Modelling Deterministic Concurrent I/O

Malcolm Dowse *
Trinity College Dublin, Ireland
dowsem@tcd.ie

Andrew Butterfield
Trinity College Dublin, Ireland
Andrew.Butterfield@cs.tcd.ie

Abstract

The problem of expressing I/O and side effects in functional languages is a well-established one. This paper addresses this problem from a general semantic viewpoint by giving a unified framework for describing shared state, I/O and deterministic concurrency. We develop a modified state transformer which lets us mathematically model the API, then investigate and machine verify some broad conditions under which confluence holds. This semantics is used as the basis for a small deterministic Haskell language extension called CURIO, which enforces determinism using runtime checks.

Our confluence condition is first shown to hold for a variety of small components, such as individual shared variables, 1-to-1 communication channels, and I-structures. We then show how models of substantial APIs (like a modification of Haskell's file I/O API which permits inter-process communication) may be constructed from these smaller components using "combinators" in such a way that determinism is always preserved. We describe combinators for product, name-indexing and dynamic allocation, the last of which requires some small extensions to cater for process locality.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Monads, effects, I/O, concurrency, determinism

1. Introduction

Both expressing and giving a semantics to I/O has been a long-standing difficulty in functional languages. This is because I/O represents a side effect that changes the state of a unique, uncopyable external world. Given that pure functional languages rely on copying as a means of ensuring referential transparency, the integration of I/O into such languages is problematic.

An early solution was to treat I/O as a special case which resulted in visible side effects—the approach adopted by ML [20] and Miranda [32]. More recent approaches have focussed on sophisticated type and effect systems [21], or have looked at ways and means for restricting programs to a form where I/O accesses are single-threaded. Two key approaches in this are the use of uniqueness types in the language Clean [1, 3] to statically ensure such single-threaded access, and the use of monads in Haskell [25, 33], which makes it structurally impossible to write I/O in other than a single-threaded manner (if we ignore the unsafe I/O operations such as `unsafePerformIO`).

This paper presents a minimal language called CURIO, which outlines a more flexible approach to both modelling I/O and expressing I/O in pure functional languages such as Haskell and Clean. The motivation for this is deterministic concurrency—relaxing single-threadedness to allow concurrent threads access to separate I/O resources. The benefits of this limited form of concurrency are: (i) the improved facility it makes available for structuring programs which conceptually consist of largely independent threads; and (ii) the ability to allow implementors to use concurrency to optimise performance, safe in the knowledge that the language semantics will be unaffected. Previously there had been few, if any, attempts to formally model this sort of behaviour, especially in the presence of inter-process communication.

CURIO is a small monadic language designed primarily for the theoretical study of I/O, concurrency and determinism. There are five key constructs, `>>=`, `return`, `action`, `par` and `test`, and determinism is enforced using run-time checks. We don't claim that the use of run-time checks is particularly novel in itself—static checks would indeed be better, and this could be promising future work. It is instead our rigorous semantics for describing the runtime checks that is the focus of this paper. CURIO's semantics is

- **expressive** One explicitly describes the API and the effects of each individual action. Programs could therefore be subject to formal correctness proofs.
- **general** A wide variety of different side effecting systems and APIs can be modelled under the same broad framework. This includes basic stores, I-structures, inter-process communication constructs, actions which dynamically allocate new structures, as well as traditional I/O, such as file system access.
- **deterministic** There is a formal condition which guarantees confluence in the presence of concurrency.
- **modular** The semantics of complicated, deterministic APIs may be given directly in terms of their smaller components using "I/O model" combinators. I/O models define both an API and the concurrency which it permits. The I/O model we give for Haskell, called $\mathfrak{io}$, is defined as:

  $$\mathfrak{term} * (\mathsf{smap}\ \mathfrak{file} * \mathsf{dmap}\ [] \ \mathfrak{chan})$$

  Here $\mathfrak{term}$, $\mathfrak{file}$ and $\mathfrak{chan}$ are models of terminal I/O, single file I/O and inter-process channel communication respectively. The `smap` combinator adds a mapping from filenames to files, the `dmap` combinator provides a facility for dynamic allocation, and the "*" combinator combines the APIs of two I/O models.

The paper is structured as follows: Section 2 presents an informal introduction to a possible extension to Haskell's I/O API. This is used as motivation for the more general CURIO system.

* Supported by Enterprise Ireland Research Grant SC/2002/0283

ICFP'06 September 16–21, 2006, Portland, Oregon, USA.

Sections 3 and 4 give a formal semantics for CURIO based on general I/O models, constructed from modified state transformers, and a (machine verified) proof of a condition under which confluence holds. Section 5 describes an extension to the I/O model concept, namely location-based models. These facilitate the modelling of dynamic allocation, and they are proved to be interconvertible with the original I/O models. Section 6 gives a description of combinators that allow us to build complex I/O models from simpler building blocks. Section 7 shows how this core language may provide a transparent, semantic back-end to a subset of Haskell's I/O API, extended to permit deterministic concurrency as described in Section 2. In Sections 8 and 9 we discuss related and future work, and Appendix A contains additional technical material.

The work in this paper significantly extends that of [10]. This earlier work did not mention inter-process communication, combinators or dynamic allocation, and the confluence proof was weaker.

2. A programmer-oriented view of CURIO

We begin in this section by giving an example of a functional I/O programming problem, and then informally describing a language and API extension to solve this.

2.1 Why deterministic concurrency?

I/O is sequenced in Haskell using monads [25, 33]. This is an elegant solution to the problem of I/O in pure functional languages, and monads obey a small set of useful algebraic laws. Monadic I/O is performed using the Haskell functions `>>=` and `return`. `return v` is a program which immediately terminates, yielding value `v`, and `m >>= f` is a program which first performs `m`, and then, if `m` terminates yielding `v`, performs `f v`, the application of function `f` to `v`. In Haskell, a program performing I/O which returns a value of type β is of type `IO` β. Example I/O operations are as follows:

`return`	::	$\forall\beta.\ \beta \rightarrow$ `IO` β
`(>>=)`	::	$\forall\beta.\ \forall\gamma.$ `IO` $\beta \rightarrow (\beta \rightarrow$ `IO` $\gamma) \rightarrow$ `IO` γ
`openFile`	::	`FilePath` $\rightarrow$ `IOMode` $\rightarrow$ `IO Handle`
`hGetChar`	::	`Handle` $\rightarrow$ `IO Char`

A recognised difficulty with using monads to enforce single-threaded access to the I/O world is that the programmer is forced to over-sequentialize their I/O code: I/O operations that are independent and which could be performed in any order still need to be given an explicit ordering in the program. In Peyton Jones' 15 year retrospective on the Haskell programming language [22] he mentioned that a more fine-grained way of partitioning I/O effects was one of two significant open challenges associated with the use of monads. What CURIO seeks to provide is a limited form of concurrency that reduces the need to over-sequentialize I/O code, while still retaining the determinism that allows us to use referential transparency when reasoning about I/O effects.

Concurrency, even when deterministic, and therefore limited, will still improve efficiency and user response time for I/O-bound programs, increase program reusability, and generally provide a better specification of program behaviour. As an example, consider a file compressor program to which we wish to give a terminal-based user interface. The file compressor and the interface should probably be understood as two relatively distinct pieces of code—one reads and writes to files, and the other communicates with the user. Yet if actions are sequenced explicitly then either user input will have to halt while a file is being encoded:

```
compressor fs = do
  cmd <- interact_with_user
  ...
  compress_whole_file fsource ftarg
```

or the two applications will have to be entangled together: (here, and throughout this paper, we use Haskell's convenient "do-notation")

```
compressor fs = do
  info <- compress_for_100ms_seconds fs
  ...
  interact_with_user
  compressor fs
```

Neither solution is particularly desirable. What we need is (i) a way of spawning processes, giving each certain permissions which don't conflict with one another—such as exclusive access to different files—and (ii) a way of creating 1-to-1 communication channels on the fly so that two processes can communicate with one another.

2.2 Concurrency and communication—`parIO` and `newChannel`

When performing concurrency, to retain determinism we must split the permissions of the original process so that they are distributed among the two new processes. When these processes terminate, the permissions should then be merged once again. Since the termination of the child processes must be synchronised, we adopt a symmetric concurrency construct in the same style as [18], in contrast to Concurrent Haskell's UNIX-like `forkIO` [24].

The question remains: how does the programmer specify the permissions, if available, that he or she would like each process to receive? Ideally the type system should handle these sorts of constraints, but for the moment we just give a basic solution: the programmer must specify the permissions directly. In our case, we may request that a process be given access to individual file handles. The call to create two parallel I/O processes is `parIO`, and has the following type, polymorphic in β and γ:

```
parIO :: ∀β. ∀γ. [Handle] → IO β → [Handle] →
    IO γ → IO (β,γ)
```

`parIO hsl ml hsr mr` concurrently executes programs `ml` and `mr`, attempting to give `ml` access to the handles in list `hsl`, and `mr` access to the handles in list `hsr`. It terminates when `ml` and `mr` have both terminated, returning their respective return values in a tuple. If there is any ambiguity or overlap in the permissions which we wish each process to have, by default the right-hand side receives access to the resource.

Our solution to inter-process communication is to only permit characters to be communicated, and to allow Haskell's `Handle` type to also refer to either the read or write handle of a communication channel. This means that when using `parIO` we can distribute read or write permissions to a channel as well. In order to guarantee determinism, we can only have one reader and one writer for any given channel, and, furthermore, the reading process may not check if data is available or perform time-outs while waiting for data.

`newChannel` creates a new communication channel, returning the new read and write handle.

```
newChannel :: IO (Handle,Handle)
```

The outline of a file compressor is given in Figure 1 using `parIO` and `newChannel`. The functions

```
wholeFile, readAccess :: FilePath → IO Handle
```

construct dummy handles which are used to indicate that a process would like complete access or read access to a file.

2.3 Process capabilities

We enforce determinism at runtime, not statically. If a process tries to perform an action which is forbidden, it fails, causing the whole

```
compressor :: [(FilePath,FilePath,Handle)] -> IO ()
compressor pdata = do
  cmd <- interact_with_user
  case cmd of
    CompressCmd fsource ftarg -> do
      (cr,cw) <- newChannel
      (retv,_) <- parIO
        [wholeFile ftarg, readAccess fsource, cw]
        (compress_and_communicate ftarg fsource cw)
        [cr, stdin, stdout]
        (compressor ((fsource,ftarg,cr):pdata))
    ExitCmd -> return ()
    ... other commands
```

Figure 1. Fragments of a file compressor

program to fail. To give a safer way for a program to deal with this, we must provide a few extra API calls to allow a process to check what actions are permitted.

```
 hAllowed  ::  Handle → IO Bool
fAllowedW  ::  FilePath → IO Bool
```

`hAllowed` checks to see if we can use the handle in question, be it a read handle or write handle. `fAllowedW` checks if a process has exclusive access to a file. Therefore, if we were to write defensive code, before attempting file access we would first check that our process had access to that file.

```
compress_and_communicate ftarg fsource cw =
  do b1 <- fAllowedW ftarg
     if b1 then (do
       ht <- openFile ftarg ReadWriteMode
       ..... compress file 'fsource'
       hClose ht)
     else (return ())
```

`parIO` defines a strict, almost syntactic binary tree of permissions (only the leaf processes are active at any one time). There is therefore no way for a process to exchange its permissions with that of another process at runtime.

3. Formal Semantics for CURIO

The API extension outlined in the previous section is a plausible one, but it is only an informal description, and the solution appears ad hoc and highly specific. In this section we describe a more general-purpose language called CURIO.

3.1 Generalising an API

Formally describing the API extension in Section 2 presents some immediate difficulties. We need a semantics for I/O actions (how do we usefully describe the meaning of file access, inter-processes communication and terminal I/O?), a formal definition of the non-interference of these actions, and a rigorous guarantee that our whole language is deterministic.

We begin by abstracting away from our API. Since we generalise over the API, we must generalise also over the monadic type constructor `IO`. The type $\texttt{Prog}_{\mathfrak{s}}\ \beta$ denotes a CURIO program which returns a value of type β, and whose API is defined by I/O model $\mathfrak{s}$. In our general framework, we define

$$\texttt{IO}\ \beta \triangleq \texttt{Prog}_{\mathfrak{io}}\ \beta$$

where $\mathfrak{io}$ is the I/O model defined, later, in Section 7.

Let us make the observation that the API calls of the previous section can be partitioned into three general classes:

- Those which perform actions, like `hPutChar`, `newChannel` and `openFile`.
- `parIO`, which is used to initiate concurrency.
- Those which check if actions can be performed, like `hAllowed`.

In place of these three classes we substitute three general primitives: `action`, `par` and `test`.

$$\begin{aligned}
\texttt{action} &:: \alpha \to \texttt{Prog}_{\mathfrak{s}}\ \nu \\
\texttt{par} &:: \forall\beta.\,\forall\gamma.\,\rho \to \texttt{Prog}_{\mathfrak{s}}\ \beta \to \texttt{Prog}_{\mathfrak{s}}\ \gamma \to \texttt{Prog}_{\mathfrak{s}}\ (\beta,\gamma) \\
\texttt{test} &:: \alpha \to \texttt{Prog}_{\mathfrak{s}}\ \texttt{Bool}
\end{aligned}$$

α is the type of primitive actions. `action` a performs an (atomic) action a, and returns a value of type ν. In Haskell, `getChar`, which reads a character from `stdin`, is just defined as primitive and at runtime executes the corresponding system call.

```
primitive getChar :: IO Char
```

Our aim, with the design of CURIO, was to retain this high level I/O interface but replace the primitive I/O actions with a semantics which tries to usefully model its behaviour. In CURIO, `getChar` is a library function implemented internally using `action` (the actual semantics will be given much later).

```
getChar = do (Left c) <- action (Left GetCh)
             return c
```

The "`Left`" constructors occur because we are selecting a variant of a sum type, produced by the I/O combinators (Section 6).

`par` is a generalisation of `parIO`. In Section 2, when performing concurrency one had to supply two lists of handles. With `par`, the parameter of type ρ abstracts away from this. `parIO` is implemented as:

```
parIO hsl ml hsr mr = par (encodeP hsl hsr) ml mr
  where encodeP hsl hsr = .....
```

where the function `encodeP` encodes the handle-lists as a single parameter of type ρ. I/O models, as defined later, monomorphically bind the three types α, ν and ρ.

`test` a returns `True` if action a is currently permitted, and can be used to implement `hAllowed`.

CURIO is monadic, so we also add `return` and `>>=`, bringing our total number of primitives to five.

$$\begin{aligned}
\texttt{return} &:: \forall\beta.\,\beta \to \texttt{Prog}_{\mathfrak{s}}\ \beta \\
(\texttt{>>=}) &:: \forall\beta.\,\forall\gamma.\,\texttt{Prog}_{\mathfrak{s}}\ \beta \to (\beta \to \texttt{Prog}_{\mathfrak{s}}\ \gamma) \to \texttt{Prog}_{\mathfrak{s}}\ \gamma
\end{aligned}$$

It is worth noting that the second argument of `>>=` is a function taking an arbitrary type β and returning a CURIO program of type $\texttt{Prog}_{\mathfrak{s}}\ \gamma$. This function argument can be expressed using the full range of constructs available in a lazy functional language, and not just the five CURIO primitives.

Finally, generalising over an API appears to be quite unusual. A reasonable question is, what can we expect to gain from this generality? There are three main benefits: (i) it shall become apparent that our language extension is applicable to many other situations; (ii) practically speaking it is easier to prove meta-properties for a small language; and (iii), we shall show later how small APIs which are unrealistic in their own right can be combined to form powerful and highly practical ones.

3.2 Semantics of actions

We now require a framework for describing the behaviour of our atomic actions. A state transformer is often satisfactory for this:

$$\mathsf{st} :: \alpha \to \omega \to (\omega,\nu)$$

Here, a mathematical function `st` defines the effect of each action of type α on a global state of type ω, and the value of type ν

which it returns. So, if we were describing the effects of actions on a store which was a mapping from variables identified by integers, to integer values, the function might be something like

$$\begin{array}{lllll} \mathsf{st}_{\mathtt{store}} & (\mathtt{Write}\ i\ v) & m & = & (m[i \mapsto v], 0) \\ \mathsf{st}_{\mathtt{store}} & (\mathtt{Read}\ i) & m & = & (m, m(i)) \end{array}$$

However, we want inter-process communication, too. A single communication channel could possibly be described in a state-like manner using a buffer implemented as a list. The sender and receiver independently manipulate the structure—the sender places a value at the end of the buffer, and the receiver takes the first item from the list.

$$\begin{array}{lllll} \mathsf{st}_{\mathtt{bfft}} & (\mathtt{Send}\ v) & vs & = & (vs +\!\!+ [v], 0) \\ \mathsf{st}_{\mathtt{bfft}} & (\mathtt{Rcve}) & (v{:}vs) & = & (vs, v) \end{array}$$

The difficulty here is that we have not specified what happens when "receiving" from an empty buffer. The receive action could return a token value indicating an error, but this would go against our desire for determinism—depending on scheduling, a receiving process may behave differently depending on whether a sending process has or has not got around to sending data.

What we need is a way of synchronising the access to shared state, and we modify our state transformer accordingly:

$$\mathsf{st} :: \alpha \to \omega \to (\omega, \nu) + \mathbb{S}$$

If $\mathsf{st}\ a\ w = \mathbb{S}$ then action a is said to be stalled (in world state w), and this indicates that it should not be performed just yet. If all actions are single-threaded, stalling makes no sense, but when other concurrent processes are also making arbitrary changes to the state, stalling allows one action to wait for another. Now, attempting to receive from an empty buffer will stall until another process has written to that buffer:

$$\begin{array}{lllll} \mathsf{st}_{\mathtt{bfft}} & (\mathtt{Rcve}) & (v{:}vs) & = & (vs, v) \\ \mathsf{st}_{\mathtt{bfft}} & (\mathtt{Rcve}) & [] & = & \mathbb{S} \end{array}$$

We make one final modification to state transformers. We add the possibility that an action can fail.

$$\mathsf{st} :: \alpha \to \omega \to (\omega, \nu) + \mathbb{S} + \bot$$

If $\mathsf{st}\ a\ w = \bot$ then action a will fail in world state w. If an action fails, then so does the whole program.

3.3 I/O contexts and I/O models

We have defined the semantics of actions of type α. Now we need to give the behaviour of ρ, which describes how process permissions are split. Each process' permissions are a set of actions, or an element of $\mathcal{P}\alpha$, the set of all subsets of α[1]. We refer to these permission sets as "I/O contexts." When initiating concurrency, the type ρ indicates how the current set of actions is to be distributed among the two new child processes. The function pf models this:

$$\mathsf{pf} :: \rho \to \mathcal{P}\alpha \to (\mathcal{P}\alpha, \mathcal{P}\alpha)$$

We have no need for a corresponding function which "combines" I/O contexts once two concurrent processes have terminated. This is because the parent process never goes away. It is merely suspended while it awaits the termination of both child processes.

We may now give the formal definition for I/O models. An I/O model is a tuple containing the functions st and pf, and it binds the types ν, α, ρ and ω:

$$\mathfrak{s} : \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega \triangleq (\mathsf{st} :: \alpha \to \omega \to (\omega, \nu) + \mathbb{S} + \bot,\ \mathsf{pf} :: \rho \to \mathcal{P}\alpha \to (\mathcal{P}\alpha, \mathcal{P}\alpha))$$

[1] Often these sets of actions will be infinite, which makes them unworkable as a data structure for a real implementation, but viewing I/O contexts as sets gives a simpler language definition. The metalanguage encoding, shown in Appendix A.1, uses a more practical solution.

This describes an API along with the potential for deterministic concurrency which it exhibits. Complete I/O models will be given in the next section, along with example programs.

3.4 Semantics of CURIO

The nondeterministic small-step semantics for CURIO is defined in Figure 2. Appendix A.1 briefly describes the meta-encoding of this language in the metalanguage, Core-Clean.

All the reduction rules describe the behaviour of what we call a world/program pair, written $w \Vdash m$, where w is the world and m is the program. This allows one to describe how a program and world state interact over time. The context $C \in \mathcal{P}\alpha$ in which a program is run also affects how the program behaves, so reduction rules are annotated with the current context. We say a CURIO program is a value if it is of the form `return` v for some v.

There are three reduction relations, $\longrightarrow^C$, $\uparrow^C$ and $\downarrow^C$.

$$\begin{array}{rcl} w \Vdash m \longrightarrow^C w' \Vdash m' & \triangleq & \text{“}w \Vdash m \text{ can reduce to } w' \Vdash m' \text{ in context } C\text{”} \\ w \Vdash m \uparrow^C & \triangleq & \text{“}w \Vdash m \text{ can fail in context C”} \\ w \Vdash m \downarrow^C & \triangleq & \text{“}w \Vdash m \text{ is in normal form in C”} \end{array}$$

CURIO is defined in a similar fashion to Haskell's I/O extensions in [23]. The behaviour of the I/O primitives is defined as an operational semantics layered on top of an implicit denotational semantics of the host language. This explains why it is necessary to mention failure, and the behaviour of an undefined CURIO program, $\bot$. A failed small-step reduction may denote a never-ending sequence of reduction steps or a runtime error in the metalanguage.

A world/program pair is in normal form if it is either a single value, or all concurrent actions it is attempting to perform are stalled. We say a world/program pair *is* in normal form, rather than can be in normal form, because despite nondeterminism it can be shown that a world/program pair is in normal form if and only if it cannot either fail or successfully reduce.

The runtime checks occur with each `action`, `test` and `par`. The behaviour of the first two is dependent on whether the action is permitted. `par` requires pf to be called and this in practice need only happen once, when the two new processes are first spawned.

4. A precondition for confluence

As is typical for concurrent languages, CURIO's small-step semantics is nondeterministic. In this section we describe determinism in CURIO, define a precondition $\mathsf{PRE}_{\mathfrak{s}}$ for I/O models which is formally proved to guarantee this, and give a few example I/O models.

4.1 Defining $\mathsf{PRE}_{\mathfrak{s}}$

To find a precondition for guaranteeing confluence we must first examine what it means for two actions to be order independent.

Let's begin with the basic case where actions cannot become stalled. By a standard definition, actions a_l and a_r are order independent if, for all w, w_2, v_l and v_r,

$$(\exists_{w_1 \in \omega}.\mathsf{st}\ a_l\ w = (w_1, v_l) \wedge \mathsf{st}\ a_r\ w_1 = (w_2, v_r)) \iff (\exists_{w'_1 \in \omega}.\mathsf{st}\ a_r\ w = (w'_1, v_r) \wedge \mathsf{st}\ a_l\ w'_1 = (w_2, v_l))$$

Under this definition, for every initial world state, if performing a_l and a_r is successful then swapping the execution order gives rise to the same resultant world state and the same return values—even though the intermediate world state may differ. So, for example, modifying two different locations in a store is order independent. This definition also guarantees that if failure occurs for one ordering, it will also occur for the other.

$$\frac{w \Vdash m \uparrow^C}{w \Vdash m \texttt{ >>= } f \uparrow^C} \qquad \frac{w \Vdash m \downarrow^C}{w \Vdash m \texttt{ >>= } f \downarrow^C} \ (m \text{ not a value}) \qquad \frac{w \Vdash m \longrightarrow^C w' \Vdash m'}{w \Vdash m \texttt{ >>= } f \longrightarrow^C w' \Vdash m' \texttt{ >>= } f}$$

$$w \Vdash \texttt{return } v \texttt{ >>= } f \longrightarrow^C w \Vdash f\ v \qquad w \Vdash \texttt{return } v \downarrow^C \qquad w \Vdash \bot \uparrow^C$$

$$\frac{a \in C \qquad \mathsf{st}\ a\ w = (w', v)}{w \Vdash \texttt{action } a \longrightarrow^C w' \Vdash \texttt{return } v} \qquad \frac{a \in C \qquad \mathsf{st}\ a\ w = \$}{w \Vdash \texttt{action } a \downarrow^C} \qquad \frac{a \in C \qquad \mathsf{st}\ a\ w = \bot}{w \Vdash \texttt{action } a \uparrow^C}$$

$$\frac{a \notin C}{w \Vdash \texttt{action } a \uparrow^C} \qquad \frac{a \in C}{w \Vdash \texttt{test } a \longrightarrow^C w \Vdash \texttt{return True}} \qquad \frac{a \notin C}{w \Vdash \texttt{test } a \longrightarrow^C w \Vdash \texttt{return False}}$$

$$\mathsf{pf}\ p\ C = (C_l, C_r) \begin{cases} w \Vdash \texttt{par } p\ (\texttt{return } v_l)\ (\texttt{return } v_r) \longrightarrow^C w \Vdash \texttt{return } (v_l, v_r) \\ \dfrac{w \Vdash m_l \longrightarrow^{C_l} w' \Vdash m_l'}{w \Vdash \texttt{par } p\ m_l\ m_r \longrightarrow^C w' \Vdash \texttt{par } p\ m_l'\ m_r} \ (m_r \neq \bot) \qquad \dfrac{w \Vdash m_l \uparrow^{C_l}}{w \Vdash \texttt{par } p\ m_l\ m_r \uparrow^C} \\ \dfrac{w \Vdash m_r \longrightarrow^{C_r} w' \Vdash m_r'}{w \Vdash \texttt{par } p\ m_l\ m_r \longrightarrow^C w' \Vdash \texttt{par } p\ m_l\ m_r'} \ (m_l \neq \bot) \qquad \dfrac{w \Vdash m_r \uparrow^{C_r}}{w \Vdash \texttt{par } p\ m_l\ m_r \uparrow^C} \\ \dfrac{w \Vdash m_l \downarrow^{C_l} \quad w \Vdash m_r \downarrow^{C_r}}{w \Vdash \texttt{par } p\ m_l\ m_r \downarrow^C} \ (m_l, m_r \text{ not both values}) \end{cases}$$

Figure 2. Nondeterministic small-step semantics for CURIO

When actions can become stalled, this definition must be extended. The crucial (and, as far as we know, original) observation is that if one action succeeds and the other is then stalled, *the stalled action must already have been stalled prior to the successful execution of the first action*. Put another way: two actions are order independent only if one cannot cause the other, if initially unstalled, to become stalled. For example, with model $\mathfrak{bfft}$, appending to the buffer cannot cause a receive action to become stalled. So these actions are order independent, since, when they are both unstalled, they also modify different parts of the same buffer.

We define two relations on actions to capture these ideas, $|||_{\mathfrak{s}}$ and $\mathsf{ally}_{\mathfrak{s}}$. Both are defined in Figure 3.

- $a_l \ |||_{\mathfrak{s}}\ a_r$: for any two actions a_l and a_r, if neither are stalled then the order in which they are executed is irrelevant—both with regard to their effect on world state and their return values.
- $\mathsf{ally}_{\mathfrak{s}}(a_l, a_r)$: if after performing a_l action a_r is stalled then it must have been stalled beforehand. The word "ally" hints at the fact that action a_l will not obstruct or hinder action a_r—they are in effect working with each other. This rules out competition for a limited resource.

We define a_l and a_r to be order independent if $a_l \ |||_{\mathfrak{s}}\ a_r$, $\mathsf{ally}_{\mathfrak{s}}(a_l, a_r)$, and $\mathsf{ally}_{\mathfrak{s}}(a_r, a_l)$, and using these, $C_l \ \Diamond_{\mathfrak{s}}\ C_r$ is defined to mean that all actions in context C_l are order independent with respect to all actions in context C_r. If, when performing concurrency with `par`, the parent process' I/O context C is split by pf into left- and right-hand processes' contexts C_l and C_r, and we wish to retain determinism, then:

- the child processes must not be able to perform actions forbidden by the parent: $C_l \subseteq C$ and $C_r \subseteq C$.
- the child processes should not interfere with one another: $C_l \ \Diamond_{\mathfrak{s}}\ C_r$

These observations motivate our definition of precondition $\mathsf{PRE}_{\mathfrak{s}}$, shown also in Figure 3.

4.2 $\mathsf{PRE}_{\mathfrak{s}}$ implies determinism

First, define $\twoheadrightarrow^C$ to be the reflexive, transitive closure of the reduction relation $\longrightarrow^C$. Then define $w \Vdash m \ \Big\downarrow^C\ w' \Vdash m'$ as meaning that both $w \Vdash m \twoheadrightarrow^C w' \Vdash m'$ and $w' \Vdash m' \downarrow^C$—that it is possible to reduce to a normal form in zero or more steps. We use $\Downarrow^C$, convergence, when this normal form is unique regardless of reduction order. $\Uparrow^C$, divergence, indicates that there is no combination of steps that lead to a normal form. The formal definition of these relations requires some details of our meta-encoding, and these can be found in Appendix A.1.

$w \Vdash m \ \Downarrow^C\ w' \Vdash m'$ implies $w \Vdash m \ \Big\downarrow^C\ w' \Vdash m'$ trivially, and the purpose of our confluence proof is to show that, if $\mathsf{PRE}_{\mathfrak{s}}$ holds, then $w \Vdash m \ \Big\downarrow^C\ w' \Vdash m'$ implies $w \Vdash m \ \Downarrow^C\ w' \Vdash m'$. The full proof [9] required many smaller lemmas, and only a brief sketch is given. For all results below we assume that $\mathsf{PRE}_{\mathfrak{s}}$ holds.

Lemma 4.1. *If $w \Vdash m \longrightarrow^C w' \Vdash m'$ then either $w = w'$ and for all w_1, $w_1 \Vdash m \longrightarrow^C w_1 \Vdash m'$, or for some action $a \in C$, $\mathsf{st}\ a\ w = (w', v')$, for some v'.*

Proof. Induction on m. □

Lemma 4.2. *If $C_l \ \Diamond_{\mathfrak{s}}\ C_r$, then if $w \Vdash m_l \longrightarrow^{C_l} w_l \Vdash m_l'$ and $w \Vdash m_r \longrightarrow^{C_r} w_r \Vdash m_r'$, then either*

- *there exists a w_2 such that both $w_r \Vdash m_l \longrightarrow^{C_l} w_2 \Vdash m_l'$ and $w_l \Vdash m_r \longrightarrow^{C_r} w_2 \Vdash m_r'$*
- *or $w_r \Vdash m_l \uparrow^{C_l}$ and $w_l \Vdash m_r \uparrow^{C_r}$.*

Proof. Case analysis, using Lemma 4.1, on whether one reduction performed an action. □

Lemma 4.3. *$w \Vdash m \uparrow^C$ implies $w \Vdash m \Uparrow^C$.*

Proof. Proof by induction over the number of reduction steps that program m cannot "escape" failure. □

$$
\begin{array}{rcl}
a_l \mid\mid\mid_{\mathsf{s}} a_r & \triangleq & \forall_{w\in\omega}.\forall_{w_2\in\omega}.\forall_{v_l\in\nu}.\forall_{v_r\in\nu}.\mathsf{st}\ a_l\ w \neq \$ \wedge \mathsf{st}\ a_r\ w \neq \$ \Longrightarrow \\
 & & (\exists_{w_1\in\omega}.\mathsf{st}\ a_l\ w = (w_1, v_l) \wedge \mathsf{st}\ a_r\ w_1 = (w_2, v_r)) \\
 & & \Longleftrightarrow \\
 & & (\exists_{w_1'\in\omega}.\mathsf{st}\ a_r\ w = (w_1', v_r) \wedge \mathsf{st}\ a_l\ w_1' = (w_2, v_l)) \\
\mathsf{ally}_{\mathsf{s}}(a_l, a_r) & \triangleq & \forall_{w\in\omega}.\forall_{w_1\in\omega}.\forall_{v\in\nu}.\mathsf{st}\ a_l\ w = (w_1, v) \wedge \mathsf{st}\ a_r\ w_1 = \$ \Longrightarrow \mathsf{st}\ a_r\ w = \$ \\
C_l \Diamond_{\mathsf{s}} C_r & \triangleq & \forall_{a_l\in C_l}.\forall_{a_r\in C_r}.a_l \mid\mid\mid_{\mathsf{s}} a_r \wedge \mathsf{ally}_{\mathsf{s}}(a_l, a_r) \wedge \mathsf{ally}_{\mathsf{s}}(a_r, a_l) \\
\mathsf{PRE}_{\mathsf{s}} & \triangleq & \forall_{p\in\rho}.\forall_{C\in\mathcal{P}\alpha}.\forall_{C_l\in\mathcal{P}\alpha}.\forall_{C_r\in\mathcal{P}\alpha}.\mathsf{pf}\ p\ C = (C_l, C_r) \Longrightarrow C_l \subseteq C\ \wedge\ C_r \subseteq C\ \wedge\ C_l \Diamond_{\mathsf{s}} C_r
\end{array}
$$

Figure 3. Formal definition of $\mathsf{PRE}_{\mathsf{s}}$ and associated relations

Lemma 4.4. *If both* $w \Vdash m \longrightarrow^C w_1 \Vdash m_1$ *and* $w_1 \Vdash m_1 \downarrow^C$ *then* $w \Vdash m \Downarrow^C w_1 \Vdash m_1$.

Proof. Induction on m, using Lemmas 4.3 and 4.2. This proves that if there is one small-step reduction before reaching normal form, then the redex must be unique. □

Lemma 4.5. *Diamond Property. If* $w \Vdash m \longrightarrow^C w_1 \Vdash m_1$ *and* $w \Vdash m \longrightarrow^C w_2 \Vdash m_2$ *then either*

- *The same redex was reduced:* $w_1 = w_2$ *and* $m_1 = m_2$.
- *There is a common reduct: for some* w_3 *and* m_3, $w_1 \Vdash m_1 \longrightarrow^C w_3 \Vdash m_3$ *and* $w_2 \Vdash m_2 \longrightarrow^C w_3 \Vdash m_3$.
- *Both sides will diverge:* $w_1 \Vdash m_1 \Uparrow^C$ *and* $w_2 \Vdash m_2 \Uparrow^C$.

Proof. Induction on m using Lemmas 4.4 and 4.3, and Lemma 4.2 as one of the base cases. □

Theorem 4.1. *Confluence. If* $w \Vdash m \Downarrow^C w_1 \Vdash m_1$ *then* $w \Vdash m \Downarrow^C w_1 \Vdash m_1$.

Proof. Induction on the number of reduction steps, using the diamond property and Lemma 4.3. □

4.3 Small Examples

We now give three small, and, on their own, largely impractical example I/O models, all of which obey $\mathsf{PRE}_{\mathsf{s}}$. These serve as easy examples of the relevant concepts and will also be used when assembling more practical I/O models using the combinators described, later, in Section 6.

For the store example given in the previous section, one possible context splitter $\mathsf{pf}_{\mathtt{store}}$ is that given in Figure 4. The context-splitting parameter $\rho_{\mathtt{store}}$ is defined to be a list of pairs, the first component of which identifies the store location, whilst the second denotes the required permission: read (`WantR`) or write (`WantW`). When performing concurrency with `par` one lists the permissions we would like to go to the left-hand process, if possible, and all the rest go the right-hand process.

```
storeExample :: Prog_store (Int,Int)
storeExample = par [(0,WantW),(1,WantR)]
  -- LHS wants to read int 1 and write to int 0
  (do {i1 <- action (Read 1); i0 <- action (Read 0)
       ; action (Write 0 i1); return i0})
  -- RHS wants to read int 1 and write to int 2
  (do {i1 <- action (Read 1); i2 <- action (Read 2)
       ; action (Write 2 i1); return i2})
```

The proof of $\mathsf{PRE}_{\mathtt{store}}$ relies on the fact that (i) all reads or writes to different integer variables are order independent, (ii) reads from the same integer variable are order independent, and (iii) that $\mathsf{pf}_{\mathtt{store}}$ guarantees that if one child process can write to a particular variable, the other must not be able to access it at all.

For an individual communication buffer, $\mathtt{bfft}$, a receive may be performed concurrently with a send, but that is really the only opportunity for deterministic concurrency. The idea for $\mathsf{pf}_{\mathtt{bfft}}$ is similar but on a smaller scale:

$$
\begin{array}{l}
\rho_{\mathtt{bfft}} \triangleq \mathtt{WantS} + \mathtt{WantR} + \mathtt{WantRS} + \mathtt{None} \\
\mathsf{pf}_{\mathtt{bfft}}\ p\ C = \\
\quad \begin{cases}
(C \setminus \{\mathtt{Rcve}\}, C \setminus \{\mathtt{Write}\ v \mid v \in \mathtt{Int}\}) & , p = \mathtt{WantS} \\
(C \setminus \{\mathtt{Write}\ v \mid v \in \mathtt{Int}\}, C \setminus \{\mathtt{Rcve}\}) & , p = \mathtt{WantR} \\
(C, \emptyset) & , p = \mathtt{WantRS} \\
(\emptyset, C) & , p = \mathtt{None}
\end{cases}
\end{array}
$$

Here we note that we can request permissions to perform zero (`None`), one (`WantR` or `WantS`) or both (`WantRS`) of the two actions available on a buffer: send or receive. In all cases, each permission can only be allocated to one process.

$\mathsf{PRE}_{\mathtt{bfft}}$ holds because the only two actions for which we must prove a non-interference property are those of the form `Rcve` and `Send` i (two concurrent `Send`s are forbidden, as are two concurrent `Rcve`s). Neither can cause the other to become stalled, and when both are unstalled they don't interfere with one another.

Our final example is model $\mathtt{istr}$, a single integer I-structure [2]. I-structures are concurrent, deterministic, write once data structures which have been used to develop elegant algorithms for computing matrices efficiently.

$$
\begin{array}{llcl}
\mathsf{st}_{\mathtt{istr}} & (\mathtt{ReadI}) & \textsc{Empty} & = & \$ \\
\mathsf{st}_{\mathtt{istr}} & (\mathtt{ReadI}) & \textsc{Full}\ i & = & (\textsc{Full}\ i, i) \\
\mathsf{st}_{\mathtt{istr}} & (\mathtt{WriteI}\ i_1) & \textsc{Empty} & = & (\textsc{Full}\ i_1, 0) \\
\mathsf{st}_{\mathtt{istr}} & (\mathtt{WriteI}\ i_1) & \textsc{Full}\ i & = & \bot \\
 & & & & \\
\mathsf{pf}_{\mathtt{istr}} & () & C & = & (C, C)
\end{array}
$$

If the structure is EMPTY a `ReadI` will wait until a concurrent `WriteI` has written to it. Writing to an already full I-structure causes failure. We have no need for I/O contexts in model $\mathtt{istr}$—all actions can be performed concurrently with one another, and $\rho_{\mathtt{istr}}$ is Haskell's unit type, `()`.

To prove $\mathsf{PRE}_{\mathtt{istr}}$ one first shows that no action can cause another to become stalled. Then one shows that the only occasion in which two actions are unstalled is in the case of two consecutive `WriteI`s (which fail, regardless of ordering), a `WriteI` and a `ReadI` to a full I-structure (in which the write always causes failure, regardless of ordering), or two `ReadI`s to a full I-structure (which are order independent).

Since model $\mathtt{istr}$ is so small, we can use it to give some example CURIO reductions. Two concurrent `ReadI`s attempting to read from an empty buffer is in normal form:

$$\textsc{Empty} \Vdash \mathtt{par}\ ()\ (\mathtt{action}\ \mathtt{ReadI})\ (\mathtt{action}\ \mathtt{ReadI}) \downarrow^{\alpha_{\mathtt{istr}}}$$

$$\begin{array}{rcl}
\rho_{\mathtt{store}} & \triangleq & \mathtt{[(Int,WantW + WantR)]} \\
\mathsf{pf}_{\mathtt{store}}\ ps\ C & = & (C \setminus \{\mathtt{Write}\ i\ v \mid v \in \mathtt{Int}, (i,\mathtt{WantW}) \notin ps \vee (i,\mathtt{WantR}) \in ps\} \setminus \{\mathtt{Read}\ i \mid (i,\mathtt{WantW}) \notin ps\}, \\
 & & C \setminus \{\mathtt{Write}\ i\ v \mid v \in \mathtt{Int}, (i,\mathtt{WantW}) \in ps \vee (i,\mathtt{WantR}) \in ps\} \setminus \{\mathtt{Read}\ i \mid (i,\mathtt{WantW}) \in ps\})
\end{array}$$

Figure 4. Context splitting function for model `store`

A `ReadI` performed concurrently with a `WriteI` cannot proceed until the I-structure is full:

$$\begin{array}{c}
\textsc{Empty} \Vdash \mathtt{par\ ()\ (action\ (WriteI\ 7))\ (action\ ReadI)} \\
\longrightarrow^{\alpha_{\mathtt{istr}}} \\
\textsc{Full}\ 7 \Vdash \mathtt{par\ ()\ (return\ 0)\ (action\ ReadI)} \\
\longrightarrow^{\alpha_{\mathtt{istr}}} \\
\textsc{Full}\ 7 \Vdash \mathtt{par\ ()\ (return\ 0)\ (return\ 7)} \\
\longrightarrow^{\alpha_{\mathtt{istr}}} \\
\textsc{Full}\ 7 \Vdash \mathtt{return\ (0,7)}
\end{array}$$

which is in normal form.

It should be noted that our I/O models make no mention of initial world states or initial, outermost contexts. We do not mention initial world states because a programmer, in general, has no control over the contents, say, of the file system. The natural choice for an outermost context is "every action". This is not made explicit anywhere because, from the point of view of proving general language properties, it is just one special case.

5. Location-based models

5.1 Summary of the problem and solution

This section tackles a specific technical difficulty which requires our attention—how to model situations where an API call returns a pointer or handle to a brand new structure. Examples of such actions from Section 2 are `newChannel` and `openFile`. The problem with these API calls is that any naïve implementation just allocates the "next" handle or structure available in a linear fashion (for example, by appending an item to a list and returning the index). This immediately means that two allocations become order dependent. In practice it usually does not matter, since the handle returned by these APIs is only a token whose literal, numeric value should not be examined. But, without a major overhaul of our semantics, there is no way for us to enforce this or state it formally.

The solution is to modify our semantics, I/O model and precondition very slightly so that actions can distinguish different calling processes. In these new, "location-based" I/O models, a process now has an associated "location" as well as a context. The location identifies the process at runtime and is given as an extra hidden parameter to both st and pf. This, on its own, allows allocation to be order independent.

This is a small, often transparent extension only for convenience, and it does not make I/O models any more powerful. To show this we give the semantics of location-based models in terms of location-free ones, and prove that confluence is still maintained. In the "location-free" I/O models an action could not directly determine a unique identity for a calling process. However, if we encode the location in the context, the process itself could still do so using the `test` command. When converting back to a normal model, a process, using `test`, determines the location in which it is running, and passes this as an extra explicit argument.

5.2 Location-based I/O models

A location must uniquely identify processes. The `par` primitive in effect creates a binary tree of processes, so we define locations, $\mathbb{L}$, to be lists of L/R values. $l_1 \preccurlyeq l_2$ means l_1 is a prefix of l_2.

$$\mathbb{L} \triangleq [\mathrm{L} + \mathrm{R}] \qquad l_1 \preccurlyeq l_2 \Leftrightarrow \exists_{l_3 \in \mathbb{L}}.\, l_2 = l_1 +\!\!+ l_3$$

Location-based models are defined below. The old functions st and pf now have an extra $\mathbb{L}$ parameter:

$$\begin{array}{l}
\mathfrak{s} : \mathsf{IOModel}'\ \nu\ \alpha\ \rho\ \omega \triangleq (\\
\quad \mathsf{st}' :: (\mathbb{L},\alpha) \to \omega \to (\omega,\nu) + \mathbb{S} + \bot, \\
\quad \mathsf{pf}' :: (\mathbb{L},\rho) \to \mathcal{P}\alpha \to (\mathcal{P}\alpha,\mathcal{P}\alpha))
\end{array}$$

Since location-based models are a small modification to location-free ones, despite appearances, we shall indicate them simply by adding a prime ($'$), so $\mathsf{IOModel}'$ and st', for example, refer to the location-based counterparts of $\mathsf{IOModel}$ and st, respectively.

The types of the five CURIO primitives are unchanged, and there are only small modifications to be made to the small-step semantics. The three reduction relations are now annotated with locations as well ($\longrightarrow^C$ now becomes $\longrightarrow^{l;C}$). This location is supplied to st' implicitly for all `action` rules. For example:

$$\frac{a \in C \qquad \mathsf{st}'\ (l,a)\ w = (w_1,v)}{w \Vdash \mathtt{action}\ a \longrightarrow^{l;C} w_1 \Vdash \mathtt{return}\ v}$$

The `par` rules are all modified so that there is also a left- and right-hand location l_l and l_r:

$$\mathtt{pf}'\ (l,p)\ C = (C_l,C_r) \qquad l_l = l +\!\!+ [\mathrm{L}] \qquad l_r = l +\!\!+ [\mathrm{R}]$$

and in the individual rules, C, C_l and C_r are changed to $l;C$, $l_l;C_l$ and $l_r;C_r$ respectively. For example:

$$\frac{w \Vdash m_l \downarrow^{l_l;C_l} \quad w \Vdash m_r \downarrow^{l_r;C_r}}{w \Vdash \mathtt{par}\ p\ m_l\ m_r \downarrow^{l;C}} \quad (m_l, m_r \text{ not both values})$$

The new precondition $\mathsf{PRE}'_{\mathfrak{s}}$ is textually almost identical. The only difference is the types of a_l, a_r and p:

$$\begin{array}{rcl}
C_l \Diamond'_{\mathfrak{s}} C_r & \triangleq & \forall_{a_l \in (\mathbb{L},C_l)}.\forall_{a_r \in (\mathbb{L},C_r)} \ldots \\
\mathsf{PRE}'_{\mathfrak{s}} & \triangleq & \forall_{p \in (\mathbb{L},\rho)} \ldots
\end{array}$$

5.3 Converting between models

The function toM' converts a location-free model to a location-based one. In the new model all extra location information is ignored. It is not difficult to show that $\mathsf{PRE}_{\mathfrak{s}}$ implies $\mathsf{PRE}'_{\mathsf{toM}'\ \mathfrak{s}}$.

$$\begin{array}{rcl}
\mathsf{toM}' & :: & \mathsf{IOModel}\ \nu\ \alpha\ \rho\ \omega \to \mathsf{IOModel}'\ \nu\ \alpha\ \rho\ \omega \\
\mathsf{toM}'(\mathsf{st},\mathsf{pf}) & \triangleq & (\lambda(l,a).\ \mathsf{st}\ a, \lambda(l,p).\ \mathsf{pf}\ p)
\end{array}$$

From now we shall work entirely with location-based models. Since the conversion to location-based models is so simple, we will just implicitly assume the above conversion.

To show that $\mathsf{PRE}'_{\mathfrak{s}}$ guarantees confluence for the new, modified language, we define a function toM which gives the semantics of location-based models by giving the equivalent location-free ones:

$$\mathsf{toM} :: \mathsf{IOModel}'\ \nu\ \alpha\ \rho\ \omega \to \mathsf{IOModel}\ \nu\ (\mathbb{L},\mathtt{Act}\ \alpha + \mathtt{Probe})\ \rho\ \omega$$

In the resultant model the location argument becomes explicit in all actions. The rough idea is that if a process' context is C, its location is l, and $a \in C$, then, when we convert everything back to location-free I/O models, action $(l_1, \mathtt{Act}\ a)$ is allowed for all locations l_1 such that $l \preccurlyeq l_1$. The extra action $(l_1, \mathtt{Probe})$ always

$$
\begin{array}{l}
(\mathsf{st}'_1,\mathsf{pf}'_1) * (\mathsf{st}'_2,\mathsf{pf}'_2) \triangleq (\mathsf{st}'_3,\mathsf{pf}'_3) \\
\quad \text{where} \\
\quad\quad \mathsf{st}'_3\ (l,\texttt{Left}\ a_1)\ (w_1,w_2)\ = \\
\quad\quad\quad \begin{cases} ((w'_1,w_2),\texttt{Left}\ v) & ,\mathsf{st}'_1\ (l,a_1)\ w_1 = (w'_1,v) \\ \$ & ,\mathsf{st}'_1\ (l,a_1)\ w_1 = \$ \\ \bot & ,\mathsf{st}'_1\ (l,a_1)\ w_1 = \bot \end{cases} \\
\quad\quad \mathsf{st}'_3\ (l,\texttt{Right}\ a_2)\ (w_1,w_2)\ = \\
\quad\quad\quad \begin{cases} ((w_1,w'_2),\texttt{Right}\ v) & ,\mathsf{st}'_2\ (l,a_2)\ w_2 = (w'_2,v) \\ \$ & ,\mathsf{st}'_2\ (l,a_2)\ w_2 = \$ \\ \bot & ,\mathsf{st}'_2\ (l,a_2)\ w_2 = \bot \end{cases} \\
\quad\quad \mathsf{pf}'_3\ (l,(p_1,p_2))\ (C_1,C_2)\ =\ ((C_{l1},C_{l2}),(C_{r1},C_{r2})) \\
\quad\quad\quad \text{where} \\
\quad\quad\quad\quad (C_{l1},C_{r1}) = \mathsf{pf}'_1\ (l,p_1)\ C_1 \\
\quad\quad\quad\quad (C_{l2},C_{r2}) = \mathsf{pf}'_2\ (l,p_2)\ C_2
\end{array}
$$

Figure 6. Definition of cartesian product combinator

fails when executed, but is *permitted* if the process' location is l, where $l \preccurlyeq l_1$. This allows a process in the old model, using `test`, to calculate its current location correctly. Further details of the proof that $\mathsf{PRE}'_{\mathfrak{s}}$ implies $\mathsf{PRE}_{\mathsf{toM}\ \mathfrak{s}}$ can be found in Appendix A.2.

6. I/O model combinators

This section defines general-purpose "glue" for combining smaller deterministic I/O models to form larger deterministic ones.

6.1 Introduction

Let us recall the model $\mathfrak{store}$, in which the world state is a mapping from integers to integers. We could also easily imagine a store of I-structures, or a store of communication buffers. Yet the definition and proof of $\mathsf{PRE}'_{\mathfrak{s}}$ for each would contain the same boilerplate—actions on different parts of the store are order independent, regardless of the individual store elements. The idea behind combinators is that one first proves small API models confluent, such as $\mathfrak{istr}$ and $\mathfrak{bffr}$, and one then develops mathematical functions which, given some confluent I/O model $\mathfrak{s}$, implements the API "store of $\mathfrak{s}$", preserving both confluence and the opportunities for determinism which each individual $\mathfrak{s}$ allows.

We present three combinators in this section, and their types can be found in Figure 5. The idea of combining API models is perhaps slightly surprising. We should emphasise that combinators were initially developed as a practical solution to the problem of machine verifying $\mathsf{PRE}'_{\mathfrak{s}}$ for large I/O models. An I/O model is just a data structure in our meta-encoding, and combinators are just functions which examine this structure, returning a new, modified one. The advantages to using combinators are (i) they are more modular, rendering unnecessary much boilerplate, (ii) they make clear the *logical* make-up of the API and (iii) since the semantics of I/O is somewhat subjective and open to debate, they let us focus on general properties of APIs, as opposed to specifics.

The important (machine verified) technical results are that, for all I/O models $\mathfrak{s}_1$ and $\mathfrak{s}_2$,

- $\mathsf{PRE}'_{\mathfrak{s}_1}$ and $\mathsf{PRE}'_{\mathfrak{s}_2}$ together imply $\mathsf{PRE}'_{\mathfrak{s}_1 * \mathfrak{s}_2}$.
- $\mathsf{PRE}'_{\mathfrak{s}_1}$ implies $\mathsf{PRE}'_{\mathsf{smap}\ \mathfrak{s}_1}$.
- for all w_0, $\mathsf{PRE}'_{\mathfrak{s}_1}$ implies $\mathsf{PRE}'_{\mathsf{dmap}\ w_0\ \mathfrak{s}_1}$.

6.2 Cartesian product combinator

This combinator is defined in full in Figure 6. $\mathfrak{s}_1 * \mathfrak{s}_2$ represents the joining of the APIs of $\mathfrak{s}_1$ and $\mathfrak{s}_2$. The world state becomes the cartesian product of the world states of $\mathfrak{s}_1$ and $\mathfrak{s}_2$, and an action can be either an action from $\mathfrak{s}_1$ or one from $\mathfrak{s}_2$. When performing concurrency, one must specify the permissions given to each subprocess for both sides of the world state.

$$
\begin{array}{l}
\mathsf{dmap}\ w_0\ (\mathsf{st}',\mathsf{pf}') \triangleq (\mathsf{st}'_{dmap},\mathsf{pf}'_{dmap}) \\
\quad \text{where} \\
\quad\quad \mathsf{st}'_{dmap}\ (l,\texttt{DAct}\ h\ a)\ p\ = \\
\quad\quad\quad \begin{cases} (p[h \mapsto w_1],\texttt{Left}\ v) & ,\mathsf{st}'\ (l,a)\ p(h) = (w_1,v) \\ \$ & ,\mathsf{st}'\ (l,a)\ p(h) = \$ \\ \bot & ,\mathsf{st}'\ (l,a)\ p(h) = \bot \end{cases} \\
\quad\quad \mathsf{st}'_{dmap}\ (l,\texttt{Next})\ p\ =\ (p,\texttt{Right}\ (l,\mathsf{len}\ p\ l))\} \\
\quad\quad \mathsf{st}'_{dmap}\ (l,\texttt{Alloc}\ (l_1,i_1))\ p\ = \\
\quad\quad\quad \begin{cases} \bot & ,l \neq l_1 \text{ or } i_1 \neq (\mathsf{len}\ p\ l) \\ (p[(l_1,i_1) \mapsto w_0],\texttt{Right}\ ([],0)), & \text{otherwise} \end{cases}
\end{array}
$$

Figure 7. Partial definition of dmap

For example, model $\mathfrak{bffr} * \mathfrak{istr}$ is an API which combines access to a buffer and an I-structure.

```
example :: Prog_bffr*istr ()
example = do (Left v) <- action (Left Rcve)
             (r1,r2) <- par (WantS,())
               (action (Right (WriteI 1)))
          ...
```

6.3 Static map combinator

The smap combinator turns an I/O model $\mathfrak{s}$ into a "store of $\mathfrak{s}$." The world state is a map from any type δ to the world state of $\mathfrak{s}$. This is similar to cartesian product, except there are a (potentially infinite) number of $\mathfrak{s}$ models, each indexed by a different δ. An action (d,a) performs action a from $\mathfrak{s}$ on the world state associated with name d. When performing concurrency, one must supply a list of how the permissions for each $d :: \delta$ are to be distributed between the left- and right-hand sides. If some d isn't mentioned in the list, then all the current permissions will be given to the right side, preventing the left side from doing anything at all with d.

For example, imagine an I/O model $\mathfrak{var}$, modelling a single integer variable.

```
var :: IOModel Int ((Write Int)+Read) (WantR+WantS) Int
```

The aforementioned $\mathfrak{store}$ model could have been defined instead as `smap var`. This would be completely identical except that the actions `(i,Write v)` and `(i,Read)` would replace `Write i v` and `Read i`. This looks less natural, but it is just as powerful.

6.4 Dynamic map combinator

dmap w_0 $\mathfrak{s}$ is an I/O model which allows processes to dynamically allocate individual $\mathfrak{s}$ models of initial world state w_0. Each action either performs an action a on a particular ω identified by handle h, `DAct` h a, or it is one of two separate steps required to create a new ω. A partial definition of dmap can be found in Figure 7. This is the only combinator to specifically require the location-based constructs of the previous section.

The basic idea is that our world state is partitioned into individual locations. Since each process is identified uniquely by a location, this allows each process to allocate "locally" without the danger of it interfering with another process doing the same at that very moment.

$$\mathsf{Pool}\ \omega \triangleq \mathbb{L} \rightarrow [\omega] \qquad \texttt{HndP} \triangleq (\mathbb{L},\texttt{Int})$$

The resultant world state of dmap is of the form Pool ω. The structure Pool ω can be understood as a map from locations to lists of ω (it is implemented, actually, using a strict binary tree of

$$
\begin{array}{rcl}
(*) & :: & \mathsf{IOModel}'\ \nu_1\ \alpha_1\ \rho_1\ \omega_1 \to \mathsf{IOModel}'\ \nu_2\ \alpha_2\ \rho_2\ \omega_2 \to \mathsf{IOModel}'\ (\texttt{Either}\ \nu_1\ \nu_2)\ (\texttt{Either}\ \alpha_1\ \alpha_2)\ (\rho_1,\rho_2)\ (\omega_1,\omega_2) \\
\mathsf{smap} & :: & \mathsf{IOModel}'\ \nu\ \alpha\ \rho\ \omega \to \mathsf{IOModel}'\ \nu\ (\delta,\alpha)\ [(\delta,\rho)]\ (\delta \to \omega) \\
\mathsf{dmap} & :: & \omega \to \mathsf{IOModel}'\ \nu\ \alpha\ \rho\ \omega \to \mathsf{IOModel}'\ (\texttt{Either}\ \nu\ \texttt{HndP})\ (\texttt{DynAction HndP}\ \alpha)\ [(\texttt{HndP},\rho)]\ (\mathsf{Pool}\ \omega)
\end{array}
$$

$$
\texttt{Either}\ \beta\ \gamma \triangleq \texttt{Left}\ \beta + \texttt{Right}\ \gamma \qquad \texttt{DynAction}\ \iota\ \alpha \triangleq \texttt{DAct}\ \iota\ \alpha + \texttt{Next} + \texttt{Alloc}\ \iota
$$

Figure 5. Type signatures for combinators

lists of ω). Since at runtime each process has a unique location, this gives each process its own unique list to which it can append new elements. The data allocated is still completely global, and may be accessed long after the creating process has gone—this approach merely ensures that the *act* of allocating data will not affect other processes. In Figure 7 we use a simple shorthand for examining and modifying pools: $p(h)$ looks up pool p at handle h, $p[h \mapsto w]$ is pool p modified so handle h maps to w, and $\mathsf{len}\ p\ l$ returns the next free index at location l in pool p.

Each element of type ω in this structure is identified uniquely by a handle of type `HndP`. This handle is a tuple containing a location, and an `Int` which identifies an individual ω for that particular location. To create a new ω a process first determines, using `Next`, the next free handle h at its location, and then allocates it with `Alloc` h. The technical reason for needing two steps is that a process must have complete access to a handle that it allocates, and I/O contexts cannot forbid actions based on their return values, only their arguments.

```
newBffr :: Prog_{dmap [] bffr} HndP
newBffr = do (Right h) <- action Next
             action (Alloc h)
             return h
```

Contexts are split in a style similar to that of smap except

- A sub-process should have access to *all* possible future handles that it or its child processes may end up using. Therefore, by default, when a process in location l splits into two processes then all permissions to access the ω identified by handle (l_h, i_h) will go the left-hand side if and only if $l +\!\!+ [\mathrm{L}] \preccurlyeq l_h$.
- The action `Alloc` h must have complete, single-threaded access to the ω identified by handle h. Therefore, when doing concurrency, if we distribute permissions to this handle, immediately `Alloc` h will be forbidden by both children. This is quite reasonable: why would you want to distribute permissions to the structure identified by an as yet unused handle?
- There is no reason to forbid action `Next`.

We are not interested in deallocation at the moment. Ideally a clever runtime system would look after this behind the scenes.

7. A real world I/O model

Finally, we can define model $\mathfrak{io}$, and it is

$$\mathfrak{term} * (\mathsf{smap}\ \mathfrak{file} * \mathsf{dmap}\ [] \ \mathfrak{chan})$$

We now briefly define $\mathfrak{term}$, $\mathfrak{file}$ and $\mathfrak{chan}$. Figure 8 shows how some of the front-end API calls from Section 2 are implemented.

Our $\mathfrak{io}$ model ignores quite a few aspects of Haskell's I/O interface. These include: those actions which are semantically transparent, and included for efficiency reasons (i.e. those to do with buffering); verbose error messages; actions which do not give us any further means of exploiting concurrency, for example, those which relate to directory structure; lazy file I/O functions such as

```
data Handle = StdInHnd | StdOutHnd | ChnWrHnd HndP
            | ChnRdHnd HndP | FileHnd String FHnd

(stdin,stdout) = (StdInHnd,StdOutHnd)

openFile :: String → IOMode → Prog_io Handle
openFile n ReadMode = do
  Right (Left (RHndP rh)) <-
    action (Right (Left (n,FNextRdHnd)))
  action (Right (Left (n,HRdOpen rh)))
  return (FileHnd n (RHnd rh))
openFile n ReadWriteMode = do
  action (Right (Left (n,FWrOpen)))
  return (FileHnd n RWHnd)

hPutChar :: Handle → Char → Prog_io ()
hPutChar hnd c = do
  case hnd of
    StdOutHnd       -> action (Left (PutCh c))
    FileHnd n RWHnd ->
      action (Right (Left (n, FWrite c)))
    ChnWrHnd h      ->
      action (Right (Right (DAct h (Send c))))
  return ()

newChannel :: Prog_io (Handle,Handle)
newChannel = do
  Right (Right (Right h)) <-
    action (Right (Right Next))
  action (Right (Right (Alloc h)))
  return (ChnRdHnd h, ChnWrHnd h)

hAllowed :: Handle → Prog_io Bool
hAllowed hnd = case hnd of
  StdInHnd    -> test (Left GetCh)
  StdOutHnd   -> test (Left (PutCh 'X'))
  FileHnd n RWHnd ->
    test (Right (Left (n,FWrite 'X')))
  FileHnd n h -> test (Right (Left (n,HGetC h)))
  ChnRdHnd h ->
    test (Right (Right (DAct h Rcve)))
  ChnWrHnd h ->
    test (Right (Right (DAct h (Send 'X'))))

wholeFile :: String -> Handle
wholeFile n = FileHnd n RWHnd

encodeP :: [Handle] → [Handle] →
    (ρ_term,([(String,ρ_file)],[(HndP,ρ_chan)]))
encodeP hsl hsr = (rhoT th,(rhoFS fsh,rhoChs ch))
  where (th,fsh,ch) = splitHandles hsl hsr
        splitHandles .. -- separate 3 handle types
        rhoFS .. -- isolate 'rho' for each file
        ...
```

Figure 8. Semantics of some high-level API calls

$$
\begin{array}{rcl}
\mathit{file} & :: & \mathsf{IOModel}'\ \nu_{\mathit{file}}\ \alpha_{\mathit{file}}\ \rho_{\mathit{file}}\ \omega_{\mathit{file}} \\
\omega_{\mathit{file}} & \triangleq & \textsc{NoFile} + \textsc{File}\ \texttt{[Char]}\ (\texttt{Either}\ (\mathsf{Pool}\ \texttt{FPtr})\ \texttt{Int}) \\
\texttt{FPtr} & \triangleq & \texttt{Active Int} + \texttt{Stale} \\
\nu_{\mathit{file}} & \triangleq & \texttt{RChar Char} + \texttt{RHndP HndP} + \texttt{RNull} + \cdots \\
\texttt{FHnd} & \triangleq & \texttt{RWHnd} + \texttt{RHnd HndP} \\
\alpha_{\mathit{file}} & \triangleq & \texttt{FDelete} + \texttt{FWrite Char} + \texttt{FWrOpen}\ \cdots \\
 & | & \texttt{HGetC FHnd} + \texttt{HClose FHnd} + \texttt{HIsEOF FHnd}\ \cdots \\
 & | & \texttt{FNextRdHnd} + \texttt{HRdOpen HndP} + \texttt{FDoesExist}\ \cdots \\
\rho_{\mathit{file}} & \triangleq & (\texttt{[FHnd]}, \texttt{[FHnd]})
\end{array}
$$

Figure 9. Fragments of model file

hGetContents. Extensions of the model to cater for these features is an obvious candidate for future work.

7.1 A file I/O model

Figure 9 contains the outline of a model of a single file. A file either doesn't exist, NoFile, or it exists and contains both the file content (a sequence of characters) and either a single write pointer or a pool of read pointers, some of which may be closed (Stale).

The primary difficulty when modelling a file is the dynamic allocation of read handles. For this reason, many aspects of file borrow directly from the dmap combinator definition. The pool is necessary because we want the opening of a file for reading to be order independent. When splitting a context one must supply the specific handles which each side requires access to (by giving two lists of FHnd, one for each process). One may use this to request complete access to the file or access to individual read handles, and like with dmap, we must ensure that a process by default has access to all the potential read handles it may need, if it has shared access to a file. The procedure for allocating a new read handle is likewise very similar to the two-step allocation process of dmap.

The concurrent allocation and deallocation of read and write pointers required a few subtle design decisions. All deallocated pointers, Stale, must be left within the pool because reusing old read handles could cause nondeterminism. For similar reasons, there is no explicit representation of a closed file: a file is closed if it only contains inactive read pointers. We found it useful to separate file actions into three categories: (i) actions, such as writing, which affect the entire file and must be explicitly sequenced; (ii) actions, such as reading a character, which only affect a single read pointer and may examine file contents but not change them; (iii) anomalous actions which must be treated separately, such as the two actions required to allocate a new read handle.

7.2 Models term and chan

Our model of terminal I/O, term, is a basic one (see Figure 10). Each character outputted to stdout by the program using PutCh gives rise instantaneously to one or more characters ready to be read from stdin. These are buffered, and will be consumed by subsequent calls to GetCh.

Model term has the exact same opportunities for deterministic concurrency as chan/bfftr. In fact, term is a complete generalisation of chan. If the world state of term is $([], \text{tchan})$,

$$\text{tchan} \triangleq \mathsf{T}\ (\lambda c.\ ([c], \text{tchan}))$$

this models a communication channel by giving the "semantics" of a user who re-inputs every character outputted to him or her.

$$
\begin{array}{rcl}
\mathsf{TermIO} & \triangleq & \mathsf{T}\ (\texttt{Char} \rightarrow (\texttt{[Char]}, \mathsf{TermIO})) \\
\omega_{\mathit{term}} & \triangleq & (\texttt{[Char]}, \mathsf{TermIO}) \\
\alpha_{\mathit{term}} & \triangleq & \texttt{GetCh} + (\texttt{PutCh Char}) \\
\nu_{\mathit{term}} & \triangleq & \texttt{Char}
\end{array}
$$

$\mathsf{st}_{\mathit{term}}$	GetCh	$([], t)$	$=$	\$
$\mathsf{st}_{\mathit{term}}$	GetCh	$((c{:}cs), t)$	$=$	$((cs, t'), c)$
$\mathsf{st}_{\mathit{term}}$	(PutCh c)	$(cs, \mathsf{T}\ f)$	$=$	$((cs +\!\!+ cs', t'), \texttt{'X'})$
	where	(cs', t')	$=$	$f\ c$

Figure 10. Partial definition of term

We omit the definition of chan. It is mostly identical to bfftr, with the exception that characters are sent, not integers[2].

8. Related Work

Curio's main contributions are in the field of deterministic concurrency and the problems with giving a semantics to/reasoning about I/O (or global state), especially in the presence of concurrency.

8.1 Deterministic concurrency

The Clean language uses uniqueness types [3, 1] to allow actions to be sequenced with respect to individual files. But, unlike Curio, there is no way for two separate program fragments performing I/O on separate pieces of world state to communicate with one another—all communication must be performed at a synchronisation point at which the two fragments of code become one. Furthermore, Clean does not give a semantics to I/O as such. The I/O interface is constructed in an ad hoc style based on Clean's graph rewriting semantics [26].

The Brisk Haskell Compiler uses rank-2 polymorphism to guarantee deterministic access to multiple global sub-states [17], including individual files. In her Ph.D. thesis [28], Eleni Spiliopoulou describes how concurrent threads may communicate deterministically. However, the language description is largely implementation-driven, with no semantics. Shared file reads are performed using lazy file I/O with an informal argument justifying why this is deterministic. Although lazy file I/O is not mentioned in Curio, we can still describe shared file reads in a rigorous fashion.

Terauchi and Aiken [29] give a means of structuring side effects under lazy evaluation via "witnesses." An ordering is enforced on side effects by making one "see" a witness of the other. The downside is that confluence is only statically decidable for some reduction strategies (call-by-need but not call-by-name, for example), and it has only been applied to simple read and write cells, not full I/O. Programs in Curio are always confluent regardless of reduction strategy, and can cater for a variety of I/O mechanisms.

Type and effect systems [21] are a standard approach to merging imperative and functional features, where pure and side effecting terms are distinguished by the type system. However, these rely on an *implicit* program evaluation strategy and thus are not directly applicable. The language Vault [8] makes use of region-based [31, 6] type and effect systems. Vault is a modification of C and has been used to provide a more secure interface to Windows 2000 device drivers. Although its type system is sensitive to specific aspects of the I/O API, it does not go quite so far as to give a semantics to I/O. Boyland [4] gives a statically decidable means of splitting permissions on mutable state, but this work is focussed on imperative

[2] One possible extension would be to allow the sender process to close the buffer. At a high level, this would allow a writer to perform hClose on the channel, letting the reader determine when communication has ended.

languages with implicit reduction orders, and the permissions do not seem to be general enough to handle communication channels.

8.2 Semantics for I/O

There are a number of elegant techniques for expressing state manipulation in functional languages which also happen to be applicable to I/O. This is true of Clean's uniqueness types, and Brisk uses a modification of lazy functional state threads [19] to structure global state. Composable Memory Transactions [16] use a form of memory transaction to permit concurrent data structures, and may be applicable to certain aspects of everyday I/O. These ad hoc techniques are useful and elegant, but we would argue that CURIO offers a more direct description of I/O and APIs.

Haskell's I/O semantics is largely derived from the work in Gordon's Ph.D. thesis [12]. In this document he develops a full operational semantics for a pure functional language using bisimulation, and extends this notion of bisimulation so that transitions, or observations, may include I/O actions. All actions are opaque, observable events and two different sequences of actions are always distinguishable. A similar approach was used to give the semantics to nondeterministic Concurrent Haskell [24] and, later, to describe Haskell's I/O, exception and foreign interface mechanisms [23]. This technique gives an elegant model of user interaction and non-termination. The main flaw to this bisimulation approach is its usefulness since it "explicitly represents the instructions issued by a program, rather than their observable effect" [24]. This style of semantics was used to prove the monad laws in [13]. However, even in Gordon's Ph.D. thesis, in order to prove useful properties about the *effects* of I/O actions, such as the order independence we ourselves want, a state transformer semantics had to be used.

There have been many attempts at reasoning about purely sequential I/O. The House functional operating system [15] allows small propositions concerning sequential monadic code to be proved correct using Programatica. Butterfield and Strong [5] performed a case study comparing the ease of formal reasoning about I/O in C, Haskell and Clean using a semantic model of the file system. This was then followed with a larger proof of a simplified version of the UNIX `make` utility [11]. Thompson [30] gives a trace-based semantics to lazy stream I/O in Miranda. Hall and Hammond's draft dynamic semantics for Haskell 1.3 [14] describes the effect of I/O actions with respect to a single system state, including a semantics for file system actions. Yet CURIO is more powerful than these in that it handles concurrency.

Sewell [27] addresses I/O in the concurrent language Pict. He develops a simplified notion of a sequential C program and the UNIX X-Windows request buffer, and proves an abstract machine correct with respect to the semantics. This is based on the experience of implementing a real language but it is mostly the same as the bisimulation approach, and does not appear to easily generalise.

9. Future Work and Conclusions

We defined the CURIO core language, gave a proof of a general confluence condition, and showed how it is powerful enough to describe the semantics of Haskell I/O and a basic language extension for deterministic concurrency and inter-process communication. The most noteworthy feature is perhaps our use of a state transformer semantics to model concurrency and communication.

Future work will include an investigation into whether the runtime checks required by CURIO could instead be performed statically, by proving a type system sound with respect to CURIO's semantics. Ideally this would make `test` unnecessary. I/O contexts are quite type-like, since a program's enclosing context does not change over time. Another limitation to our system is that communication channels currently only permit the sending of characters, and fully polymorphic channels would be far more desirable. Furthermore we also plan to use CURIO as the basis for actual proofs about the behaviour of I/O, and develop general notions of program equivalence for CURIO programs. Some progress in this area has already been reported in [9]. Extending the io model to cover more of the Haskell API, and the APIs of other languages would also be worthwhile.

References

[1] Peter Achten and Rinus Plasmeijer. The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, January 1995.

[2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Language Systems*, 11(4):598–632, 1989.

[3] Erik Barendsen and Sjaak Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[4] John Boyland. Checking Interference with Fractional Permissions. In Radhia Cousot, editor, *SAS*, volume LNCS2694, pages 55–72. Springer, 2003.

[5] Andrew Butterfield and Glenn Strong. Proving Correctness of Programs with I/O — a Paradigm Comparison. In Thomas Arts and Markus Mohnen, editors, *Proceedings of IFL2001*, volume LNCS2312, pages 72–87, 2001.

[6] Karl Crary, David Walker, and Greg Morrisett. Typed Memory Management in a Calculus of Capabilities. In *POPL '99: Proceedings of the 26th Symposium on Principles of Programming Languages*, pages 262–275, New York, NY, USA, 1999. ACM Press.

[7] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem Proving for Functional Programmers. In Thomas Arts and Markus Mohnen, editors, *Proceedings of IFL2001*, volume LNCS2312, pages 55–71. Springer-Verlag, 2001.

[8] Robert DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *PLDI '01: Proceedings of the 2001 Conference on Programming Language Design and Implementation*, pages 59–69, New York, NY, USA, 2001. ACM Press.

[9] Malcolm Dowse. *A Semantic Framework for Deterministic Functional Input/Output*. PhD thesis, Department of Computer Science, University of Dublin, 2006.

[10] Malcolm Dowse, Andrew Butterfield, and Marko van Eekelen. Reasoning about Deterministic Concurrent Functional I/O. In Clemens Grelck and Frank Huch, editors, *Proceedings of IFL 2004*, volume LNCS3474, pages 177–194. Springer-Verlag, 2005.

[11] Malcolm Dowse, Glenn Strong, and Andrew Butterfield. Proving 'make' Correct — I/O Proofs in Haskell and Clean. In Ricardo Peña and Thomas Arts, editors, *Proceedings of IFL 2002*, volume LNCS2670, pages 68–83, 2002.

[12] Andrew Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[13] Andrew 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.

[14] Cordelia Hall and Kevin Hammond. A Dynamic Semantics for Haskell (draft), May 20 1993.

[15] Thomas Hallgren, Mark Jones, Rebekah Leslie, and Andrew Tolmach. A Principled Approach to Operating System Construction in Haskell. In *ICFP '05: Proceedings of the 10th International Conference on Functional Programming*, pages 116–128, New York, NY, USA, 2005. ACM Press.

[16] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[17] Ian Holyer and Eleni Spiliopoulou. Concurrent Monadic Interfacing. In Kevin Hammond, Antony Davie, and Chris Clack, editors, *Proceedings of IFL '98*, volume LNCS1595, pages 73–89. Springer-Verlag, 1999.

[18] Mark Jones and Paul Hudak. Implicit and Explicit Parallel Programming in Haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, August 1993.

[19] John Launchbury and Simon Peyton Jones. Lazy Functional State Threads. In *PLDI '94: Proceedings of the 1994 Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, Florida, June 20–24, 1994.

[20] Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[21] Flemming Nielson and Hanne Riis Nielson. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances*, pages 114–136, London, UK, 1999. Springer-Verlag.

[22] Simon Peyton Jones. Wearing the Hair Shirt: A retrospective on Haskell. Invited talk at POPL 2003.

[23] Simon Peyton Jones. Tackling the Awkward Squad — monadic input/output, concurrency, exceptions, and foreign language calls in Haskell. In CAR Hoare, M Broy, and R Stein-brueggen, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, pages 47–96. IOS Press, 2001.

[24] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd Symposium on Principles of Programming Languages*, pages 295–308, New York, NY, USA, 1996. ACM Press.

[25] Simon Peyton Jones and Philip Wadler. Imperative Functional Programming. In *POPL '93: Proceedings of the 20th Symposium on Principles of Programming Languages*, pages 71–84, New York, NY, USA, 1993. ACM Press.

[26] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[27] Peter Sewell. On Implementations and Semantics of a Concurrent Programming Language. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 391–405, London, UK, 1997. Springer-Verlag.

[28] Eleni Spiliopoulou. *Concurrent and Distributed Functional Systems*. PhD thesis, University of Bristol, Department of Computing Science, August 1999.

[29] Tachio Terauchi and Alex Aiken. Witnessing Side-Effects. In *ICFP '05: Proceedings of the 10th International Conference on Functional Programming*, pages 105–115, New York, NY, USA, 2005. ACM Press.

[30] Simon Thompson. Interactive Functional Programs: a method and a formal semantics. Technical Report 48*, Computing Laboratory, University of Kent,, November 1987.

[31] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[32] David Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, December 1986.

[33] Philip Wadler. The Essence of Functional Programming. In *POPL '92: Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.

A. Additional Material

The proofs were machine verified using the LCF-based Sparkle [7] proof-assistant, and using Core-Clean as a metalanguage. We use Haskell syntax instead as it will be more familiar to the average reader. Full details concerning these proofs, including all results in this paper, may be found in [9].

A.1 Meta-encoding

I/O models are defined as follows.

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

The functions `af` and `wa` form `st`, the type `c` denotes contexts, `ap` gives the actions each context permits, and `pf` is `pf`. CURIO's five primitives are encoded with the following algebraic type:

```
data Prog v a p = Ret v | Action a
  | Bind (Prog v a p) (v -> Prog v a p)
  | Test a (Prog v a p) (Prog v a p)
  | Par p (Prog v a p) (Prog v a p) (v -> v -> v)
```

Small-step reduction is implemented with the function `next`, and `run` continually applies `next`, only terminating when a normal form is reached:

```
next :: IOModel v a p w c -> c -> Guess ->
  (w,Prog v a p) -> Maybe (w,Prog v a p)
run :: IOModel v a p w c -> c -> [Guess] ->
  (w,Prog v a p) -> (w,Prog v a p)
```

The `Guess` parameter is used to implement nondeterminism, and the reduction relations are defined in terms of these functions:

$$
\begin{array}{lcl}
w \Vdash m \downarrow^c & \iff & \exists_{g \in \texttt{Guess}}.\texttt{next}_\texttt{s}\ g\ c\ (w,m) = \texttt{Nothing} \\
w \Vdash m \uparrow^c & \iff & \exists_{g \in \texttt{Guess}}.\texttt{next}_\texttt{s}\ g\ c\ (w,m) = \bot \\
w \Vdash m \Uparrow^c & \iff & \forall_{gs \in [\texttt{Guess}]}.\texttt{run}_\texttt{s}\ gs\ c\ (w,m) = \bot \\
w \Vdash m \longrightarrow^c w_1 \Vdash m_1 & & \\
\qquad & \iff & \exists_{g \in \texttt{Guess}}.\texttt{next}_\texttt{s}\ g\ c\ (w,m) = \texttt{Just}\ (w_1,m_1) \\
w \Vdash m \Big\downarrow^c w_1 \Vdash m_1 & & \\
\qquad & \iff & \exists_{gs \in [\texttt{Guess}]}.\texttt{run}_\texttt{s}\ gs\ c\ (w,m) = (w_1,m_1) \\
w \Vdash m \Downarrow^c w_1 \Vdash m_1 & & \\
\qquad & \iff & \forall_{gs \in [\texttt{Guess}]}.\texttt{run}_\texttt{s}\ gs\ c\ (w,m) = (w_1,m_1)
\end{array}
$$

A.2 Encoding location-based models

The trick to the encoding is being able to convert $(\mathbb{L},\mathcal{P}\alpha)$ to and from the context $\mathcal{P}(\mathbb{L},\texttt{Act}\ \alpha + \texttt{Probe})$. This encoding and decoding is as follows.

$$
\begin{array}{rcl}
\mathsf{encode} & :: & (\mathbb{L},\mathcal{P}\alpha) \to \mathcal{P}(\mathbb{L},\texttt{Act}\ \alpha + \texttt{Probe}) \\
\mathsf{encode}\ (l,C) & \triangleq & \{(l_1,\texttt{Act}\ a) \mid l \preccurlyeq l_1, a \in C\} \\
 & & \quad \cup \{(l_1,\texttt{Probe}) \mid l \preccurlyeq l_1\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{decode} & :: & \mathcal{P}(\mathbb{L},\texttt{Act}\ \alpha + \texttt{Probe}) \to (\mathbb{L},\mathcal{P}\alpha) \\
\mathsf{decode}\ C' & \triangleq & (\bigcap\{l \mid (l,\texttt{Probe}) \in C'\}, \\
 & & \quad \{a \mid (l,\texttt{Act}\ a) \in C'\})
\end{array}
$$

$\bigcap L$, where L is a set of locations, returns the largest list $l \in \mathbb{L}$ such that, for all $l_1 \in L$, $l \preccurlyeq l_1$, or, if $L = \emptyset$, it returns [].

The proof that PRE'_s implies $\mathsf{PRE}_{\mathsf{toM}\ \mathsf{s}}$ relies on the correctness of the above encoding and decoding.

The definition of toM is

$$
\begin{array}{l}
\mathsf{toM}\ (\mathsf{st}',\mathsf{pf}') \triangleq (\mathsf{st}_0,\mathsf{pf}_0) \\
\quad \text{where} \\
\qquad \mathsf{st}_0\ (l,\texttt{Act}\ a)\ w = \mathsf{st}'\ (l,a)\ w \\
\qquad \mathsf{st}_0\ (l,\texttt{Probe})\ w = \bot \\
\qquad \mathsf{pf}_0\ p\ C' = (C' \cap C'_l, C' \cap C'_r) \\
\qquad\quad \text{where} \\
\qquad\qquad (l,C) = \mathsf{decode}\ C' \\
\qquad\qquad (C_l,C_r) = \mathsf{pf}'\ (l,p)\ C \\
\qquad\qquad C'_l = \mathsf{encode}\ (l +\!\!+ [\mathrm{L}],C_l) \\
\qquad\qquad C'_r = \mathsf{encode}\ (l +\!\!+ [\mathrm{R}],C_r)
\end{array}
$$