

A Toolkit for Structuring I/O

Malcolm Dowse and Andrew Butterfield*

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

Trinity College Dublin, Ireland

Abstract

We give a quick presentation of the language CURIO, which gives a semantics to concurrent I/O in pure functional languages by way of modelling the API directly. Central to CURIO is the fact that an I/O model must obey a pre-condition – some broad axiomatic properties – so that program execution can be deterministic.

We then present some mechanisms for modelling the dynamic allocation of data, combinators which can be used to build larger (deterministic) APIs from smaller ones, and show how they may be used to encode a small, useful subset of the functionality of Haskell’s file I/O library.

1 INTRODUCTION

In lazy languages, the order in which expressions are evaluated is hard to predict so one is forced instead to concentrate on the logic of the program. This has often been considered one of their great selling points since optimisations then, ideally at least, become side issues which we tinker with once the program is logically correct. It also allows us to write software at a more abstract level, almost at the level of a specification, making programs more general and reusable.

However, this particular approach seems to vanish in the presence of I/O. In (plain monadic) Haskell [10] all actions must be explicitly sequenced. This is not abstract at all, and is no more a specification of I/O than any C program. What is needed is a way to specify the extent to which the ordering of actions may be loosened; to permit concurrency while still ensuring determinism.

For many people this seems confusing because surely if a program is deterministic then by definition it always does the same thing – so why bother with concurrency at all? The answer is that concurrency improves program efficiency and user response time for I/O bound programs, and also increases program reusability.

As an example, let’s say I have written a file-encoder and want to give it a user interface. The encoder 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, or the two applications will have to be entangled together.

The CURIO language was designed with a view to solving these sorts of problems, and can be seen as a further generalisation of the principles of lazy languages.

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

The central contribution of this paper is a description of how we went about constructing a more large scale, practical I/O model for CURIO, and the numerous technical problems which had to be solved in doing so.

All results in this paper have been machine-verified using the Sparkle proof-assistant [2], and the language is encoded using Core-Clean as a meta-language. We use Haskell syntax instead because it will be more familiar to most readers. We omit all the actual proof details in this document. Instructions on how one can obtain the machine-readable form of the proofs is available from:

<http://www.cs.tcd.ie/Malcolm.Dowse/IOToolkit>

2 THE CURIO LANGUAGE – A QUICK INTRODUCTION

CURIO is a small functional language with concurrency, interprocess communication and monadic constructs which together let the user write expressive programs which perform I/O. This expressivity is due to our ability to enforce determinism.

The Concurrent Haskell language [9] is probably the most fully-fledged semantics for I/O in a lazy functional language. CURIO’s semantics is different however, since it is *inductive* rather than co-inductive. There exists a “world-state” which the program interacts with and modifies when doing I/O, and the observable effect of a program is its resultant world-state. In Concurrent Haskell all actions are observable, whereas with CURIO it is only the cumulative effect of a finite number of actions over a program’s lifetime that is observable.

What makes the semantics of CURIO somewhat more expressive (if less elegant) is that it ascribes to each action a precise effect on world state, and therefore allows us to distinguish actions which do and do not interfere with one another. Insofar as a language semantics should be a vehicle for compiler optimisations and formal proofs, this would appear to make it more powerful – but we cannot tell at this early stage.

2.1 Language Primitives

In reality, CURIO is less a full language specification than a rigorous semantics for a collection of powerful I/O primitives which can be wrapped around a pure functional language using solely its denotational semantics. A program in CURIO is any element of the type $\text{Prog}_{v\alpha\rho} \beta$ and the five I/O primitives are as follows: (the three types v , α and ρ are bound by the particular I/O model, and these are explained later.)

$$\begin{aligned} (>>=) &:: \forall\beta. \forall\gamma. \text{Prog}_{v\alpha\rho} \beta \rightarrow (\beta \rightarrow \text{Prog}_{v\alpha\rho} \gamma) \rightarrow \text{Prog}_{v\alpha\rho} \gamma \\ \text{return} &:: \forall\beta. \beta \rightarrow \text{Prog}_{v\alpha\rho} \beta \\ \text{action} &:: \alpha \rightarrow \text{Prog}_{v\alpha\rho} v \\ \text{test} &:: \forall\beta. \alpha \rightarrow \text{Prog}_{v\alpha\rho} \beta \rightarrow \text{Prog}_{v\alpha\rho} \beta \rightarrow \text{Prog}_{v\alpha\rho} \beta \\ \text{par} &:: \forall\beta. \forall\gamma. \forall\varepsilon. \rho \rightarrow \text{Prog}_{v\alpha\rho} \beta \rightarrow \text{Prog}_{v\alpha\rho} \gamma \rightarrow (\beta \rightarrow \gamma \rightarrow \varepsilon) \rightarrow \text{Prog}_{v\alpha\rho} \varepsilon \end{aligned}$$

Each of the five primitives correspond to a different data constructor in a higher-order non-strict algebraic data type in the meta language. Central to these language primitives is the notion of an I/O context. Each (sub-) program is executed within a context $c \in \zeta$, and that context dictates which actions that (sub-) program can perform, and therefore affects the program's outcome.

A more complete explanation of the semantics may be found elsewhere [3, 4], but very briefly: `>>=` and `return` are the familiar monadic functions in Haskell; `action a` performs primitive action a , which may stall (it fails if a is not permitted by the context); `test a mt mf` performs program m_t if action a is permitted in the current context, and program m_f if it is not. Program `par p ml mr (*)` runs m_l and m_r concurrently, splitting the current context in the way determined by p . If m_l and m_r reduce in their own contexts to `return vl` and `return vr` respectively then `par p ml mr (*)` will reduce to `return vl * vr`.

Concurrency introduces non-determinism, and reduction proceeds in a non-deterministic fashion until the world/program pair is in normal form – meaning either that the program is of the form `return v` for some v , or that the program contains actions which are stalled.

2.2 I/O Models

We are now in a position to explain where the types v , α , ρ and ζ come from. These, together with the type ω and four functions, describe the world-state, the effect of actions, I/O contexts and how contexts are divided among sub-programs in the presence of concurrency.

An I/O model in the metalanguage is a 4-tuple parameterised by five types:¹

$$\begin{aligned} \mathfrak{s} :: \text{IOModel } v \alpha \rho \omega \zeta &\triangleq \langle \text{af} :: \alpha \rightarrow \omega \rightarrow (\omega, v) \\ &\quad , \text{wa} :: \alpha \rightarrow \omega \rightarrow \text{Bool} \\ &\quad , \text{ap} :: \zeta \rightarrow \alpha \rightarrow \text{Bool} \\ &\quad , \text{pf} :: \rho \rightarrow \zeta \rightarrow (\zeta, \zeta) \rangle \end{aligned}$$

2.2.1 The API – `af` and `wa`

The functions `af` and `wa` define the API. `af :: $\alpha \rightarrow \omega \rightarrow (\omega, v)$` defines the state-transformer for each action. Elements of type α identify I/O actions which can be performed and for any action $a :: \alpha$, `af a :: $\omega \rightarrow (\omega, v)$` is the state-transformer for that action. Type v is typically a sum-type capable of storing `Ints`, `Bools`, `Chars` or any other value that an action might need to return. If communication is to be permitted then there must be occasions in which an action is waiting for something to occur and cannot proceed. The function `wa :: $\alpha \rightarrow \omega \rightarrow \text{Bool}$` indicates exactly this. We say an action a is *stalled* (in world state w) if `wa a w = TRUE`.

¹Unless otherwise indicated we assume the existence of some arbitrary, implicit I/O model called \mathfrak{s} which binds the five types v , α , ρ , ω and ζ , and the four functions `af`, `wa`, `ap` and `pf`.

$$\begin{aligned}
& \parallel_{\mathfrak{s}}, \text{ally}_{\mathfrak{s}} : \alpha \rightarrow \alpha \rightarrow \mathbb{B} & \sqsubseteq_{\mathfrak{s}}, \diamond_{\mathfrak{s}} : \zeta \rightarrow \zeta \rightarrow \mathbb{B} \\
c_1 \sqsubseteq_{\mathfrak{s}} c_2 & \triangleq \forall a \in \alpha. \text{ap } a \ c_1 \implies \text{ap } a \ c_2 \\
c_l \diamond_{\mathfrak{s}} c_r & \triangleq \forall a_l \in \alpha. \forall a_r \in \alpha. \text{ap } c_l \ a_l \wedge \text{ap } c_r \ a_r \implies a_l \parallel_{\mathfrak{s}} a_r \wedge \text{ally}_{\mathfrak{s}}(a_l, a_r) \wedge \text{ally}_{\mathfrak{s}}(a_r, a_l)
\end{aligned}$$

FIGURE 1. Relations on actions and contexts

Figure 1 shows two relations on actions and the API. $a_1 \parallel_{\mathfrak{s}} a_2$ means that for any two actions a_1 and a_2 , 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. $\text{ally}_{\mathfrak{s}}(a_1, a_2)$ indicates that if action a_1 is not stalled then performing it cannot cause action a_2 , if also not stalled, to then become stalled. That is, a_1 will not impede a_2 .

2.2.2 I/O Contexts – *ap* and *pf*

I/O contexts are elements of the type ζ and the function $\text{ap} :: \zeta \rightarrow \alpha \rightarrow \text{Bool}$ defines the actions permitted by any context. Note that the API does not mention contexts at all. A context merely determines whether or not an action can be performed, not the effect of the action on world state if it is.

Figure 1 defines two important relations on contexts.

- $c_1 \sqsubseteq_{\mathfrak{s}} c_2$: any action permitted by context c_1 is also permitted by c_2 .
- $c_l \diamond_{\mathfrak{s}} c_r$: for all actions a_l and a_r , permitted by contexts c_l and c_r respectively, it is true that $a_l \parallel_{\mathfrak{s}} a_r$, $\text{ally}_{\mathfrak{s}}(a_l, a_r)$ and $\text{ally}_{\mathfrak{s}}(a_r, a_l)$.

The function $\text{pf} :: \rho \rightarrow \zeta \rightarrow (\zeta, \zeta)$ is used to constrain the behaviour of sub-programs running concurrently. When a process $\text{par } p \ m_l \ m_r \ (*)$ is run in context c it forks into two processes m_l and m_r running in contexts c_l and c_r respectively, where $\text{pf } p \ c = (c_l, c_r)$.

We have proved separately [3], using the Sparkle proof-assistant [2], the powerful (and difficult) result that if pf obeys certain properties then reduction is confluent. This means that regardless of the non-deterministic interleaving of the execution of sub-processes, a program will either always diverge or always converge to the same resultant world-state and program. These properties form a pre-condition on an I/O model, written $\text{PRE}_{\mathfrak{s}}$, and it is stated formally as:

$$\text{PRE}_{\mathfrak{s}} \triangleq \forall p \in \rho. \forall c \in \zeta. \forall c_l \in \zeta. \forall c_r \in \zeta. \text{pf } p \ c = (c_l, c_r) \implies c_l \sqsubseteq_{\mathfrak{s}} c \wedge c_r \sqsubseteq_{\mathfrak{s}} c \wedge c_l \diamond_{\mathfrak{s}} c_r$$

It can be summarised as follows. Firstly, no child process should be allowed to perform an action forbidden by the parent context: $c_l \sqsubseteq_{\mathfrak{s}} c$ and $c_r \sqsubseteq_{\mathfrak{s}} c$. Secondly, the order in which actions in the two child sub-programs are performed should be irrelevant: $c_l \diamond_{\mathfrak{s}} c_r$.

$$\begin{aligned}
\text{term} &:: \text{IOModel } v_{\text{Term}} \alpha_{\text{Term}} \rho_{\text{Term}} \omega_{\text{Term}} \zeta_{\text{Term}} \triangleq \langle \text{af}_{\text{Term}}, \text{wa}_{\text{Term}}, \text{ap}_{\text{Term}}, \text{pf}_{\text{Term}} \rangle \\
v_{\text{Term}} &\triangleq \text{Char} & \alpha_{\text{Term}} &\triangleq \text{PUTC Char} \mid \text{GETC} \\
\omega_{\text{Term}} &\triangleq ([\text{Char}], \text{TermIO}) & \rho_{\text{Term}}, \zeta_{\text{Term}} &\triangleq \text{TCXT Bool Bool} \\
&& \text{TermIO} &\triangleq \text{TERMIO} (\text{Char} \rightarrow ([\text{Char}], \text{TermIO})) \\
\text{wa}_{\text{Term}} (\text{PUTC } c) (cs, t) && &\triangleq \text{FALSE} \\
\text{af}_{\text{Term}} (\text{PUTC } c) (cs, \text{TERMIO } f) && &\triangleq ((cs++fst (f c), \text{snd } (f c)), ' ') \\
\text{wa}_{\text{Term}} \text{GETC} (cs, t) && &\triangleq \text{null } cs \\
\text{af}_{\text{Term}} \text{GETC} ((c:cs), t) && &\triangleq ((cs, t), c) \\
\text{ap}_{\text{Term}} (\text{TCXT } b _) (\text{PUTC } c) && &\triangleq b \\
\text{ap}_{\text{Term}} (\text{TCXT } _ b) \text{GETC} && &\triangleq b \\
\text{pf}_{\text{Term}} (\text{TCXT } bp_p \ bg_p) (\text{TCXT } bp \ bg) && &\triangleq \\
&& &(\text{TCXT } (bp \&\& \ bg_p) (bg \&\& \ bg_p), \text{TCXT } (bp \&\& \ \text{not } bp_p) (bg \&\& \ \text{not } bg_p))
\end{aligned}$$

FIGURE 2. A Model for Terminal I/O

We make no mention at all of initial world states because a program can be assumed to have no control over the initial environment in which it runs (i.e. which files exist). Also we do not discuss the initial I/O context in which an entire program runs. This should probably permit the maximum number of actions possible.

2.3 Examples

2.3.1 Terminal I/O

As a small example, consider the complete model of terminal I/O given in Figure 2. There are two actions: `PUTC c` writes character `c`, and `GETC` reads a character. The model is not perfect, since it does not capture any interesting temporal properties of `stdin/stdout`. There is no notion of absolute time – each outputted `Char` gives rise to 0 or more inputted characters instantaneously. It is adequate, nonetheless, and obeys the simple property that if characters are available for input then outputting characters cannot change that. The model also lets us exploit the fact that `stdin` and `stdout` are two separate handles, and we may want a process to wait for input on one whilst another outputs data on the other. (This is probably only useful for filtering programs such as `grep`. Interactive programs would usually require a lock-step synchronisation of input and output).

The proof of PRE_{term} is not difficult. Reading may take place in parallel with writing in this semantics since (1) if a `Char` is ready for input then writing a

```

putStr :: String → Progterm ()
putStr []      = return ()
putStr (c:cs) = action (PutC c) >> putStr cs

stdPermissions :: Progterm String
stdPermissions = do
  in  <- test GetC (return "stdin") (return "no stdin")
  out <- test (PutC 'x') (return "stdout") (return "no stdout")
  return (in ++ ", " ++ out)

getPutC :: Char → Progterm Char
getPutC c = par (Tcxt True False) (action (PutC c))
              (action GetC) (\_ c1 -> c1)

```

FIGURE 3. Three Terminal I/O programs

character will not change that and (2) if a character is not available for input then the read will stall until one is. The function pf_{Term} guarantees that if many processes are running concurrently then at most one can call PUTC and at most one can perform GETC.

We give three example programs in Figure 3. `putStr` writes a string to `stdout` (or fails if that is not allowed). `stdPermissions` returns a string which indicates which actions the current context permits. The program `getPutC c` outputs character `c` whilst concurrently requesting a character from input. The entire program returns the character that was read. Here $\text{Prog}_{\text{term}} \beta$ is used as a convenient short-hand for $\text{Prog}_{\text{Term}} \alpha_{\text{Term}} \rho_{\text{Term}} \beta$ – a notation used throughout this paper. We also make use of Haskell’s monadic `>>` operator and `do`-notation.

2.3.2 Communication Channels

Consider the type `TermIO` once again. One possible instance of it is `loopBack`, defined as follows:

```
loopBack = TermIO (\c -> ([c], loopBack))
```

This really just models a user who re-inputs each character which was outputted. By constructing an I/O model with these exact semantics one then has a one-to-one “communication channel” allowing the writer process to stream data to the reader process. Let us call this model `chan`. It is modelled as follows. The types ρ_{Chan} and ζ_{Chan} are omitted but they are effectively the same as that of `term`.

$$\omega_{\text{Chan}} \triangleq [\text{Char}] \quad \nu_{\text{Chan}} \triangleq \text{Char} \quad \alpha_{\text{Chan}} \triangleq \text{CHPUTC Char} \mid \text{CHGETC}$$

w_{Chan}	$(\text{CHPUTC } c)$	cs	\triangleq	FALSE
a_{fChan}	$(\text{CHPUTC } c)$	cs	\triangleq	$(cs \text{++} c, ' ')$
w_{Chan}	CHGETC	cs	\triangleq	$\text{null } cs$
a_{fChan}	CHGETC	$(c : cs)$	\triangleq	(cs, c)

3 DYNAMICALLY ALLOCATING DATA

One of the difficulties with a global, state-based semantics is how one deals with the dynamic allocation of data. Consider the allocation of handles for the shared reading of a file. A handle is effectively a pointer to an internal system structure. However, if we loosen the order in which handles are allocated then processes may acquire different handles on different program executions – they will be just as useful, but not indistinguishable – and the world state will be subtly different also.

The bottom line is that despite our global state and API we still need a mechanism for allocating data locally within the global state. The technical problem can be solved by supplying the state-transformer with a tiny piece of additional information for certain actions. This information is a value of type `Loc`, a list of L/R constructors, which indicates at run-time the current branch in the tree of concurrent processes that the action’s caller inhabits. This allows an action to allocate a structure in global world-state without the danger of it “stepping on the toes” of another process doing the same at that very moment. The data allocated is itself still 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.

The `Pool` β structure was developed for this reason. It is, abstractly, a finite lookup table from `HndP` to β , partitioned into locations. Each `HndP` contains both a `Loc`, the location, and a `Indx`, the index within the location. Pools are modified and queried with `ovwP` and `lkpP` (which obey the expected properties of lookup table). To allocate a new pool element one must first determine the next free index with respect to some location using `nextP`. Then one combines this location with the index to give a valid handle, and calls `newP` with the initial value. A `Pool` β is implemented internally using a strict² binary tree of lists of β .

$$\begin{aligned}
\text{Loc} &\triangleq [\text{L} \mid \text{R}] & \text{Indx} &\triangleq \text{Int} & \text{HndP} &\triangleq (\text{Loc}, \text{Indx}) \\
\text{ovwP}, \text{newP} &:: \text{HndP} \rightarrow \beta \rightarrow \text{Pool } \beta \rightarrow \text{Pool } \beta \\
\text{lkpP} &:: \text{HndP} \rightarrow \text{Pool } \beta \rightarrow \beta \\
\text{nextP} &:: \text{Loc} \rightarrow \text{Pool } \beta \rightarrow \text{Indx}
\end{aligned}$$

Locations can be encoded into I/O models in an ad-hoc manner by (1) extending the context ζ to include the current location, (2) modifying `pf` so that when a context with location l is split, the left- and right-hand contexts have locations

²A great deal of strictness annotation was required in order to prove the technical results in this paper. We omit all these details, however, since it makes the presentation clearer.

$$\begin{aligned}
\mathfrak{s} :: \text{IOModelL } \nu \alpha \rho \omega \zeta &\triangleq \langle \text{af}' :: (\text{Loc}, \alpha) \rightarrow \omega \rightarrow (\omega, \nu) \\
&\quad , \text{wa}' :: (\text{Loc}, \alpha) \rightarrow \omega \rightarrow \text{Bool} \\
&\quad , \text{ap}' :: \zeta \rightarrow \alpha \rightarrow \text{Bool} \\
&\quad , \text{pf}' :: \text{Loc} \rightarrow \rho \rightarrow \zeta \rightarrow (\zeta, \zeta) \\
&\quad , \text{ig}' :: \alpha \rightarrow \text{Bool} \rangle \\
\Diamond'_s &: \zeta \rightarrow \zeta \rightarrow \mathbb{B} \\
c_l \Diamond'_s c_r &\triangleq \forall a_l \in \alpha. \forall l_l \in \text{Loc}. \forall a_r \in \alpha. \forall l_r \in \text{Loc}. \text{ap}' c_l a_l \wedge \text{ap}' c_r a_r \wedge \\
&\quad (\text{ig}' a_l \vee \text{ig}' a_r \vee l_l \neq l_r) \implies (l_l, a_l) \parallel_s (l_r, a_r) \wedge \\
&\quad \text{ally}_s((l_l, a_l), (l_r, a_r)) \wedge \text{ally}_s((l_r, a_r), (l_l, a_l)) \\
\text{PRE}'_s &\triangleq \forall p \in \rho. \forall c \in \zeta. \forall c_l \in \zeta. \forall c_r \in \zeta. \text{pf } p c = (c_l, c_r) \implies c_l \sqsubseteq_s c \wedge c_r \sqsubseteq_s c \wedge c_l \Diamond'_s c_r
\end{aligned}$$

FIGURE 4. Definition of Location-Based I/O Models

$l++[\text{L}]$ and $l++[\text{R}]$, and (3) making the user supply the current location as an argument to those actions which require it.

This ad-hoc encoding works OK, but because locations are so useful and only ought to be added once, it is worthwhile to devise a new sort of “location-based” I/O model, $\text{IOModelL } \nu \alpha \rho \omega \zeta$, as defined in Figure 4. It is like $\text{IOModel } \nu \alpha \rho \omega \zeta$ except that the behaviour of certain actions is affected by a location, and there is an extra function ig' which indicates whether a given action ignores the location information. The all-important pre-condition PRE_s becomes PRE'_s – it is also similar except for the additional constraint that two location dependent actions cannot interfere with one another if their respective locations differ.

The functions $\text{LMt} \circ \text{M}$ and $\text{Mt} \circ \text{LM}$ defined in Figure 5 are used to convert a location-based I/O models into a normal I/O model and back again, preserving the respective pre-conditions. $l_1 \preceq l_2$ is equivalent to the statement that there exists a l_3 such that $l_1 ++ l_3 = l_2$.

Theorem 3.1.

- PRE_s implies $\text{PRE}'_{\text{Mt} \circ \text{LM } s}$.
- PRE'_s implies $\text{PRE}_{\text{LMt} \circ \text{M } s}$.

The function $\text{Mt} \circ \text{LM}$ is very simple – $\text{Mt} \circ \text{LM } s$ embeds all the actions in s directly into the more sophisticated framework by adding the information that no action is location dependent.

$\text{LMt} \circ \text{M}$ can be viewed as giving the semantics of location-based I/O models by converting them back to a traditional I/O model which contains extra information. In the resultant I/O model one must supply an additional location argument to each action, which may be ignored, depending on the action. The context also contains

$$\begin{aligned}
\text{DAction } \alpha &\triangleq \text{ PROBE} \mid \text{ ACT } \alpha \\
\text{MtO} \text{LM} &:: \forall v. \forall \alpha. \forall \rho. \forall \omega. \forall \zeta. \text{IOModel } v \ \alpha \ \rho \ \omega \ \zeta \rightarrow \text{IOModelL } v \ \alpha \ \rho \ \omega \ \zeta \\
\text{LMtO} \text{M} &:: \forall v. \forall \alpha. \forall \rho. \forall \omega. \forall \zeta. \text{IOModelL } v \ \alpha \ \rho \ \omega \ \zeta \rightarrow \\
&\quad \text{IOModel } v \ (\text{Loc}, \text{DAction } \alpha) \ \rho \ \omega \ (\text{Loc}, \zeta) \\
\text{MtO} \text{LM} \langle \text{af}, \text{wa}, \text{ap}, \text{pf} \rangle &\triangleq \langle \\
&\quad \lambda(l, a). \lambda w. \text{af } a \ w, \\
&\quad \lambda(l, a). \lambda w. \text{wa } a \ w, \\
&\quad \text{ap}, \\
&\quad \lambda. \text{pf}, \\
&\quad \lambda a. \text{TRUE} \\
&\rangle \\
\text{LMtO} \text{M} \langle \text{af}', \text{wa}', \text{ap}', \text{pf}', \text{ig}' \rangle &\triangleq \langle \\
&\quad \lambda(l, \text{ACT } a). \lambda w. \text{af}' \ (l, a) \ w, \\
&\quad \lambda(l, \text{ACT } a). \lambda w. \text{wa}' \ (l, a) \ w, \\
&\quad \lambda(l_c, c). \lambda(l_a, a). \text{case } a \ \text{of} \\
&\quad \quad \text{ACT } a \rightarrow \text{ap}' \ c \ a \ \&\& \ (\text{ig}' \ a \ \parallel \ l_c \preceq l_a) \\
&\quad \quad \text{PROBE} \rightarrow l_c \preceq l_a, \\
&\quad \lambda p. \lambda(l, c). \text{case } (\text{pf}' \ l \ p \ c) \ \text{of} \\
&\quad \quad (c_l, c_r) \rightarrow ((l++[\text{L}], c_l), (l++[\text{R}], c_r)) \\
&\rangle
\end{aligned}$$

FIGURE 5. Converting between **IOModel** and **IOModelL**.

a location l , and when a context is split this location is modified accordingly to be $l++[\text{L}]$ and $l++[\text{R}]$ for the left and right-hand sides respectively. Finally, there is an extra “helper” action **PROBE** which should never actually be performed. Instead, a program, using `test` repeatedly to query whether **PROBE** is permitted, can discover the location in which it is running. This enables a program to supply the hidden extra argument to other location dependent actions.

After converting a location-based model back to a traditional one, it is possible to completely hide the extra **LOC** argument by replacing `action` and `test` with `actionL` and `testL` as follows: (actions and return values in \mathfrak{s} are of type α and v respectively, and `testL`, unlike `test`, returns a **Bool** indicating if the action is permitted by the current context.)

```

actionL ::  $\alpha \rightarrow \text{Prog}_{\text{LMtO} \text{M } \mathfrak{s}} \ v$ 
actionL a = probeLoc >>= \l -> action (l, Act a)

testL ::  $\alpha \rightarrow \text{Prog}_{\text{LMtO} \text{M } \mathfrak{s}} \ \text{Bool}$ 
testL a = probeLoc >>= \l ->
  test (l, Act a) (return True) (return False)

```

The function `probeLoc` :: $\text{Prog}_{\text{LMtO} \text{M } \mathfrak{s}} \ \text{Loc}$ (definition omitted) uses `test` and **PROBE** to determine a process’s location. This is rather inefficient, and it is possible that location-based I/O models should completely replace the normal I/O models in the language semantics. This, however, is future work and we will not consider it here.

4 COMBINATORS FOR I/O MODELS

In this section we take the first steps towards modelling a real functional I/O API and develop some tools for this purpose. Haskell’s `System.IO` library is a fully-fledged I/O interface, and it is our rough basis for this work. We choose to ignore some aspects of this I/O interface. For example: those which are semantically transparent, and included for efficiency reasons (ie. those to do with buffering); verbose error messages; actions which do not allow us any means of exploiting concurrency, for example, those which relate to directory structure; lazy file I/O functions such as `hGetContents` which read file contents lazily, and would need an operational semantics of lazy evaluation, such as Launchbury’s [7].

4.1 Combinators

We now give three general “combinators” for I/O models – one of the central contributions of this paper. These let one build large confluent formal models of I/O (ie. those that obey PRE’) by combining existing, smaller confluent models.

$$\begin{aligned}
 (*) &:: \text{IOModelL } v_1 \alpha_1 \rho_1 \omega_1 \zeta_1 \rightarrow \text{IOModelL } v_2 \alpha_2 \rho_2 \omega_2 \zeta_2 \rightarrow \\
 &\quad \text{IOModelL } (\text{Either } v_1 v_2) (\text{Either } \alpha_1 \alpha_2) (\rho_1, \rho_2) (\omega_1, \omega_2) (\zeta_1, \zeta_2) \\
 \text{smap} &:: \text{IOModelL } v \alpha \rho \omega \zeta \rightarrow \text{IOModelL } v (\text{String}, \alpha) \\
 &\quad [(\text{String}, \text{Splitter } \rho)] (\text{String} \rightarrow \omega) (\text{String} \rightarrow \text{Cxt } \zeta) \\
 \text{DynAction } \iota \alpha &\stackrel{\Delta}{=} \text{DYNACT } \iota \alpha \mid \text{NEXT} \mid \text{ALLOC } \iota \\
 \text{dmap} &:: \omega \rightarrow \text{IOModelL } v \alpha \rho \omega \zeta \rightarrow \\
 &\quad \text{IOModelL } (\text{Either } v \text{HndP}) (\text{DynAction HndP } \alpha) \\
 &\quad [(\text{HndP}, \text{Splitter } \rho)] (\text{Pool } \omega) (\text{HndP} \rightarrow \text{Cxt } \zeta)
 \end{aligned}$$

$s_1 * s_2$ represents the cartesian product of I/O models s_1 and s_2 . The world state becomes the cartesian product of the world states of s_1 and s_2 , and an action can be either an action from s_1 or one from s_2 . Similarly, a context must be able to determine whether actions from either s_1 or s_2 are allowed, and when performing concurrency, one must specify the permissions given to each sub-program for both sides of the world state.

The `smap` combinator turns an I/O model s into one in which world state is a map from `String` to the world state of s . (`String` could really be any type with an equality relation defined on it.) This is similar to the cartesian product, except there are an infinite number of s models, each indexed by a different `String`. An action (n, a) performs action a from s on the world state associated with name n . When performing concurrency, one must supply a list of how the permissions for each `String` are to be distributed between the left- and right-hand sides. If a `String` n 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 n . If a `String` occurs twice then the second one is ignored.

How do we give no permissions at all to a particular side? The type constructors **Cxt** and **Splitter** are used for this purpose:

$$\mathbf{Cxt} \zeta \triangleq \dots \quad \mathbf{Splitter} \rho \triangleq \mathbf{ALLEFT} \mid \mathbf{SPLIT} \rho \mid \mathbf{ALLRIGHT}$$

The type **Cxt** ζ denotes the context ζ extended to guarantee the existence of a context which permits no actions and a context which permits every action (we do not give its definition.) **Splitter** ρ is used to split contexts of type **Cxt** ζ . **SPLIT** p splits the context as specified above using p , **ALLEFT** gives all permissions to the left-hand side and **ALLRIGHT** gives all permissions to the right-hand side.

$\mathbf{dmap} \ w_0 \ s$ is an I/O model in which the world state is **Pool** ω (where ω is the world state type of s), and when a new ω is created dynamically its initial state is w_0 . Each action either performs an a action on a particular ω identified by handle h , **DynAct** $h \ a$, or it is one of two separate steps required to create a new ω . To create a new ω a process first determines the next free handle at its location using **NEXT** and then allocates it with **ALLOC** h . The reason for this two-step process is that a process must have complete access to a handle that it allocates (it must be the “everything” element of type **Cxt** ζ), and I/O contexts cannot forbid actions based on their return values, only their arguments.

DynAct $h \ a$ is location dependent if and only if a was location dependent originally. **ALLOC** and **NEXT** are both always location dependent. Contexts are split in a style roughly similar to that **smap** except that the default behaviour differs. A sub-process should, by default, have access to all possible future handles it might need. (Contexts cannot “know” dynamic facts about the world state, such as whether a handle is in use, so one cannot prevent a programmer from distributing access to an as yet unused handle.) So if a process running in location l splits into two processes then, unless otherwise specified, all permissions to access the ω identified by handle (l_h, i_h) will go the left-hand side if and only if $l \mathbf{++}[\mathbf{L}] \preceq l_h$.

Theorem 4.1. *The important technical results are*

1. \mathbf{PRE}'_{s_1} and \mathbf{PRE}'_{s_2} imply $\mathbf{PRE}'_{s_1 * s_2}$
2. \mathbf{PRE}'_s implies $\mathbf{PRE}'_{\mathbf{smap} \ s}$
3. \mathbf{PRE}'_s implies $\mathbf{PRE}'_{\mathbf{dmap} \ w_0 \ s}$

4.2 A File I/O model

Figure 6 contains the outline of a simple location-based model of a single file. A file either doesn't exist, **NOFILE**, or it exists and contains both the file contents, a sequence of characters, and either a single write-pointer or a pool of read-pointers, some of which may be closed. 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 non-determinism. Depending on the interleaving of process execution, two

ω_{File}	\triangleq	NOFILE FILE [Char] (Either (Pool FPtr) Int)
FPtr	\triangleq	ACTIVE Int STALE
v_{File}	\triangleq	RCHAR Char RHNDP HndP RBOOL Bool RNULL ...
FHnd	\triangleq	RWHND RHND HndP
α_{File}	\triangleq	FDELETE FWRITE Char FWROPEN ... HGETC FHnd HCLOSE FHnd HISEOF FHnd ... FNEXTRDHND HRDOPEN HndP FDOESEEXIST ...
ζ_{File}	\triangleq	FWRITECXT FREADCXT (HndP \rightarrow Bool) FNONECXT
ρ_{File}	\triangleq	([FHnd], [FHnd])

FIGURE 6. Fragments of a file Model

processes could obtain different handles. Also, for the same reason there is no explicit representation of a closed file. A file is closed if the pool of read-pointers contains only inactive pointers. If the structure was reset when all pointers became inactive then this could also introduce non-determinism.

The I/O context associated with a file is either FWRITECXT, indicating that all actions are possible, FREADCXT m , indicating that the process has access to certain read-handles, and FNONECXT, indicating that the process cannot perform any action whatsoever.

There are roughly three sorts of actions on files.

- Those which affect the entire file. These are only permitted by the FWRITECXT context and must be explicitly sequenced, admitting no concurrency at all. These include: opening a file for writing; writing to a file; deleting a file.
- Those actions which affect only a single pointer and may examine the file contents, but not change them. These require an extra parameter of type FHnd, and two actions on two different read-handles are order independent. These actions include: reading a character; checking for end-of-file; closing an open handle; checking the file size.
- Anomalous actions which must be treated separately. These include the two location-dependent actions required of allocating a new read handle.

Like with `dmap`, when splitting a context, one must supply the specific handles which each side requires access to. This is done by giving two lists of FHnd, one for the left-hand process, one for the right-hand process. If both sides request the same permissions then the requests of one side will be turned down, and requesting write-access immediately causes the other side to be given no permissions at all.

There is also the same possibility that although a process will be able to create a new handle, due to the way in which contexts had been split it will not have access to it. The same default scheme is employed as was used with `dmap`. The procedure for allocating new read handles is similar to allocation with `dmap`: first one acquires the next read-handle using `FNEXTRDHND`; then one allocates the new read pointer using `HRDOPEN`. These are the only location dependent actions in the file model, and no file actions stall.

This is still a far cry from a normal I/O interface. In particular the opening of a file requires three separate actions – two when opening for reading and another separate action when opening for writing. This is solved by wrapping these actions in libraries which provide a convenient interface, and a unified notion of handle.

5 A UNIFIED I/O LIBRARY

Finally, using the above combinators we can combine all our scraps into one unified I/O model `io` as follows:

$$\begin{aligned} \text{io} &\triangleq \text{LMt oM io}' & \text{chan}' &\triangleq \text{Mt oLM chan} & \text{term}' &\triangleq \text{Mt oLM term} \\ \text{io}' &\triangleq \text{term}' * (\text{smap file} * \text{dmap [] chan}') \end{aligned}$$

This lets the user (1) do terminal I/O, (2) access a potentially infinite number of files, each identified by a different string and (3) dynamically allocate communication channels, whose initial world state is an empty buffer `[]`, to allow processes to communicate. One can also, when splitting contexts, allocate to a particular process access to a particular file, a collection of files, or, perhaps, full access to one file, read access to another and write access to a particular channel.

The type of model `io'` is enormous:

```
io' :: IOModelL
      (Either v_Term (Either v_File (Either v_Chan HndP)))
      (Either alpha_Term (Either (String, alpha_File) (DynAction HndP alpha_Chan)))
      (rho_Term, ([ (String, Splitter rho_File) ], [ (HndP, Splitter rho_Chan) ]))
      (omega_Term, (String -> omega_File, Pool omega_Chan))
      (zeta_Term, (String -> Cxt zeta_File, HndP -> Cxt zeta_Chan))
```

This is quite cumbersome to use but it is only a back-end ultimately. In practice the sum and product types become transparent by using a suitable set of libraries. The Haskell 98 Revised Report [10] defines a collection of built-in I/O actions. Both terminal and file I/O are performed using a `Handle` type, and these include

```
stdin, stdout :: Handle
  fOpen      :: String -> IOMode -> Prog_io Handle
  fClose     :: Handle -> Prog_io ()
  hGetChar   :: Handle -> Prog_io Char
  hPutChar   :: Handle -> Char -> Prog_io ()
```

```

programWithLogFile :: Progio ()
programWithLogFile = do
  (cr,cw) <- chOpen -- open a new communication channel
  parIO
    [FileHnd "logfile.txt" RWHnd, cr] (process_data_and_log cr)
    [] (rest_of_program cw)

interactiveLoop :: Progio ()
interactiveLoop = do
  cmd <- readNextCmd stdin
  case cmd of
    EncodeFile fn -> do
      parIO [FileHnd fn RWHnd] (fEncode fn) [] interactiveLoop
      return ()
    ExitProgram -> return ()
  -- ... other actions, recursively calling 'interactiveLoop'

```

FIGURE 7. Two Program Skeletons written using I/O Libraries

We give a semantics to these and also extend the interface: `chOpen` allocates a new communication channel, returning the separate read handle and write handle. `canRead` and `canWrite` determine at run-time if the current I/O context permits reading/writing to a particular handle. `parIO` gives a convenient way of distributing permissions to each process when doing concurrency – each process is accompanied by a list of handles which it would like to have access to, and if these are underspecified then the default for the various I/O models will apply.

```

chOpen  :: Progio (Handle, Handle)
canRead :: Handle → Progio Bool
canWrite :: Handle → Progio Bool
parIO  :: ∀β. ∀γ. [Handle] → Progio β → [Handle] → Progio γ → Progio (β, γ)

```

Figure 8 shows the implementation of some of the above functions, and Figure 7 gives two basic example programs. In the first example, a program creates a separate process for logging data, and sends it data to log via a channel, and in the second an interactive program spawns a new process which encodes a file. In both cases, as is always the case with CURIO, the termination of two concurrent processes spawned with a single `par` must be synchronised.

6 RELATED WORK, CONCLUSIONS AND FUTURE WORK

There have been a number of attempts at deterministic, concurrent I/O in the functional language literature. Probably the most famous approach is that of Clean's

uniqueness types [1] which allow the world-state to be split into its component files. Work on concurrent monadic interfacing [6] uses rank-2 polymorphism to guarantee concurrent access to different files in a monadic setting. Ultimately, however, we are concerned not only with deterministic concurrency but also explaining the actual API with a view to eventually giving correctness proofs for programs. Existing semantics for I/O have been co-inductive [5, 8], so CURIO is rather different.

Future work will focus on the development of a more elegant lattice-like structure for I/O contexts which would automatically include such notions as a context which permits all actions and a context which permits no actions. We also plan to investigate whether the run-time checks CURIO requires could be performed statically using the type-checker. Ideally we would like to abolish the use of `test` entirely by forcing the type-checker to detect these problems. I/O contexts are quite type-like, since a program's enclosing I/O context does not change over time.

REFERENCES

- [1] P. Achten and R. Plasmeijer. The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, January 1995.
- [2] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers. In T. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop, IFL2001*, number 2312 in LNCS, pages 55–71. Springer-Verlag, 2001.
- [3] M. Dowse and A. Butterfield. The Curio Language Specification and its Machine-Verified Confluence Proof. Technical Report TCD-CS-2005-56, Trinity College Dublin, July 2005.
- [4] M. Dowse, A. Butterfield, and M. van Eekelen. Reasoning about deterministic concurrent functional I/O. In C. Grelck and F. Huch, editors, *Proceedings of IFL 2004*, volume LNCS3474, pages 177–194. Springer-Verlag, 2005.
- [5] A. 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.
- [6] 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.
- [7] J. Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154, New York, NY, USA, 1993. ACM Press.
- [8] S. Peyton Jones. Tackling the awkward squad – monadic input/output, concurrency, exceptions, and foreign language calls in Haskell. In C. Hoare, M. Broy, and R. Stein-brueggen, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, pages 47–96. IOS Press, 2001.
- [9] 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.
- [10] S. Peyton Jones, P. Hudak, P. Wadler, and J. Hughes, editors. *Haskell 98 Language and Libraries, the Revised Report*. CUP, April 2003.

$$\text{Handle} \triangleq \text{STDINHND} \mid \text{STDOUTHND} \mid \text{FILEHND String FHnd} \\ \mid \text{CHANRDHND HndP} \mid \text{CHANWRHND HndP}$$

stdin, stdout :: Handle

```
stdin = StdInHnd
stdout = StdOutHnd
```

fOpen :: String → IOMode → Prog_{i_o} Handle

```
fOpen fn ReadMode = do
  Right (Left (RHndP rh)) <-
    actionL (Right (Left (n, FNextRdHnd)))
  actionL (Right (Left (n, HRdOpen rh)))
  return (FileHnd n (RHnd rh))
fOpen fn ReadWriteMode = do
  actionL (Right (Left (n, FWrOpen)))
  return (FileHnd n RWHnd)
```

hPutChar :: Handle → Char → Prog_{i_o} ()

```
hPutChar hnd c = do
  case hnd of
    StdOutHnd      -> actionL (Left (PutC c))
    FileHnd n RWHnd ->
      actionL (Right (Left (n, FWrite c)))
    ChanWrHnd h    ->
      actionL (Right (Right (DynAct h (ChPutC c))))
  return ()
```

chOpen :: Prog_{i_o} (Handle, Handle)

```
chOpen = do
  Right (Right (Right h)) <-
    actionL (Right (Right Next))
  actionL (Right (Right (Alloc h)))
  return (ChanRdHnd h, ChanWrHnd h)
```

canRead :: Handle → Prog_{i_o} Bool

```
canRed hnd = case hnd of
  StdInHnd      -> testL (Left GetC)
  FileHnd n h   -> testL (Right (Left (n, HGetC h)))
  ChanRdHnd h   -> testL (Right (Right (DynAct h ChGetC)))
  _             -> return False
```

FIGURE 8. The Semantics of High-Level I/O Actions