

# Reasoning About Deterministic Concurrent Functional I/O

Malcolm Dowse<sup>1,\*</sup>, Andrew Butterfield<sup>1</sup>, and Marko van Eekelen<sup>2</sup>

<sup>1</sup> Trinity College Dublin, Ireland

{Malcolm.Dowse, Andrew.Butterfield}@cs.tcd.ie

<sup>2</sup> University of Nijmegen, The Netherlands

M.vanEekelen@science.ru.nl

**Abstract.** This paper develops a language for reasoning about concurrent functional I/O. We assume that the API is specified as state-transformers on a single world state. We then prove that under certain conditions evaluation in this language is deterministic, and give some examples. All properties were machine-verified using the Sparkle proof-assistant and using Core-Clean as a meta-language.

## 1 Introduction

In pure functional languages, I/O is usually achieved using some method of explicitly sequencing actions on the external world (monads [13]; unique types [1]). However, solutions to I/O tasks are sometimes more easily expressed and understood when written in a style that allows for the specification of concurrent I/O.

“All I/O operations [are] strictly sequenced along a single “trunk”. Sometimes, though, such strict sequencing is unwanted.” [13]

In Clean [14], a limited form of concurrent I/O is permitted. The unique type-system allows the global world state to be “split” into distinct parts resulting in, for example, a file and the rest of the file system. This leaves the relative ordering of actions on each distinct part of the global-state unspecified, but since the regions are distinct, evaluation still remains deterministic.

Concurrent Haskell [12], on the other hand, introduces powerful concurrency primitives into the language. Processes may perform I/O, fork and communicate with one another. Although this has many practical uses, it has the unavoidable effect of introducing non-determinism.

In this paper we introduce a small language with monad-like constructs which is designed for reasoning about the effect of concurrency in state-based functional I/O. We then show how given some pre-conditions on the state and some extra run-time checks we call *contexts* one can loosen the explicit sequencing of

---

\* Supported by Enterprise Ireland Basic Research Grant SC-2002-283.

actions without introducing non-determinism. The resultant language is rather dynamic in nature, requiring more run-time checks. However, it does provide greater flexibility compared with the strict sequencing of I/O actions currently required by Haskell if we wish to retain deterministic behaviour.

## 1.1 This Paper

Section 2 introduces our state-based model of I/O, APIs and contexts. Section 3 then implements a non-deterministic language with monad-like I/O and a `fork` concurrency primitive, and Section 4 proves that under certain specific conditions the language is confluent with respect to evaluation. Finally, Sections 5 and 6 give detailed examples of contexts being used to allow safe concurrency on separate parts of the one file system.

We use `Core-Clean` as a meta-language for implementing the API, modelling global-state and sequencing I/O actions. Our only axioms are pieces of `Clean/Core-Clean` code. Proofs were all machine-verified using the `Sparkle` proof-assistant [4], a semi-automated LCF-style proof-assistant designed specifically for reasoning about `Core-Clean`. `Core-Clean` supports a large subset of the functionality of the `Clean` programming language, including parametric polymorphism and strictness annotation – but not I/O, hence the need for meta-proofs.

This work generally builds on existing research by the authors [2, 6, 5] into examining how the different ways of expressing (functional) I/O affect our ability to do formal reasoning. We use `Clean` to model the language, but our results are in no way bound by `Clean`'s actual implementation.

## 1.2 Related Work

An enormous amount of literature has been published on the subjects of concurrency, state and I/O in functional languages. Most of it has only a limited relevance to our work, which is concerned, ultimately, with the semantics of I/O with a view to proving useful properties about actual programs.

For us, state is strictly global with a fixed interface. This distinguishes our work from literature on memory allocation, deallocation and sharing. When explicit concurrency is mentioned with reference to functional languages, it usually includes inter-process communication and other typically non-deterministic constructs (for example, the CCS-style approach adopted in [12]). Most mathematical results concerning functional I/O tend to be high-level and axiomatic (monads and monad-transformers [11]; using CCS to structure the ordering of actions [7]). In general, the specifics of actual APIs are ignored.

Deterministic concurrent I/O in functional languages has, however, been studied before. This has mainly been in implementation-driven attempts to provide a smoother I/O interface: the `Clean` file system API [14]; some implementation techniques for deterministic concurrency [3]; state-splitting using a special form of lazy functional state-thread [9]. Building models of the specifics of the I/O system is not new either [10, 8]. As with our approach, these systems are also state-based.

Nonetheless, this paper, as far as we know, is the first to tackle the semantic issues of concurrency in state-based I/O, incorporating a full-blown formal model of the global state as part of the language's semantics. Our results also have been machine verified.

### 1.3 A Note on Sparkle and Clean

Occasionally we modify the actual Clean syntax in this paper to make up for small shortcomings in Sparkle. In the actual Clean code: functions are usually used in an uncurried fashion; records are replaced with tuples; lambda abstractions are replaced by named functions; `Chars` are replaced with `Ints`.

Sometimes strict tuple and strict list types are used:

```
:: STup a b = STup !a !b
:: SList :a = SCons !a !(SList a) | SNull
```

For the purposes of clarity, we change these types to `!(a,b)` and `![a]` respectively, retaining the names of the standard operations (`fst`, `length` etc.)

The benefit of using Sparkle is that lazy functional semantics are already encoded within the theorem prover. We only need to wrap the pure language in a small exterior which allows I/O to be expressed. Any small pitfalls concerning name-capture, strict/lazy semantics or the Hindley-Milner type system will (hopefully) be detected automatically.

One disadvantage of using Sparkle is that it is not ideal for modelling a world-state. For example, there is, as of yet, no facility for modelling sets easily in Sparkle. Also, since we are always reasoning about actual programs which may not terminate, all types must necessarily contain a bottom element.

We use Clean syntax, but it is mostly very similar to that of Haskell. The most obvious difference is that a type `a -> b -> c` in Haskell is written as `a b -> c` in Clean.

## 2 State-Based I/O and Contexts

In this paper we incorporate a `fork`-like primitive into I/O in a functional language. To introduce concurrency without causing non-determinism we must be able to isolate certain classes of legitimate I/O actions which don't interfere with one another. Only actions with this property can be performed concurrently.

The solution is based around *contexts*. A context identifies a set of permitted actions. Each program fragment is executed within a particular context and, along with the global state, it affects what the program does and is allowed to do. If the current context forbids certain actions then any attempt to execute these actions will result in a catchable run-time error.

### 2.1 Modelling I/O

To talk about non-interference we resort to an entirely state-based model of I/O. The meaning of each action is defined as a state-transformer on some global state.

Similarly, we regard the meaning of a whole program as being the resultant effect it has on global state.<sup>1</sup>

An instance of the Clean record type `IOSystem` defines both the semantics of each I/O action and how concurrency can be performed.

```
:: IOSystem v a p w c ::= { af :: a w -> (w,v)
                          , ap :: c a -> Bool
                          , pf :: p c -> (c,c)}
```

The above structure – three functions parameterised by five types, and one additional pre-condition which we discuss below – is enough to formally model an I/O system with deterministic concurrency. The following subsections explain each component in turn.

## 2.2 af – The Semantics of Actions

The function `af :: a w -> (w,v)` models the effect of each action on the world state. The type `w` is that of the world, or global state. `a` is the API, with each element identifying an I/O action that can be performed. Type `v` denotes return values. This is typically a sum type capable of storing `Ints`, `Bools`, `Chars` or any other value that an action needs to return.

For any action `a`, `af a :: w -> (w,v)` defines the state-transformer for that action.

## 2.3 ap - Contexts and Non-interference

Contexts are denoted by the type `c` and the meaning of each context is defined by `ap :: c a -> Bool`. `ap c a` is a `Bool` which indicates whether action `a` is permitted by context `c`. A context `c` can be thought of as the set of actions `a` such that `ap c a = True`.

The original purpose of contexts was to isolate certain groups of actions which don't interfere. We can say that two actions `al` and `ar` won't interfere with one another if, for all world states, the order in which they are performed is irrelevant. This is expressed as `al ||| ar`:

$$\begin{aligned} a_l \parallel a_r &\triangleq \forall w. \forall w_2. \forall v_l. \forall v_r \\ &(\exists w_1. \mathbf{af} \ a_l \ w = (w_1, v_l) \wedge \mathbf{af} \ a_r \ w_1 = (w_2, v_r)) \\ &\Leftrightarrow \\ &(\exists w_1. \mathbf{af} \ a_r \ w = (w_1, v_r) \wedge \mathbf{af} \ a_l \ w_1 = (w_2, v_l)) \end{aligned}$$

---

<sup>1</sup> This means that any two non-terminating programs become indistinguishable – a rather worrying problem. The CCS approach to modelling I/O [12] doesn't suffer from this, however, and since we see no immediate reason why one can't have the best of both worlds, this will be a topic of future work.

Now we can define two important relations on contexts:

$$\begin{aligned} c_1 \parallel c_2 &\triangleq \forall a_1. \forall a_2. \mathbf{ap} \ c_1 \ a_1 \wedge \mathbf{ap} \ c_2 \ a_2 \implies a_1 \parallel a_2 \\ c \sqsubseteq c_1 &\triangleq \forall a. \mathbf{ap} \ c \ a \implies \mathbf{ap} \ c_1 \ a \end{aligned}$$

$c_1 \parallel c_2$  states that for all actions  $a_1$  permitted under context  $c_1$  and for all actions  $a_2$  permitted under context  $c_2$ , the ordering of actions  $a_1$  and  $a_2$  is irrelevant.  $c \sqsubseteq c_1$  states that if an action is permitted to run in context  $c$  then it will also be permitted in context  $c_1$ .

$\parallel$  is symmetric and  $\sqsubseteq$  is a pre-order – both by definition.

## 2.4 pf - Enforcing Non-interference

Assume that we have modelled our API and created a set of contexts which model all the different permissions a program might well be allowed to have. We would now like to guarantee that if a program running in context  $c$  performs a `fork` which results in two programs with contexts  $c_l$  and  $c_r$ , then

1. neither process will interfere with one another:  $c_l \parallel c_r$ .
2. neither process will be capable of performing an action forbidden by the enclosing parent context:  $c_l \sqsubseteq c \wedge c_r \sqsubseteq c$ .

We solve this problem by defining a function  $\mathbf{pf} :: \mathbf{p} \ c \ \rightarrow \ (c, c)$ , which we assume obeys that very pre-condition:

$$\mathbf{PRE}_{\mathbf{af}, \mathbf{ap}, \mathbf{pf}} \triangleq \forall c. \forall c_l. \forall c_r. \forall p. \mathbf{pf} \ p \ c = (c_l, c_r) \implies c_l \parallel c_r \wedge c_l \sqsubseteq c \wedge c_r \sqsubseteq c$$

A value of type  $\mathbf{p}$  is an extra parameter which gives the programmer some flexibility with regard to how he wishes the current context to be split. If a program is running in context  $c$ , the programmer `forks` supplying the value  $p$ , and  $\mathbf{pf} \ p \ c = (c_l, c_r)$ , then the new left- and right-hand processes will execute in contexts  $c_l$  and  $c_r$  respectively. When both of these terminate, the execution of the parent process will continue again in context  $c$ .

To prevent visual clutter, for the rest of the paper we assume the existence of some implicit `IOSystem` called  $\mathbf{s}_{\mathbf{af}, \mathbf{ap}, \mathbf{pf}}$ . We assume this defines the functions  $\mathbf{af}$ ,  $\mathbf{ap}$  and  $\mathbf{pf}$ , and binds the types  $\mathbf{v}$ ,  $\mathbf{a}$ ,  $\mathbf{w}$ ,  $\mathbf{c}$  and  $\mathbf{p}$ . Unless stated otherwise, all results generalise over all of these values.

## 3 A Language with Concurrent I/O

In this section we define a language which implements concurrency as described in the previous section in a functional style.

### 3.1 Syntax

The language is defined directly in Clean. Programs are elements of the higher-order algebraic data-type `Prog v a p`.

```

:: Prog v a p = Bind (Prog v a p) (v -> Prog v a p)
              | Ret v
              | Act a (Prog v a p)
              | Par p (Prog v a p) (Prog v a p) (v v -> v)

```

**Ret** and **Bind** are similar in spirit to Haskell’s monadic **return** and **>>=** respectively. **Ret v** returns **v** without changing the global state. **Bind m f** performs **m**, and if **m** terminates with some resultant value **v**, it then performs the program **f v**. The program **Act a m** performs action **a** if it is permitted in the current context. If it isn’t permitted it runs program **m** instead, where **m** would typically act as a sort of exception handler, providing an alternative return value indicating that the action wasn’t allowed. **Par p ml mr vf** runs **ml** and **mr** in parallel, splitting the context as determined by **p**. If both **ml** and **mr** terminate with values **vl** and **vr** respectively, the whole program yields a return value **vf vl vr**.

Since we are using a normal algebraic type, the type signatures of the four constructors are quite a lot weaker than we would like – especially those of **Bind** and **Par**. The upshot is that all return values must be an element of the same fixed type **v**.

These problems could be solved using an extra type variable to indicate the program’s resultant return type (as opposed to the return type of actions) and some existential typing. We don’t do this. One reason is that Sparkle has no facilities for reasoning about existential types. More importantly, the problem doesn’t really limit the flexibility of our results. Since the type-variable **v** isn’t constrained in any way, intuitively it can be made polymorphic. The only reason that we use an algebraic type at all is to make explicit the fact that there are four and only four ways of constructing a **Prog**.

### 3.2 Single-Step Reduction Rules

We define the operational semantics for the language at a high level of abstraction using non-deterministic single-step reduction.

We use the following syntactic sugar, choosing infix notation for **Bind**, **Par** and the function **vf :: v v -> v**.

$$\begin{aligned}
m_l \parallel^{*p} m_r &\triangleq \text{Par } p \ m_l \ m_r \ (*) \\
m \gg= f &\triangleq \text{Bind } m \ f \\
w \Vdash m &\longrightarrow_c w_1 \Vdash m_1 \triangleq \text{“program } m \text{ with world-state } w \text{ may single-step} \\
&\text{reduce under context } c \text{ to program } m_1 \text{ with} \\
&\text{world-state } w_1\text{.”}
\end{aligned}$$

The seven reduction rules can be found in Figure 1.

The first two refer to the interaction between **Ret** and **Bind**: evaluation always proceeds (recursively) from left to right. The third and fourth describe the behaviour of **Act a m** as described above. The final three rules refer to

concurrency. The first of these states that when both sides are finished then the parallel execution of both has finished. It is the final two rules which introduce non-determinism into the single-step semantics: if the left-hand-side or the right-hand-side can be reduced then any arbitrary one may chosen.

$$w \Vdash \mathbf{Ret} v \gg\!\!\gg f \longrightarrow_c w \Vdash f v \quad (1)$$

$$\frac{w \Vdash m \longrightarrow_c w' \Vdash m'}{w \Vdash m \gg\!\!\gg f \longrightarrow_c w' \Vdash m' \gg\!\!\gg f} \quad (2)$$

$$\frac{\mathbf{af} a w = (w', v') \quad \mathbf{ap} c a = \mathbf{True}}{w \Vdash \mathbf{Act} a m \longrightarrow_c w' \Vdash \mathbf{Ret} v'} \quad (3)$$

$$\frac{\mathbf{ap} c a = \mathbf{False}}{w \Vdash \mathbf{Act} a m \longrightarrow_c w \Vdash m} \quad (4)$$

$$\frac{}{w \Vdash (\mathbf{Ret} v_l) \parallel^{*p} (\mathbf{Ret} v_r) \longrightarrow_c w \Vdash \mathbf{Ret} v_l * v_r} \quad \mathbf{pf} p c = (c_l, c_r) \quad (5)$$

$$\frac{w \Vdash m_l \longrightarrow_{c_l} w' \Vdash m'_l}{w \Vdash m_l \parallel^{*p} m_r \longrightarrow_c w' \Vdash m'_l \parallel^{*p} m_r} \quad \mathbf{pf} p c = (c_l, c_r) \quad (6)$$

$$\frac{w \Vdash m_r \longrightarrow_{c_r} w' \Vdash m'_r}{w \Vdash m_l \parallel^{*p} m_r \longrightarrow_c w' \Vdash m_l \parallel^{*p} m'_r} \quad \mathbf{pf} p c = (c_l, c_r) \quad (7)$$

**Fig. 1.** Single-Step Reduction Rules

### 3.3 Implementation

We implement single-step reduction as a Clean function which modifies the program/state pair given the context it has to be reduced in. However, the above language aims to leave the order in which concurrent actions are performed unspecified. Since we are reasoning about an implementation in a deterministic language like Core-Clean it is necessary to emulate this randomness or lack of knowledge. One approach might be for the reduction function to return a list containing the possible resultant programs after single-step reduction. Instead, the solution we chose was to supply an extra argument to the single-step reduction function.

```

:: Random ::= ![Bool]
next :: (IOSystem v a p w c) c Random (Prog v a p, w) -> (Prog v a p, w)

```

`next` implements single-step reduction. The extra argument mentioned is of type `Random` and acts as a sort of random number generator. One individual

**Random** (that is, strict list of **Bool**<sup>2</sup>) is consumed for each single-step reduction – one boolean value for each syntactic level of parallelism. The boolean values are only referred to when a non-deterministic choice needs to be made between reducing the left- or right-hand-side of a parallel computation. If it's **False**, go left; if it's **True**, go right. If the list isn't long enough the evaluator just defaults to the value **False**.

We can now define the implementation of single-step reduction by existentially quantifying over the possible **Random** values.

$$w \Vdash m \longrightarrow_c w_1 \Vdash m_1 \triangleq \exists r. r \neq \perp \wedge \text{next } \mathbf{s}_{\text{af}, \text{ap}, \text{pf}} \ c \ r \ (m, w) = (m_1, w_1)$$

### 3.4 Evaluation

Evaluation is the process of continually single-step reducing a program until it becomes a single value of the form **Ret v**. Although we can write a function which does just that, in order to prove properties formally we must be precise about the number of reduction steps required.

```
:: Random2 ::= ![Random]
rdce :: Int (IOSystem v a p w c) c Random2 (Prog v a p, w) -> (Prog v a p, w)
```

**rdce** iterates the single-step reduction function **next** a specific non-negative number of times. It requires a value of type **Random2** (a strict list of **Random**) as a parameter because each application of **next** on its own needs a fresh **Random**. With every iteration another **Random** is consumed from the list, defaulting to **[]** if the list is exhausted.

Like with reduction, we again define some more syntactic sugar.

$$w \Vdash m \Downarrow^c \langle v, w_1 \rangle \triangleq \exists q. q \neq \perp \wedge \exists i. \text{rdce } i \ \mathbf{s}_{\text{af}, \text{ap}, \text{pf}} \ c \ q \ (m, w) = (\text{Ret } v, w_1)$$

$$w \Vdash m \Downarrow^c \langle v, w_1 \rangle \triangleq \forall q. q \neq \perp \implies \exists i. \text{rdce } i \ \mathbf{s}_{\text{af}, \text{ap}, \text{pf}} \ c \ q \ (m, w) = (\text{Ret } v, w_1)$$

The first,  $w \Vdash m \Downarrow^c \langle v, w_1 \rangle$ , states that  $w \Vdash m$  may possibly evaluate to  $w_1 \Vdash \text{Ret } v$  in context  $c$ , depending on which non-deterministic choices are made.  $w \Vdash m \Downarrow^c \langle v, w_1 \rangle$ , on the other hand, is stronger. It states that  $w \Vdash m$  *always* evaluates to  $w_1 \Vdash \text{Ret } v$  in context  $c$  (which, of course, if true, implies that it possibly can). It is the second, stronger property which we want and the purpose of the confluence proof is to show that if  $\text{PRE}_{\text{af}, \text{ap}, \text{pf}}$  holds then the two are in fact the same: if a program can evaluate to some resultant state then it won't ever do anything else.

<sup>2</sup> An infinite, lazy stream of **Bool** might seem more appropriate but this isn't the case. Laziness also introduces partiality and we then need a messy pre-condition on every stream ensuring that each individual **Bool** is defined. With a strict list  $r$  we get this condition automatically by asserting simply that  $r \neq \perp$ .

Quite often we need to be explicit about the number of reduction steps performed. In these situations we annotate the evaluation with that number, as follows:

$$w \Vdash m \Downarrow_i^c \langle v, w_1 \rangle \triangleq \exists q. q \neq \perp \wedge \text{rdce } i \text{ s}_{\text{af}, \text{ap}, \text{pf}} c q (m, w) = (\text{Ret } v, w_1)$$

$$w \Vdash m \Downarrow_i^c \langle v, w_1 \rangle \triangleq \forall q. q \neq \perp \implies \text{rdce } i \text{ s}_{\text{af}, \text{ap}, \text{pf}} c q (m, w) = (\text{Ret } v, w_1)$$

### 3.5 Failure

Since the language's semantics and individual programs contain arbitrary functions, it is clear that reduction can fail or not terminate. We define failure simply to be the case that no number of reduction steps would ever yield a single value. It can happen in a number of different ways:

- An action fails - that is, for some action  $a$ ,  $\text{ap } a w = \perp$ .
- When running program  $m \ggg f$ ,  $f$  doesn't terminate when applied to  $m$ 's return value.
- A program is syntactically ill-defined. For example:  $\perp \parallel^* \perp$ .
- The functions  $\text{ap}$  and  $\text{pf}$  happen to return  $\perp$ .
- The program loops infinitely, continually performing I/O actions.

## 4 Proving Confluence

In this section we show that if  $\text{PRE}_{\text{af}, \text{ap}, \text{pf}}$  holds then the non-deterministic single-step semantics are confluent with respect to program evaluation. In other words, the arbitrary choices made when reducing any given program have no effect on the resultant global-state and return value.

The full confluence proof is large and requires many smaller results. Only the more important ones are shown below.

**Lemma 1.** *If  $\text{PRE}_{\text{af}, \text{ap}, \text{pf}}$  holds and single-step reducing a program is successful, either it didn't change the world-state at all or it performed a single action which was permitted by that program's context.*

*Proof.* Structural induction over  $\text{Prog}^3$ . The only way the world-state can be changed is by performing an action. Because of the properties guaranteed by  $\text{PRE}_{\text{af}, \text{ap}, \text{pf}}$ , no forbidden action can be performed at a deeper level which might have been forbidden at the top level.

**Lemma 2.**

$$\begin{aligned} & \text{PRE}_{\text{af}, \text{ap}, \text{pf}} \wedge (\exists p. \exists c. \text{pf } p c = (c_l, c_r)) \implies \\ & (\exists w_1. w \Vdash m_l \longrightarrow_{c_l} w_1 \Vdash m_{l1} \wedge w_1 \Vdash m_r \longrightarrow_{c_r} w_2 \Vdash m_{r1}) \\ & \iff \\ & (\exists w_1. w \Vdash m_r \longrightarrow_{c_r} w_1 \Vdash m_{r1} \wedge w_1 \Vdash m_l \longrightarrow_{c_l} w_2 \Vdash m_{l1}) \end{aligned}$$

<sup>3</sup> This isn't usually possible. On this one occasion we have no need to reason about  $f$  in a program of the form  $m \ggg f$ .

If two contexts don't interfere then the order in which one single-step reduces those two program in the different contexts is irrelevant.

*Proof.* A direct application of Lemma 1 and the non-interference properties guaranteed by  $\text{PRE}_{af, ap, pf}$ . If both both  $m_l$  and  $m_r$  performed an action then their order was irrelevant.

**Lemma 3.**

$$\begin{aligned} & \text{PRE}_{af, ap, pf} \wedge (\exists p. \exists c. pf \ p \ c = (c_l, c_r)) \implies \\ & (\exists w_1. w \Vdash m_l \Downarrow_{i_l}^{c_l} \langle v_l, w_1 \rangle \wedge w_1 \Vdash m_r \Downarrow_{i_r}^{c_r} \langle v_r, w_2 \rangle) \\ & \iff \\ & (\exists w_1. w \Vdash m_r \Downarrow_{i_r}^{c_r} \langle v_r, w_1 \rangle \wedge w_1 \Vdash m_l \Downarrow_{i_l}^{c_l} \langle v_l, w_2 \rangle) \end{aligned}$$

The evaluation order of two programs in two non-interfering contexts is irrelevant.

*Proof.* Induction over both  $i_l$  and  $i_r$ . Each single-step reduction in  $m_l$  is, in turn, exchanged with the other reductions in  $m_r$  so that it happens after  $m_r$  instead of before it, applying Lemma 2.

**Lemma 4.**

$$\frac{w \Vdash m \ggg f \Downarrow_i^c \langle v_2, w_2 \rangle}{\exists i_1. \exists v_1. \exists w_1. i_1 \geq 0 \wedge w \Vdash m \Downarrow_{i_1}^c \langle v_1, w_1 \rangle \wedge w_1 \Vdash f \ v_1 \Downarrow_{i-i_1-1}^c \langle v_2, w_2 \rangle}$$

If  $m \ggg f$  evaluates with some specific reduction order then there exists a specific reduction order for  $m$  which yields a value  $v_1$  and another reduction order for  $f \ v_1$  which, together, has the same resultant effect.

*Proof.* Induction on  $i$ . The initial **Random2** list, which determines the reduction order, is effectively sliced into two parts. The first part is the ordering for  $m$ , the second that for  $f \ v_1$ .

**Lemma 5.**

$$\frac{w \Vdash m \Downarrow_{i_1}^c \langle v_1, w_1 \rangle \quad w_1 \Vdash f \ v_1 \Downarrow_{i_2}^c \langle v_2, w_2 \rangle}{w \Vdash m \ggg f \Downarrow_{i_1+i_2}^c \langle v_2, w_2 \rangle}$$

If evaluation of  $m$  is confluent, always returning value  $v_1$ , and  $f \ v_1$  is also confluent, then so is the evaluation of  $m \ggg f$ .

*Proof.* Induction on  $i_1$ .

**Lemma 6.**

$$\begin{aligned}
 w \Vdash m_l \parallel^{*p} m_r \Downarrow_i^c \langle v, w_2 \rangle \wedge \text{PRE}_{af, ap, pf} \wedge \text{pf} p c = (c_l, c_r) \\
 \implies \\
 \exists i_1. \exists v_l. \exists v_r. \exists w_1. \left( \begin{array}{l} i_1 \geq 0 \wedge v = v_l * v_r \wedge \\ w \Vdash m_l \Downarrow_{i_1}^{c_l} \langle v_l, w_1 \rangle \wedge \\ w_1 \Vdash m_r \Downarrow_{i-i_1-1}^{c_r} \langle v_r, w_2 \rangle \end{array} \right)
 \end{aligned}$$

If  $m_l \parallel^{*p} m_r$  evaluates with some arbitrary reduction order then there exists a reduction order for  $m_l$  and a reduction order for  $m_r$  such that executing both separately, one after the other, has exactly the same effect.

*Proof.* Induction on  $i$ . We split the **Random2** list used for both  $m_l$  and  $m_r$ , filtering the **Random** values into two separate lists depending on which of the two sub-programs received each value originally.

**Lemma 7.** if  $\text{pf} p c = (c_l, c_r)$  and  $\text{PRE}_{af, ap, pf}$ ,

$$\frac{w \Vdash m_l \Downarrow_{i_l}^{c_l} \langle v_l, w_1 \rangle \quad w_1 \Vdash m_r \Downarrow_{i-i_l-1}^{c_r} \langle v_r, w_2 \rangle \quad i \geq 0}{w \Vdash m_l \parallel^{*p} m_r \Downarrow_i^c \langle v_l * v_r, w_2 \rangle}$$

If evaluation of  $m_l$  and  $m_r$  are both confluent on their own when run sequentially in two non-interfering contexts  $c_l$  and  $c_r$ , then  $m_l \parallel^{*p} m_r$  is also confluent when run in an enclosing context  $c$ .

*Proof.* Induction over  $i$ . Depending on the random values, each reduction step in the parallel computation may pick either to reduce  $m_l$  or  $m_r$ . If it's  $m_l$  it is relatively easy. If  $m_r$  has to be single-step reduced we must show that reducing it once at the start is no different to doing it after  $m_l$  has been fully evaluated.

One awkward technicality is the proof that failure propagates sensibly: if neither  $m_l$  nor  $m_r$  fail, then it must be shown that failure cannot occur at a higher level either, regardless of reduction order.

**Theorem 1.** Confluence

$$\text{if } \text{PRE}_{af, ap, pf}, \text{ then } w \Vdash m \Downarrow_i^c \langle v, w_1 \rangle \implies w \Vdash m \Downarrow_i^c \langle v, w_1 \rangle$$

*Proof.* Strong induction over  $i$ . The base case,  $i = 0$ , is trivial since  $m$  is just a value. In the inductive case we perform case analysis on the three different recursive constructors (**Bind**, **Act** and **Par**). For each constructor, we

1. Decompose  $w \Vdash m \Downarrow_i^c \langle v, w_1 \rangle$  into the separate, sequential evaluation of  $m$ 's constituent sub-programs.

2. Apply the inductive hypothesis, thus guaranteeing that the different sub-programs are confluent when run in isolation. (The number of reduction steps in each sub-program must be strictly less than  $i$  – and in this case it always is.)
3. Show that the confluence of reduction is preserved when the sub-programs are executed as individual parts of the one program again.

Proving this for `Act` is not difficult. For `Bind`, steps 1 and 3 above are performed by Lemmas 4 and 5, and for `Par`, by Lemmas 6 and 7.

**Corollary 1.** *If  $\text{PRE}_{af, ap, pf}$ , then if a program can fail, it always will.*

*Proof.* By contradiction. If it didn't always fail, then it would for some reduction order succeed in evaluating, and therefore, by confluence, always evaluate.

## 5 An Example: A File System

### 5.1 Design Criteria

We want the file system to contain a potentially infinite number of files, each file containing any finite amount of data. Files can be opened for shared reading with multiple read-pointers, as well as (non-shared) reading and writing. Files can also be created and deleted.

The design and implementation of the file system is also influenced by the fact that we want the behaviour of certain actions to be independent of one another. Most notably:

- Actions on different files.
- Actions on different read-handles of the same file.

The model is not meant to be industrial strength. Nonetheless, we would like to think that it is a plausible simplification, capturing many reasonable everyday properties one would expect of a real file system.

### 5.2 File System State

The FS type models our file system:

```

:: FS      ::= MapN (Maybe FData)
:: OpenSt  = Closed | Open ![Maybe Ptr] | ReadWrite !Ptr
:: Hnd     = ReadH !Nam !FileDes | WriteH !Nam
:: Maybe a = Just !a | Nothing

:: FData   ::= !(Data, OpenSt)           :: Data ::= ![Char]
:: MapN d  ::= Nam -> d                  :: Nam   ::= Int
:: FileDes ::= Int                       :: Ptr   ::= Int

```

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 (strict) list

of characters and an `OpenSt` which indicates whether the file is closed, open for (shared) reading or open for both reading and writing.

If it's open for reading then the file maintains a list of active pointers. This list grows with each new shared-read. Each time a handle is closed it is replaced by `Nothing`, and if all handles have been closed then the entire file is then closed. If the file is open for writing then it just stores the one pointer. Once files are opened, they are accessed via handles of type `Hnd`. This structure contains enough information to find out where the required file-pointer is stored within the global-state.

### 5.3 File API and Return Values

There are fourteen primitive file system actions in our API, which we encode as an algebraic type. As in [5], we trim the meaning of each action down to its bare, logical minimum. That is: lots of actions, each doing a very specific task.

```

:: FSAction = FOpen Nam Bool | HClose Hnd | FClose Nam | HRead Hnd
             | HWrite Hnd Char | HNext Hnd | HEOF Hnd | HRewind Hnd
             | HValid Hnd | FISOpen Nam | FISRead Nam | FCreate Nam
             | FDelete Nam | FExists Nam

```

Seven of the fourteen actions act on handles (which themselves refer to specific files); the others just act on filenames.

Each action can return information using the dynamic return type `RV`.

```

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

```

We also define two useful look-up functions with the following types and the obvious definitions.

```

actNam :: FSAction -> Nam           hndNam :: Hnd -> Nam

```

### 5.4 Defining a State-Transformer

The common patterns of behaviour to do with non-interference mentioned above are much easier to guarantee and reason about if they are enforced directly.

We can be certain that two actions on different pieces of global state don't interfere if:

- They only modify their own piece of global state.
- Both their resultant return value and the way that they modify the state is solely determined by the action's own parameters and the original value of that piece of local state.

For this reason, all API-calls are modelled as state-transformers on individual *files*. Additionally, API-calls on file-handles are (mostly) modelled as state-transformers on individual *file-pointers* (for one specific file).

**Actions on Individual Files.** The meaning of each action is given by a function of type `(Maybe FData) -> (Maybe FData, RV)`. This is converted into a state-transformer on FS with the function `liftMapN` (the details are omitted).

```
liftMapN :: !Nam (d -> (d,r)) (MapN d) -> (MapN d, r)
```

**Lemma 8.** *if  $n_1 \neq n_2$ , then, for any two state-transformers  $f_1$  and  $f_2$ , the execution of `liftMapN n1 f1` and `liftMapN n2 f2` is order independent.*

*Proof.* Relatively easy in the absence of failure. The definition of `liftMapN` must strictly evaluate the local state before and after the (local) state-transformer is applied to guarantee that failure propagates symmetrically.

**Actions on Individual Pointers.** Certain actions only modify the value of one pointer in a file, the identity of that pointer being determined by a handle. These actions may examine the contents of that file, but cannot examine the values of any other file-pointers. This interface is enforced using `liftPtrsMapN`.

```
liftPtrsMapN :: (![Char] (Maybe Ptr) -> (Maybe Ptr, r)) Hnd ->
  (Maybe FData -> (Maybe FData, r))
```

**Lemma 9.** *If  $h_1$  and  $h_2$  are both read-handles referring to the same file but referring to different pointers, then the execution of `liftPtrsMapN f1 h1` and `liftPtrsMapN f2 h2` is order independent.*

*Proof.* Not too difficult, but does require a library of standard strict list theorems.

## 5.5 Implementing the API

Finally, we build the full API state-transformer `afFS` using the `liftMapN` function. An individual API-call may also employ `liftPtrsMapN`.

To save space we just show the implementation of `FOpen` and `HRead`.<sup>4</sup>

```
afFS :: FSAction FS -> (FS,RV)
afFS a w = liftMapN (actNam a) (actFn a) w

actFn :: (Maybe FData) -> (Maybe FData, RV)
actFn (FOpen n b) = fOpen_ n b
actFn (HRead h)   = liftPtrsMapN hRead_ h
actFn (HWrite h c) = // .....
```

```
fOpen_ :: !Nam Bool (Maybe FData) -> (Maybe FData, RV)
fOpen_ n False (Just (cs,Closed)) =
  (Just (cs, Open [Just 0]), RHnd (ReadH n 0))
```

<sup>4</sup> The `HRead` action is a little different to a normal POSIX-style read. It only reads the character - to increment the file pointer afterwards one must then use `HNext`.

```
fOpen_ n False (Just (cs,Read ps)) =
  (Just (cs, Open ps++[Just 0]), RHnd (ReadH n (length ps)))
fOpen_ n True  (Just (cs,Closed))  =
  (Just (cs, ReadWrite 0), RHnd (WriteH n))
```

```
hRead_ :: ![Char] (Maybe Ptr) -> (Maybe Ptr, RV)
hRead_ cs (SJust p) = (SJust p, RChar (cs!!p))
```

## 5.6 Resultant Properties

The above file system partially defines an `IOSystem` with type variables `v`, `a` and `w` bound to `RV`, `FSAction` and `FS` respectively. The function `afFS` gives the semantics of each action.

The file system implementation obeys two important properties.

**Lemma 10.** *If  $actNam\ a_1 \neq actNam\ a_2$ , then  $a_1 \parallel a_2$*

*Proof.* A direct consequence of Lemma 8.

**Lemma 11.** *If  $a_1$  and  $a_2$  are actions which act on the same file but different read-pointers, then  $a_1 \parallel a_2$*

*Proof.* The actions `HRead`, `HEOF`, `HNext`, `HRewind` are straightforward (using Lemma 9) since they are defined directly using `liftPtrsMapN`. `HValid` and `HClose` are a little special and require more work.

## 6 File System Contexts

### 6.1 Partitioning Contexts on Files

A simple but useful example is to identify contexts with sets of files that a program is allowed to access. The context is a map from `Nam` to `Bool` – a look-up table used to determine if files of a particular name can be accessed.

```
:: CEasy ::= MapN Bool
:: PEasy ::= [Nam]
```

```
apEasy :: CEasy FSAction -> Bool
apEasy c a = c (actNam a)
```

```
pfEasy :: PEasy CEasy -> (CEasy,CEasy)
pfEasy p c = (\n -> c n && not (isMember n p),
              \n -> c n && isMember n p)
```

**Lemma 12.**  $\text{PRE}_{afFS, apEasy, pfEasy}$

*Proof.* The properties  $c_l \sqsubseteq c$  and  $c_r \sqsubseteq c$  are trivial ( $(c\ n \ \&\&\ \text{not}\ (\text{isMember}\ n\ p))$  and  $(c\ n \ \&\&\ \text{isMember}\ n\ p)$  both imply  $(c\ n)$ ). The property  $c_l \parallel c_r$  is proved by first showing that any respective left- and right-hand action  $a_l$  and  $a_r$

permitted by these contexts can't act on the same file. (If it did, that filename simultaneously would and would not be an element of the list  $p$ .) Once we know this, use Lemma 10.

The lemma proved above guarantees confluence. The program `Par ns ml mr vf` runs programs `ml` and `mr` in parallel. It attempts to give `mr` access to as many of the files referred to in `ns` as possible. `ml` is allowed to access any remaining files.

## 6.2 Example of File-Based Partitioning

Figure 2 gives a small example of file-based partitioning.

The program `totalLength ns` calculates concurrently the sum-total of the lengths of each of the files named in list `ns`. If one filename appears twice in `ns`, only one process will be able to access it. If a file is “locked-out” by the current context, `fileLength` will simply return 0.

```
totalLength :: [Nam] -> Prog RV FSAction PEasy
totalLength ns = foldr (\n1 m1 ->
  Par [n1] m1 (fileLength n1)
      (\(RInt i1) (RInt i2) -> RInt (i1+i2))
  (Ret (RInt 0))) ns

fileLength :: Nam -> Prog RV FSAction PEasy
fileLength n =
  Bind (Act (FOpen n) (Ret RNull))
    (\v -> case v of
      RNull -> Ret (RInt 0)    // (if file access is denied)
      RHnd h -> Bind (fileLenLoop n 0)
                    (\l -> Bind (Act (HClose h) undef) (\_ -> Ret 1)))

// keep incrementing the handle, counting the length.
fileLenLoop :: Hnd Int -> Prog RV FSAction PEasy
```

**Fig. 2.** Computing File Lengths

## 6.3 Shared Reads

The context data doesn't have to just include what files the programmer is allowed to access. It can also be modified to include information about what specific actions the programmer is allowed to perform on those files.

By modifying the contexts to incorporate information about which specific handles one is allowed to read from, a limited kind of shared reads are permitted. It is limited because an `FOpen` on one file cannot run concurrently with any action on that file – all read-handles must be opened before any concurrency takes place at all. The problem is that while contexts can enforce a property such as “don't allow a `HRead` read from handle `h`”, it can't be expected to enforce a property like “don't permit a `FOpen` to run if it possibly could return a handle `h`”.

So the current file system model is slightly inadequate. However, by modifying it a little so as to separate the creation of a handle from the opening of a file, this problem should be avoidable. This will be the subject of future work.

## 7 Conclusions and Future Work

We have developed a language for expressing and reasoning about state-based I/O with concurrency. By adding contexts to the language it can be shown that with certain pre-conditions concurrent evaluation is deterministic.

Future work shall involve more sophisticated and realistic I/O models, including models of non-state-based I/O such as stream I/O. Stream I/O is perhaps the most pressing issue since at the moment two concurrent sub-programs are unable communicate with one another. Also important is the ability to properly distinguish non-terminating programs. Two other possible future directions are the development of Hoare-like proof rules for reasoning about languages with context and an investigation into whether types could be used to statically check some of the run-time properties required by contexts.

## Acknowledgments

The authors would like to thank the anonymous referees, and Glenn Strong and Shane O’Conchuir. Their helpful suggestions improved the quality of this paper.

## References

1. E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.
2. 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.
3. D. Carter. Deterministic concurrency. Master’s thesis, Department of Computer Science, University of Bristol, September 1994.
4. 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.
5. M. Dowse, A. Butterfield, M. van Eekelen, M. de Mol, and R. Plasmeijer. Towards machine verified proofs for I/O. Technical Report NIII-R0415, University of Nijmegen, April 2004.
6. 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 IFL 2002*, volume LNCS2670, pages 68–83, 2002.
7. 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.

8. C. Hall and K. Hammond. A dynamic semantics for haskell (draft), May 20 1993.
9. I. Holyer and E. Spiliopoulou. Concurrent monadic interfacing. In *IFL '98, 10th International Workshop, Selected Papers, London, UK, September 1998*, pages 73–89. Lecture Notes in Computer Science, Volume 1595, Springer Verlag, June 1999.
10. P. Hudak and R. S. Sundaresh. On the expressiveness of purely functional I/O systems. Technical report, Yale University, 1989.
11. M. P. Jones and L. Duponcheel. Composing monads. Technical report, Yale University, December 1993.
12. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In ACM, editor, *POPL '96: Florida, 21–24 January 1996*, pages 295–308, New York, NY, USA, 1996. ACM Press.
13. S. Peyton Jones and P. Wadler. Imperative functional programming. In ACM, editor, *POPL '93: Charleston, January 10–13, 1993*, pages 71–84, New York, NY, USA, 1993. ACM Press.
14. R. Plasmeijer and M. van Eekelen. Concurrent clean version 2.0 language report. <http://www.cs.kun.nl/~clean/>, December 2001.

## A Sparkle Section Files

The Sparkle section files, which contains the proofs of all lemmas, theorems and numbered equations in this paper, can be obtained from:

[http://www.cs.tcd.ie/research\\_groups/fmg/archive/ifl2004proofs.html](http://www.cs.tcd.ie/research_groups/fmg/archive/ifl2004proofs.html)