guages and Computer Architecture, Copenhagen, June 9–11, 1993, pages
136–145. ACM Press, 1993.

An Operational Semantics for I/O in a Lazy Functional Language

Andrew D. Gordon
Programming Methodology Group,
Department of Computer Science,
Chalmers University of Technology,
412 96 Gothenburg, Sweden.
gordon@cs.chalmers.se

Abstract

I/O mechanisms are needed if functional languages are to be suitable for general purpose programming and several implementations exist. But little is known about semantic methods for specifying and proving properties of lazy functional programs engaged in I/O. As a step towards formal methods of reasoning about realistic I/O we investigate three widely implemented mechanisms in the setting of teletype I/O: synchronised-stream (primitive in Haskell), continuation-passing (derived in Haskell) and Landin-stream I/O (where programs map an input stream to an output stream of characters). Using methods from Milner's CCS we give a labelled transition semantics for the three mechanisms. We adopt bisimulation equivalence as equality on programs engaged in I/O and give functions to map between the three kinds of I/O. The main result is the first formal proof of semantic equivalence of the three mechanisms, generalising an informal argument of the Haskell committee.

1 Introduction and motivation

In the absence of formal semantics and of ways to prove program properties, any mechanism for input/output (I/O) in a lazy functional language is bound to be contentious.

The tension is plain. On the one hand, lazy functional languages are advocated on the basis of their simple semantics and the ease with which program properties can be proved. On the other, I/O is concerned with state and communication and hence does not apparently fit into the framework of functional semantics and proof techniques used with lazy functional languages.

The tension is immediate. Several different I/O mechanisms have been implemented for lazy languages such as Haskell, but virtually no work has been done to develop semantic methods to cope with I/O. Given these mechanisms, functional programs are capable of accessing complex operating system and window system facilities. Such mechanisms are needed to make general purpose programming feasible in a lazy language. But it is unconvincing to advocate a functional language on the basis both of its suitability for formal methods and of the expressiveness of its I/O, unless the I/O mechanism can be well integrated into the language's semantics.

As a response to this tension, we advocate operational semantics and bisimulation equivalence as the basis of a simple yet powerful theory of functional programming and I/O. We show in this paper how methods from the CCS theory of concurrency can be applied to specify and prove properties of functional programs engaged in I/O.

We develop here an operational semantics for functional I/O that was first used by Holmström in his semantics of PFL [9]. Holmström used a continuation-passing style to embed CCS-like operations for communication and concurrency in a functional language. Starting with an evaluation relation for the host language, he defined the meaning of the embedded operations in the style of a labelled transition system, as used in CCS [18]. A labelled transition system is a way to formalise the idea that an agent (such as a functional program engaged in I/O) can perform an action (such as input or output of a character) and then become a successor agent. This style of semantics is attractive for at least three reasons. First, it can model a wide variety of nondeterministic and concurrent computation: witness the CCS school of concurrency theory. Second, the evaluation relation for the host language is unmodified; any property of the host language without I/O will still hold after the I/O mechanism has been added. Third, the method complements an operational language definition such as that of SML [19].

We go further than Holmström by developing an equational theory of functional I/O based on labelled transitions. In the context of teletype I/O (input from a keyboard, output to a printer) we give an operational semantics for three I/O mechanisms: *Landin-stream*, *synchronised-stream* and *continuation-passing* I/O. We adopt bisimulation equivalence from CCS as equality on programs engaged in I/O. Bisimilarity is an equivalence on agents induced by their operational behaviour: two agents are bisimilar iff whenever one can perform an action, the other can too such that their two successors are bisimilar. We verify that mappings between these I/O mechanisms are bisimulation-preserving. Those between Landin-stream and continuation-passing I/O are original, while the mappings between synchronised-stream and continuation-passing I/O were discovered during the design of Haskell [12, 11], but have not hitherto been verified formally.

The method presented here is not immediately applicable to *side-effecting* I/O mechanisms, such as the "pseudo functions" `read` and `write` in LISP 1.5 [17] and their descendants in say SML. Programmers using lazy languages are

σ, τ	$::=$	X	(type variable)
	$\mid$	$\mathtt{Int}$	(natural numbers)
	$\mid$	$(\sigma_1 \mathtt{->} \sigma_2)$	(functions)
	$\mid$	μ	(algebraic datatype)
μ	$::=$	$(\mathtt{data}\, X = \mathtt{dc}_1 \mid \cdots \mid \mathtt{dc}_n)$	(algebraic datatype, X bound in each $\mathtt{dc}_i$, constructors distinct)
$\mathtt{dc}$	$::=$	$(K\, \sigma_1 \cdots \sigma_m)$	($\mathtt{data}$-clause, $m \geq 0$)
$\mathtt{c}$	$::=$	$\underline{n}$	(literal, $n \in \mathbb{N}$)
	$\mid$	$(\backslash x{::}\sigma \mathtt{->} \mathtt{e})$	(abstraction, x bound in $\mathtt{e}$)
	$\mid$	$(K^{\mu}\, \mathtt{e}_1 \cdots \mathtt{e}_m)$	(constructor application)
$\mathtt{e}$	$::=$	x	(value variable)
	$\mid$	$\mathtt{c}$	(canonical term)
	$\mid$	$(\mathtt{e}_1 \underline{\oplus} \mathtt{e}_2)$	(arithmetic, $\oplus \in \{+, -, \times, =, <\}$)
	$\mid$	$(\mathtt{e}_1\, \mathtt{e}_2)$	(call-by-name application)
	$\mid$	$(\mathtt{e}_1 \hat{}\, \mathtt{e}_2)$	(call-by-value application)
	$\mid$	$\mathtt{rec}^{\sigma}(x.\, \mathtt{e})$	(recursion, x bound in $\mathtt{e}$)
	$\mid$	$(\mathtt{case}^{\mu}\ \mathtt{e}\ \mathtt{of}\ \mathtt{cc}_1 \mid \cdots \mid \mathtt{cc}_n)$	($\mathtt{case}$-term)
$\mathtt{cc}$	$::=$	$(K \mathtt{->} \mathtt{e})$	($\mathtt{case}$-clause)

Figure 1: Syntax of $\mathcal{H}$

encouraged not to concern themselves with evaluation order, which can be left to the implementation, and to use the property that expressions simply denote values when reasoning about programs. Hence side-effecting I/O is unsuitable for lazy languages: the programmer must be concerned with evaluation order, expressions may denote sequences of side-effects and established implementation techniques may no longer be valid. This paper is concerned only with I/O mechanisms for lazy languages; we leave the development of a theory of side-effecting I/O as an important open problem.

2 $\mathcal{H}$, a small functional language

In this section we define syntax, operational semantics and contextual equality for a small functional language. $\mathcal{H}$ is essentially a core fragment of Haskell; it would be impractical to work with the full language. The focus of this paper is functional I/O and so $\mathcal{H}$ is treated here only briefly; full details and proofs can be found elsewhere [7].

The syntax of $\mathcal{H}$ is given by a BNF grammar in Figure 1. Variables σ and τ range over *types* and variable X over an infinite set of type variables. There is a type `Int` of numbers, together with function types and algebraic types, μ. Each algebraic type is a potentially-recursive sum-of-products, specified by a list of `data`-clauses, $\mathtt{dc}_i$, each of which contains a unique constructor, K, and a list of argument types. For instance, given a type σ, the algebraic type of σ-lists is simply $(\mathtt{data}\, X = \mathtt{Nil} \mid \mathtt{Cons}\, \sigma\, X)$. As in Haskell we call this type $[\sigma]$. Type variables are used to express recursion; X is bound by the `data` construct and occurs free in the `Cons` `data`-clause. We can define `Bool` to be the algebraic type $(\mathtt{data}\, X = \mathtt{False} \mid \mathtt{True})$. Algebraic types of essentially this form are found in Haskell and SML.

Let variable x range over an infinite set of (term) variables, $\mathtt{e}$ range over $\mathcal{H}$ *terms*, and $\mathtt{c}$ range over those terms that are *canonical*. Intuitively, canonical terms represent *values*, the outcomes of computation. The term syntax departs from Haskell in three significant ways. First, a call-by-value function application, $(\mathtt{e}_1 \hat{}\, \mathtt{e}_2)$, is included; call-by-value is not expressible in Haskell but is frequently included in lazy language implementations.[1] Call-by-value is included mainly because it was used in certain programs in the author's dissertation [7], but also for technical reasons explained in the penultimate paragraph of this section. Second, recursive functions or data are constructed using a recursion operator, $\mathtt{rec}(x.\, \mathtt{e})$, rather than recursion equations. Finally, a `case`-term in $\mathcal{H}$ simply discriminates between the different constructors of an algebraic type. The `case`-clauses must exactly match the `data`-clauses in the algebraic type; there is no general pattern-matching as in Haskell. For instance, here is the null-list predicate:

```
\xs -> case xs of Nil -> True | Cons ->(\x -> \xs -> False).
```

We adopt some standard syntactic conventions. We identify syntax up to alpha-conversion; write $\mathtt{e}_1 \equiv \mathtt{e}_2$ iff terms $\mathtt{e}_1$ and $\mathtt{e}_2$ are syntactically identical up to systematic renaming of bound variables. Write $\mathtt{e}_1\{\mathtt{e}_2/x\}$ for the outcome of subtituting $\mathtt{e}_2$ for each free occurrence of x in $\mathtt{e}_1$, with change of bound variables in $\mathtt{e}_1$ as needed to avoid variable capture. A *context*, $\mathcal{C}$, is a "term with one or more holes"; write $\mathcal{C}[\mathtt{e}]$ for the term obtained by filling in each hole in $\mathcal{C}$ with term $\mathtt{e}$.

$\mathcal{H}$ can be given a monomorphic type system. Let an *environment*, Γ, be a finite map from variables to closed types. The type system of $\mathcal{H}$ is a structurally defined *type assignment* relation, consisting of sentences of the form $\Gamma \vdash \mathtt{e} :: \tau$, where τ is a closed type assigned to term $\mathtt{e}$ given environment Γ. It is straightforward to write down structural rules to define this relation [7]; here we omit the details. Terms bear sufficient type information to make type assignments unique.

Let a *program* be a closed well-typed term. The operational semantics of $\mathcal{H}$ is an *evaluation relation*, written $\mathtt{e} \Rightarrow \mathtt{c}$, where $\mathtt{e}$ is a program and $\mathtt{c}$ is a canonical program, given inductively by the evaluation rules in Figure 2. It is convenient in Figure 2 to let variable ℓ range over both numbers $\mathbb{N}$ and truth-values $\{T, F\}$, and to define $\underline{T}$ and $\underline{F}$ to mean `True` and `False` respectively. Hence one can infer $\underline{1} \underline{=} \underline{0} \Rightarrow \mathtt{False}$, for instance.

Evaluation is deterministic and is lazy in the sense that algebraic type constructors do not evaluate their arguments (as in Haskell, but contrary to SML, say). It is conventional to say that a program $\mathtt{e}$ *converges* and to write $\mathtt{e}\Downarrow$ to mean $(\exists \mathtt{c}.\, \mathtt{e} \Rightarrow \mathtt{c})$. Conversely, say that program $\mathtt{e}$ *diverges* and write $\mathtt{e}\Uparrow$ to mean that $\mathtt{e}$ does not converge. It is convenient to have a named divergent program. At each closed type σ, let $\bot^{\sigma}$ be the term $\mathtt{rec}^{\sigma}(x.\, x)$; we have that $\bot^{\sigma}\Uparrow$.

We follow Plotkin's seminal study of the semantics of the typed higher-order functional language PCF [24], and adopt contextual equality as the equality on $\mathcal{H}$ terms:

Definition 1 Contextual order, $\sqsubseteq$, *is the relation such that* $\mathtt{e} \sqsubseteq \mathtt{e}'$ *iff for any context* $\mathcal{C}$ *such that* $\mathcal{C}[\mathtt{e}]$ *and* $\mathcal{C}[\mathtt{e}]$ *are programs of type* `Int`, $\mathcal{C}[\mathtt{e}]\Downarrow$ *implies* $\mathcal{C}[\mathtt{e}']\Downarrow$.

Contextual equality, $=$, *is the relation such that* $\mathtt{e} = \mathtt{e}'$ *iff* $\mathtt{e} \sqsubseteq \mathtt{e}'$ *and* $\mathtt{e}' \sqsubseteq \mathtt{e}$. ■

For the purpose of this paper, to reason about functional I/O specified operationally, it is essential that both contextual order and equality are operationally adequate, where

[1] For instance, $(\mathtt{e}_1 \hat{}\, \mathtt{e}_2)$ is expressible as $(\mathrm{strict}\, \mathtt{e}_1\, \mathtt{e}_2)$ in Mark Jones' Gofer system.

$$\mathtt{c} \Rightarrow \mathtt{c}$$

$$\frac{\mathtt{e}_1 \Rightarrow \underline{\ell_1} \qquad \mathtt{e}_2 \Rightarrow \underline{\ell_2}}{(\mathtt{e}_1 \underline{\oplus} \mathtt{e}_2) \Rightarrow \underline{\ell_1 \oplus \ell_2}} \qquad \frac{\mathtt{e}\{\mathtt{rec}(x.\,\mathtt{e})/x\} \Rightarrow \mathtt{c}}{\mathtt{rec}(x.\,\mathtt{e}) \Rightarrow \mathtt{c}}$$

$$\frac{\mathtt{e}_1 \Rightarrow (\backslash x \mathtt{->} \mathtt{e}_3) \qquad \mathtt{e}_3\{\mathtt{e}_2/x\} \Rightarrow \mathtt{c}}{(\mathtt{e}_1\, \mathtt{e}_2) \Rightarrow \mathtt{c}}$$

$$\frac{\mathtt{e}_1 \Rightarrow (\backslash x \mathtt{->} \mathtt{e}_3) \qquad \mathtt{e}_2 \Rightarrow \mathtt{c}_2 \qquad \mathtt{e}_3\{\mathtt{c}_2/x\} \Rightarrow \mathtt{c}}{(\mathtt{e}_1 \hat{}\, \mathtt{e}_2) \Rightarrow \mathtt{c}}$$

$$\frac{\mathtt{e} \Rightarrow (K^{\mu}\, \mathtt{e}_1 \cdots \mathtt{e}_m) \quad \mathtt{cc}_i \equiv (K \mathtt{->} \mathtt{e}') \quad (\mathtt{e}'\, \mathtt{e}_1 \cdots \mathtt{e}_m) \Rightarrow \mathtt{c}}{(\mathtt{case}\ \mathtt{e}\ \mathtt{of}\ \mathtt{cc}_1 \mid \cdots \mid \mathtt{cc}_n) \Rightarrow \mathtt{c}}$$

Figure 2: Operational semantics of $\mathcal{H}$

a relation $\mathcal{R}$ is *operationally adequate* when it possesses the following properties:

- If $\mathtt{e} \Rightarrow \mathtt{c}$ then $\mathtt{c}\mathcal{R}\mathtt{e}$.
- $\mathtt{e}\Uparrow$ iff $\mathtt{e}\mathcal{R}\bot$.
- $\mathtt{e}\Downarrow$ iff for some canonical $\mathtt{c}$, $\mathtt{c}\mathcal{R}\mathtt{e}$.

For the purpose of supporting established techniques for reasoning about functional programs [2], it is essential that contextual equality and order have a range of properties, including those in the following informal summary:

- Congruence, that contextual order is a substitutive preorder and contextual equality is a substitutive equivalence relation;
- An exhaustion principle, *Strachey's law*, that for any program e there is an equal canonical program, or else that e equals $\bot$;
- Beta and eta laws analogous to those for untyped λ-calculus;
- Strictness laws indicating how $\bot$ propagates through programs;
- *Canonical exclusivity*, that canonical programs c and c′ are equal just when the outermost syntactic constructor of c and its immediate subterms are respectively equal to those of c′.
- Structural induction principles for algebraic types.

For a detailed axiomatisation of such properties for Miranda see Thompson's paper [29], and Bird and Wadler's book for a good introduction to proofs of functional program properties. Note that monotonicity of functions, that $\mathtt{e}_1 \sqsubseteq \mathtt{e}_1$ implies $\mathtt{f}\,\mathtt{e}_1 \sqsubseteq \mathtt{f}\,\mathtt{e}_2$, follows from the substitutivity of contextual order. It is appropriate to refer to the exhaustion principle above as Strachey's law, as it formalises his principle that the "characteristic feature of an expression is its value" [28]. Side-effecting I/O in the style of SML, say, precludes Strachey's law; a program with a side-effect equals neither $\bot$ nor any canonical program. Strachey's law is one aspect of what is generally called "referential transparency" [26].

For the remainder of this paper we will use Haskell notation to denote $\mathcal{H}$ types and programs. One might view $\mathcal{H}$ as a core functional language of about the same level as FLIC [22]; translations from the level of Haskell to FLIC or $\mathcal{H}$ are well-known. We will usually omit type information from terms. We will reason about contextual equality in the informal way exemplified by Bird and Wadler and implicitly by appeal to the properties stated above. Statement and proof of these properties can be found in the author's dissertation [7], where a recent result of Howe's [10] is used to develop Abramsky's *applicative bisimulation* [1], which is an alternative and tractable characterisation of contextual equality for $\mathcal{H}$. In PCF and Haskell, applicative bisimulation would be finer grained than contextual equality, but in $\mathcal{H}$ the two equivalences coincide because of the presence of call-by-value applications and `case`-terms [10]. The standard example is that applicative bisimulation distinguishes $\bot$ and `\x ->` $\bot$. In $\mathcal{H}$, these are contextually distinct too, witness the context $(\backslash \mathtt{f} \mathtt{->} \underline{0})\hat{}[]$ (which produces a term of type `Int`) but in PCF or Haskell (which have no call-by-value applications) the two are contextually indistinguishable.

Of course, the properties above might also be proved via domain-theoretic denotational semantics.

3 Semantics of three I/O mechanisms

We adopt labelled transition systems from the theory of CCS [18] to give semantics for Landin-stream, continuation-passing and synchronised stream I/O. In each of the mechanisms there is a single $\mathcal{H}$ type whose programs can be *executed* to interact with the teletype. A program is *executable* iff it is of this type. For instance, executable programs using Landin-stream I/O are of the stream transformer type `[Char]->[Char]`. Let variables p and q range over executable programs. We formalise the execution of programs as a labelled transition system.

Definition 2 *The set of* actions*, ranged over by α, is produced by the following grammar:*

$$\begin{array}{rcll} \alpha & ::= & n & \textit{(input character } n \in \mathbb{N}) \\ & | & \overline{n} & \textit{(output character } n \in \mathbb{N}) \end{array}$$

A labelled transition system *is a family of binary relations indexed by actions, $\{\xrightarrow{\alpha} \mid \alpha \text{ is an action}\}$, such that if $\mathtt{p} \xrightarrow{\alpha} \mathtt{q}$ then p and q are executable programs.*

The intuitive meaning of transition $\mathtt{p} \xrightarrow{n} \mathtt{q}$ is that program p can input the character n from the keyboard to become program q. Similarly, the intuitive meaning of transition $\mathtt{p} \xrightarrow{\overline{n}} \mathtt{q}$ is that program p can output the character n to the printer to become program q. For the sake of simplicity, define the type of characters, `Char`, to be `Int`.

We begin with a semantics of continuation-passing I/O in §3.1. In §3.2 we introduce combinators for programming both kinds of stream-based I/O and use them in §3.3 to give semantics to synchronised-streams. In §3.4 we show that a semantics of Landin-stream I/O cannot be based directly on the operational semantics of $\mathcal{H}$, essentially because we cannot test whether a function examines the value of its argument. Our solution to this problem is to add to $\mathcal{H}$ a simple exception mechanism, in §3.5, to yield $\mathcal{HX}$. Finally, in §3.6 we give a semantics for Landin-stream I/O based on $\mathcal{HX}$.

3.1 Continuation-passing I/O

In *continuation-passing I/O*, the executable type is an algebraic type with a constructor corresponding to each kind of expressible imperative activity. In the case of teletype I/O we have:

```
data CPS = INPUT (Char -> CPS)
         | OUTPUT Char CPS
         | DONE
```

PFL [9] was the first functional language to take the continuation-passing mechanism as primitive. In earlier work, Karlsson programmed continuation-passing operations on top of a synchronised-stream mechanism [15]. A similar datatype was used by Plotkin in the Pisa notes [25] as semantics for side-effecting I/O. Several languages, such as Perry's Hope+C [21], use continuation-passing I/O. The mechanism is so-called because of the similarity between the argument to `INPUT` and continuations as used in denotational semantics.

The intended meaning of `CPS`-programs is easily given.

- `INPUT k` is to mean "input a character n from the keyboard and then execute ($\mathtt{k}\,\underline{n}$)."
- `OUTPUT` $\underline{n}$ `p` is to mean "output character n to the printer and then execute `p`."
- `DONE` is to mean "terminate immediately."

These intended meanings are reflected in the following two rules, which together define a labelled transition system for `CPS`-programs.

$$\frac{\mathtt{p} \Rightarrow \mathtt{INPUT\ k}}{\mathtt{p} \xrightarrow{n} \mathtt{k}\,\underline{n}} \qquad \frac{\mathtt{p} \Rightarrow \mathtt{OUTPUT\,v\,q} \qquad \mathtt{v} \Rightarrow \underline{n}}{\mathtt{p} \xrightarrow{\overline{n}} \mathtt{q}}$$

3.2 Stream transformers

The two remaining I/O mechanisms, synchronised-stream and Landin-stream I/O, are based on stream transformers. A stream is a list type whose cons operation is lazy, such as $[\sigma]$ in $\mathcal{H}$. Stream transformers in $\mathcal{H}$ have the general type:

```
type ST inp out = [inp] -> [out]
```

The idea is simple: a stream transformer maps a stream of values of type `inp` into a stream of values of type `out`. This mapping represents an interactive computing device that consumes values of type `inp` and produces values of type `out`. Intuitively, if the device has been offered the sequence of values $\mathtt{in}_1, \ldots, \mathtt{in}_n$ for consumption, applying the stream transformer to the stream $(\mathtt{in}_1 : \ldots : \mathtt{in}_n : \bot)$ yields a stream containing the sequence of values the device can produce. The list cons operation, :, has to have lazy semantics so that the partial list $(\mathtt{in}_1 : \ldots : \mathtt{in}_n : \bot)$ does not simply equal $\bot$. Implementations of stream-based I/O [14] typically represent the undefined value at the end of a partial list as a memory cell that can be instantiated to hold the next input character and to point to a fresh undefined value. Such a technique is intuitively correct, but we leave open the question of how to verify formally that it correctly implements the semantics to be given here.

Stream transformers for stream-based I/O have typically been written using explicit construction of the output list and explicit examination of the input list [8, 14]. Such a programming style can be hard to read. We can avoid explicit mention of input and output lists by using the following combinators to construct stream transformers:

```
getST :: (inp -> ST inp out) -> ST inp out
putST :: out -> ST inp out -> ST inp out
nilST :: ST inp out

getST k xs = case xs of (x:xs') -> k x xs'
putST x f xs = x : f xs
nilST xs = []
```

A programmer can use the combinators above to construct stream transformers; to give semantics to stream-based I/O we use combinators `giveST`, `nextST` and `skipST`. The intention is that `giveST` feeds an input value to a stream transformer, `nextST` tests whether a stream transformer can produce an output value without any further input, and `skipST` consumes an output value from a stream transformer.

```
data Maybe a = Yes a | No

giveST :: inp -> ST inp out -> ST inp out
nextST :: (ST inp out) -> Maybe out
skipST :: ST inp out -> ST inp out

giveST c f xs = f (c:xs)
nextST f      = case f ⊥ of
                    [] ->     No
                    (x:xs) -> Yes x
skipST f xs   = tail(f xs)
```

The technique of using a mock argument $\bot$ to test whether a stream transformer is ready to produce output was discovered by the Haskell committee [12, 23]. Of course, if the next output from a stream transformer `f` depends on the next value in its input stream, then `nextST f` will loop.

The following proposition relates the six combinators introduced in this section.

Proposition 1 *For all suitably-typed programs* `u`, `v`, `k`, `f`*:*

(1) (`giveST u (getST k)`) = (`k u`)

(2) (`nextST(putST v f)`) $\Rightarrow$ (`Yes v`)

(3) (`nextST(nilST)`) $\Rightarrow$ `No`

(4) (`skipST(putST v f)`) = `f`

Proof. Straightforward calculations. ∎

3.3 Synchronised-stream I/O

In *synchronised-stream I/O*, the stream transformer produces a stream of requests and consumes a stream of acknowledgements. The requests and acknowledgements are in one-to-one correspondence: the computing device specified by a stream transformer alternates between producing an output request and consuming an input acknowledgement. It is the programmer's burden to ensure that the value of each request does not depend on the corresponding acknowledgement. Synchronised-stream I/O was first reported as the underlying implementation technique for Karlsson's Nebula operating system [15]. It was independently discovered by Stoye [27], and O'Donnell [20]. It is the mechanism underlying KAOS [4, 31] and Haskell I/O [12, 11] (where the mechanism is named a *dialogue*).

Here is the type SS of executable programs in the setting of teletype I/O, together with intended meanings of some example programs:

```
type SS = ST Ack Req
data Req = Get | Put Char
data Ack = Got Char | Did
```

- `putST Get (getST k)` is to mean "input a character n from the keyboard and then execute (`k` (`Got` $\underline{n}$))."
- `putST (Put` $\underline{n}$`) (getST k)` is to mean "output character n to the printer and then execute (`k Did`)."
- `nilST` is to mean "terminate immediately."

A wide range of imperative activity can be expressed using this mechanism—witness Haskell I/O. We define an auxiliary function for use in examining the acknowledgement obtained from a `Get` request:

```
outGot :: Ack -> Char
outGot (Got x) = x
```

The semantics of synchronised-streams can be given for SS-programs in $\mathcal{H}$ as the labelled transition system inductively defined by the following two rules:

$$\frac{\mathtt{nextST\ f} \Rightarrow \mathtt{Yes\ r} \qquad \mathtt{r} \Rightarrow \mathtt{Get}}{\mathtt{f} \xrightarrow{n} \mathtt{giveST(Got}\ \underline{n}\mathtt{)(skipST\ f)}}$$

$$\frac{\mathtt{nextST\ f} \Rightarrow \mathtt{Yes\ r} \qquad \mathtt{r} \Rightarrow \mathtt{Put\ v} \qquad \mathtt{v} \Rightarrow \underline{n}}{\mathtt{f} \xrightarrow{\overline{n}} \mathtt{giveST\ Did\ (skipST\ f)}}$$

We state a lemma to show that this formal semantics correctly reflects the informal intended meanings given for synchronised-stream programs—apart from termination.

Lemma 2 *Suppose* `k::Char -> SS` *and* `h::SS` *are programs. Define programs* `f` *and* `g` *to be:*

$$\begin{aligned} \mathtt{f} &= \mathtt{putST\ Get\ (getST\ k)} \\ \mathtt{g} &= \mathtt{putST\ (Put\ v)\ (getST\ k)} \end{aligned}$$

Then we have:

(1) $\mathtt{nextST\ f} \Rightarrow \mathtt{Yes\ Get}$

(2) $\mathtt{nextST\ g} \Rightarrow \mathtt{Yes(Put\ v)}$

(3) $\mathtt{f} \xrightarrow{n}\!\!= \mathtt{k\ (Got}\ \underline{n}\mathtt{)}$

(4) $\mathtt{g} \xrightarrow{\overline{n}}\!\!= \mathtt{k\ Did}$ *if* $\mathtt{v} \Rightarrow \underline{n}$.

(The juxtaposition $\xrightarrow{\overline{n}}\!\!=$ *denotes the composition of relations* $\xrightarrow{\overline{n}}$ *and* $=$*.)*

Proof. Parts (1) and (2) follow from the definitions of `nextST`, `putST` and `getST`. For parts (3) and (4), we can calculate the following transitions and equations using Proposition 1:

$$\begin{aligned} \mathtt{f} &\xrightarrow{n} \mathtt{giveST(Got}\ \underline{n}\mathtt{)(skipST\ f)} \\ &= \mathtt{giveST(Got}\ \underline{n}\mathtt{)(getST\ k)} \\ &= \mathtt{k\ (Got}\ \underline{n}\mathtt{)} \\ \mathtt{g} &\xrightarrow{\overline{n}} \mathtt{giveST\ Did\ (skipST\ g)} \\ &= \mathtt{giveST\ Did\ (getST\ k)} \\ &= \mathtt{k\ Did} \qquad \blacksquare \end{aligned}$$

3.4 Landin-stream I/O and $\mathcal{H}$

The simplest kind of stream transformer used for I/O is one that maps a stream of input characters to a stream of output characters. We call such a mechanism *Landin-stream I/O* in honour of Landin [16], who suggested that streams "would be used to model input/output if ALGOL 60 included such." Henderson [8] was the first implementor of character-based I/O based on Landin-stream I/O. Executable programs are stream transformers of type `LS`, with the following intended meanings:

```
type LS = ST Char Char
```

- `getST k` is to mean "input a character n from the keyboard and then execute (`k` $\underline{n}$)."
- `putST` $\underline{n}$ `f` is to mean "output character n to the printer and then execute `f`."
- `nilST` is to mean "terminate immediately."

We wish to implement this intended meaning using the operational semantics of $\mathcal{H}$. Given a function `f::LS` we are to compute whether `f` can output a character with no further input, or whether `f` needs an input character before producing more output, or whether `f` can terminate. More precisely, we need a function `ready` of the following type

```
data RWD out = R | W out | D
ready :: ST inp out -> RWD out
```

and satisfying the equations:

$$\begin{aligned} \mathtt{ready(putST}\ \underline{n}\ \mathtt{f)} &= \mathtt{W}\ \underline{n} \\ \mathtt{ready(getST\ k)} &= \mathtt{R} \\ \mathtt{ready(nilST)} &= \mathtt{D} \end{aligned}$$

We show that in $\mathcal{H}$ there is no such program. Consider programs `e1` and `e2` of type `LS`:

```
e1 = getST (\x -> putST 205 nilST)
e2 = putST 205 nilST
```

It is not hard to see that for any `xs` the following equations hold:

$$\begin{aligned} \mathtt{e1\ xs} &= \mathtt{(case\ xs\ of\ (x : xs') \text{->} [}\underline{205}\mathtt{])} \\ \mathtt{e2\ xs} &= \mathtt{[}\underline{205}\mathtt{]} \end{aligned}$$

and hence that $\mathtt{e1} \sqsubseteq \mathtt{e2}$ and $\mathtt{e2} \not\sqsubseteq \mathtt{e1}$ by Strachey's law. To see why there can be no function `ready` that obeys the equations shown above, we assume there is and derive a contradiction. We have $\mathtt{ready(e1)} = \mathtt{R}$ and $\mathtt{ready(e2)} = (\mathtt{W}\ \underline{205})$, and $\mathtt{R} \not\sqsubseteq (\mathtt{W}\ \underline{205})$. But $\mathtt{e1} \sqsubseteq \mathtt{e2}$ so by monotonicity we have $\mathtt{ready(e1)} \sqsubseteq \mathtt{ready(e2)}$. Contradiction.[2]

Intuitively, the problem is that in $\mathcal{H}$ there is no way to tell whether a term depends on the value of one of its subterms, such as an element of the input stream. In the next section we remedy this by adding an exception mechanism to $\mathcal{H}$.

[2] John Hughes showed me this argument in 1988.

$$\mathcal{EC} ::= ([]\underline{\oplus}\mathsf{e}) \mid ([]\,\mathsf{e}) \mid ([]\,\hat{}\,\mathsf{e}) \mid (\mathtt{case}\ []\ \mathtt{of}\ \mathsf{cc}_1\ \mid \cdots \mid \mathsf{cc}_n)$$

$$\frac{\mathsf{e} \Rightarrow \mathtt{bang}^{\sigma} \qquad \mathcal{EC}[\mathsf{e}] :: \tau}{\mathcal{EC}[\mathsf{e}] \Rightarrow \mathtt{bang}^{\tau}}$$

$$\frac{\mathsf{e}_1 \Rightarrow \underline{\ell} \qquad \mathsf{e}_2 \Rightarrow \mathtt{bang}^{\sigma} \qquad (\mathsf{e}_1\underline{\oplus}\mathsf{e}_2)::\tau}{(\mathsf{e}_1\underline{\oplus}\mathsf{e}_2) \Rightarrow \mathtt{bang}^{\tau}}$$

$$\frac{\mathsf{e}_1 \Rightarrow (\backslash x \mathbin{\texttt{->}} \mathsf{e}) \qquad \mathsf{e}_2 \Rightarrow \mathtt{bang}^{\sigma} \qquad (\mathsf{e}_1\,\hat{}\,\mathsf{e}_2)::\tau}{(\mathsf{e}_1\,\hat{}\,\mathsf{e}_2) \Rightarrow \mathtt{bang}^{\tau}}$$

$$\frac{\mathsf{e}_1 \Rightarrow \mathsf{c} \qquad \mathit{Mute}(\mathsf{c})}{(\mathsf{e}_1\ \mathtt{??}\ \mathsf{e}_2) \Rightarrow \mathsf{c}} \qquad \frac{\mathsf{e}_1 \Rightarrow \mathtt{bang}^{\sigma} \qquad \mathsf{e}_2 \Rightarrow \mathsf{c}}{(\mathsf{e}_1\ \mathtt{??}\ \mathsf{e}_2) \Rightarrow \mathsf{c}}$$

Figure 3: Evaluation in $\mathcal{HX}$

3.5 $\mathcal{HX}$: $\mathcal{H}$ plus one exception

The exception mechanism in SML, say, has the following property: during evaluation of a term, if demand arises for a subterm which evaluates to an exception, then the exception propagates to the outermost level, unless a handler intervenes. Adding such an exception mechanism is a way to modify evaluation in $\mathcal{H}$ to formalise the notion of demand for the next character in a stream. Another motivation is to obtain a fully abstract denotational semantics; this is the purpose of Cartwright and Felleisen's recent extension of PCF with exceptions [3].

We consider a language $\mathcal{HX}$ obtained from $\mathcal{H}$ by adding just one exception, the canonical term `bang`. Raising an exception is represented by a program evaluating to `bang`, which is present at every type. For the sake of brevity, we say the program has *banged*. Program `bang` bangs. In general, if a program needs to evaluate several subterms before terminating, and evaluation of any one of the subterms bangs, then the whole program bangs. The only exemptions from this rule are programs of the form (e_1 `??` e_2). If evaluation of e_1 returns an answer or diverges, then evaluation of the whole program does so too. But if evaluation of e_1 bangs, then the whole program behaves the same as e_2.

To obtain $\mathcal{HX}$ from $\mathcal{H}$, we add new canonical terms, $\mathtt{bang}^{\sigma}$, and non-canonical terms (e_1 `??` e_2), subject to the following typing rules:

$$\Gamma \vdash \mathtt{bang}^{\sigma} :: \sigma \qquad \frac{\Gamma \vdash \mathsf{e}_1 :: \sigma \qquad \Gamma \vdash \mathsf{e}_2 :: \sigma}{\Gamma \vdash (\mathsf{e}_1\ \mathtt{??}\ \mathsf{e}_2) :: \sigma}$$

The rest of the syntax and typing system of $\mathcal{H}$ is as before. Define the predicate $\mathit{Mute}(\mathsf{c})$ on canonical terms to hold iff for no type τ does $\mathsf{c} \equiv \mathtt{bang}^{\tau}$. The evaluation relation for $\mathcal{HX}$ is the binary relation on $\mathcal{HX}$ programs, $\Rightarrow$, defined inductively by the evaluation rules from Figures 2 and 3. The rule for call-by-value evaluation in Figure 2 is modified to apply only when $\mathit{Mute}(\mathsf{c}_2)$. Contextual order and equality are defined as before. The same symbols $\Rightarrow$, $\sqsubseteq$ and $=$ are used to denote relations in both $\mathcal{H}$ or $\mathcal{HX}$, and are labelled with the language name when necessary.

A similar theory to the one sketched for $\mathcal{H}$ can be derived for $\mathcal{HX}$. In particular, Strachey's law still applies, although now there is an additional canonical program, `bang`, at each type. For instance, at the type `Int` we have that every program either equals $\perp$, $\underline{n}$ for some n, or `bang`. Program $\perp$ is less than the others in contextual order, $\sqsubseteq$. The others are mutually incomparable, because they are distinct and canonical. Details can be found elsewhere [7]. The two languages can be compared as follows.

Proposition 3 *Let $\mathcal{H}^0$ and $\mathcal{HX}^0$ be the sets of programs in $\mathcal{H}$ and $\mathcal{HX}$ respectively.*

(1) $\mathcal{H}^0 \subset \mathcal{HX}^0$

(2) *If* $\mathsf{e} \in \mathcal{H}^0$ *and* $\mathsf{e} \Rightarrow^{\mathcal{HX}} \mathsf{c}$ *then* $\mathsf{c} \in \mathcal{H}^0$.

(3) *For any* $\mathsf{e}, \mathsf{c} \in \mathcal{H}^0$, $\mathsf{e} \Rightarrow^{\mathcal{HX}} \mathsf{c}$ *iff* $\mathsf{e} \Rightarrow^{\mathcal{H}} \mathsf{c}$.

(4) *For any* $\mathsf{e}, \mathsf{e}' \in \mathcal{H}^0$, $\mathsf{e} \sqsubseteq^{\mathcal{HX}} \mathsf{e}'$ *implies* $\mathsf{e} \sqsubseteq^{\mathcal{H}} \mathsf{e}'$.

(5) *For any* $\mathsf{e}, \mathsf{e}' \in \mathcal{H}^0$, $\mathsf{e} =^{\mathcal{HX}} \mathsf{e}'$ *implies* $\mathsf{e} =^{\mathcal{H}} \mathsf{e}'$.

(6) *There are* $\mathsf{e}, \mathsf{e}' \in \mathcal{H}^0$ *with* $\mathsf{e} \sqsubseteq^{\mathcal{H}} \mathsf{e}'$ *but not* $\mathsf{e} \sqsubseteq^{\mathcal{HX}} \mathsf{e}'$.

(7) *There are* $\mathsf{e}, \mathsf{e}' \in \mathcal{H}^0$ *with* $\mathsf{e} =^{\mathcal{H}} \mathsf{e}'$ *but not* $\mathsf{e} =^{\mathcal{HX}} \mathsf{e}'$.

Proof. Part (1) follows by definition. Parts (2) and (3) follow by induction on depth of inference. Parts (4) and (5) follow from the definition of contextual order and equality; any $\mathcal{H}$ context is also an $\mathcal{HX}$ context. For parts (6) and (7) consider $\mathsf{e} \equiv (\backslash\mathtt{x} \mathbin{\texttt{->}} \mathtt{x}\underline{\times}\perp)$ and $\mathsf{e}' \equiv (\backslash\mathtt{x} \mathbin{\texttt{->}} \perp)$. We have $\mathsf{e} =^{\mathcal{H}} \mathsf{e}'$ but not $\mathsf{e} \sqsubseteq^{\mathcal{HX}} \mathsf{e}'$ (consider the context ([] `bang`)). ■

In the remainder of this paper we work with $\mathcal{HX}$ instead of $\mathcal{H}$. The only reason we do so is to model demand for a lazy input stream. Parts (3)–(5) of the proposition assure us that any $\mathcal{HX}$ evaluation, order or equality deduced about $\mathcal{H}$ programs in fact implies the corresponding $\mathcal{H}$ property. Parts (6) and (7) indicate that contextual order and equality in $\mathcal{HX}$ are finer grained than in $\mathcal{H}$, intuitively because an exception can detect whether a function examines its argument.

3.6 Landin-stream I/O and $\mathcal{HX}$

Given $\mathcal{HX}$, we can define an operational semantics for Landin-stream I/O. First, we find that the argument that there can be no function `ready` in $\mathcal{H}$ does not hold in $\mathcal{HX}$. In $\mathcal{HX}$ we have that the programs `e1` and `e2` are incomparable, because `e1(bang)` = `bang`, `e2(bang)` = `[`$\underline{205}$`]`, and `bang` and `[`$\underline{205}$`]` are incomparable.

Intuitively, to tell in $\mathcal{HX}$ whether a term depends on the value of one of its subterms, replace the subterm with `bang` and use the handler operator `??` to see if the whole term bangs. We can define `ready` in $\mathcal{HX}$ as follows

```
ready f =
    (case (f bang) of
        [] -> D
        (x:_) -> W x)
    ?? R
```

and one can calculate that the conditions on `ready` given in §3.4 are satisfied. The semantics of Landin-streams can be given for LS-programs in $\mathcal{HX}$ as the labelled transition system inductively defined by the following two rules:

$$\frac{\mathtt{ready\ f} \Rightarrow \mathtt{R}}{\mathtt{f} \xrightarrow{n} \mathtt{giveST}\ \underline{n}\ \mathtt{f}} \qquad \frac{\mathtt{ready\ f} \Rightarrow \mathtt{W\ v} \qquad \mathtt{v} \Rightarrow \underline{n}}{\mathtt{f} \xrightarrow{\overline{n}} \mathtt{skipST\ f}}$$

The following lemma shows that this formal semantics correctly reflects the informal intended meanings given for Landin-stream programs—apart from termination, which we have not formalised.

Lemma 4

(1) $\mathtt{ready(getST\ k)} \Rightarrow \mathtt{R}$

(2) $\mathtt{ready(putST\ v\ k)} \Rightarrow \mathtt{W\ v}$

(3) $\mathtt{getST\ k} \xrightarrow{n}= (\mathtt{k}\ \underline{n})$

(4) $\mathtt{putST\ v\ p} \xrightarrow{\overline{n}}= \mathtt{p}$ *if* $\mathtt{v} \Rightarrow \underline{n}$

Proof. Parts (1) and (2) follow from the definitions of `ready`, `getST` and `putST`. For parts (3) and (4), we can calculate the following transitions:

$$\begin{array}{rcl} \mathtt{getST\ k} & \xrightarrow{n} & \mathtt{giveST}\ \underline{n}\ (\mathtt{getST\ k}) \\ \mathtt{putST\ v\ p} & \xrightarrow{\overline{n}} & \mathtt{skipST}\ (\mathtt{putST\ v\ p}) \end{array}$$

These, together with Proposition 1 establish the required results. ■

4 Bisimilarity of programs engaged in I/O

Following Holmström's method [9], we have given labelled transition semantics for continuation-passing and synchronised-stream I/O based on $\mathcal{H}$, and for Landin-stream I/O based on $\mathcal{HX}$. In this section we adopt (strong) bisimilarity from CCS [18] as a characterisation of identical I/O behaviour. Unless otherwise stated, the evaluation, contextual order and equality relations are those of $\mathcal{HX}$.

Definition 3 *Define function* $\langle\cdot\rangle$ *to be the function over binary relations on* $\mathcal{HX}$ *programs such that* $\mathtt{p}\langle\mathcal{S}\rangle\mathtt{q}$ *iff*

(1) *whenever* $\mathtt{p} \xrightarrow{\alpha} \mathtt{p}'$ *there is* $\mathtt{q}'$ *with* $\mathtt{q} \xrightarrow{\alpha} \mathtt{q}'$ *and* $\mathtt{p}'\mathcal{S}\mathtt{q}'$;

(2) *whenever* $\mathtt{q} \xrightarrow{\alpha} \mathtt{q}'$ *there is* $\mathtt{p}'$ *with* $\mathtt{p} \xrightarrow{\alpha} \mathtt{p}'$ *and* $\mathtt{p}'\mathcal{S}\mathtt{q}'$.

A bisimulation *is a binary relation on programs,* $\mathcal{S}$, *such that* $\mathcal{S} \subseteq \langle\mathcal{S}\rangle$. Bisimilarity, $\sim$, *is the union of all bisimulations.*

Proposition 5

(1) *Function* $\langle\cdot\rangle$ *is monotonic.*

(2) *Bisimilarity is the greatest fixed-point of* $\langle\cdot\rangle$ *and is the greatest bisimulation.*

(3) $\mathtt{p} \sim \mathtt{q}$ *iff there is a bisimulation* $\mathcal{S}$ *such that* $\mathtt{p}\mathcal{S}\mathtt{q}$.

(4) *Bisimilarity is an equivalence relation.*

Proof. Part (1) follows easily from the definition. (2) follows from the Knaster-Tarski theorem from fixed-point theory [5]. (3) For the forwards direction, take the bisimulation $\mathcal{S}$ to be $\sim$ itself. For the backwards direction, we have $\mathcal{S} \subseteq \sim$, so $\mathtt{p}\mathcal{S}\mathtt{q}$ implies $\mathtt{p} \sim \mathtt{q}$. Part (4) is straightforward (see Milner's book for details). ■

This definition of bisimulation equivalence is very simple, but for two reasons one might wish to develop it further. First, although each of the three I/O mechanisms has a notion of program termination we have not modelled termination in the labelled transition system. Hence a program that immediately terminates is bisimilar to one that diverges. Second, we have assumed that teletype input is observable. Consider two Landin-stream programs `f` and `g`:

```
f xs = ⊥
g xs = case xs of
         [] -> ⊥
         (_:xs) -> g xs
```

Given an input stream, `g` unravels it forever whereas `f` loops immediately. We have $\mathtt{f} =^{\mathcal{H}} \mathtt{g}$ but $\mathtt{f} \neq^{\mathcal{HX}} \mathtt{g}$ and $\mathtt{f} \not\sim \mathtt{g}$ (because `g` forever inputs characters whereas `f` diverges). One might argue that they have indistinguishable behaviour because neither ever produces output. On the other hand, it seems reasonable to distinguish them on the ground that teletype input is observable to the operating system, if not always to the end user.

4.1 Bisimilarity strictly contains contextual equality

First we prove that contextual equality (in $\mathcal{HX}$) is a subset of bisimilarity.

Proposition 6 *For any* $\mathtt{p}$ *and* $\mathtt{q}$, $\mathtt{p} = \mathtt{q}$ *implies* $\mathtt{p} \sim \mathtt{q}$.

Proof. By Proposition 5(3) it suffices to show that the relation of contextual equality on programs is a bisimulation. We have to show that for any programs $\mathtt{p}$ and $\mathtt{q}$, $\mathtt{p} = \mathtt{q}$ implies that $\mathtt{p}\langle=\rangle\mathtt{q}$, which is to say:

(1) whenever $\mathtt{p} \xrightarrow{\alpha} \mathtt{p}'$ there is $\mathtt{q}'$ with $\mathtt{q} \xrightarrow{\alpha} \mathtt{q}'$ and $\mathtt{p}' = \mathtt{q}'$;

(2) whenever $\mathtt{q} \xrightarrow{\alpha} \mathtt{q}'$ there is $\mathtt{p}'$ with $\mathtt{p} \xrightarrow{\alpha} \mathtt{p}'$ and $\mathtt{p}' = \mathtt{q}'$.

For (1), suppose that $\mathtt{p} \xrightarrow{\alpha} \mathtt{p}'$, and proceed by analysis of the six rules by which this inference can be derived. We show the details of the CPS rule for input.

Suppose that $\mathtt{p} \Rightarrow \mathtt{INPUT\ k}$, $\alpha = n$, and hence that $\mathtt{p}' \equiv \mathtt{k}\,\underline{n}$. By operational adequacy we have $\mathtt{p} = \mathtt{INPUT\ k}$ and also that $\mathtt{q}\Downarrow$. Then by operational adequacy and canonical exclusivity, there is a program $\mathtt{k}'$ such that $\mathtt{q} = \mathtt{INPUT\ k}'$ and $\mathtt{k} = \mathtt{k}'$. By the CPS input rule we have that $\mathtt{q} \xrightarrow{\alpha} \mathtt{k}'\,\underline{n}$ and that $\mathtt{k}'\,\underline{n} = \mathtt{p}'$ as required.

Examination of the other rules follows a similar pattern to prove (1), and then (2) follows by symmetry. ■

The force of this result is that the theory of contextual equality can be used to prove properties of the execution behaviour of executable programs. The proof makes essential appeal to operational adequacy.

Second, we have that bisimilarity does not imply contextual equality.

Proposition 7 *There are program pairs,* $\mathtt{p}$ *and* $\mathtt{q}$, *in each of the types* `CPS`, `LS` *and* `SS` *such that* $\mathtt{p} \sim \mathtt{q}$ *but not* $\mathtt{p} = \mathtt{q}$.

Proof. Witness program pair $\mathtt{Write} \perp \mathtt{Done}$ and $\perp$ in type `CPS`, and pair $\mathtt{putST} \perp \mathtt{nilST}$ and $\perp$ in each of the types `SS` and `LS`. ■

Intuitively the proof depends on contextual equality distinguishing more "junk" programs than bisimilarity. Given a richer I/O model there would be more significant distinctions. Suppose we extended the `CPS` algebraic type with a new constructor `Par::CPS -> CPS -> CPS`, with intended meaning that `Par p q` is to be the parallel execution of programs `p` and `q`, as in PFL. Then if $\mathtt{p} \neq \mathtt{q}$, programs `Par p q` and `Par q p` would be contextually unequal (because `Par` is the constructor of an algebraic type) but bisimilar (because as in CCS both lead to the parallel execution of `p` and `q`).

```
ss2cps f = case nextST f of
  No -> DONE
  Yes r -> case r of
    Get ->
      INPUT (\c ->
        ss2cps (giveST (Got c) (skipST f)))
    Put v ->
      OUTPUT v (ss2cps (giveST Did (skipST f)))

cps2ss p = case p of
  INPUT k ->
    putST Get (getST (\ack ->
      cps2ss (k (outGot ack))))
  OUTPUT c q ->
    putST (Put c) (getST (\ack -> cps2ss q))
  DONE -> nilST
```

Figure 4: Translation between SS and CPS in $\mathcal{HX}$ (and $\mathcal{H}$)

4.2 Bisimilarity coincides with trace equivalence

Given its simple sequential nature, one would expect the semantics of teletype I/O to be determinate. The following result makes this precise.

Proposition 8 *For any program* p, $\mathtt{p} \xrightarrow{\alpha} \mathtt{p}'$ *and* $\mathtt{p} \xrightarrow{\alpha} \mathtt{p}''$ *implies* $\mathtt{p}' \equiv \mathtt{p}''$.

Proof. By inspection of each of the inference rules. ■

Given this determinacy, bisimilarity can alternatively be characterised in terms of traces. If $s = \alpha_1, \ldots, \alpha_n$ is a finite sequence of actions, say that s is a *trace* of program p iff there are programs $\mathtt{p}_i$ with $\mathtt{p} \xrightarrow{\alpha_1} \mathtt{p}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} \mathtt{p}_n$. Two programs are *trace equivalent* iff they have the same set of traces.

In a nondeterministic calculus like CCS, trace equivalence does not in general imply bisimilarity. Given the determinacy result above, however, it is not hard to show that the two equivalences coincide. We omit the proof, but see Milner's book for a more general result [18, Chapter 9].

5 Translation between the three mechanisms

We show that each of the three mechanisms has equivalent expressive power in the following sense. If p is an executable program with respect to one mechanism, then for each other mechanism, there is a function f such that f(p) is an executable program with respect to the other mechanism, and p and f(p) are bisimilar.

We show in Figure 4 functions ss2cps and cps2ss to map between the types SS and CPS, and in Figure 5 functions ls2cps and cps2ls to map between the types LS and CPS. The main result of the paper is that the three I/O mechanisms are equivalent in the following sense.

Proposition 9

(1) *For any* SS*-program* f, f $\sim$ (ss2cps f).

(2) *For any* CPS*-program* p, p $\sim$ (cps2ss p).

(3) *For any* LS*-program* f, f $\sim$ (ls2cps f).

(4) *For any* CPS*-program* p, p $\sim$ (cps2ls p).

```
ls2cps f =
    case ready f of
        R ->   INPUT (\c -> ls2cps (giveST c f))
        W c -> OUTPUT c (ls2cps (skipST f))
        D ->   DONE

cps2ls p =
    case p of
        INPUT k ->     getST (\c -> cps2ls (k c))
        OUTPUT c q ->  putST c (cps2ls q)
        DONE ->        nilST
```

Figure 5: Translation between LS and CPS in $\mathcal{HX}$

Proof. We only prove part (1); the other parts follow by similar arguments [7]. It suffices to show that relation $\mathcal{S}$ below is a bisimulation.

$$\mathcal{S} \stackrel{\text{def}}{=} \{(\mathtt{f}, \mathtt{ss2cps\ f}) \mid \mathtt{f} \text{ is an SS-program}\}$$

We are to show that $\mathcal{S} \subseteq \langle\mathcal{S}\rangle$. Let f be any SS-program and we have that (ss2cps f)::CPS. Hence the synchronised-stream rules apply to f and the continuation-passing rules to (ss2cps f). We are to show that (f, ss2cps f) $\in \langle\mathcal{S}\rangle$. We proceed by analysis of the evaluation behaviour of (nextST f). There are five cases to consider.

(1) (nextST f)$\Uparrow$ or (nextST f) $\Rightarrow$ bang

(2) (nextST f) $\Rightarrow$ No

(3) (nextST f) $\Rightarrow$ Yes r and either r$\Uparrow$ or r $\Rightarrow$ bang

(4) (nextST f) $\Rightarrow$ Yes r and r $\Rightarrow$ Get

(5) (nextST f) $\Rightarrow$ Yes r and r $\Rightarrow$ Put v

Here are the possible transitions from f and (ss2cps f).

(1,2,3) There are no transitions from either (ss2cps f) or f.

(4) The only transitions of f are of the form f $\xrightarrow{n}$ (giveST (Got $\underline{n}$) (skipST f)) for any n.
We have (ss2cps f) evaluates to
INPUT(\c -> ss2cps(giveST (Got c) (skipST f))).
So the only transitions of (ss2cps f) are of the form (ss2cps f) $\xrightarrow{n}$ ss2cps(giveST (Got $\underline{n}$) (skipST f)) for any n.

(5) There is no transition from f unless v $\Rightarrow$ $\underline{n}$, when f $\xrightarrow{\overline{n}}$ (giveST Did (skipST f)).
We have (ss2cps f) evaluates to
OUTPUT v (ss2cps(giveST Did (skipST f))).
So there is no transition from (ss2cps f) unless v $\Rightarrow$ $\underline{n}$, when (ss2cps f) $\xrightarrow{\overline{n}}$ (ss2cps(giveST Did (skipST f))).

One can see that in each case the conditions for (f, ss2cps f) $\in \langle\mathcal{S}\rangle$ are satisfied, so part (1) follows from Proposition 5(3). ■

A simple corollary of this proposition is that each of the four translation functions is a bijection, up to $\sim$, and hence that the three types are in bijection, up to $\sim$. The point of the proposition is that any one of the three mechanisms can be taken as primitive, and execution of a program using one of the other mechanisms can be simulated by its translation into the primitive mechanism. The Haskell committee discovered the `ss2cps` and `cps2ss` translations and chose to make synchronised-streams primitive because no efficient implementation of `ss2cps` was known [23].

One might wonder whether a similar result could be proved for contextual equality instead of bisimilarity. We can show that none of the mapping functions is bijective up to contextual equality, and so the given translations do not establish bijections between the three types.

Proposition 10 *Neither* `ss2cps` *nor* `ls2cps` *is injective, and neither* `cps2ss` *nor* `cps2ls` *is surjective up to contextual equality.*

Proof. (`ss2cps`) Witness `f` $\equiv$ `getST(\x -> doneST)` and `g` $\equiv$ `getST(\x -> f)` of type `SS`. Both these programs examine the first acknowledgement before producing a request, and hence both are mapped to $\perp$ by `ss2cps`. Since `g` examines two elements of the input stream, whereas `f` only examines one, the two are not contextually equal. Hence `ss2cps` is not injective.

(`ls2cps`) Define $\mathtt{h}_i$ to be a family of `LS`-programs indexed by the character i given by

`\xs -> case xs of Nil -> [`$\underline{i}$`] | Cons -> `$\perp$.

For each i we have `ls2cps`$(\mathtt{h}_i)$ = `Input(\c -> `$\perp$`)`, but $\mathtt{h}_i = \mathtt{h}_j$ only when $i = j$. So `ls2cps` is not injective.

(`cps2ss`, `cps2ls`) One can check by case analysis that no `CPS`-program is mapped to `SS`-program `f` above, and no `CPS`-program is mapped to any of the `LS`-programs $\mathtt{h}_i$. Hence neither `cps2ss` nor `cps2ls` is surjective. ■

This result is further evidence that bisimilarity is the appropriate equivalence as far as I/O behaviour is concerned; contextual equality makes distinctions between programs with identical I/O behaviour.

6 Conclusions, related work and discussion

The main contribution of this paper is a framework in which to study functional I/O. We considered three mechanisms suitable for lazy languages, and gave an operational semantics for each. We needed a simple exception mechanism to model demand for lazy input streams. We showed how the notion of bisimilarity from CCS is a suitable equivalence on programs engaged in I/O. The main result is the first formal proof of the equivalence of three of the most widely implemented functional I/O mechanisms, generalising an informal argument of the Haskell committee.

The three mechanisms are equivalent in the specific sense that there are bijections between them expressible in the lazy language itself. For a general framework for comparing language expressiveness see Felleisen's study [6] of various dialects of Scheme.

There are several literature surveys of work on functional I/O [13, 21, 7]. There has been little previous work on semantics for I/O in lazy languages. Thompson [30] studied Landin-stream I/O using traces. Williams and Wimmers [34] developed algebraic laws for I/O in FL (a call-by-value language) in which each function has an extra, implicit *history* parameter to express I/O. Hudak and Sundaresh [12] informally compared various I/O mechanisms. They gave a semantics to synchronised stream I/O, but using an informally defined nondeterministic merge operator. There has been no previous comparison of I/O mechanisms based on a formal semantics.

A gulf separates the source-level semantics of the stream-based I/O mechanisms given here from their imperative implementation [14]. The gulf is particularly wide between the semantics and implementation of lazy input streams. The only other semantics of stream-based I/O is Thompson's trace-based work [30], which is domain-theoretic and also distant from practical implementations. In contrast, the operational semantics we gave for continuation-passing I/O corresponds fairly closely to an interpretive implementation [21]. It is an open question how to relate abstract specifications of I/O to efficient implementations using side-effects [23].

The low-level mechanisms discussed here can lead to a clumsy programming style; various high-level combinators have emerged from experience of functional I/O [30], the best-known being monadic I/O [4, 33, 23]. The author's dissertation includes a semantics for a particular form of monadic I/O [7]; it remains future work to relate monadic I/O to the three mechanisms studied here. Another direction of future work is to extend the treatment of continuation-passing I/O to accommodate more of Holmström's PFL [9], and to model termination.

Landin-stream I/O is good for teletype I/O but is not general purpose. Lazy input streams admit elegant parsing techniques, which cannot be based so simply on synchronised-stream or continuation-passing teletype I/O. Landin-stream programs can be written in either Haskell [11] or Hope+C [21] because both I/O mechanisms provide operations to obtain a lazy input stream, and hence Landin-stream I/O can be simulated.

The principal merit of synchronised-stream I/O over continuation-passing I/O is that the former can efficiently simulate the latter via a function such as `cps2ss`. Simulation of the former by the latter using a function such as `ss2cps` is inefficient in Haskell [12], although performance can be improved using side-effects [23]. Synchronised-stream programs suffer from problems relating to synchronisation between input and output streams that do not arise with continuation-passing [21] and have a more complex semantics. High-level combinators such as for monadic I/O can be implemented on top of either mechanism [4, 23]. If all user programs are to use such combinators there seems to be no reason to choose synchronised-stream I/O as the underlying mechanism rather than continuation-passing I/O.

7 Acknowledgements

This is a revised version of Chapter 7 of my dissertation [7], written at the University of Cambridge Computer Laboratory. I extend my thanks again to all those who helped me complete my PhD. I am grateful also to Simon Peyton Jones, Phil Wadler and the anonymous referees for useful comments.

References

[1] Samson Abramsky. The lazy lambda calculus. In Turner [32], pages 65–116.

[2] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall International, 1988.

[3] Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *Proc. of the 19th ACM Symp. on Principles of Programming Languages*, pages 328–342, 1992.

[4] J. Cupitt. A brief walk through KAOS. Technical Report 58, Computing Laboratory, University of Kent at Canterbury, February 1989.

[5] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[6] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[7] Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis, Computer Laboratory, University of Cambridge, February 1993. Available as Technical Report 285.

[8] Peter Henderson. Purely functional operating systems. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982.

[9] Sören Holmström. PFL: A functional language for parallel programming. In *Declarative Programming Workshop*, pages 114–139. University College, London, 1983. Extended version published as Report 7, Programming Methodology Group, Chalmers University. September 1983.

[10] Douglas J. Howe. Equality in lazy computation systems. In *Proc. of the 4th IEEE Symp. on Logic in Computer Science*, pages 198–203, 1989.

[11] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, et al. Report on the functional programming language Haskell: A non-strict, purely functional language version 1.2. *ACM SIGPLAN Notices*, 27(5), March 1992. Section R.

[12] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional I/O systems. Research Report YALEU/DCS/RR–665, Yale University Department of Computer Science, March 1989.

[13] S. B. Jones and A. F. Sinclair. Functional programming and operating systems. *The Computer Journal*, 32(2):162–174, April 1989.

[14] Simon B. Jones. Abstract machine support for purely functional operating systems. Technical Report PRG-34, Programming Research Group, Oxford University Computing Laboratory, August 1983.

[15] Kent Karlsson. Nebula: A functional operating system. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1981.

[16] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101,158–165, February and March 1965.

[17] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Mass., 1962.

[18] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.

[20] John T. O'Donnell. Dialogues: A basis for constructing programming environments. In *ACM Symp. on Language Issues in Programming Environments*, pages 19–27, 1985. SIGPLAN Notices 20(7).

[21] Nigel Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Department of Computing, Imperial College, London, June 1991.

[22] Simon L. Peyton Jones. FLIC—a functional language intermediate code. *ACM SIGPLAN Notices*, 23(8):30–48, August 1988.

[23] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proc. 20th ACM Symp. on Principles of Programming Languages, Charleston, South Carolina, January 1993*. ACM Press, 1993.

[24] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[25] Gordon D. Plotkin. The category of complete partial orders: a tool for making meanings. Unpublished lecture notes for the Summer School on Foundations of Artificial Intelligence and Computer Science, Pisa., June 1978.

[26] Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990.

[27] William Stoye. Message-based functional operating systems. *Science of Computer Programming*, 6(3):291–311, 1986.

[28] Christopher Strachey. Fundamental concepts in programming languages. Unpublished lectures given at the International Summer School in Computer Programming, Copenhagen, August 1967.

[29] Simon Thompson. A logic for Miranda. *Formal Aspects of Computing*, 1(4):339–365, October–December 1989.

[30] Simon Thompson. Interactive functional programs: A method and a formal semantics. In Turner [32], pages 249–286.

[31] David Turner. An approach to functional operating systems. In Turner [32], pages 199–217.

[32] David Turner, editor. *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[33] Philip Wadler. Comprehending monads. In *Proc. of the 1990 ACM Conf. on Lisp and Functional Programming*, pages 61–78, June 1990.

[34] John H. Williams and Edward L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Conf. Record of the 15th ACM Symp. on Principles of Programming Languages*, pages 169–179, January 1988.