A Language for Reasoning about Concurrent Functional I/O

Malcolm Dowse, Trinity College Dublin
(Malcolm.Dowse@cs.tcd.ie)

Andrew Butterfield, Trinity College Dublin
(Andrew.Butterfield@cs.tcd.ie)

IFL 2004,
Lübeck, Germany,
2004-09-08

Supported by Enterprise Ireland Basic Research Grant SC-2002-283
Overview

Our research is concerned with reasoning formally about I/O and effects.

In this talk we present:

- A language for expressing and reasoning about concurrent functional I/O.
  - a state-transformer model of world-state.
- “Contexts”, used to encode which actions programs are allowed to perform.
- A pre-condition which guarantees deterministic evaluation.
- An implementation in Core-Clean.
- Overview of a confluence proof in Sparkle.
- Some examples: a file-system.
Concurrent I/O in Pure Functional Languages

In pure functional languages I/O is expressed using some means of explicitly sequencing actions.

The most well-known example is monads.

At times it is more elegant and expressive to be able to leave the ordering of actions unspecified.

“All I/O operations [are] strictly sequenced along a single “trunk”. Sometimes, though, such strict sequencing is unwanted.” [“Imperative Functional Programming”, Peyton Jones, Wadler (1993)]
Different Solutions

The languages Haskell and Clean offer two different solutions to the problem.

• Clean
  – The unique type-system allow a limited amount of concurrency.
  – Actions can be sequenced on separate, discrete parts of the I/O system.
  – For example, individual files.
  – Since the concurrent actions don’t interfere with each other, the language remains deterministic.

• (Concurrent) Haskell
  – Adds powerful concurrency primitives to the language.
  – Forking, synchronisation, inter-process communication.
  – Useful in practice.
  – But I/O semantics becomes nondeterministic.

For reasoning, a deterministic semantics is usually more desirable.
An Initial Attempt

An initial attempt to extend the existing language is shown below:

\[
\begin{align*}
: & : \text{Prog } v \ a = \text{Bind} \ (\text{Prog } v \ a) \ (v \rightarrow \text{Prog } v \ a) \\
& \quad \mid \text{Ret } v \\
& \quad \mid \text{Act } a \\
& \quad \mid \text{Par} \ (\text{Prog } v \ a) \ (\text{Prog } v \ a) \ (v \ v \rightarrow v)
\end{align*}
\]

The language is the same except for the extra Par construct.

- Par \( m_1 \ m_2 \ vf \) runs \( m_1 \) in parallel with \( m_2 \).
- If \( m_1 \) and \( m_2 \) return values \( v_1 \) and \( v_2 \) respectively, it returns \( vf \ v_1 \ v_2 \).

Some simple syntactic abbreviations:

\[
\begin{align*}
\text{Bind } m \ f & \overset{\text{def}}{=} m \gg f \\
\text{Par } m_1 \ m_2 (\ast) & \overset{\text{def}}{=} m_1 \ | m_2
\end{align*}
\]
Initial Attempt - Reduction Rules

The meaning of our system is defined with \( \text{af} :: a \ w \rightarrow (w,v) \).

\[
\begin{align*}
    w \vdash \text{Ret } v \gg f & \implies w \vdash f \ v \\
    w \vdash m & \implies w_1 \vdash m_1 \\
    w \vdash m \gg f & \implies w_1 \vdash m_1 \gg f \\
    \text{af } a \ w = (w_1,v_1) & \\
    w \vdash \text{Act } a & \implies w_1 \vdash \text{Ret } v_1 \\
    w \vdash (\text{Ret } v_l)^* (\text{Ret } v_r) & \implies w \vdash \text{Ret } v_l * v_r \\
    w \vdash m_l & \implies w_1 \vdash m_1 \\
    w \vdash m_l \parallel m_r & \implies w_1 \vdash m_1 \parallel m_r \\
    w \vdash m_r & \implies w_1 \vdash m_1 \\
    w \vdash m_l \parallel m_r & \implies w_1 \vdash m_l \parallel m_1
\end{align*}
\]
Constraining Concurrent Execution

Problem: Arbitrary concurrency (in general) results in non-determinism.

Solution:

• Isolate specific actions which permit (deterministic) concurrency.

• Supply extra information indicating
  – What these actions are.
  – How we test at run-time when concurrency is legitimate.

• We add
  – Two extra functions.
  – Two extra types.
  – A pre-condition.
Expressing Non-Interference

When can two actions run concurrently without affecting each other?

If the order in which they are executed is irrelevant. \( \text{swap}(a_1, a_2) \) expresses this formally.

\[ \text{swap}(a_1, a_2) \overset{\text{def}}{=} \forall (*). \text{fseq} (\text{Act} a_1) (\text{Act} a_2) (*) \cong \text{fseq} (\text{Act} a_2) (\text{Act} a_1) (\text{flip} (*)) \]

File-System model example: Actions on different files may be executed concurrently.

The above definition makes use of function \( \text{fseq} \).

\[ \text{fseq} :: (\text{Prog} v a) (\text{Prog} v a) (v v \to v) \to (\text{Prog} v a) \]

\[ \text{fseq} m1 m2 vf = \text{Bind} m1 (\lambda v1 \to \text{Bind} m2 (\lambda v1 \to \text{Ret} (vf \ v1 \ v2))) \]

\( \text{fseq} a1 a2 vf \) performs action \( a1 \), then action \( a2 \).

It returns value \( (vf \ v1 \ v2) \), where \( v1 \) and \( v2 \) are the two actions’ respective return values.
• Our approach is to assume each program-fragment is given certain permissions at run-time.
  – eg. the files it is allowed to modify.

• We call these permissions contexts.

• A context acts as a child-lock on certain actions.

• We require a second function, \( \text{ap} : \text{a c} \to \text{Bool} \).
  – Type c represent contexts.
  – ap gives the meaning of a context.
  – \((\text{ap a c})\) is a Boolean value which indicates if action a is permitted by context c.

• All reduction rules are annotated.
  – \( w_0 \vdash m_0 \overset{c}{\longrightarrow} w_1 \vdash m_1 = \text{“}m_0 \text{ reduces to } m_1 \text{ in context } c\text{”} \).
Context Properties

Two important relations on contexts are:

\[ c_1 \sqsubseteq c_2 \overset{\text{def}}{=} \forall a. \text{ap} \ a \ c_1 \implies \text{ap} \ a \ c_2 \]

\[ c_1 ||| c_2 \overset{\text{def}}{=} \forall a_1. \forall a_2. \text{ap} \ a_1 \ c_1 \land \text{ap} \ a_2 \ c_2 \implies \text{swap}(a_1, a_2) \]

\( c_1 \sqsubseteq c_2 = \) “Any action permitted by \( c_1 \) is also permitted by \( c_2 \)”.

\( c_1 ||| c_2 = \) “Actions permitted by \( c_1 \) cannot interfere with actions permitted by \( c_2 \)”.

\( \sqsubseteq \) is a pre-order and **|||** is symmetric (both by definition).
Splitting Contexts

• Suppose a program run in context $c$ executes two programs in parallel.
• These two programs also have contexts, say, $c_l$ and $c_r$.
• We must ensure
  – $c_l$ and $c_r$ don’t permit actions disallowed by $c$.
  – Actions permitted by $c_l$ don’t interfere with actions permitted by $c_r$.
• Or, formally, $c_l \sqsubseteq c \land c_r \sqsubseteq c \land c_l \parallel c_r$
  – it is not clear how to check this property, so we enforce it.
  – The solution is to introduce a third function $\text{pf} :: p \ c \rightarrow (c, c)$.
• Type $p$ is the type of a parameter to the concurrent execution, indicating how the contexts are to be split.
• We assume it has been proved that $\text{pf}$ splits contexts so as to adhere to the above rule:

$$\text{pf} \ p \ c = (c_l, c_r) \implies c_l \sqsubseteq c \land c_r \sqsubseteq c \land c_l \parallel c_r$$
Modifications (1)

Originally we needed:

- three types (\( v, a \) and \( w \) – return values, actions and world-state).
- one function \( af \) (for describing the meaning of each action).

Now we add:

- two additional functions (\( ap \) and \( pf \)).
- two additional types (\( c \) and \( p \)).
- one pre-condition.

The full model is encoded with the record-type:

\[
\text{:: CSystem } v \ a \ p \ w \ c := \{ \ af :: a \ w \to (w,v) \\
\quad , \ ap :: c \ a \to \text{Bool} \\
\quad , \ pf :: p \ c \to (c,c) \}
\]

The assumed pre-condition is:

\[
\text{pf } p \ c = (c_l,c_r) \implies c_l \sqsubseteq c \land c_r \sqsubseteq c \land c_l \parallel c_r
\]
Modifications (2)

The language is rewritten as follows:

\[
:: \text{Prog } v \ a \ p = \text{Bind } (\text{Prog } v \ a \ p) (v \to \text{Prog } v \ a \ p) \\
| \text{Ret } v \\
| \text{Act } a (\text{Prog } v \ a \ p) \\
| \text{Par } p (\text{Prog } v \ a \ p) (\text{Prog } v \ a \ p) (v \to v)
\]

We add:

- A parameter of type \( p \) to the parallelism construct.
  - If \( \text{Par } p \ m_l \ m_r (\ast) \) is run in context \( c \) then \( \text{pf } p \ c \) determines the contexts in which programs \( m_l \) and \( m_r \) are run.
  - We write this program as \( m_l \ \parallel^{*} p \ m_r \).

- An “exception” mechanism to primitive actions.
  - An action \( a \) might not be permitted.
  - In such cases, \( \text{Act } a \ m_e \) performs program \( m_e \) instead.
**Non-Deterministic Single-Step Rules**

\[
\begin{align*}
\text{af } a \, w &= (w_1, v_1) \quad \text{ap } c \, a = \text{True} \\
\frac{w \vdash \text{Act } a \, m_e}{w_1 \vdash \text{Ret } v_1}
\end{align*}
\]  \hspace{1cm} (1)

\[
\frac{\text{ap } c \, a = \text{False}}{w \vdash \text{Act } a \, m_e \rightarrow_c w_1 \vdash m_e}
\]  \hspace{1cm} (2)

\[
\frac{w \vdash (\text{Ret } v_l) \parallel \text{(Ret } v_r)}{w_1 \vdash \text{Ret } v_l \ast v_r} \quad \text{pf } p \, c = (c_l, c_r)
\]  \hspace{1cm} (3)

\[
\frac{w \vdash m_l \rightarrow_{c_l} w_1 \vdash m_1}{w \vdash m_l \parallel m_r \rightarrow_c w_1 \vdash m_l \parallel m_r} \quad \text{pf } p \, c = (c_l, c_r)
\]  \hspace{1cm} (4)

\[
\frac{w \vdash m_r \rightarrow_{c_r} w_1 \vdash m_1}{w \vdash m_l \parallel m_r \rightarrow_c w_1 \vdash m_l \parallel m_1} \quad \text{pf } p \, c = (c_l, c_r)
\]  \hspace{1cm} (5)

The rules for Ret and Bind remain unchanged for any context.
A Meta-Encoding for Non-Determinism (1)

The previous rules have been given at a high-level. How should one model non-determinism in a deterministic language like Core-Clean?

- The resultant program/world state is only one of a possible list of states.
  - The program is updated using a function of type \( a \rightarrow [a] \), not \( a \rightarrow a \).
  - “A can reduce to B” = “B is an element of the list resulting from doing A”.
  - “A always reduces to B” = “B is every element of the list resulting from doing A”.
  - Problem: We really want a set, not a list. Sparkle can’t model sets easily.

- The possible choices are determined in advance by the (presumable) random value of an extra parameter.
  - We quantify over all possible values of this parameter.
  - “A can reduce to B” = “There exists a random value such that A reduces to B”.
  - “A always reduces to B” = “A reduces to B for all (sensible) random values”.

A Meta-Encoding for Non-Determinism (2)

Our system takes the second approach.

:: Random = ![Bool]

nextC :: (CSystem v a p w c) c Random (Prog v a p, w) -> (Prog v a p, w)

nextC s c r (m,w) = // ....

- nextC implements single-step reduction.
- The extra parameter is Random, a strict list of Bool.
- One random Bool is consumed for each syntactic level of concurrency.
  - False? If possible reduce the left-hand-side.
  - True? If possible reduce the right-hand-side.
  - No values left? Default to False.

The implementation of the reduction rules is as follows: (We assume s is an implicit CSystem.)

\[
\begin{align*}
  w \vdash m & \quad \longrightarrow_c \quad w_1 \vdash m_1 \quad \text{def} \quad \text{nextC } s \ c \ r \ (m, w) = (m_1, w_1) \\
  w \vdash m & \quad \longrightarrow_c \quad w_1 \vdash m_1 \quad \text{def} \quad \exists r. r \neq \bot \land w \vdash m \quad \longrightarrow_c \quad w_1 \vdash m_1
\end{align*}
\]
Deterministic Reduction Rules (with Randomness)

\[
\begin{align*}
&w \vdash (\text{Ret } v_l) \parallel (\text{Ret } v_r) \xrightarrow{r} w \vdash \text{Ret } v_l \ast v_r & \text{pf } p \ c = (c_l, c_r), r \neq \bot & \\
&w \vdash m_l \parallel m_r \xrightarrow{p} w_1 \vdash m_1 & \text{pf } p \ c = (c_l, c_r) & \\
&w \vdash m_r \xrightarrow{p} w_1 \vdash m_1 & \text{pf } p \ c = (c_l, c_r) & \\
&w \vdash (\text{Ret } v_l) \parallel m_r \xrightarrow{p} w_1 \vdash (\text{Ret } v_l) \parallel m_1 & \\
&w \vdash m_l \xrightarrow{r} w_1 \vdash m_1 & \text{pf } p \ c = (c_l, c_r) & \\
&w \vdash m_l \parallel m_r \xrightarrow{p} w_1 \vdash [\text{False}: r] m_1 \parallel m_r & \text{pf } p \ c = (c_l, c_r) & \\
&w \vdash m_r \xrightarrow{r} w_1 \vdash m_1 & \text{pf } p \ c = (c_l, c_r) & \\
&w \vdash m_l \parallel m_r \xrightarrow{p} w_1 \vdash [\text{True}: r] m_1 \parallel m_1 & \text{pf } p \ c = (c_l, c_r) &
\end{align*}
\]

The rules for Ret, Act and Bind remain true for any (defined) Random value.
Evaluation

Evaluation is the process of single-step reducing a program until it is a single value.

To fully evaluate a program we need a potentially infinite number of Randoms – one for each individual reduction step.

The function $\text{rdceC}$ therefore requires input of type $\text{Random2}$ – a strict list of Random.

\[
\text{:: Random2 := ![Random]} \quad \text{rdceC :: Int (CSys v a p w c) c Random2 (Prog v a p, w) \rightarrow (Prog v a p, w)}
\]

Evaluation and divergence is expressed as follows.

\[
\text{w \vdash m} \Downarrow \langle w_1, v_1 \rangle \overset{c.r}{=} \exists i. \text{rdceC i s c r} (m, w) = (\text{Ret } v_1, w_1)
\]

\[
\text{w \vdash m} \Uparrow \overset{c.r}{=} \neg \exists v_1. \exists w_1. \exists i. \text{rdceC i s c r} (m, w) = (\text{Ret } v_1, w_1)
\]

The behaviour of a program/world pair is determined by its context $c$ and a value $r$ of type $\text{Random2}$.
Proving Confluence

- The confluence proof amounts to saying that the random value \( r \) is irrelevant.
- Reduction isn’t deterministic, but, if the pre-condition holds, evaluation/divergence is.
- Proof: Long and messy...
  - At its core, a large double-induction.
    - Strong induction over the number of reduction steps.
    - Structural induction over \( \text{Prog} \) type.
  - Expressing evaluation as a function (no existential quantification).
  - Single step reducing a program only performs at most one (permitted) action.
    (induction over \( \text{Prog} \)).
  - If \( c_l \ ||\ | \ c_r \) then a single-step reduction of an action in \( c_l \) can be swapped with one in \( c_r \).
  - Transforming Random2 values into a more useful canonical form.
  - Admissibility issues...
File-System Example (1)

The old file-system model:

:: FSCall ::= (FSAction, Nam)
:: FSAction = FRead | FOpen | FClose | FWrite !Char | FCreate |
            | FDel | FExists | FEOF | FOpened
:: RV    = RInt !Int | RChar !Char | RBool !Bool | RNull
:: FS    = // ...

The three types were bounds as follows:

- \( v = RV \)
- \( a = FSCall \)
- \( w = FS \)

.. and we showed how actions on different files don’t interfere with each other.

\[
\text{getName } a_1 \neq \text{getName } a_2 \implies swap(a_1, a_2) \quad (11)
\]

getName :: FSCall -> Nam
getName = snd
File-System Example (2)

We want to enforce this with contexts in our language. We choose:

- \( c = (\text{Nam} \to \text{Bool}) \), a map from filenames to \text{Bool}.
  - Indicates what files a program is allowed to access.

- \( p = [\text{Nam}] \), a list of filenames.
  - Indicates which specific files the right-hand-process shall be allowed to access, if they’re not already forbidden.
  - The left-hand-process will be allowed access all remaining (permitted) files.

\[
\begin{align*}
ap &:: (\text{Nam} \to \text{Bool}) \text{FSCall} \to \text{Bool} \\
ap \ cm \ a &= c \ (\text{name} \ a) \\

pf &:: [\text{Nam}] (\text{Nam} \to \text{Bool}) \to (\text{Nam} \to \text{Bool}, \text{Nam} \to \text{Bool}) \\
pf \ ns \ cm &= (\ n \to \ cm \ n \land \not \ (\text{isMember} \ n \ ns), \\
&\quad \ n \to \ cm \ n \land \text{isMember} \ n \ ns)
\end{align*}
\]
**File-System Example (3)**

These can easily be shown to obey the required pre-condition.

\[c_l \subseteq c \equiv c \text{ (getName } a\text{) } \&\& \text{ not (isMember (getName } a\text{) } ns) \Rightarrow c \text{ (getName } a\text{)}\]

\[c_r \subseteq c \equiv c \text{ (getName } a\text{) } \&\& \text{ isMember (getName } a\text{) } ns \Rightarrow c \text{ (getName } a\text{)}\]

\[c \text{ (getName } al\text{) } \&\& \text{ not (isMember (getName } al\text{) } ns) \land \]

\[c \text{ (getName } ar\text{) } \&\& \text{ isMember (getName } ar\text{) } ns \Rightarrow \]

\[\text{swap}(al, ar)\]

The first two are trivial.

In the final theorem we show by contradiction that \((\text{getName } al) \neq (\text{getName } ar)\).

This fact is enough to show that the actions don't interfere.
**File-System Example (4)**

`totalLength ns` calculates, concurrently, the sum total of the lengths of each file in `ns`.

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

fileLength :: Nam -> Prog RV FSCall [Nam]
fileLength n =
  Bind (Act (FOpen,n) (Ret (RInt 0)))(\v -> case v of
    RInt _ -> Ret (RInt 0)
    RNull -> Bind (fileLenLoop n 0)
             (\l -> mseq (Act (FClose,n) undef) (Ret l))

fileLenLoop :: Nam Int -> Prog RV FSCall [Nam]
fileLenLoop n l = ifelse (Act (FEOF,n) undef) (Ret (RInt 1))
                   (mseq (Act (FRead,n) undef) (fileLenLoop n (l+1))
```
**File-System – Finer Granularity**

Not all actions on the same file interfere with one another.

The following table indicates all the actions for which this is true in our file-system model:

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOpen, FClose</td>
</tr>
<tr>
<td>FRead, FWrite</td>
</tr>
<tr>
<td>FCreate, FDel</td>
</tr>
<tr>
<td>FEOF</td>
</tr>
<tr>
<td>FOpened</td>
</tr>
<tr>
<td>FExists</td>
</tr>
</tbody>
</table>

For example: A process incapable of creating or destroying a file won’t interfere with a process which can only query the existence of that file.

This suggests there are opportunities for a more fine-grained concurrency on shared state.

And, given a more sophisticated model with file-handles: shared reads.
Conclusions

- A language for expressing and reasoning about concurrent I/O.
  - We guarantee the non-interference of concurrent processes.
  - Processes have contexts which determine what actions they can perform.
  - Evaluation is shown to be deterministic.
- The language was implemented in Clean.
- All properties were proven with Sparkle.
  - Proof-assistants keep you honest!
- Some simple file-system examples.
Future Work

• A family of complex I/O models with interesting concurrency results.
  – Shared reads.
  – Stream processors / GUIs.
  – A more sophisticated file-system.

• Reasoning about programs with context and concurrency.
  – Determinism is good.
  – .. but extra run-time checks are bad.

• Getting the type-system to enforce properties.
  – Reducing the amount of extra run-time information.
  – Eliminating the “exceptions” required for actions.