Towards a Denotational Semantics for Concurrent State Transformers

Peter Thiemann
Fakultät für Informatik, Universität Tübingen
Sand 13, D-72072 Tübingen, Germany
E-mail: thiemann@informatik.uni-tuebingen.de

ABSTRACT

Concurrent state transformers are an extension of state transformers investigated by Launchbury, Peyton Jones, Wadler, and others by concurrency primitives. A denotational semantics for state transformers executing in parallel is defined using standard domain theoretic techniques. Using the semantics it is proved that concurrent state transformers still form a monad and that the concurrent semantics is a proper extension of the sequential one. Finally, we show that an important property of the fix-pointing state transformer is also preserved.

1. Introduction

In a recent sequence of papers, Launchbury, Peyton Jones, and Wadler [19,20,13,8,9] have introduced the concept of state transformers into purely functional programming languages. State transformers integrate I/O and computations on mutable variables without compromising referential transparency. The key is to use an abstract datatype of computations (a monad) with an explicit operator for sequential composition. Thus the program uniquely specifies the sequence of read/write operations on variables (or files) — even in a lazy language. An extension of the same approach has been used to explicitly describe parallel computations in functional languages [7] and to uncover implicit parallelism [18].

The latter application is the subject of the present paper. In [9] a simple denotational semantics for sequential lazy state transformers is given. However, many interesting applications depend on deliberately breaking the sequentiality implicit in the whole approach. Furthermore, the works [7,18] explicitly provide the means to introduce parallelism. Therefore, we investigate the semantics of concurrently executing state transformers. Our semantics is based on the standard resumption approach to model concurrency denotationally.

Moreover, the semantics is crucial in our attempt to prove the correctness of an effect-directed transformation (developed in [18]) which makes hidden parallelism in state transformers explicit. A semantics that appropriately models state transformers running in parallel is essential to for the correctness proof and also for the soundness proof of the simple effect system outlined in that work.

Overview After defining some notation for standard domain constructions, we introduce state transformers (Sec. 3) and review their sequential denotational semantics (Sec. 4). Next (Sec. 5) we introduce non-sequential primitive state transformers. Section 6 develops the semantics for concurrent state transformers. Section 7 provides some results towards the correctness proof of the transformation from [18].

2. Notation

Our notation for domain theoretic constructions follows [10]. We work in the category of SFP domains, where all domain equations involving the functors $\_{}_\perp$ (lifting with embedding **up** and projection **down**), $\_ \times \_$ (with projections $\mathbf{on}_i$), $\_ \oplus \_$ (coalesced sum with injections $\mathbf{in}_i$ and a "case analysis" function $[f_1, \ldots, f_n] \in D_1 \oplus \ldots \oplus D_n \to D$ for $f_i \in D_i \to D$), $\_ \to \_$ (continuous function space using $\circ$ for function composition, **id** for the identity function, and the usual notation and conventions for application and typed abstraction), $\_ \circ\!\!\to \_$ (strict continuous function space with the projection **strict** which makes its argument function strict), $\_^\natural$ (convex powerdomain with set former $\{\!|\_|\!\}$, set union $\_ \uplus \_$, and extension $\mathbf{ext}\_$ of functions: if $f \in D \to E^\natural$ then $\mathbf{ext}\ f \in D^\natural \to E^\natural$ is the unique extension which maps $\{\!|d|\!\}$ to $f d$, for all $d \in D$) have minimal solutions. $\mathbf{fix} \in (D \to D) \to D$ is the uniform fixpoint operator. Each of the operators really denotes a family of operators indexed by domains, e.g. $\mathbf{fix}_D$ and $\mathbf{strict}_{D \to E}$, but we omit the indices since they can be inferred from the context. The ordering relation on domains is denoted by $\sqsubseteq$ with $\perp$ denoting the least element. The domain O is the familiar two point domain with elements $\perp$ and $\top$, ordered by $\perp \sqsubset \top$. In domain expressions, the operator $\times$ takes precedence over $\to$ and $\oplus$.

We write a typed abstraction as $\lambda v \in V \ldots$ to denote a continuous function, but we often omit the typing if it is obvious from context. Also, when defining a function $D_1 \oplus \ldots \oplus D_n \to E$ whose domain is a sum we write $\lambda d_i \in D_i.e$ in place of $[\perp_{D_1 \to E}, \ldots, \lambda d_i \in D_i.e, \ldots, \perp_{D_n \to E}]$ when no confusion arises.

3. State Transformers

In this section, we briefly review state transformers and their application to integrate state-based computations in purely functional languages without causing side-effects. Readers that are familiar with [13,9] may skip this section.

A state transformer is a value of the abstract type `ST s a`. Intuitively, it encapsulates a computation involving some state `s` which produces a value of type `a` when it is executed. The basic operations on state transformers are the functions `returnST` of type `a -> ST s a` and `thenST` of type `ST s a -> (a -> ST s b) -> ST s b`. `returnST` plays the role of the identity function and `thenST` substitutes for function application: `returnST x` is a transformer which returns `x` and leaves the state unchanged, `thenST m f` is a computation which runs `m` first, extracts its result, say `x`, and continues computing `f x`.

The functions `returnST` and `thenST` are assumed to handle the underlying state in a single-threaded manner, i.e. the state is propagated through the whole computation without duplication. Therefore, if primitive operations that manipulate the state are added to `ST s a`, their sequence of execution is fixed. That, in turn, makes it possible to have primitive operations that manipulate the state destructively. For example, they are allowed to physically overwrite a variable with its new value.

Mutable variables holding values of type `a` are elements of an abstract type `MutVar s a`. To ensure the proper sequencing of the operations on mutable variables they are all expressed as state transformers. Here is a signature of primitive operations [9]:

```
newVar   :: a -> ST s (MutVar s a)
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

Here, `newVar x` creates a computation which yields a new mutable variable initialized to `x`, `readVar v` (is a computation which) reads the contents of variable `v`, and `writeVar v x` (is a computation which) assigns `x` to variable `v`.

Now we can construct simple computations from the building blocks described above: the transformer `incrVar` below accepts a variable holding a number and updates it with the incremented value and the transformer `swap` exchanges the values of two variables.[a]

```
incrVar :: MutVar s Int -> ST s ()
incrVar v = readVar v `thenST` \x -> writeVar v (x+1)

swap :: MutVar s a -> MutVar s a -> ST s ()
swap r s = readVar r `thenST` \v -> readVar s `thenST` \w ->
           writeVar r w `thenST` \_ -> writeVar s v
```

A further interesting primitive operation is

```
fixST :: (a -> ST s a) -> ST s a
```

The intended semantics of `fixST f` is the state transformer that you get when `f` is fed back its own output as an argument.[b] It is used to write so-called cyclic programs. We include it here because our translation algorithm mentioned in the introduction depends on it. Furthermore, its concurrent semantics poses some interesting problems.

So far, we have only introduced building blocks for computations involving mutable variables, but we have not yet provided a means to actually execute a computation. Execution is done through the function `runST`. The value of `runST m` is the value obtained by executing the computation `m`.[c] It should be noted (and is reflected by the sequential semantics in the next section) that the intended semantics of running `m` is a lazy one. That is, the reads/writes in `m` are only executed as far as necessary to fulfill the demand on `runST m`. But if any one read/write must be performed, it is assured that all preceding read/writes **must** be performed as well.

Finally, the development in this section is all implemented in the context of the Glasgow Haskell compiler and also in the current version of the Gofer system.

---

[a] The examples are written in Haskell [6]: ``m `thenST` f`` is infix notation (right associative) for the function application `thenST m f`, `x :: t` is a type declaration, and `\v -> e` is a lambda abstraction where the body extends as far to the right as possible.

[b] The formal definition is shown in the next section.

[c] By providing `runST` with a special type, it is statically guaranteed that `m` does not use variables created outside of its own scope. These issues are not in the scope of the current paper and we refer to the discussion in [9,17].

4. Sequential Semantics

In this section we provide a sequential semantics for a non-strict lambda-calculus extended with the state transformer primitives introduced in the previous section. The semantics follows [9], but we have recast it as a parameterized semantics in the style of [1], since we will provide another instantiation in the sections to follow. Notice that the definitions of the constants make extensive use of our conventions to simplify notation. For example, the full definition of *returnST* is

$$\mathit{returnST}^s = \mathit{mkfun}^s(\lambda v \in V.\mathbf{in}_3(\lambda\sigma.(v,\sigma))).$$

We also defer a discussion of the definition of *readVar* etc. to the definition of their concurrent counterparts.

The language of types is defined by

$$\begin{array}{llllll} T & ::= & \texttt{B} & \mid\ T_1\,\texttt{->}T_2 & \mid\ \texttt{ST}\ T_1\ T_2 & \mid\ \texttt{MutVar}\ T_1\ T_2 \\ & & \text{base type} & \text{function type} & \text{state transformer} & \text{mutable variable.} \end{array}$$

Each type constructor C is interpreted by a functor $\mathcal{F}^s_C$ as follows:

$$\begin{array}{lcl} \mathcal{F}^s_{\texttt{B}} & = & B_\perp \\ \mathcal{F}^s_{\texttt{->}}(V_1,V_2) & = & V_1 \to V_2 \\ \mathcal{F}^s_{\texttt{ST}}(\_,V) & = & S_V \to (V \times S_V) \\ \mathcal{F}^s_{\texttt{MutVar}}(\_,\_) & = & \mathbf{N}_\perp \end{array}$$

where B is some set of base values and a state $S = S_V$ is defined by

$$S_V = (\mathbf{N}_\perp \circ\!\!\to V_\perp)_\perp.$$

Natural numbers are used to model variable names. If $\sigma \in S$ is not $\perp$ it maps natural numbers to either $\mathbf{up}\ v$ (where $v \in V$ is the contents of the location, perhaps $\perp$) or to $\perp$, indicating a free location.

Putting things together, the domain of values is the solution of the equation

$$\begin{array}{rcccccccc} V & = & \mathcal{F}^s_{\texttt{B}} & \oplus & \mathcal{F}^s_{\texttt{->}}(V,V) & \oplus & \mathcal{F}^s_{\texttt{ST}}(V,V) & \oplus & \mathcal{F}^s_{\texttt{MutVar}}(V,V) \\ \text{with typical element} & & & & \ni f & & \ni m & & \ni l. \end{array}$$

The syntax of expressions is defined by

$$\begin{array}{lcl} e & ::= & x \mid k \mid e_1\ e_2 \mid \lambda x.e \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ k & ::= & \texttt{returnST} \mid \texttt{thenST} \mid \texttt{newVar} \mid \texttt{readVar} \mid \texttt{writeVar} \mid \texttt{fixST} \mid \texttt{runST} \mid \ldots \end{array}$$

and their parameterized semantics by (with environments $\rho \in \mathit{Var} \to V$)

$$\begin{array}{lcl} \mathcal{E}[\![x]\!]\rho & = & \rho[\![x]\!] \\ \mathcal{E}[\![k]\!]\rho & = & \mathcal{B}[\![k]\!] \\ \mathcal{E}[\![e_1\ e_2]\!]\rho & = & \mathit{apply}(\mathcal{E}[\![e_1]\!]\rho)(\mathcal{E}[\![e_2]\!]\rho) \\ \mathcal{E}[\![\lambda x.e]\!]\rho & = & \mathit{mkfun}(\lambda y.\mathcal{E}[\![e]\!]\rho[x \mapsto y]) \\ \mathcal{E}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]\rho & = & \mathcal{E}[\![e_2]\!](\mathit{fix}\lambda\rho'.\rho[x \mapsto \mathcal{E}[\![e_1]\!]\rho']) \end{array}$$

$$\begin{array}{lcl} \mathcal{B}[\![\texttt{returnST}]\!] & = & \mathit{returnST} \\ \mathcal{B}[\![\texttt{thenST}]\!] & = & \mathit{thenST} \\ \mathcal{B}[\![\texttt{newVar}]\!] & = & \mathit{newVar} \\ \mathcal{B}[\![\texttt{readVar}]\!] & = & \mathit{readVar} \\ \mathcal{B}[\![\texttt{writeVar}]\!] & = & \mathit{writeVar} \\ \mathcal{B}[\![\texttt{fixST}]\!] & = & \mathit{fixST} \\ \mathcal{B}[\![\texttt{runST}]\!] & = & \mathit{runST} \end{array}$$

We use $\mathcal{E}^s[\![\_]\!]$ to indicate that we use the standard sequential interpretation of the operators as defined by:

$$
\begin{array}{lcl}
apply^s & = & [\bot, \lambda f \in V \to V.f, \bot, \bot] \\
mkfun^s & = & \mathbf{in}_2 \\
fix^s & = & \mathbf{fix} \\
returnST^s & = & \lambda v.\lambda\sigma.(v,\sigma) \\
thenST^s & = & \lambda m.\lambda f.\lambda\sigma.(\lambda(v,\sigma).fv\sigma)(m\sigma) \\
newVar^s & = & \lambda v.\mathbf{strict}\lambda\sigma.(l, \mathbf{up}\ (\mathbf{down}\ \sigma[l \mapsto \mathbf{up}\ v])) \quad \text{where } l \neq \bot \wedge \mathbf{down}\ \sigma l = \bot \\
readVar^s & = & \mathbf{strict}\lambda l.\mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda v.(\mathbf{down}\ v, \sigma))(\mathbf{down}\ \sigma l) \\
writeVar^s & = & \mathbf{strict}\lambda l.\lambda v.\mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda z.(\bot, \mathbf{up}\ (\mathbf{down}\ \sigma[l \mapsto \mathbf{up}\ v])))(\mathbf{down}\ \sigma l) \\
fixST^s & = & \lambda f.\lambda\sigma.\mathbf{fix}\ \lambda(v,\sigma').fv\sigma \\
runST^s & = & \lambda m.\mathbf{on}_1(m(\mathbf{up}\ \bot_S))
\end{array}
$$

5. Introducing concurrency

The primitive operation `interleaveST`[d][13] can be used to express system programming concepts in a higher-order lazy functional programming language. Its type is `ST s a -> ST s a` and it spawns its argument state transformer as an independent computation outside the sequential computation. Some applications are shown in [9].

We will consider the semantics of a parallelizing version of `interleaveST` which really forks off its argument computation as a separate thread which shares the state (with all mutable variables) with the rest of the computation. Both versions of `interleaveST`, the sequential one as well as the parallelizing one, are inherently unsafe[e] as they allow two independent threads of computation to access the same set of mutable variables simultaneously.

Another primitive, which is better behaved semantically, is `forkST` with signature specified as

```
forkST :: ST s a -> ST s b -> ST s (a,b)
```

The sequential semantics of `forkST m1 m2` (as considered in [18]) is a computation which independently performs `m1` and `m2` in a demand-driven interleaved manner. The result of `forkST m1 m2` is a pair consisting of the result of `m1` and `m2`.

A parallel version of `forkST` [7,18] actually spawns both computations `m1` and `m2` as independent threads and collect the results of both. It corresponds to fork/join parallelism. It is this version that will be considered in our semantics.

6. Semantics

We develop a denotational semantics for a non-strict lambda-calculus extended with concurrent state transformers created using the operations mentioned above. That is, with respect to Sec. 4 the syntax of expressions is extended by

$$k ::= \ldots \mid \texttt{interleaveST} \mid \texttt{forkST}$$

Our semantics is based on standard domain theory [5,4]. It models the interleaved, branching time execution of state transformers where all accesses (variable allocation,

[d]Also know as `unsafeInterleaveST` to the Glasgow Haskell compiler.

[e]ie they break referential transparency.

read, write) to the store are atomic. The result of running such a state transformer is finally flattened to a linear time description.

6.1. Semantic Domains

State transformations are expressed by means of *resumptions*, i.e. they are decomposed into sequences of atomic pieces so that they may be interleaved. Thus it is possible to explicitly construct all interleavings of two (or more) such sequences. Our resumptions are specified by

$$R = S \to (V \times S \oplus R \times S)^{\natural}$$

that is, a resumption $r \in R$ is a mapping from a state in S to a set of alternatives. For set construction the natural (Plotkin) powerdomain [14] is used. As these sets are used to express possible outcomes of computations it does not matter that the Plotkin powerdomain does not contain the empty set: any computation returns a result which may be $\perp$. Another reason for choosing the Plotkin powerdomain is the fact that it can keep, eg, $\{\!|\perp, 1|\!\}$ and $\{\!|1|\!\}$ apart (which are equal in the lower powerdomain). Therefore, the possibility of nontermination is modeled appropriately: the first set denotes "either nontermination or 1" while the second set denotes "definitely 1".

Each of the elements of such a set of alternatives is either a pair (v, σ) of a value and a state, or a pair (r, σ) of a resumption and a state. The former (v, σ) denotes the termination of a state transformer which yields the final answer v in state σ. The latter (r, σ) denotes an intermediate result of a computation which has changed the state to σ so far and is willing to proceed (resume) with r.

In the standard treatment of resumptions (see e.g. [16,10]) only the interleaved execution of imperative statements is discussed. In that context, resumptions only change the state and do not return a value. The presence of a return value leads to additional complications.

We are now ready to give the interpretation of the `ST` type constructor. Let $A = (\mathbf{N} \times \mathbf{N})_\perp$ be *allocation information*, which is used for the allocation of new variables and is discussed in the next section. Let further $R^? = R_\perp \oplus O$ denote a possibly absent resumption, i.e. $\mathbf{in}_1(r)$ denotes the presence of the resumption r and $\mathbf{in}_2(\top)$ denotes the absence of a resumption.

$$\mathcal{F}^c_{\mathtt{ST}}(\_, V) = R^? \to A \to (V \to R^? \to A \to R) \to R$$

A concurrent state transformer, m say, maps a (possibly absent) resumption $r \in R^?$, allocation information $a \in A$, and a continuation $k \in V \to R^? \to A \to R$ to another resumption. The idea is that r represents the resumption which is going to be interleaved to m (or none if nothing is interleaved). It is passed (possibly changed) to the continuation. The allocation information a is also passed to the continuation. The continuation k takes — apart from the passed values just discussed — the result of the subject transformer and yields a resumption.

Please note that continuations are not only used in tail call position. Only in the sequential fragment, our continuations are used in the standard way. In the description of the parallelizing operations they appear nested inside of other function applications.

The interpretation of the other type constructors `B`, `->`, and `MutVar` remains unchanged. However, the domain equation for the domain of values must be adjusted to account for sets of results as follows:

$$V = (\mathcal{F}^c_{\texttt{B}} \oplus \mathcal{F}^c_{\texttt{->}}(V,V) \oplus \mathcal{F}^c_{\texttt{ST}}(V,V) \oplus \mathcal{F}^c_{\texttt{MutVar}}(V,V))^\natural.$$

Accordingly, we have to set

$$\begin{array}{lcl} \mathit{apply}^c & = & \mathbf{ext}[\bot, \lambda f \in V \to V.f, \bot, \bot] \\ \mathit{mkfun}^c & = & \lambda f \in V \to V.\{\!|\mathbf{in}_2 f|\!\} \\ \mathit{fix}^c & = & \mathbf{fix} \end{array}$$

For brevity, we define $M = \mathcal{F}^c_{\texttt{ST}}(V,V)$ for transformers, $L = \mathcal{F}^c_{\texttt{MutVar}}(V,V)$ for locations, and $K = V \to R^? \to A \to R$ for continuations.

6.2. Auxiliary Functions

Memory Allocation The allocation information $A = (\mathbf{N} \times \mathbf{N})_\bot$ is used to ensure that threads executing in parallel use disjoint addresses to allocate memory. This makes it easier to compare different interleavings of allocations. Memory allocation is parameterized by a pair $(e,i) \in A$ such that, for a state σ, $\mathbf{alloc}(e,i)\sigma$ returns the minimum l such that (1) location l is free in state σ: $(\mathbf{down}\ \sigma)l = \bot$ and (2) $l \equiv i \pmod{2^e}$.

The execution of a state transformer starts with allocation information $a = (0,0)$ and every `forkST` or `interleaveST` also splits the allocation information (e,i) into $\acute{a} := (e+1, i)$ and $\grave{a} := (e+1, i+2^e)$. Let AT denote the set of all binary trees where the root node is labeled with $(0,0)$ and the sons of a node labeled (e,i) are labeled with $(e+1,i)$ and $(e+1,i+2^e)$. The following lemma is proved by induction on n with $\mathit{Loc}(e,i) = \{k2^e + i \mid k \in \mathbf{N}\}$.

Lemma 1 *For every cut $(e_1,i_1),\ldots,(e_n,i_n)$ of some $t \in AT$, the family of sets $(\mathit{Loc}(e_j,i_j) \mid 1 \le j \le n)$ is a partition of $\mathbf{N}$.*

Sequential Composition When `runST` executes a state transformer, it accepts a value of type M and must transform it into the set (in the powerdomain sense) of all possible return values. To this end we construct a function which flattens a resumption into a function which transforms an initial state into a set of pairs of return value and final state.

$$\begin{array}{l} \mathbf{compose} \in R \to S \to (V \times S)^\natural \\ \mathbf{compose} = \mathbf{fix}\ \lambda h.\lambda r \in R.\mathbf{ext}[\lambda(v,\sigma).\{\!|(v,\sigma)|\!\}, \lambda(r,\sigma).hr\sigma] \circ r \end{array}$$

The function $\mathbf{attach}_1$ accepts a value v_1 and a resumption r which finally returns v_2 and constructs a resumption which finally returns (v_1,v_2). $\mathbf{attach}_2$ is defined analogously with v_1 and v_2 exchanged in the lambda abstractions.

$$\begin{array}{ll} \mathbf{attach}_1 \in V \to R \to R & \\ \mathbf{attach}_1 = & \lambda v_1 \in V.\mathbf{fix}\ \lambda h \in R \to R.\lambda r \in R. \\ & \quad \mathbf{ext}[\ \lambda(v_2,\sigma).\{\!|\mathbf{in}_1((v_1,v_2),\sigma)|\!\}, \\ & \qquad \lambda(r_2,\sigma).\{\!|\mathbf{in}_2(hr_2,\sigma)|\!\}] \circ r \end{array}$$

The function **append** $\in R^? \to A \to K \to R \to R$ accepts arguments for a continuation, a continuation, and a resumption. It builds a new resumption which models the sequential execution of the argument resumption and then the continuation.

$$\begin{array}{rl} \mathbf{append} = & \lambda r_0 \in R^?.\lambda a \in A.\lambda k \in K. \\ & \mathbf{fix}\ \lambda h \in R \to R.\lambda r \in R.\mathbf{ext}[\ \lambda(v,\sigma).\mathbf{in}_2(kvr_0a,\sigma), \\ & \qquad\qquad\qquad\qquad\qquad \lambda(r,\sigma).\mathbf{in}_2(hr,\sigma)] \circ r \end{array}$$

The function **yield** has type K and plays the role of the identity continuation.

$$\mathbf{yield} = \lambda v \in V.\lambda r' \in R^?.\lambda a \in A.\lambda\sigma.\{\!|\mathbf{in}_1(v,\sigma)|\!\}$$

Parallel Composition We present two operators for parallel composition of resumptions. One of them is used to model interleaved computations on the state. It maps $R \times R^?$ to R by ignoring the result of its second argument (but not its effect on the state). The other models the operation `fork`, such that the result of the parallel composition of two resumptions is a resumption which finally returns a pair. We consider the latter operation first:

$$\begin{array}{rl} \parallel & \in R \times R \to R \\ \_ \parallel \_ & = \ \mathbf{fix}\ \lambda h \in R \times R \to R.\lambda(r_1,r_2) \in R \times R. \\ & \qquad \mathbf{ext}[\lambda(v_1,\sigma).\{\!|\mathbf{in}_2(\mathbf{attach}_1\ v_1\ r_2,\sigma)|\!\}, \lambda(r_1',\sigma).\{\!|\mathbf{in}_2(h(r_1',r_2),\sigma)|\!\}] \circ r_1 \\ & \ \uplus \ \ \mathbf{ext}[\lambda(v_2,\sigma).\{\!|\mathbf{in}_2(\mathbf{attach}_2\ v_2\ r_1,\sigma)|\!\}, \lambda(r_2',\sigma).\{\!|\mathbf{in}_2(h(r_1,r_2'),\sigma)|\!\}] \circ r_2 \end{array}$$

The function $\parallel$ defines the symmetric parallel composition operation which is used to define `forkST`. Basically, it nondeterministically chooses to execute one step of one of the two resumptions r_1 or r_2. If the application of r_1, say, to the current state delivers a final result (v_1,σ), the result resumption is constructed by attaching r_2 to its end and modifying its return value. Otherwise, the intermediate result is (r_1',σ) and the result resumption is obtained by interleaving r_1' and r_2.

$$\begin{array}{rl} \backslash\!\!\backslash & \in R \times R^? \to R \\ \_ \backslash\!\!\backslash \_ & = \ \mathbf{fix}\ \lambda h \in R \times R^? \to R.\lambda(r_1,r_2) \in R \times R^?. \\ & [\lambda r_2. \quad \mathbf{ext}[\lambda(v_1,\sigma).\{\!|\mathbf{in}_2(\mathbf{attach}_0\ v_1\ r_2,\sigma)|\!\}, \lambda(r_1,\sigma).\{\!|\mathbf{in}_2(h(r_1,r_2),\sigma)|\!\}] \circ r_1 \\ & \qquad \uplus \ \ \mathbf{ext}[\lambda(v_2,\sigma).\{\!|\mathbf{in}_2(r_1,\sigma)|\!\}, \lambda(r_2,\sigma).\{\!|\mathbf{in}_2(h(r_1,r_2),\sigma)|\!\}] \circ r_2, \\ & \lambda z \in O.r_1]r_2 \end{array}$$

The definition is quite similar to the one for $\parallel$, except that first r_2 is tested for emptiness (and r_1 returned in the affirmative case) and then r_1 and r_2 are unwound as in $\parallel$. When r_2 terminates, its result is ignored and the (rest of the) resumption r_1 is simply appended. In the unraveling of r_1, the function $\mathbf{attach}_0$ is used in place of $\mathbf{attach}_1$. It unwinds the resumption r_2, ignores its result, and replaces it by v_1. Its definition is similar to $\mathbf{attach}_1$ and therefore omitted.

By construction, $\backslash\!\!\backslash$ has $\mathbf{in}_2(\top)$ as a right identity. We assume that $\backslash\!\!\backslash$ associates to the left.

6.3. Sequential Operations

First, we specify the concurrent interpretations of the basic sequential operations.

$$returnST^c = \ \lambda v \in V.\lambda r \in R^?.\lambda a \in A.\lambda k \in K.kvra$$

This reflects the fact that `returnST` gets the value to be returned as an argument v. It simply calls its continuation k on v and threads the arguments r and k through.[f]

$$\begin{aligned}
\mathit{thenST}^c = \ & \lambda m \in M.\lambda f \in V \to M. \\
& \lambda r \in R^?.\lambda a \in A.\lambda k \in K. \\
& mra(\lambda v.\lambda r.\lambda a.fvrak)
\end{aligned}$$

Bearing in mind that m is a state transformer and f is a function delivering one, we see that m is executed with a continuation that extracts the value v returned by m and executes fv with the threaded arguments and the original continuation.[g]

$$\begin{aligned}
\mathit{readVar}^c = \ & \lambda l \in L. \\
& \lambda r \in R^?.\lambda a \in A.\lambda k \in K. \\
& \mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda v.\{\!|\mathbf{in}_2(k(\mathbf{down}\ v)ra, \sigma)|\!\})(\mathbf{down}\ \sigma l)
\end{aligned}$$

All operations that affect the state or are affected by it, have to create an explicit step of the resumption, as exemplified by the *readVar* operation above. The first argument l is the location of type `MutVar s v`. As in the sequential case when a free location is dereferenced, we have $\mathbf{down}\ \sigma l = \perp$. Applying $\mathbf{strict}\lambda v \ldots$ to it forces the whole result to be $\perp$. However, it is possible to have l bound to $\perp$: then $\mathbf{down}\ \sigma l = \mathbf{up}\ \perp$ and the continuation k is called on $\mathbf{down}\ (\mathbf{up}\ \perp) = \perp$.

Finally, $\mathbf{strict}\lambda\sigma \ldots$ makes it impossible to ignore a non-terminating computation which involves state changes. The presence of this strict function forces all preceding read/write operations on the state to be performed before the variable l is read.

$$\begin{aligned}
\mathit{writeVar}^c = \ & \lambda l \in L.\lambda v \in V. \\
& \lambda r \in R^?.\lambda a \in A.\lambda k \in K. \\
& \mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda z.\{\!|\mathbf{in}_2(k\perp ra, \mathbf{up}\ ((\mathbf{down}\ \sigma)[l \mapsto \mathbf{in}_1(v)]))|\!\})(\mathbf{down}\ \sigma l)
\end{aligned}$$

The *writeVar* operation takes a location l and a new value v and creates a resumption step with the state appropriately changed. It also checks with $(\mathbf{strict}\lambda z \ldots)(\mathbf{down}\ \sigma l)$ whether l is uninitialized in the same manner as *readVar* does. The comments on $\mathbf{strict}\lambda\sigma \ldots$ apply here as well.

$$\begin{aligned}
\mathit{newVar}^c = \ & \lambda v \in V. \\
& \lambda r \in R^?.\lambda a \in A.\lambda k \in K. \\
& \mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda l.\{\!|\mathbf{in}_2(klra, \mathbf{up}\ ((\mathbf{down}\ \sigma)[l \mapsto \mathbf{up}\ (v)]))|\!\})(\mathbf{alloc}\ a\ \sigma)
\end{aligned}$$

The creation of a new variable also forces the execution of all preceding read/writes.[h] If the allocation operation does not succeed, the whole result becomes $\perp$ by virtue of $\mathbf{strict}\lambda l \ldots$.

$$\begin{aligned}
\mathit{runST}^c = \lambda m \in M.\ & \mathbf{ext}\ \mathbf{on}_1(\mathbf{compose} \\
& (m\ (\mathbf{in}_2\top)(0,0)\mathbf{yield})(\mathbf{up}\ \perp))
\end{aligned}$$

[f] Technically, we would have to write $\mathit{returnST}^c = \mathit{mkfun}^c(\lambda v \in V.\{\!|\mathbf{in}_3(\lambda r \ldots)|\!\})$ but we leave that implicit for the benefit of the reader.

[g] Here, $\lambda m \in M \ldots$ really abbreviates $\mathbf{ext}[\perp, \perp, \lambda m \in M \ldots, \perp]$.

[h] This is not strictly necessary but slightly more efficient in an implementation. It coincides with the sequential semantics given here and in [9] as well as with the read implementation. A non-strict allocation operation can be programmed using `interleaveST`.

The *runST* function performs as outlined above. It starts its argument transformer m with no computation interleaved ($\mathbf{in}_2\top$), the initial allocation information $(0, 0)$, and the identity continuation **yield**. The thus constructed resumption is then transformed to a function of type $S \to (V \times S)^\natural$ by **compose**. The resulting function is applied to an initial empty state $\mathbf{up}\ \bot$ (remember that $\bot \in S$ means that the state is not available anymore). Extracting the value component of this set of pairs yields the desired set of values.

6.4. Parallel Operations

The first operation that we treat is `interleaveST`. The presence of `interleaveST` explains the necessity of the continuation argument. The result of `interleaveST m` is the interleaved execution of `m` and the rest of the computation, captured by the continuation. A slight complication is introduced since the result of `m` must be passed as an argument to the continuation.

$$
\begin{aligned}
\mathit{interleaveST}^c = \;& \lambda m \in M. \\
& \lambda r \in R^?.\lambda a \in A.\lambda k \in K. \\
& \lambda\sigma.\ (\lambda(r_m, r_k, v_m).(r_k \setminus\!\!\setminus \mathbf{in}_1 r_m)\sigma) \\
& \quad (\mathbf{fix}\ \lambda(r_m, r_k, v_m).\ (m(\mathbf{in}_1(r_k \setminus\!\!\setminus r))\ \text{à}\ \mathbf{yield}, \\
& \qquad\qquad\qquad\qquad\quad \lambda\sigma.\mathbf{ext}\ k v_m(\mathbf{in}_1(r_m \setminus\!\!\setminus r))\ \text{á}\ \mathbf{yield}, \\
& \qquad\qquad\qquad\qquad\quad \mathbf{ext}\ \mathbf{on}_1(\mathbf{compose}((r_m \setminus\!\!\setminus \mathbf{in}_1 r_k) \setminus\!\!\setminus r)\sigma)))
\end{aligned}
$$

The variable r_m, r_k, and v_m denote the resumption of m (interleaved with r_k and r), the resumption of the continuation (interleaved with r_m and r), and the set of values obtained from running r_m. These three definition are mutually dependent which makes it necessary to use the fixpoint. Notice that $v \in V^\natural$, i.e. a set of values in V which are sets themselves. The latter is the reason for $\mathbf{ext}k\ldots$ in the expression for r_k.

Unfortunately, we obtain some artifacts if m depends on state changes induced by k. In that case some "branches" of r_k run on a value which they can never get in practice. This is due to the fact that v_m is a fixpoint as a set, not a set of fixpoints.

First, the resumption r_m for m is constructed. Then the value set v_m of values returned from r_m is computed by composition of the interleaved executions of r_m with the resumption r_k (capturing the computation in the continuation) and the resumption r which is interleaved. The resumption r_k is defined simultaneously since v_m and r_k are mutually dependent.

The description of `forkST` is much more simple and symmetric. The reason being that `forkST` does not run its arguments in parallel with its continuation, but joins its argument computations first.

$$
\begin{aligned}
\mathit{forkST}^c = \;& \lambda m_1 \in M.\lambda m_2 \in M. \\
& \lambda r \in R^?.\lambda a \in A.\lambda k \in K. \\
& \quad (\lambda(r_1, r_2).\mathbf{append}\ r\ a\ k\ (r_1 \parallel r_2)) \\
& \quad \mathbf{fix}\ \lambda(r_1, r_2).(m_1(\mathbf{in}_1(r_2 \setminus\!\!\setminus r))\ \text{á}\ \mathbf{yield}, \\
& \qquad\qquad\qquad\quad m_2(\mathbf{in}_1(r_1 \setminus\!\!\setminus r))\ \text{à}\ \mathbf{yield})
\end{aligned}
$$

6.5. The Fixpointing State Transformer

In order to obtain a description for `fixST` we must take into account that interleaved computations can influence the value produced by a state transformer. The "feedback value" is therefore found by taking the fixpoint of $(\lambda(v, \sigma').\mathbf{compose}\ldots)$ which runs all possible interleavings of fv and r and extracts the set of possible values. The resulting resumption is constructed by supplying this set of arguments to f and running it with the real continuation k. Notice, that there may be artificial feedback values, due to the fact that we have the fixpoint of a function from sets to sets and not a set of fixpoints.

$$
\begin{aligned}
\mathit{fixST}^c = \;& \lambda f \in V \to M. \\
& \lambda r \in R^?.\lambda a \in A.\lambda k \in K. \\
& \lambda\sigma.\ \mathbf{ext}(\lambda(v,\sigma').fvrak\sigma) \\
& \quad (\mathbf{fix}\ (\mathbf{ext}\lambda(v,\sigma').\mathbf{compose}(fvra\ \mathbf{yield} \mathbin{\backslash\!\backslash} r)\sigma))
\end{aligned}
$$

7. Properties

7.1. It's a monad

One of the pleasing properties of the sequential state transformers is the fact that they form a monad. That is, the function `thenST` is associative with `returnST` as a unit in the following sense:

$$
\begin{aligned}
\mathit{thenST}\ (\mathit{returnST}\ x)\ f &= f\ x && (1)\\
\mathit{thenST}\ m\ (\mathit{returnST}) &= m && (2)\\
\mathit{thenST}(\mathit{thenST}\ m\ f)\ g &= \mathit{thenST}\ m\ (\lambda v.\mathit{thenST}\ (f\ v)\ g) && (3)
\end{aligned}
$$

It is very important to establish these properties as they form the very basis of all program transformations applicable to programs written in monadic style. Fortunately, these equations can be proved in our concurrent semantics, too.

Theorem 1 *The concurrent interpretations returnST and thenST form a monad.*

Proof 1 *We only demonstrate (2), the other proofs proceed similar.*

$$
\begin{aligned}
& \mathit{thenST}\ m\ (\mathit{returnST}) \\
= \;& (\lambda mfrak.mra(\lambda vra.fvrak))\ m\ (\lambda vrak.kvra) \\
= \;& \lambda rak.mra(\lambda vra.(\lambda vrak.kvra)vrak) \\
= \;& \lambda rak.mra(\lambda vra.kvra) \\
= \;& \lambda rak.mrak \\
= \;& m
\end{aligned}
$$

7.2. It's a Well-Behaved Extension

Whenever we execute a state transformer which contains neither of the parallelization operators `interleaveST` and `forkST` it returns a singleton set as result.

Theorem 2 *Let e be a closed expression which does not contain* `interleaveST` *and* `forkST`. *Then $\mathcal{E}^c[\![e]\!]$ is a singleton set.*

This is immediate for all constants and for abstractions. Some work is required to establish it for all expressions. First, we need some definitions to state a stronger hypothesis which we can base an inductive proof on.

Definition 1 *A resumption $r \in R$ is* deterministic *iff for all $n \in \mathbf{N}$ there is an $x \in \mathcal{F}^c_{\mathtt{B}} \oplus \mathcal{F}^c_{\mathtt{->}}(V,V) \oplus \mathcal{F}^c_{\mathtt{ST}}(V,V) \oplus \mathcal{F}^c_{\mathtt{MutVar}}(V,V) \times S$ such that for all σ $(r \downarrow n)\sigma \in \{\bot, \{\!|x|\!\}\}$, where*

$$\begin{array}{lcl} r \downarrow 0 & = & r \\ r \downarrow (i+1) & = & \mathbf{ext}[\ \lambda(v,\sigma).\{\!|\mathbf{in}_1(v,\sigma)|\!\}, \lambda(r,\sigma).(r \downarrow i)\sigma] \circ r \end{array}$$

Definition 2 *A value $v \in V$ is* well-behaved, *$WB(v)$, if $v = \bot$ or*

(WBS) *v is a singleton: $\exists x.v = \{\!|x|\!\}$,*

(WBF) *if $v = \{\!|\mathbf{in}_2 f|\!\}$ then f is a function which satisfies $\forall v' \in V.WB(v') \Rightarrow WB(fv')$,*

(WBM) *if $v = \{\!|\mathbf{in}_3 m|\!\}$ then $r := m(\mathbf{in}_2\top)(0,0)\mathbf{yield}$ is deterministic and $\forall \sigma \in S.WB(\sigma) \Rightarrow WB(\mathbf{ext}\ \mathbf{on}_1(\mathbf{compose}\ r\sigma))$.*

A state $\sigma \in S$ is well-behaved, *$WB(\sigma)$, if $\sigma = \bot$ or $\forall l \in L.\mathbf{down}\ \sigma l \neq \bot \Rightarrow WB(\mathbf{down}\ (\mathbf{down}\ \sigma l))$.*

An environment $\rho \in Var \to V$ is well-behaved, *$WB(\rho)$, if for all $x \in dom(\rho)$ $WB(\rho[\![x]\!])$ holds.*

Now we are in the position to state and prove the following lemma.

Lemma 2 *For all e which do not contain* `interleaveST` *and* `forkST` *and ρ with $free(e) \subseteq dom(\rho)$ we have $WB(\rho) \Rightarrow WB(\mathcal{E}^c[\![e]\!]\rho)$.*

Proof 2 *We only show some representative cases of the induction on e.*

$\boxed{e \equiv \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2}$ *By fixpoint induction, starting with $\mathcal{E}^c[\![e_1]\!]\rho[x \mapsto \{\!|\bot|\!\}]$ and using the inductive assumption for e_1 in every step, we establish $WB(\rho')$ where $\rho' = \mathbf{fix}\ \lambda\rho'.\rho[x \mapsto \mathcal{E}^c[\![e_1]\!]\rho']$.*

By appeal to the inductive assumption for e_2 and ρ' we obtain $WB(\mathcal{E}^c[\![e_2]\!]\rho')$ and we are done.

$\boxed{e \equiv \mathtt{returnST}}$ *Let $v = returnST = \{\!|\mathbf{in}_2 \underbrace{(\lambda v \in V.\{\!|\mathbf{in}_3(\lambda rak.kvra)|\!\})}_{=:f}|\!\}$. Property* ***WBS*** *holds trivially and* ***WBM*** *is void. For* ***WBF*** *we have to show $WB(v') \Rightarrow WB(fv')$ for all $v' \in V$. This is equivalent to showing*

$$\forall v' \in V.WB(v') \Rightarrow WB\{\!|\mathbf{in}_3(\lambda rak.kv'ra)|\!\}$$

which is easily seen, since **WBS** hold trivially and **WBF** is void. For **WBM** we have the following:

$$\begin{array}{rl} & (\lambda rak.kv'ra)(\mathbf{in}_2\top)(0,0)\mathbf{yield} \\ = & \mathbf{yield}v'(\mathbf{in}_2\top)(0,0) \\ = & (\lambda vra\sigma.\{\!\!\{\mathbf{in}_1(v,\sigma)\}\!\!\})v'(\mathbf{in}_2\top)(0,0) \\ = & \lambda\sigma.\{\!\!\{\mathbf{in}_1(v',\sigma)\}\!\!\} \end{array}$$

Therefore, fv' is deterministic and for all states σ we have

$$\begin{array}{rl} & \mathbf{ext}\ \mathbf{on}_1(\mathbf{compose}(\lambda\sigma.\{\!\!\{\mathbf{in}_1(v',\sigma)\}\!\!\})\sigma) \\ = & \mathbf{ext}\ \mathbf{on}_1((\mathbf{fix}\ \lambda h.\lambda r\in R.\mathbf{ext}[\lambda(v,\sigma).\{\!\!\{(v,\sigma)\}\!\!\},\lambda(r,\sigma).hr\sigma]\circ r)(\lambda\sigma.\{\!\!\{\mathbf{in}_1(v',\sigma)\}\!\!\})\sigma) \\ = & \mathbf{ext}\ \mathbf{on}_1((\mathbf{ext}[\lambda(v,\sigma).\{\!\!\{(v,\sigma)\}\!\!\},\lambda(r,\sigma).(\mathbf{fix}\ \ldots)r\sigma]\{\!\!\{\mathbf{in}_1(v',\sigma)\}\!\!\})) \\ = & \mathbf{ext}\ \mathbf{on}_1\{\!\!\{(v',\sigma)\}\!\!\} \\ = & v' \end{array}$$

which is well-behaved by assumption, and thus we have $WB(v)$.

$\boxed{e\equiv\texttt{readVar}}$ Let $r := (\lambda rak.\mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda v.\{\!\!\{\mathbf{in}_2(k(\mathbf{down}\ v)ra,\sigma)\}\!\!\})(\mathbf{down}\ \sigma l))$ and $g := \mathbf{ext}\lambda l\in L.\{\!\!\{\mathbf{in}_3 r\}\!\!\}$ then $v = \mathit{readVar}^c = \{\!\!\{\mathbf{in}_2 g\}\!\!\}$. For v, **WBS** is obvious and **WBM** is void. To show **WBM** we have to look at the body of g (where **WBS** is obvious and **WBF** is void) and then — to show **WBM** — at resumption r. The following calculation shows that r is deterministic:

$$\begin{array}{rl} & (\lambda rak.\mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda v.\{\!\!\{\mathbf{in}_2(k(\mathbf{down}\ v)ra,\sigma)\}\!\!\})(\mathbf{down}\ \sigma l))(\mathbf{in}_2\top)(0,0)\mathbf{yield} \\ = & \mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda v.\{\!\!\{\mathbf{in}_2(\mathbf{yield}(\mathbf{down}\ v)(\mathbf{in}_2\top)(0,0),\sigma)\}\!\!\})(\mathbf{down}\ \sigma l) \\ = & \mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda v.\{\!\!\{\mathbf{in}_2((\lambda vr'a.\lambda\sigma.\{\!\!\{\mathbf{in}_1(v,\sigma)\}\!\!\}))(\mathbf{down}\ v)(\mathbf{in}_2\top)(0,0),\sigma)\}\!\!\})(\mathbf{down}\ \sigma l) \\ = & \mathbf{strict}\lambda\sigma.(\mathbf{strict}\lambda v.\{\!\!\{\mathbf{in}_2(\lambda\sigma.\{\!\!\{\mathbf{in}_1(\mathbf{down}\ v,\sigma)\}\!\!\},\sigma)\}\!\!\})(\mathbf{down}\ \sigma l) \\ = & \mathbf{strict}\lambda\sigma.\{\!\!\{\mathbf{in}_2(\lambda\sigma.\{\!\!\{\mathbf{in}_1(\mathbf{down}\ (\mathbf{down}\ \sigma l),\sigma)\}\!\!\},\sigma)\}\!\!\} \end{array}$$

Therefore, r is deterministic. For any $\sigma\in S$, if $WB(\sigma)$ we can calculate that $\mathbf{ext}\ \mathbf{on}_1(\mathbf{compose}\ r\sigma)$ yields $\mathbf{down}\ \sigma l$ which is WB by assumption on σ.

Proof 3 *(of Thm. 2) Immediate since the empty environment is WB.*

7.3. `fixST` *Deserves its Name*

The following lemma is a property of (sequential) `fixST` which forms the basis of our translation algorithm in [18].

Lemma 3 *Let* $v := \mathbf{on}_1(\mathcal{E}^s[\![\texttt{fixST}\lambda x.e]\!]\rho\sigma)$. *Then* $v = \mathbf{on}_1(\mathcal{E}^s[\![e]\!]\rho[x\mapsto v]\sigma)$.

Its proof is a straightforward calculation. Its statement for the concurrent semantics is as follows:

Lemma 4 *Let* $v_m := \mathcal{E}^c[\![\texttt{fixST}\lambda x.e]\!]\rho$. *If* $v_m = \{\!\!\{\mathbf{in}_3 m\}\!\!\}$ *and* $r_m = m(\mathbf{in}_2\top)(0,0)\mathbf{yield}$ *and* $w_m(\sigma) = \mathbf{ext}\ \mathbf{on}_1(\mathbf{compose}\ r_m\sigma)$ *and* $v_e(\sigma) = \mathcal{E}^c[\![e]\!]\rho[x\mapsto w_m(\sigma)] = \{\!\!\{\mathbf{in}_3(m'(\sigma))\}\!\!\}$ *and* $r_e(\sigma) = (m'(\sigma))(\mathbf{in}_2\top)(0,0)\mathbf{yield}$ *and* $w_e(\sigma) = \mathbf{ext}\ \mathbf{on}_1(\mathbf{compose}\ (r_e(\sigma))\sigma)$ *then* $\forall\sigma.w_m(\sigma) = w_e(\sigma)$.

8. Related Work

Denotational treatments of concurrent programming constructs in imperative languages are discussed e.g. in the textbook [16] and in the handbook article [10]. They also use standard domain theory and resumptions. In our case, additional complications arise because state transformers not only have an effect on the store, but do compute a final value, too. The presence of the fixpoint operation forces us to consider both the history and the future of a computation. The history affects the semantics since it may have spawned interleaved computations. The future is respected in the semantics through the use of continuations.

A different approach to a denotational semantics for concurrency is pursued by de Bakker and others [3,2]. Their framework is not domain theory, but the theory of complete metric spaces where the Banach fixpoint theorem guarantees the existence of unique fixpoints of contractive functions. In [2] a denotational semantics for a parallel object-oriented language (with message passing) is given. The techniques employed there also make use of continuations and resumptions.

9. Conclusion

Although we have succeeded in giving a denotational semantics for concurrent state transformers which has some pleasant properties, there are some shortcomings:

1. The semantics is not particularly abstract. For example $\mathcal{E}^c[\![\texttt{readVar x}]\!]$ is different from $\mathcal{E}^c[\![\texttt{readVar x `thenST` \textbackslash z -> readVar x}]\!]$ although the expressions are operationally equivalent.
2. It is not clear whether it really models lazy state: reading or writing a variable in one thread correctly forces all preceding reads and writes in that thread to happen, but it also forces reads and writes in other threads that happen to be interleaved "in front of" the offending read/write operation.
3. It does not model the sequential case where `interleaveST` and `forkST` just initiate independent threads that proceed on demand only. Instead, our semantics models eager, but non-strict evaluation (similar to Id [11]). It could also serve as a semantics of concurrent computations in $\lambda_{\mathbf{var}}$ [12].

10. References

1. S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1, 1990.
2. P. America, J. W. de Bakker, J. N. Kok, and J. J. M. M. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83:152–205, 1989.

3. J. W. de Bakker and J. H. A. Warmerdam. Four domains for concurrency. *Theoretical Comput. Sci.*, 90:127–149, 1991.
4. C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, Cambridge, MA, 1992.
5. C. A. Gunter and D. S. Scott. *Semantic Domains*, volume B of *Handbook of Theoretical Computer Science*, chapter 12. Elsevier Science Publishers, Amsterdam, 1990.
6. Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):R1–R164, May 1992.
7. M. P. Jones and P. Hudak. Implicit and explicit parallel programming in Haskell. Research Report YALEU/DCS/RR-982, Yale University, New Haven, CT 06520-2158, Aug. 1993.
8. J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, Fla, USA, June 1994. ACM Press. ACM SIPLAN Notices, v29, 6.
9. J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 1995. to appear.
10. P. D. Mosses. *Denotational Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11. Elsevier Science Publishers, Amsterdam, 1990.
11. R. S. Nikhil. Id (Version 90.1) Reference Manual. CSG Memo 284-2, MIT Laboratory for Computer Science, Cambridge, MA, July 1991.
12. M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda calculus. In POPL1993 [15], pages 43–57.
13. S. L. Peyton Jones and P. L. Wadler. Imperative functional programming. In POPL1993 [15], pages 71–84.
14. G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5:452–487, 1976.
15. *Proc. 20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, Jan. 1993. ACM Press.
16. D. A. Schmidt. *Denotational Semantics, A Methodology for Software Development*. Allyn and Bacon, Inc, Massachusetts, 1986.
17. P. Thiemann. Safe sequencing of assignments in purely functional programming languages. Technical Report WSI-93-16, Wilhelm-Schickard-Institut, Tübingen, Germany, Nov. 1993.
18. P. Thiemann. Terminated references and automatic parallelization for state transformers. In U. S. Reddy, editor, *ACM SIGPLAN Workshop on State in Programming Languages*, San Francisco, CA, Jan. 1995. University of Illinois.
19. P. L. Wadler. Comprehending monads. In *Proc. Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, 1990. ACM.
20. P. L. Wadler. The essence of functional programming. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, Jan. 1992.