f_0 v_1$ evaluates to v_2 (proven directly by Lemma 26); Lemmas 30 and 31; If m diverges, then $m \triangleright f$ diverges, and if m evaluates to v_1 and $f v_1$ diverges, then $m \triangleright f$ diverges (proven by contradiction using Lemma 26); Lemma 32: Evaluation is deterministic (proven using Lemma 22).

4.6 Program Equivalence

The monadic language is finalised by developing a theory of program equivalence. The equivalence relation chosen is as follows:

$$m_1 \cong m_2 \equiv \forall w. \forall v_0. \forall w_0. \frac{\langle m_1, w \rangle \Downarrow \langle w_0, v_0 \rangle}{\langle m_2, w \rangle \Downarrow \langle w_0, v_0 \rangle}$$

It states that two programs m_1 and m_2 are equivalent if for all initial world-states w , and all resultant values v_0 and resultant world-states w_0 , $\langle m_1, w \rangle$ evaluates to $\langle w_0, v_0 \rangle$ if and only if $\langle m_2, w \rangle$ also does. It amounts to saying that m_1 and m_2 both agree on whether they terminate and also on their resultant world-state and return-value if they do.

The fact that \cong defines an equivalence relation is trivial.

$$(m_1 \triangleright f_2) \triangleright f_3 \cong m_1 \triangleright (\lambda v. f_2 v \triangleright f_3) \quad (33)$$

$$\mathbf{MRet} v \triangleright f \cong f v \quad (34)$$

$$m \triangleright \mathbf{MRet} \cong m \quad (35)$$

Fig. 6. Monad Laws

Figure 6 contains the three monad-laws from [4]: associativity, left-unit and right-unit respectively. The three are proven by taking into account the fact that any sub-program not or may not diverge. Lemma 29 is used to split the evaluation into its constituent stages.

Finally the two following rules are proven. These show how the substitution of one equivalent programs for another within a larger program yields an equivalent larger program. This is, in effect, a form of contextual equivalence.

$$\frac{m_1 \cong m_2}{m_1 \triangleright f \cong m_2 \triangleright f} \quad (36)$$

$$\frac{\forall v. f_1 v \cong f_2 v}{m \triangleright f_1 \cong m \triangleright f_2} \quad (37)$$

5 File-System Proofs

In this section we prove some properties about two small (but not trivial) programs which perform file I/O. To do this, it is first necessary to develop some basic control-flow components.

5.1 Control-Flow

We define and prove properties about two control-flow constructs: sequencing and conditionals.

```
mseq :: Monad Monad -> Monad
mseq m1 m2 = MBind m1 (\_ -> m2)

mifelse :: Monad Monad Monad -> Monad
mifelse mc mt mf =
  MBind mc (\(RBool b) -> if b mt mf)
```

$$\text{mseq (mseq m1 m2) m3} \cong \text{mseq m1 (mseq m2 m3)} \quad (38)$$

$$\frac{\langle \text{mseq m1 m2}, w \rangle \Downarrow \langle w_2, v_2 \rangle}{\exists v_1. \exists w_1. \langle \text{m1}, w \rangle \Downarrow \langle w_1, v_1 \rangle \wedge \langle \text{m2}, w_1 \rangle \Downarrow \langle w_2, v_2 \rangle} \quad (39)$$

$$\frac{\langle \text{mc}, w \rangle \Downarrow \langle w_1, \text{RBool True} \rangle \quad \langle \text{mt}, w_1 \rangle \Downarrow \langle w_2, v \rangle}{\langle \text{mifelse mc mt mf}, w \rangle \Downarrow \langle w_2, v \rangle} \quad (40)$$

$$\frac{\langle \text{mc}, w \rangle \Downarrow \langle w_1, \text{RBool False} \rangle \quad \langle \text{mf}, w_1 \rangle \Downarrow \langle w_2, v \rangle}{\langle \text{mifelse mc mt mf}, w \rangle \Downarrow \langle w_2, v \rangle} \quad (41)$$

$$\exists w_1. \left(\begin{array}{c} \langle \text{mifelse mc mt mf}, w \rangle \Downarrow \langle w_2, v \rangle \implies \\ \langle \text{mc}, w \rangle \Downarrow \langle w_1, \text{RBool True} \rangle \wedge \langle \text{mt}, w_1 \rangle \Downarrow \langle w_2, v \rangle \\ \vee \\ \langle \text{mc}, w \rangle \Downarrow \langle w_1, \text{RBool False} \rangle \wedge \langle \text{mf}, w_1 \rangle \Downarrow \langle w_2, v \rangle \end{array} \right) \quad (42)$$

Fig. 7. Control-Flow Lemmas

These functions obey the properties in Figure 7.

Sequencing is achieved with `mseq`. `mseq m1 m2` does `m1` and then `m2`. It is associative (Lemma 38) and the sequencing of two programs may be split into the separate execution of each (Lemma 39).

`mifelse mc mt mf` is a conditional which performs `mc` and if the result is `RBool True` then it does `mt` (Lemma 40), if the result is `RBool False` then it does `mf` (Lemma 41) and if the result is neither it fails. A conditional may be split into one of two different evaluation orders depending on the result of the program `mc` (Lemma 42).

All of these results are natural consequences of the given definitions and results proven in Section 4. Lemmas 39 and 42 are by far the most heavily used later on, allowing a large program to be split easily into its individual parts.

5.2 File-System API properties

Figure 8 contains four lemmas which explain the behaviour of the `FRead`, `FWrite` and `FEOF` API-calls (lemmas corresponding the others are omitted for brevity). These are mostly a direct translation of the file-system implementations in Section 3. One fact worth stating is that these lemmas do not formally say anything about when precisely the various API-calls fail.

$$\frac{\frac{[w]_n = \text{Just } (\text{cs}, \text{Open } p) \langle \text{MAct } (\text{FRead}, n), w \rangle \Downarrow \langle w_1, v \rangle}{[w_1]_n = \text{Just } (\text{cs}, \text{Open } (p+1)) \wedge v = \text{RChar } (\text{cs}!p)} \quad (43)}{[w]_n = \text{Just } (\text{cs}, \text{Open } p) \langle \text{MAct } (\text{FWrite } c, n), w \rangle \Downarrow \langle w_1, v \rangle} \frac{}{[w_1]_n = \text{Just } (\text{take } p \text{ cs} ++ [c] ++ \text{drop } (p+1) \text{ cs}, \text{Open } (p+1)) \wedge v = \text{RNull} \wedge p >= 0 \wedge p <= \text{length } \text{cs}} \quad (44)$$

$$\frac{[w]_n = \text{Just } (\text{cs}, \text{Open } p) \langle \text{MAct } (\text{FEOF}, n), w \rangle \Downarrow \langle w_1, \text{RBool True} \rangle}{[w_1]_n = [w]_n \wedge p = \text{length } \text{cs}} \quad (45)$$

$$\frac{[w]_n = \text{Just } (\text{cs}, \text{Open } p) \langle \text{MAct } (\text{FEOF}, n), w \rangle \Downarrow \langle w_1, \text{RBool False} \rangle}{[w_1]_n = [w]_n \wedge p \geq 0 \wedge p < \text{length } \text{cs}} \quad (46)$$

Fig. 8. API-Properties

5.3 Two I/O Proofs

The two proofs are small examples of how one might expect to reason about the I/O model.

Writing data to a file The `writeLoop` function writes a list of characters to a file. The file is assumed to exist, be open and, for the proof, to have its file-pointer pointing to the end of the file. Note that there is no use of any API-call's return-value in this example – only sequencing is used.

```
writeLoop :: Nam [Char] -> Monad
writeLoop n []      = MRet RNull
writeLoop n [c:cs] =
  mseq (MAct (FWrite c,n)) (writeLoop n cs)
```

The main lemma is in Figure 9. It states that given the assumptions mentioned above, where the data `cs0` is the data already stored in file `n`, if `writeLoop n cs1` succeeds it will result in data `cs0++cs1`, with a pointer value of `length cs0+length cs1` (the end of the file). The lemma contains some hidden pre-conditions which are omitted for clarity: lists `cs0` and `cs1` must be finite, and the every element of `cs1` must be defined.

The proof is an induction over the length of `cs1`. In the inductive case, let us say `cs1` is `[c:cs]`. Character `c` is written to the end of the file, incrementing the pointer. The inductive case can then be applied using `cs0++[c]` as the new `cs0` and `cs` as the new `cs1`.

$$\frac{[w]_n = \text{Just } (\text{cs0}, \text{Open } (\text{length cs0})) \langle \text{writeLoop } n \text{ cs1}, w \rangle \Downarrow \langle w_1, v \rangle}{[w_1]_n = \text{Just } (\text{cs0}+\text{cs1}, \text{Open } (\text{length cs0} + \text{length cs1}))} \quad (47)$$

Fig. 9. writeLoop Property

Reading the length of a file `fileLength n` is a program which opens a file, reads the contents of the file from beginning to end counting the number of data items, then closes the file returning that number. The guts of the program is in the `fileLenLoop` function. This function reads from an open file until EOF, with the current length-count passed as a parameter. The read is just used to move the pointer forward – the character returned is discarded. The `fileLength` wrapper function opens the file, calls `fileLenLoop n` with an initial count-value of 0 and when it finishes, the file is closed and the length is returned.

Unlike `writeLoop`, this example contains conditionals and the use of an API-call’s return-value (EOF, in this case).

```
fileLength :: Nam -> Monad
fileLength n =
  MBind (mseq (MAct (FOpen,n))
        (fileLenLoop n (RInt 0)))
        (\l -> mseq (MAct (FClose,n)) (MRet l))

fileLenLoop :: Nam RV -> Monad
fileLenLoop n (RInt l) =
  mifelse (MAct (FEof,n))
    (MRet (RInt l))
    (mseq (MAct (FRead,n))
          (fileLenLoop n (RInt (l+1))))
```

$$\frac{[w]_n = \text{Open } (\text{cs}, \text{length cs} - \text{p}) \langle \text{fileLenLoop } n \text{ (RInt p1)}, w \rangle \Downarrow \langle w_1, v \rangle}{v = \text{RInt } (\text{p}+\text{p1})} \quad (48)$$

$$\frac{[w]_n = \text{Just } (\text{cs}, \text{Closed}) \langle \text{fileLength } n, w \rangle \Downarrow \langle w_1, v \rangle}{v = \text{RInt } (\text{length cs})} \quad (49)$$

Fig. 10. fileLength Properties

There are two lemmas – one for each function. Both are to be found in Figure 10.

The main one, not surprisingly, is Lemma 48 which pertains to `fileLenLoop`. It states that if file `n` contains data `cs` and is open with its pointer pointing `p` characters away from the end of the file, then if `fileLenLoop n (RInt`

`p1`) succeeds it will return the integer `p+p1`. There are also some hidden pre-conditions: `cs` must be finite with every element defined, `p1` must be defined, and $0 \leq p \leq \text{length } cs$.

To prove it, we induct over `p` – the number of characters the pointer is away from the end-of-file. In the base case, `p=0`, `FEOF` returns `True` and `fileLenLoop` returns the value `p1` it was initially given. In the inductive case, `FEOF` returns `False`, a character is read, then `fileLenLoop` is called inductively incrementing `p1`. We know that after reading character `c` the pointer will be one character closer to the end of the file, so the inductive hypothesis can be applied with the new `cs` being the old `cs++[c]`.

Lemma 49 states that if a file `n` exists, it contains data `cs` and is closed then if `fileLength n` succeeds it will return `length cs`. There is also the omitted pre-condition that `cs` be finite and each element of it be defined. The proof is mostly straight-forward, requiring some properties of `FOpen` and `FClose`, and Lemma 48 with `p1=0`.

6 Simple Concurrency

In this section, we present a model of concurrency in which the non-interference laws proven in Section 2 are finally used.

6.1 Intermingle

A function `intermingle` is defined as follows:

```
intermingle :: [[c]] -> [[c]]
intermingle css
  | length (flatten css) == 0 = [[]]
  | otherwise                 =
    flatten (map (\(fr,cs,bk) -> case cs of
      []       -> []
      [c:cs1] -> map (\cs -> [c:cs])
                    (intermingle (fr++[cs1]++bk)))
              (partitions3 css))

partitions3 :: [c] -> [[c],c,[c]]
partitions3 cs = partitions3a [] cs
  where
    partitions3a csf [] = []
    partitions3a csf [c:cs] =
      [(csf,c,cs) : partitions3a (csf++[c]) cs]
```

`intermingle` takes a list of lists `css` and returns a list of all the different ways in which the values of `css` can be intermingled such that the ordering in

each individual sub-list in `css` is still respected. For example:

```
intermingle [[1],[2,3],[4]] ⇒
  [[1,2,3,4],[1,2,4,3],[1,4,2,3],[2,1,3,4],
   [2,1,4,3],[2,3,1,4],[2,3,4,1],[2,4,1,3],
   [2,4,3,1],[4,1,2,3],[4,2,1,3],[4,2,3,1]]
```

The `intermingle` function is similar in spirit to “interleave” in CSP’s trace-semantic [19], whose purpose is to build a trace of all the possible ways a number of non-interacting processes can behave when placed in parallel. We are using `intermingle` for the same sort of purpose.

6.2 Non-determinism and Concurrency

Figure 11 contains the one single result from this section. In a nutshell, it states that given some primitive actions of type `FSCall`, the resultant effect of executing these actions on any file-system is only dependent on the relative ordering of actions on the same file – actions on different files may be executed in any arbitrary order.

In more detail, it states this: given a list of lists of actions `css` of type `[[FSCall]]` such that the list is finite and each sub-list is also finite, if `css` is structured in such a way that only actions on file 0 are stored in the first sub-list, actions on file 1 are stored in the second etc. (remember, the type `Nam` is a synonym for `Int`) then regardless of which particular ordering of these actions is chosen from `intermingle css`, the result of executing those actions will be the same. Each individual `intermingle` is compared with effect of a single specific one, namely the straight-forward flattening of `css`.

$$\begin{aligned}
 \text{finite2 } \text{css} \wedge (\forall i. 0 \leq i < \text{length } \text{css}. \forall j. 0 \leq j < \text{length } \text{css}!!i. \text{snd } (\text{css}!!i!!j) = i) \implies \\
 \quad \forall i. 0 \leq i < \text{intermingle } \text{css}. \\
 \quad \text{seq } (\text{map } (\text{fst } \circ \text{explain}) (\text{flatten } \text{css})) \\
 \quad = \\
 \quad \text{seq } (\text{map } (\text{fst } \circ \text{explain}) ((\text{intermingle } \text{css})!!i))
 \end{aligned} \tag{50}$$

Fig. 11. `intermingle` Property

The proof, by most standards, is enormous. At its core it is an induction over the number of actions that are left to be processed in the given list of lists, and in the inductive case it is necessary to exchange the action that is to be performed next in `flatten css` with that in `intermingle bss`. Using Lemma 1 (the non-interference property) and the given pre-condition it can be shown that these two values can indeed be exchanged in the overall sequence without any harmful effects.

6.3 Discussion

This section shows up how the properties exhibited by an explicit world-state can yield interesting ways of expressing I/O. Since the ordering of actions on files is irrelevant, there is the possibility of one using the model as a basis for expressing file I/O that is operationally non-deterministic yet still referentially-transparent.

The model of concurrency is still rough – a list of lists of files is hardly a plausible model of computation – but we see no reason why this shouldn't be a stepping-stone to more sophisticated ways of expressing I/O.

7 Conclusions and Future Work

A file-system model has been developed as an example model of the external world-state. On top of this a monadic language has been built, and an associated theory developed, which allows one to reason about the effect of lazy functional programs which perform I/O. Two example areas of application for this work are given: proofs about the observable effects of real programs (Section 5), and proofs about the feasibility of expressing real I/O in a more flexible programming style (Section 6).

In conclusion, we believe that this paper highlights that there are benefits to be had in the field of functional programming from modelling the external world-state explicitly. There is nothing about our specific file-system API that is special, so the file-system proofs outlined are not real theoretical results in any sense. Nonetheless, the overall approach gives us a means of proving useful properties of real-world programs within some practical framework.

As the title of this paper would suggest, we believe the use of tool-support has been crucial for this work – the complexity of both the external model and lazy functional semantics are such that proving properties by-hand on a large scale would be fraught with the possibility of errors.

Our immediate future work shall be an attempt to extend the monadic algebraic type in Section 4 so it can handle concurrency.

8 Acknowledgements

Malcolm Dowse would like to thank the Clean group for the advice and encouragement he received during his five-month stay as a guest researcher at the University of Nijmegen. Much of the work underpinning the contents of this paper was performed during that time.

References

1. P. Achten. A functional framework for deterministically interleaved interactive programs. In *Programming Languages: Implementations, Logics and Programs (PLILP)*, volume LNCS982, pages 451–452, 1995.

2. P. Achten and R. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, Jan. 1995.
3. E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.
4. R. J. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998.
5. A. Butterfield and G. Strong. Proving correctness of programs with I/O – a paradigm comparison. In T. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop, IFL2001*, volume LNCS2312, pages 72–87, 2001.
6. M. Carlsson and T. Hallgren. FUDGETS – A graphical user interface in a lazy functional language. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 321–330, 1993.
7. A. Courtney and C. Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.
8. M. de Mol, M. van Eekelen, and R. Plasmeijer. Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop, IFL2001*, number LNCS2312, page 55. Springer-Verlag, 2001.
9. M. Dowse, G. Strong, and A. Butterfield. Proving make correct – I/O proofs in Haskell and Clean. In R. Peña and T. Arts, editors, *Proceedings of the 14th International Workshop, IFL2002*, volume LNCS2670, pages 68–83, 2002.
10. A. D. Gordon. An operational semantics for I/O in a lazy functional language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 136–145, New York, NY, USA, June 1993. ACM Press.
11. A. D. Gordon. Bisimilarity as a theory of functional programming: mini-course. Notes Series BRICS-NS-95-3, BRICS, Department of Computer Science, University of Aarhus, July 1995.
12. J. Launchbury and S. L. P. Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, Florida, June 20–24, 1994.
13. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
14. E. Nöcker and S. Smetsers. Partially strict non-recursive data types. *Journal of Functional Programming*, 3(2):191–215, 1993.
15. S. Peyton Jones and et al, editors. *Haskell 98 Language and Libraries, the Revised Report*. CUP, Apr. 2003.
16. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 295–308, New York, NY, USA, 1996. ACM Press.
17. S. L. Peyton Jones and P. Wadler. Imperative functional programming. In ACM, editor, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 71–84, New York, NY, USA, 1993. ACM Press.
18. R. Plasmeijer and M. van Eekelen. Concurrent clean version 2.0 language report. <http://www.cs.kun.nl/~clean/>, December 2001.
19. S. Schneider. *Concurrent and Real-time Systems, the CSP Approach*. John Wiley and Sons, Chichester, England, first edition, 2000.

A Sparkle Files

The Sparkle section-files for all proofs in this paper and information on how they can be viewed is available from <http://www.cs.kun.nl/~marko/sparkle>.

Lemmas 1-7 are in section `channel`, Lemmas 8-42 are in section `mnext`, Lemmas 43-49 are in section `mnext_fs`, and Lemma 50 is in section `intermingle_channel`.

The proof names are as follows:

1: <code>cwld_swap_theorem</code>	2: <code>cwld_failure_theorem</code>
3: <code>channel_strict_result</code>	4: <code>channel_undefined</code>
5: <code>cwld_==</code>	6: <code>cwld_<></code>
7: <code>cret_<></code>	8: <code>mnext_MAct</code>
9: <code>mnext_MBind_MRet</code>	10: <code>mnext_MBind</code>
11: <code>mnext_MRet_reverse</code>	12: <code>mnext_MAct_reverse</code>
13: <code>mnext_MBind_reverse</code>	14: <code>mnext_MBind_becomes_MBind</code>
15: <code>miter_0</code>	16: <code>miter_1</code>
17: <code>miter_+1</code>	18: <code>miter_+</code>
19: <code>meval_miter</code>	20: <code>miter_MBind</code>
21: <code>miter_MRet_becomes_MRet</code>	22: <code>miter_deterministic</code>
23: <code>miter_becomes_MBind</code>	24: <code>miter_MBind_2</code>
25: <code>MBind_explained</code>	26: <code>miter_MBind_splice</code>
27: <code>evaluation_MRet</code>	28: <code>evaluation_MBind</code>
29: <code>evaluation_MBind_splice</code>	30: <code>divergence_MBind</code>
31: <code>divergence_MBind_right</code>	32: <code>evaluation_deterministic</code>
33: <code>==_MBind</code>	34: <code>==_MBind_right</code>
35: <code>==_Monad_associative</code>	36: <code>==_Monad_left_unit</code>
37: <code>==_Monad_right_unit</code>	38: <code>==_mseq_associative</code>
39: <code>mseq_splice</code>	40: <code>mifelse_True</code>
41: <code>mifelse_False</code>	42: <code>mifelse_splice</code>
43: <code>FRead</code>	44: <code>FWrite</code>
45: <code>FEOF_True</code>	46: <code>FEOF_False</code>
47: <code>writeLoop</code>	48: <code>fileLenLoop</code>
49: <code>fileLength</code>	50: <code>intermingle_channel</code>