Reactive Objects in a Functional Language
An escape from the evil "I"

Johan Nordlander *Magnus Carlsson*

Department of Computing Science
Chalmers University of Technology
S - 412 96 Göteborg, Sweden
Email: {nordland,magnus}@cs.chalmers.se

Abstract

We present an extension to Haskell which supports reactive, concurrent programming with objects, *sans* the problematic blocking input. We give a semantics together with a number of programming examples, and show an implementation based on a preprocessor and a library implementing seven monadic constants.

1 Introduction

With the advent of Haskell 1.3 the monadic I/O model has become well established [PH96]. At the top level, a Haskell program is now a sequence of imperative commands that transforms a state consisting of the real world and/or some program state into a final configuration. In a pure state transformational approach, carrying a monolithic program state around is likely to complicate modular design; however, this problem can to a large extent be circumvented by introducing first-class references in the monadic framework [LJ94]. Taken together, these additions make the resulting Haskell programs — on the top level at least — more and more reminiscent of programs written in traditional imperative languages like Pascal.

Just like I/O in traditional languages, though, straightforward monadic I/O imposes a rather rigid structure on environment interaction. Especially, when a program says "read input", the outside world has nothing to do but to follow order, since otherwise execution is effectively stuck. Examples of this scenario are numerous, the average computer user is probably all too familiar with primitive programs that continuously alternate between telling the user what to do and reading back responses. It goes without saying that putting programs structured this way in the context of an inherently concurrent and nondeterministic environment is bound to be problematic.

In their POPL '96 paper, Peyton Jones, Gordon, and Finne identify the need for *semantically visible concurrency* to alleviate these problems [PGF96]. Their proposed language, Concurrent Haskell, allows multiple execution threads to be initiated, and provides a mechanism for synchronisation and communication between such threads via atomically-mutable state variables. Thus, when one thread blocks for input, another thread can be created that maintains the capability of interacting with the environment in alternative ways. As with the sequential I/O model of Haskell, a central property of Concurrent Haskell is a stratified semantics that limits non-determinism and state-manipulation to the top-level of a program, while leaving the purely functional semantics of expression evaluation unaffected.

Undeniably, the primitives offered by Concurrent Haskell are at a very low level. This choice has been deliberate, though, since the purpose of these primitives is really to act as basic building blocks in the construction of various higher-level abstractions. However, as far as mastering concurrency is concerned, this situation is not very much different from the challenge that faced Algol programmers equipped with pointers and semaphore libraries three decades ago [Dij65]. That is, while important properties like liveness and mutual exclusion certainly *can* be maintained in a Concurrent Haskell system, there is nothing in the language that imposes such a structure on programs. This is a rather paradoxical quality for a purely functional language, which has deliberately abandoned low-level features like assignments and jumps in order to impose good structure on ordinary, non-concurrent computations.

What one ideally would like to see is a language that, in the spirit of Concurrent Haskell, combines the virtues of explicit concurrency and unobstructed functional programming, but which also commits itself to some top-level form that can be found both abstract and flexible, as well as intuitively appealing. In this paper we make an experimental attempt towards this goal, by means of a Haskell extension based on the notion of *reactive objects*. As a slight nod to the object-oriented community, we call our extension O'Haskell.

2 Reactive Objects

Our approach to concurrency in O'Haskell starts out from an identification of a state and a process. This means that there can be no state-container without an associated thread of control in our language, neither can there be any control threads without associated local states. Such a state/process combination is called an *object*, and in the sequel we will use this term interchangeably with our notion of a process.

The code of an object is an expression in the monad `O s ()`, which is a refinement of Haskell's interaction monad

IO () into an *reacting state monad* with a state type s.[1] To improve the readability of expressions in this monad, we provide an extension to the do-syntax of Haskell 1.3 that allows the state of an object to be inspected and assigned in a Pascal-like manner.

The crucial difference between the IO monad and our replacement is now the following: whereas the value return () represents a terminated program/process in Haskell/Concurrent Haskell, we will interpret this value as an *inactive process* that just passively maintains its state. Such a process may indeed become runnable in the future, since the monadic code of an object can be extended with new fragments (using the standard operator ≫) by the reception of *messages.* These messages can be of two forms: either an asynchronous action, that lets the sender continue immediately and thus introduces concurrency, or a synchronous request, which allows a value to be passed back to the waiting sender. In both cases, the receiving object reacts by starting a reduction of its updated monadic expression, which will, in the absence of a non-terminating reduction, eventually restore the receiver to an inactive state again. A significant feature of this scheme is that the actions and requests of an object implicitly form a critical region.

Objects are introduced using a template, which defines the initial state of an object together with a *communication interface.* Executing a template expression in the O monad creates a new object that is an instance of the template, which is the O'Haskell equivalent of forking off a process. The following code fragment defines a template for a simple counter object:

```
newCounter =
    template
        val := 0
    with
        ( action
            val := val + 1
        , request
            return val
        )
```

In this case, the communication interface consists of an asynchronous action that increments the counter, and a synchronous request for reading its current value. The silly program below illustrates how a counter object is created and then passed an *increment* message, before its current value is requested. At the end of the sequence the new counter is actually forgotten, and will become garbage.

```
main = do
    (inc,read) ← newCounter
    inc
    one ← read
    return ()
```

Synchronous and asynchronous message sending are the only communication primitives offered by O'Haskell, and neither of these blocks the sender more than temporarily.[2] This means that indefinite blocking for input is confined to the *passive case only*, where *any* defined message may be received. As a consequence, an O'Haskell program cannot impose any order on the events it is set to handle, and the liveness of a system will be upheld by default. These are properties that we consider extremely important for the robustness of a concurrent system. We actually take the absence of blocking input commands as our definition of the term *reactive*, and consider it to be the most significant feature of our concurrency proposal. We will return to this issue several times in the coming sections.

[1]The absence of the letter I in the name of our monad is *not* coincidental!

[2]See section 6.1 for a further discussion on the validity of this statement.

3 Semantics

In our presentation of the formal semantics of O'Haskell we will assume the existence of an operational semantics for expression evaluation, with a small step reduction relation $\mapsto$. For the sake of completeness we give such a semantics in appendix A, but it should be kept in mind that the exposition that will follow does not depend on the actual choices made in this definition. In particular, our concurrency extension should be equally applicable to a call-by-value language.

3.1 Syntactic transformations

We begin by transforming away the explicit naming of state variables and the template syntax, as shown in figure 1. The translation is parameterised over a tuple-pattern p, which matches the state introduced by the closest enclosing template expression. If no template is in scope, p equals (). Only the interesting cases are shown, the translation extends trivially to the whole expression syntax. We do however restrict variables bound by let and \ to be distinct from the variables in p.

After translation, our concurrency extension is visible only in the presence of seven primitive constants, of which four are the common state monad operations recast to our O type:

```
return  ::  t → O s t
>>=     ::  O s t → (t → O s t') → O s t'
set     ::  s → O s ()
get     ::  O s s
```

As usual we write a ≫ b as a shorthand for a >>= _ → b.

The remaining three constants correspond to the action, request, and template constructs, respectively. These constants all involve a second built-in type Ref s, which is the type of primitive *reference* values that uniquely determine a particular object at run-time.

```
new  ::  s' → (Ref s' → t) → O s t
act  ::  Ref s' → O s' () → O s ()
req  ::  Ref s' → O s' t → O s t
```

References in O'Haskell bear some resemblance to Concurrent Haskell's MVars, although it should be observed that a reference is used to identify a process rather than some passive storage location or synchronisation point. This view is influenced by the Actors programming model, which regards the agents/processes as the sole identity-carrying entities during a computation [Agh86].

References are mostly accessed indirectly, however, via the actions and requests an object exports through its communication interface. The common case where an interface

$$\begin{array}{llll}
a, b & ::= & \ldots \mid \underline{\mathsf{do}}\ \bar{c} \mid \underline{\mathsf{template}}\ \overline{x := a}\ \underline{\mathsf{with}}\ b \mid \underline{\mathsf{action}}\ \bar{c} \mid \underline{\mathsf{request}}\ \bar{c} & \text{Expressions} \\
c & ::= & a \mid x \leftarrow a \mid x := a & \text{Commands}
\end{array}$$

$$\begin{array}{lcll}
[\![\ \underline{\mathsf{do}}\ c\ ]\!]_p & = & [\![\ c\ ]\!]_p & \\
[\![\ \underline{\mathsf{do}}\ x \leftarrow a\ ;\ \bar{c}\ ]\!]_p & = & [\![\ a\ ]\!]_p \ \texttt{>>=}\ [\![\ \backslash x \rightarrow \underline{\mathsf{do}}\ \bar{c}\ ]\!]_p & \\
[\![\ \underline{\mathsf{do}}\ c\ ;\ \bar{c}\ ]\!]_p & = & [\![\ c\ ]\!]_p \ \texttt{>>}\ [\![\ \underline{\mathsf{do}}\ \bar{c}\ ]\!]_p & \\
 & & & \\
[\![\ \underline{\mathsf{template}}\ \overline{x := a}\ \underline{\mathsf{with}}\ b\ ]\!]_p & = & [\![\ \mathsf{new}\ (a_1, \ldots, a_n)\ \backslash\mathsf{self} \rightarrow b\ ]\!]_{(x_1, \ldots, x_n)} & \\
 & & & \\
[\![\ \underline{\mathsf{action}}\ \bar{c}\ ]\!]_p & = & \mathsf{act\ self}\ [\![\ \underline{\mathsf{do}}\ \bar{c}\ ]\!]_p & \\
[\![\ \underline{\mathsf{request}}\ \bar{c}\ ]\!]_p & = & \mathsf{req\ self}\ [\![\ \underline{\mathsf{do}}\ \bar{c}\ ]\!]_p & \\
 & & & \\
[\![\ x := a\ ]\!]_p & = & [\![\ \mathsf{set}\ p[a/x]\ ]\!]_p & \text{if } x \in \textsc{fv}(p) \\
[\![\ a\ ]\!]_p & = & \mathsf{get}\ \texttt{>>=}\ \backslash p \rightarrow [\![\ a\ ]\!]_p & \text{if } \textsc{fv}(a) \cap \textsc{fv}(p) \neq \emptyset \\
[\![\ a\ ]\!]_p & = & [\![\ a\ ]\!]_p & \text{otherwise}
\end{array}$$

Figure 1: Translation of syntactic sugar

completely hides the existence of its underlying reference value is therefore directly supported by our syntactic sugar, by the implicit binding of a variable self in the body of a template. This corresponds to what John Reynolds calls *procedural abstraction*, where information hiding is achieved by partially applying an access procedure to the structure that needs protection [Rey94].[3]

The counter example discussed in the section 2 looks as follows in its desugared variant:

```
newCounter =
    new 0 \self ->
        ( act self
            (get >>= \val -> set (val+1))
        , req self
            (get >>= \val -> return val)
        )

main =
    newCounter >>= \(inc,read) ->
    inc >>
    read >>= \one ->
    return ()
```

3.2 Dynamic semantics

We now turn to the dynamic aspects of O'Haskell. The stratified semantics approach that is an important part of Concurrent Haskell will be repeated here, although the actual language we define will of course be different. We have tried, though, to use a formulation that as far as possible follows the presentation of Concurrent Haskell, in order to simplify comparison.

First we need to specify that >>=, act, and req are strict in their first argument, and that >>= reduces in the usual monadic manner. We do this by extending appendix A with further evaluation contexts and an evaluation rule.

$$\begin{array}{lll}
\mathcal{E} & ::= & \ldots \mid \mathcal{E}\ \texttt{>>=}\ a \mid \mathsf{act}\ \mathcal{E} \mid \mathsf{req}\ \mathcal{E} \\
\textsc{Bind} & & \mathsf{return}\ a\ \texttt{>>=}\ b \ \mapsto\ b\ a
\end{array}$$

Next we define a small language of process terms, that will enable us to capture the state of a complete O'Haskell system.

$$\begin{array}{llll}
P & ::= & a_n^b & \text{Atomic process (object)} \\
 & \mid & P \parallel P' & \text{Parallel composition} \\
 & \mid & \nu n.P & \text{Reference generation}
\end{array}$$

The atomic process a_n^b corresponds to our notion of an object referenced by n, executing a monadic expression a in the state b. This form is restricted by the requirement that if $b :: s$, then $n :: \mathsf{Ref}\ s$ and $a :: \mathsf{O}\ s\ ()$. Furthermore, we require that no pair of objects are tagged with the same reference n. Our reaction rules introduced below all obey these restrictions.

Following the polyadic π-calculus we also adopt the *chemical solution* metaphor, which uses a structural congruence relation $\equiv$ to abstract away from syntactical differences between equivalent process terms [Mil91, BB90]. Using this metaphor we may safely assume that any pair of objects in a system that are willing to interact can be brought together syntactically (as if they were molecules floating around in a chemical solution). The definition of $\equiv$ from [Mil91] is immediately applicable to our process terms; we include the relevant rules in appendix B.

We are now ready to define how a solution of processes may evolve. This is captured by means of a *reaction relation* $\rightarrow$ between process terms. Thanks to the generality of the chemical framework we only have to specify the axioms of $\rightarrow$; a non-deterministic relation between arbitrary complex process terms is automatically obtained by incorporating the structural reaction rules of the π-calculus (recapitulated in appendix C). In the definition of $\rightarrow$ we sometimes take the liberty of matching against atomic processes using a simplified pattern a_n; this should be interpreted as an assertion

[3]Reynolds's contrasting notion, *type abstraction*, would mean exposing the object reference and its state type in the interface and then encapsulating that knowledge within some scope by means of an existential type.

$$\mathcal{M} \quad ::= \quad [\,] \mid \mathcal{M} \mathbin{\mathsf{>\!\!>\!=}} a$$

$$\begin{array}{lrcl}
\textsc{Eval} & a_n & \rightarrow & b_n \qquad \text{if } a \mapsto b \\
\\
\textsc{Set} & \mathcal{M}[\mathsf{set}\ b]_n^a & \rightarrow & \mathcal{M}[\mathsf{return}\ ()]_n^b \\
\textsc{Get} & \mathcal{M}[\mathsf{get}]_n^a & \rightarrow & \mathcal{M}[\mathsf{return}\ a]_n^a \\
\\
\textsc{New} & \mathcal{M}[\mathsf{new}\ a\ b]_m & \rightarrow & \nu n.(\mathcal{M}[\mathsf{return}\ (b\ n)]_m \parallel (\mathsf{return}\ ())_n^a) \\
 & & & \qquad \text{where } n \notin \text{FN}(\mathit{lhs}) \\
\\
\textsc{Ego} & \mathcal{M}[\mathsf{act}\ m\ a]_m & \rightarrow & (\mathcal{M}[\mathsf{return}\ ()] \mathbin{\mathsf{>\!\!>}} a)_m \\
\textsc{Act} & \mathcal{M}[\mathsf{act}\ n\ a]_m \parallel b_n & \rightarrow & \mathcal{M}[\mathsf{return}\ ()]_m \parallel (b \mathbin{\mathsf{>\!\!>}} a)_n \\
\\
\textsc{Req} & \mathcal{M}[\mathsf{req}\ n\ a]_m \parallel b_n & \rightarrow & \mathcal{M}[\mathsf{\_syn}]_m \parallel (b \mathbin{\mathsf{>\!\!>}} a \mathbin{\mathsf{>\!\!>\!=}} \mathsf{\_rep}\ m)_n \\
\textsc{Reply} & \mathcal{M}[\mathsf{\_syn}]_m \parallel \mathcal{M}'[\mathsf{\_rep}\ m\ a]_n & \rightarrow & \mathcal{M}[\mathsf{return}\ a]_m \parallel \mathcal{M}'[\mathsf{return}\ ()]_n
\end{array}$$

Figure 2: Semantics of reaction

that the state component of process n is the same on both sides of a rule. Figure 2 shows the axioms of $\rightarrow$.

The first rule, Eval, connects the semantics of reaction with the semantics of expression evaluation. This rule will in particular enable reduction of monadic expressions that are redexes according to the Bind rule above. However, Bind only allows a return expression on its left hand side, which is necessary to preserve the semantics of $\mapsto$. The remaining constants in the O monad must be dealt with directly by $\rightarrow$. We therefore define the notion of a *reaction context* $\mathcal{M}$ to single out the head of a sequence of >>= applications in an atomic process. Using $\mathcal{M}$, the Set and Get rules implement the standard semantics of a state monad.

Process creation is defined by New. Note that new processes are born in an inactive state. Axiom Ego captures the case where the sender and the receiver of an asynchronous message are identical, while the general case is handled by Act. Since arriving code fragments are appended to the receiver irrespective of its current activities, our semantics actually specifies a *message buffer* for each object.

The final and most complex case is synchronous communication, which is defined in terms of two internal constants, _syn and _rep, that are not accessible to the programmer. Req differs from Act by the attachment of _rep to the message sent, and by putting the sender in a blocking state. Rule Reply will subsequently be applicable to resolve this situation, provided that the receiver never loops indefinitely and never sends a req back to the waiting process. The impact of these preconditions, and their relevance to our claim that O'Haskell supports liveness by default is discussed in section 6.1.

Direct programmer access to _syn and _rep would be problematic for at least the following reasons: (1) both constants are hard to type in a way that is sound in general, and (2) unrestricted use of _syn would immediately destroy the reactive character of our language. Note also that req is not redundant; it cannot be faithfully simulated by asynchronous messages in two directions, since the original sender has no means of filtering out a reply before handling other messages that might be pending. What req actually offers is a very restricted, but convenient form of input, whose termination *does not depend on any external events that have yet to occur.* This stands in contrast to the *read* primitive of most traditional languages, which returns only at the will of a potential user.

We exemplify our semantic rules by showing how the desugared counter example of section 3.1 reduces. The program environment, which is supposed to initiate execution, is modelled as a process m with an empty state () and an initial monadic expression main. To reduce clutter, we only write out ν-binders where they are introduced, and skip trivial applications of rules Eval and Bind. For the same reason, some eager evaluation is also implicitly performed.

$\nu m.\mathsf{main}_m$
$\rightarrow$ Eval (1)
$(\mathsf{newCounter} >\!\!>\!= (\mathsf{inc},\mathsf{read}) \rightarrow \ldots)_m$
$\rightarrow$ Eval (2)
$(\mathsf{new}\ 0\ \backslash\mathsf{self} \rightarrow (\mathsf{act\ self}\ \ldots,\mathsf{req\ self}\ \ldots) >\!\!>\!= \ldots)_m$
$\rightarrow$ New (3)
$\nu n.(\mathsf{return}\ ((\backslash\mathsf{self} \rightarrow \ldots)\ n) >\!\!>\!= \ldots)_m \parallel (\mathsf{return}\ ())_n^0$
$\rightarrow$ Eval (4)
$(\mathsf{act}\ n\ (\mathsf{get} >\!\!>\!= \ldots) >\!\!> \mathsf{req}\ \ldots)_m \parallel (\mathsf{return}\ ())_n^0$
$\rightarrow$ Act (5)
$(\mathsf{req}\ n\ \ldots)_m \parallel (\mathsf{get} >\!\!>\!= \backslash\mathsf{val} \rightarrow \mathsf{set}\ ((\mathsf{val}+1)))_n^0$
$\rightarrow$ Get (6)
$(\mathsf{req}\ n\ \ldots)_m \parallel (\mathsf{set}\ (0+1))_n^0$
$\rightarrow$ Set (7)
$(\mathsf{req}\ n\ (\mathsf{get} >\!\!>\!= \ldots) >\!\!>\!= \ldots)_m \parallel (\mathsf{return}\ ())_n^1$
$\rightarrow$ Req (8)
$(\mathsf{\_syn} >\!\!>\!= \ldots)_m \parallel (\mathsf{get} >\!\!>\!= \ldots >\!\!>\!= \mathsf{\_rep}\ m)_n^1$
$\rightarrow$ Get (9)
$(\mathsf{\_syn} >\!\!>\!= \ldots)_m \parallel (\mathsf{return}\ 1 >\!\!>\!= \mathsf{\_rep}\ m)_n^1$
$\rightarrow$ Eval (10)
$(\mathsf{\_syn} >\!\!>\!= \ldots)_m \parallel (\mathsf{\_rep}\ m\ 1)_n^1$
$\rightarrow$ Reply (11)
$(\mathsf{return}\ 1 >\!\!>\!= \backslash\mathsf{one} \rightarrow \mathsf{return}\ ())_m \parallel (\mathsf{return}\ ())_n^1$
$\rightarrow$ Eval (12)
$(\mathsf{return}\ ())_m \parallel (\mathsf{return}\ ())_n^1$

To illustrate the non-determinism inherent in the reaction relation we give an alternative reduction sequence for steps 7-8 above, which also shows buffering of a message to an active object.

$$\begin{array}{lr}
\quad\vdots & (6)\\
(\mathsf{req}\ n\ (\mathsf{get}\ \gg\!=\ \ldots)\ldots)_m\ \ ||\ \ (\mathsf{set}\ (0{+}1))^0_n & \\
\to \textsc{Req} & (7)\\
(\mathsf{\_syn}\ \gg\!=\ \ldots)_m\ \ ||\ \ (\mathsf{set}\ (0{+}1)\ \gg\ \mathsf{get}\ \gg\!=\ \ldots)^0_n & \\
\to \textsc{Set} & (8)\\
(\mathsf{\_syn}\ \gg\!=\ \ldots)_m\ \ ||\ \ (\mathsf{get}\ \gg\!=\ \ldots)^1_n & \\
\quad\vdots &
\end{array}$$

We have not developed any theory around our semantics so far, although this would certainly be an interesting research project. The similarities between our formulation and the presentation of Concurrent Haskell are so strong, however, that we expect the main results about the latter language to directly carry over to our case. This includes the important property that the deterministic, purely functional semantics of expression evaluation ($\mapsto$) is not affected by its inclusion into a non-deterministic, imperative context ($\to$).

4 Typing issues

Like monadic operators in general, actions, requests, and templates are first-class values in O'Haskell. This means, intuitively, that it should be possible to send a communication interface along from one object to another, to obtain dynamically evolving communication patterns. Unfortunately, the typings given to our primitive constants given in the previous section severely limit this possibility. The problem has to do with instantiating the state component of the O monad. Recall that for an action value this type stands for the internal state of the *sender*, so the problems we encounter are not an indication of insufficient object encapsulation. Still, the state component is necessary in general, since it ensures that all code fragments of an object read and assign to the same type of state. What is problematic is that all interesting action values in O'Haskell will be lambda-bound, and thus subject to monomorphic instantiation only. The net result is that communication interfaces can only be shared between objects that have the same internal state type!

There might be several ways out of this dilemma, including the first-class structures proposed by Mark Jones [Jon96], or the use of explicit, polymorphic coercion functions that get the static typings right but behave as the identity function at runtime. We have found, though, that the object-oriented concept of *subtyping* solves the problem very neatly, and we will present our program examples in the next section using the alternative typings that this solution gives rise to.

Our approach to subtyping in polymorphic languages is covered in detail in [Nor97]; for the purposes of this paper it suffices to know that subtyping relations between type constants must be explicitly declared by the programmer, via *polymorphic subtype axioms.* In this spirit, our concurrency primitives can be given more precise types if the following types and subtype axioms are provided as built-in:

```
Template t   <   O s t
    Action   <   O s ()
 Request t   <   O s t
```

The alternative typings for new, act, and req now become

```
new   ::   s -> (Ref s -> t) -> Template t
act   ::   Ref s -> O s () -> Action
req   ::   Ref s -> O s t -> Request t
```

We argue informally that this refinement is sound, since none of the constants above reads or writes the local state of its executing object. Hence, the narrow types we have introduced may be safely promoted to the state monad O s, for any state s. What the subtyping axioms allow us to do is to choose this s anew *at each occurrence* of a particular constant.

Struct-like types still have their place in O'Haskell, though, because of their natural correspondence to the idea of a communication interface. They also fit quite nicely into our subtyping machinery, although we will not exploit this aspect here. Appendix D contains an extension of core Haskell with struct values that we will rely on in the next section. Taking both structures and subtypes into account, here is (finally) our preferred formulation of the by now well-known counter program:

```
struct Counter =
    inc    :: Action
    read   :: Request Int

newCounter :: Template Counter
newCounter =
    template
        val := 0
    with struct
        inc = action
            val := val + 1
        read = request
            return val

main = do
    c <- newCounter
    c.inc
    one <- c.read
    return ()
```

5 Examples

Classical examples used as a test-bed for new concurrency proposals are mainly concerned with *coordination problems*, e.g. how shared, mutable data can be protected from concurrent access, and *communication problems*, e.g. how buffers, channels, and the like can be built from available primitives. However, these problems often have less relevance to the class of message-based languages where O'Haskell belongs. So, let us from the outset emphasise that O'Haskell directly supports the following basic needs:

- **Critical sections.** A shared data structure is equivalent to an object in our model, and the actions and requests an object provides are mutually exclusive. Thus, the code fragments that constitute these services can be considered as automatically protected critical sections.
- **Message buffering.** Our semantics for message sending already provides unbounded buffering of messages.

This means that there is no need to implement separate queuing mechanisms when the ordering of messages sent must be preserved. We believe that the vast majority of buffer applications used in existing concurrent systems fits into this many-to-one communication pattern.

- **A signal system.** The use of signals to synchronise various events in a system is greatly simplified in our model, since actions have first-class status, and objects are only temporarily in a state where they cannot respond to any input.

Other, more specific communication and synchronisation requirements may of course occur in practice, but our experience with the language so far suggests that the programming style enforced by O'Haskell is surprisingly flexible, and that problems which initially seem to call for both blocking input commands and shared state variables often benefit from a reactive reformulation. Subsections 5.2 and 5.3 describe how some classical synchronisation facilities can be encoded in O'Haskell. We will, however, begin this section at the other end of the spectrum, by sketching a framework for a high-level application: an event-driven, interactive program with a graphical user interface.

5.1 Event-driven, interactive programs

Graphical user interfaces are standard on today's personal computers, and with them has come a style of programming that can be characterised as *event-driven*. In traditional languages, this style consists of structuring a program around an *event-loop*, which repeatedly does a blocking system-call to get the next event-structure, performs case-analysis on this structure, and then executes one of its branches depending on the actual event received. In the normal case, these branches never perform any blocking operations themselves; all input to the program is concentrated to the top of the event-loop.

Evidently, O'Haskell natively supports this programming style. From the view of the operating system, an application is nothing more than a process that responds to a certain set of events/messages, and the application, in turn, just sees the operating system as a process with another, specific communication interface. A (greatly simplified) specification of these interfaces might look like this:

```
struct GUIApp =
    redraw :: Action
    click  :: Point → Action
    drag   :: Point → Action
    key    :: Char → Action

struct Window =
    size   :: Request Point
    resize :: Point → Action
    clear  :: Action
    line   :: Point → Point → Action

someGUIApp :: Window → Template GUIApp
```

Here, someGUIApp can be a template for any kind of interactive program; the only thing an operating system (which is supposed to instantiate the template) needs to know is that the resulting process supports the GUIApp interface, and the template implementor, in turn, must only assume that there will be some Window interface available at runtime, where drawing commands can be sent. We like to call this division of responsibilities *communication by contract*.

So far the normal case. But what about exceptions to this scheme, for example when a graphical application needs to present a dialog box, or a popup menu, which has to consume all input until it closes? Is this not a situation where a blocking input abstraction would be helpful?

A reactive approach to this problem needs to make a clear distinction between opening a popup menu, say, and reacting on its completion. The former activity is naturally modelled as sending a message, while the latter part is equivalent to the *arrival* of a message. Since actions are first-class values, the code that creates a popup menu object could easily be parameterised with respect to the actions that handle selection. So, if we assume that the service of creating and opening a popup menu is available through some WindMgr interface, and that the click-handling code of our graphical application asks for this service, a reactive solution could follow this outline:

```
struct WindMgr =
    ...
    popup :: Point → [(String,Action)] → Action

...
    click pt = action
        windmgr.popup pt
            [ ("Foo",action A)
            , ("Bar",action B) ]
```

The important thing to notice here is that code fragments A and B above are only lexically within the scope of click — there is no "subroutine" relationship implied, and the execution of click will be over as soon as the popup message is sent. In due course of time, one of the actions at A or B will be executed, provided that the user chooses to select anything at all from the menu. A nice consequence of this reactive scheme is that our graphical application will still be able to respond to any message directed to it while the popup menu is open. This is really as it should be: there is no reason why, for example, redrawing a window should be postponed just because there is a special form of graphical input device open in front of it.

5.2 Semaphore encoding

A semaphore is a synchronisation device that in its simplest form is just a "token" that can be claimed and released, and where the claiming operation may block if the token is not available [Dij65]. According to our definition, a semaphore is clearly not a reactive abstraction, but as we have mentioned, O'Haskell provides both synchronisation and communication by more abstract means.

However, there might be situations where the implementation of synchronisation mechanisms is actually a part of the programming problem, for example in the simulation of a railway system. Hence it might be necessary to encode semaphores reactively, and we need to show how this can be done.

The key step towards a reactive implementation is to lift out the client code that is supposed to run after a successful claim, and put it into a separate action. Then the responsibility for triggering this action can be put on the semaphore, quite similar to the *continuation passing style* sometimes used in functional languages to encode stateful computations.

Here is the semaphore implementation:

```
struct Semaphore =
    claim   :: Action → Action
    release :: Action

semaphore :: Template Semaphore
semaphore =
    template
        active := False
        wakeup := []
    with struct
        claim grant = action
            if not active then
                active := True
                grant
            else
                wakeup := wakeup ++ [grant]
        release = action
            case wakeup of
                []      → active := False
                w:ws    → wakeup := ws; w
```

One can imagine many variations on this theme, e.g. letting claim be a boolean request that returns True in the successful case instead of triggering grant. This would allow a client to take special action if the claim is granted immediately.

5.3 Queue encoding

An alternative to the many-to-one buffering mechanism inherent in O'Haskell would be a many-to-many communication scheme, where data is communicated via a distinguished *queue* process. In order to implement such a queue in O'Haskell, however, we need to reconsider the *remove* operation, since it is supposed to block if no data is available. The reactive way of removing an item must be a two-phase operation: first a consumer *announces* its readiness to receive some data, then, perhaps later on when data has arrived, the queue process sends a message with the data to the consumer process. The code looks as follows:

```
struct Queue a =
    insert      :: a → Action
    announce    :: (a → Action) → Action

queue :: Template (Queue a)
queue =
    template
        packets := []
        servers := []
    with struct
        insert p = action
            case servers of
                []      → packets := packets ++ [p]
                s:ss    → servers := ss; s p
        announce s = action
            case packets of
                []      → servers := servers ++ [s]
                p:ps    → packets := ps; s p

producer q =
    template
        ...
    with let produce = action
                x ← ... produce an x
                q.insert x
                produce
    in produce

consumer q =
    template
        ...
    with let consume x = action
                ... x ... consume x
                q.announce consume
    in q.announce consume

main = do
    q   ← queue
    p1  ← producer q
    p2  ← producer q
    c1  ← consumer q
    c2  ← consumer q
    p1; p2; c1; c2
```

Note that this "polarity switch" of the remove operation does not really clutter up the code; it is still as symmetrical as it would be in a language with blocking read operations. Turning the queue into a bounded buffer would mainly just require switching the polarity of the insert operation as well. The real benefit of this reactive encoding is that all processes involved can easily be extended to handle additional messages, without affecting the basic communication pattern.

5.4 Interrupt Service Routines

As an example of how the concept of a *hardware interrupt* fits into the reactive style, we give an implementation of a timer process, that allows its clients to "sleep" for a specified number of ticks. Sleeping should be interpreted reactively, however, meaning that a process passively awaits any message, one of which will signal that a certain amount of time has passed. This action must be supplied as a parameter to the timer each time a timing task is started. We do not specify the communication interface for the timer process as a struct value, since one of its actions, tick, is only meant to be installed in some interrupt vector table, and the other, start, should preferably be made available through some general, operating-system-like interface that we do not wish to consider any further.

```
newTimer =
    template
        time := 0
        pend := []
    with let
        start t sig = action
            pend := insert (time+t,sig) pend
    check_pending = do
        case pend of
            (t,sig):pend' | time >= t ->
                pend := pend'
                sig
                check_pending
            _ -> done

    tick = action
        time := time + 1
        check_pending

in (start, tick)
```

A notable feature of this example is the use of a recursive procedure in the interrupt service routine tick. Such a call is completely local to a process, and cannot be interspersed with other messages. Compare this with the recursive message sending that is performed by the producer processes of section 5.3.

It is interesting to see that safe communication between an interrupt service routine and an ordinary process (a rather tricky task in most programming languages) can be handled with the same mutual exclusion machinery that is used in ordinary interprocess communication. The reason behind this is that, in effect, *all* messages are modelled as interrupts in O'Haskell, and it does not matter whether some of them are actually generated by hardware.

5.5 A telnet client

We conclude this section with a slightly larger example: a rudimentary implementation of a Telnet client. The resulting code has a very appealing structure, centered around the intuitive fact that the state of a Telnet process is its current connection. We have not implemented any Telnet-specific handshaking, though, since that would just mean repeating the pattern used in the handling of open/close acknowledgements.

```
struct Connection =
    send       :: Packet -> Action
    close      :: Action
    peer       :: Request Host

struct Client =
    connected :: Connection -> Action
    deliver    :: Packet -> Action
    closed     :: Action

struct Tcp =
    open       :: Host -> Port -> Client -> Action
    listen     :: Port -> Client -> Action

struct Telnet =
    connect    :: Host -> Action
    keypress   :: Char -> Action
    disconnect :: Action
```

```
telnet :: Tcp -> Screen -> Template Telnet
telnet tcp screen =
    template
        you_server := Nothing
    with let me_client = struct
                connected c = action
                    you_server := Just c
                    screen.puts "[Connected]\n"
                deliver pkt = action
                    screen.putc (mkchar pkt)
                closed = action
                    you_server := Nothing
                    screen.puts "[Disconnected]\n"
    in struct
        connect host = action
            tcp.open host telnet_port me_client
        keypress ch = action
            case you_server of
                Just c      -> c.send (mkpkt ch)
                Nothing     -> screen.beep
        disconnect = action
            case you_server of
                Just c      -> c.close
                Nothing     -> done
```

The type definitions used in this example clearly demonstrate the principle of *communicating by contract*. Notice how the use of a connection is effectively prohibited until it is established (by splitting the interface to Tcp into two struct types). Note also how the Telnet process exhibits two different interfaces at the same time, depending on which level in the protocol hierarchy it is seen from. Finally, one interesting consequence of the reactive implementation is actually visible in the body of keypress, where the user is notified by a beep if characters are entered before a reliable connection has been established. This would not have been so straightforward in a language that had modelled tcp.open as a blocking input operation.

6 Discussion

6.1 Liveness

The decision to abolish blocking input commands is a radical one, but as we have seen, the program structure that emerges as a result has some interesting merits. As we have mentioned, it keeps programmers from imposing any order on the events that a program is set up to handle, and it makes the important liveness property hold by default.

The latter property needs some clarification. What we really would like to state is that every active object in an O'Haskell system is guaranteed to react to any message in a finite amount of time, but clearly there are some preconditions that must hold for such a statement to be true.

Firstly, an object that is stuck in a cycle of objects blocked on synchronous requests to each other will never react to any more messages. However, this is a detectable exception on the same level as division by zero, and one which is highly unlikely to occur in practice due to the relative sparseness of requests that we anticipate in real programs.

Secondly, an object which calls a non-terminating recursive procedure will obviously not be able to proceed. But this is an instance of the problem of guaranteeing termination in general, and we see no reason to treat the O type specially in this respect. It is worth noting here that the liveness of an O'Haskell system does not depend on any *non-terminating* properties that must be proved (c.f. [HPS96]).

With these observations in mind, we feel justified in claiming that an O'Haskell program upholds the liveness property *by default*, that is, liveness holds trivially for all processes unless the programmer constructively destroys it by writing inherently erroneous code. It is our belief that O'Haskell in this sense is less sensitive to programming mistakes, than a language where the liveness property crucially depends upon active cooperation from the programmer.

6.2 Preserving message ordering

Our choice to specify that messages should be queued is also worth some comments. Common practice in concurrent languages is to leave this issue unspecified, in order to facilitate distributed implementations across an unreliable network [PGF96, Agh86]. We base our decision on the following arguments:

- Many simple programs would be unduly complicated if message ordering was not preserved (c.f. the counter example in section 2).
- Processes on an unreliable network can be conveniently accessed via a library of local *proxy* objects. These proxies can present a reliable or an unreliable connection, at the free choice of the implementor. Implementing a proxy is greatly simplified, however, if communication with the local clients is order preserving.
- The only operation which cannot be faithfully simulated by a proxy is the synchronous request. Thus, a network object will need to have a more limited interface than a corresponding local one. This is a necessary restriction if we consider liveness to be important, and blocking input to be harmful. On the other hand, if the network in question is considered so reliable that a blocking input operation really would have done no harm, then the network can equally well be made a part of the language implementation, since it guarantees the language semantics.

7 Implementation

Since O'Haskell is defined as an extension to ordinary Haskell, it has turned out to be straightforward to build a compiler for O'Haskell by performing just the syntactic transformations in figure 1, and providing the seven primitive constants of the O monad in a library module. We actually have two implementations of this module at the moment; one being written in pure Haskell using a datatype with constructors for each primitive. This has allowed us to program the reaction semantics in figure 2 by pattern matching, but has resulted in a rather inefficient implementation. In addition, not all semantic rules can be typed in pure Haskell, which we have circumvented using explicit type casts.

It turns out that a more pragmatic implementation, that also uses concurrent evaluation, can actually be built from the primitives provided by Concurrent Haskell. These primitives are available in Lennart Augustsson's eminent 2nd generation Haskell compiler (HBCC) [Aug96], which also provides "real", first class structures à la Mark Jones, to our delight. This implementation is short and is included below, as it might help the Concurrent Haskellate reader in understanding certain aspects of O'Haskell's reactive semantics.

To make a complete comparison between the two languages, we also provide a translation from Concurrent Haskell into O'Haskell, which can be found in appendix E.

7.1 O'Haskell in Concurrent Haskell

The type of objects is built on top of the IO monad, and supplies commands with a mutable variable that holds the local state.

```
type O s t     = IOVar s -> IO t
```

The type O is recognised as the standard reader monad.[4]

```
returnO a      = \_ -> return a
m `bindO` f    = \v -> do a <- m v
                          f a v
```

The state-related operations boils down to accessing and manipulating the mutable variable supplied to the commands.

```
get            = \v -> readIOVar v
set s          = \v -> writeIOVar v s
```

The type of object references encapsulates a message queue (called *channel* in [PGF96]) of commands to be executed by the object.

```
type Ref s     = Chan (O s ())
```

Sending an asynchronous action to an object implies writing the command to this object's channel.

```
act r c        = \_ -> putChan r c
```

The synchronous request is a little more involved, here we create a temporary MVar to mediate the answer.

```
req r c        = \_ -> do ans <- newMVar
                          putChan r
                             (\v -> do a <- c v
                                       putMVar ans a)
                          takeMVar ans
```

Each object has an associated server thread which forever reads commands from the object channel and executes them.

```
objproc        :: IOVar s -> Chan (O s ()) -> IO ()
objproc v r    = do c <- getChan r
                    c v
                    objproc v r
```

The last primitive to define is new, which creates a fresh mutable variable for the state, and a new command queue. Finally, new forks off a server thread for the new object.

```
new s iface    = \_ -> do v <- newIOVar s
                          r <- newChan
                          forkIO (objproc v r)
                          return (iface r)
```

[4]To avoid any confusion regarding overloading, we let O'Haskell's basic monad operations be denoted by returnO and bindO in this subsection.

8 Related work

Much work has been done in extending functional languages with concurrency features. Within the lazy functional community, *stream-based* approaches have a fairly long tradition [Sto86, Tur87, Hen82, HC95]. A common characteristic of these solutions is that communication is directed towards a particular *process*, and that all input streams for a process are merged into one before reception. This means that stream-based processes do not constrain the order in which different messages are received, a feature quite similar to the reactive property of our proposal. The drawback of the stream-based school is that it requires a rather heavy use of disjoint sum types in the merging of messages, and that the coupling between output and input is very loose, thus ruling out synchronous constructions like our requests.

The connection between a stream processor and an O'Haskell object can be illustrated by considering the type SP (Either a b) (Either c d) (taken from the *Fudgets* library for concurrent programming in Haskell [HC95]). This type stands for a stream processor which reacts to messages of type a or b by outputting messages of type c or d. Such a stream processor is naturally modelled in O'Haskell as a template parameterised over a pair of output actions, presenting an interface with the input actions:

$$(c \rightarrow \mathsf{Action}, d \rightarrow \mathsf{Action}) \rightarrow$$
$$\mathsf{Template}\ (a \rightarrow \mathsf{Action}, b \rightarrow \mathsf{Action})$$

A different approach is to separate the concept of a process and a communication destination. This is the common view in most process calculi, and it has been adapted in many functional languages as well [PGF96, Sch95, Rep92, Car86, Hol83]. An immediate benefit of these systems is that most message typing problems disappear, even for very complex communication patterns. However, since a *channel* (as we might collectively call the passive communication points) must be addressed explicitly in both the send and receive operations, the ability to simultaneously wait for any kind of message is lost. One remedy is to introduce the *choice* operator known from process calculi [Mil91], but its complexity generally makes it hard to implement efficiently, and great care must also be taken to avoid loss of abstraction when it is used [Rep92]. Choice-free encodings of object-like structures using multiple threads and locks are described in [PT94], while Concurrent Haskell seems to assume tagging by means of a datatype in its encoding of iterated choice [PGF96]. What O'Haskell offers is direct support for reactive reception of synchronous and asynchronous messages of multiple types, without the need for multiplexing by tagging, or coordination by additional processes.

Our unified view of objects as processes stems from the Actor model [Agh86]. Like objects in this model, an O'Haskell object can basically do three things in reaction to a message: (1) send messages, (2) create objects, and (3) update its local state. In the Actor model, these activities all occur in parallel, and reaction is thus atomic. We allow a *sequence* of operations in each action body, partly because this blends better with the monadic framework, and partly because we consider sequential imperative programming to be quite natural and something we wish to support. Furthermore, while state change in the pure Actor model is equivalent to specifying a replacement behaviour, we rely exclusively on state variables, which has the desired effect of making the state of an object decoupled from the set of messages it is willing to receive.

Erlang and UFO are two functional languages that are also influenced by the Actor model [AVWW96, Sar93]. Erlang endorses a sequential view of processes like we do, but relies on tail calls to keep a process alive, and utilises an untyped, Prolog-like pattern-matching mechanism to specify a current set of accepted messages. UFO is typed and has encapsulated state variables, but the object/process correspondence is vague and sequencing comes from data dependence only. Message acceptance can furthermore be restricted by manipulating predefined pseudo-variables. Neither of these languages can be called reactive in our meaning of the word.

An interesting characterisation of concurrency proposals for functional languages is of course whether they are able to preserve a pure semantics of expression evaluation. Of the works cited above, [HC95, Tur87, Sto86, PGF96, Sch95, Hol83] belong to this category, while [Hen82, Rep92, Car86, AVWW96, Sar93] do not. As we have mentioned before, retaining an unobstructed evaluation semantics has been one of the primary goals in the design of O'Haskell.

There is a host of concurrent object-oriented languages that might relate to O'Haskell in the sense that they support both concurrency and some typical object-oriented features (see e.g. [AYW93]). We are not yet ready to comment on these in any detail; however, it seems to us that many concurrent object-oriented designs are mostly concerned with maximising parallelism, in order to achieve good performance on massive multiprocessors [Pap92]. This might be the reason why it is hard to find any well-developed notion of functions in these proposals. Like the designers of Concurrent Haskell, we take the standpoint that performance-enhancing parallelism should be *semantically transparent*, a task for which pure functional languages are ideally suited [Ham94].

Objects in O'Haskell bear some resemblance to the *monitor* concept developed by Hoare [Hoa74]. Monitors were introduced to automatise the typical pattern where a semaphore is always released by the same process that has claimed it, as a means of protecting a critical section. The other distinct use of a semaphore, as identified by Dijkstra in [Dij68], feeds input to a particular process by means of a counting semaphore which is exclusively claimed by the receiver. This latter requirement is handled in a monitor by explicitly managed *condition variables*, something which significantly complicates the monitor semantics. We might say that our objects generalise the ordinary use of a monitor to cater for the input-feeding needs as well.

Reactive is a term coined by Pnueli and further used by Berry and Benveniste to describe systems that maintain an interaction with their environments [Pnu86, BB91]. The word is also sometimes seen as a synonym for the *synchronous* approach to *deterministic* reactive programming advocated by the latter authors. Although execution of an O'Haskell program is neither deterministic nor synchronised by a global clock, we think that our somewhat narrow definition of the term is still appropriate. The reason for this is that, in common with the synchronous school, an O'Haskell system can always be considered to react "immediately" just by utilising a sufficiently fast processor.

The work reported in this paper is an extension and reformulation of previous work by the first author [Nor95].

9 Conclusion and further work

In this paper we have defined an extension to Haskell that supports semantically visible concurrency by means of *reactive objects*. The core of our approach is a deviation from the common view of a process as a long thread of control interspersed with active resting points, towards the more object-oriented notion of a process as a collection of unordered, terminating code fragments connected by a common mutable state. Our communication primitives notably lack any blocking input facilities — a central feature of our reactive model is that all input to a process is confined to the passive, resting state that every object returns to when it is not executing.

We have developed a prototype implementation, which has enabled us to run several quite interesting example programs. Our experience with the language and the programming style it supports is encouraging, still the obvious path for further work starts with the development of more realistic applications that will tell us whether this reactive style, that looks very promising in the small, scales up.

Another branch of work concerns the language implementation. We would like to integrate the subtyping approach in [Nor97] with the preprocessor, and do a direct implementation of the library primitives in terms of the underlying runtime machinery.

Even though we believe that most O'Haskell programs by default uphold the reactive property, it might be interesting to investigate possibilities to formally guarantee this (and other properties), for example by using a more informative type system.

Acknowledgements

Thomas Hallgren and Björn von Sydow.

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Aug96] Lennart Augustsson. The Haskell Compiler HBCC. Personal communication, augustss@cs.chalmers.se, 1996.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[AYW93] G. Agha, A. Yonezawa, and P. Wegner, editors. *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1993.

[BB90] G. Berry and G. Boudol. The Chemical Abstract Machine. In *ACM Principles of Programming Languages*, pages 81–94, San Francisco, CA, January 1990.

[BB91] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-time Systems. Technical Report 1445, INRIA-Rennes, 1991.

[Car86] L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, LNCS 242. Springer Verlag, 1986.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, New York, 1965. Academic Press.

[Dij68] E.W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, 1968.

[Ham94] K. Hammond. Parallel Functional Programming: An Introduction (Invited Paper). In *PASCO '94*, Linz, Austria, September 1994.

[HC95] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In *Advanced Functional Programming*, LNCS 925, pages 137–182. Springer Verlag, 1995.

[Hen82] P. Henderson. Purely Functional Operating Systems. In *Functional Programming and its Applications*, pages 177–189. Cambridge University Press, 1982.

[Hoa74] C.A.R. Hoare. Monitors: An Operating Systems Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hol83] Sören Holmström. PFL: A Parallel Functional Language and Its Implementation. Technical Report PMG-7, Programming Methodology Group, Chalmers University of Technology, 1983.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *ACM Principles of Programming Languages*, St Petersburg, FL, January 1996. ACM Press.

[Jon96] M.P. Jones. Using Parameterized Signatures to Express Modular Structure. In *ACM Principles of Programming Languages*, St Petersburg, FL, January 1996. ACM Press.

[LJ94] J. Launchbury and S.L. Peyton Jones. Lazy Functional State Threads. In *ACM Programming Languages Design and Implementation*, Orlando, FL, 1994. ACM Press.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Lab for Foundations of Computer Science, Edingburgh, October 1991.

[Nor95] Johan Nordlander. Lazy Computations in an Object-oriented Language for Reactive Programming. In *Proc. ACM SIGPLAN Workshop on State in Programming Languages*, San Fransisco, CA, January 1995. Available as Technical Report UIUCDCS-R-95-1900, University of Illinois at Urbana-Champaign.

[Nor97] Johan Nordlander. Pragmatic Subtyping in Polymorphic Languages. Forthcoming, 1997.

[Pap92] M. Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In *ECOOP '91 Workshop on Object-Based Computing*, LNCS 612. Springer Verlag, 1992.

[PGF96] S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Principles of Programming Languages*, pages 295–308, St Petersburg, FL, January 1996. ACM Press.

[PH96] J. Peterson and K. Hammond, editors. *The Haskell 1.3 Report.* YALEU/DCS/RR-1106. Yale University Research Report, 1996.

[Pnu86] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends. In *Current Trends in Concurrency*, LNCS 224. Springer Verlag, 1986.

[PT94] B. Pierce and D.N. Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming*, Sendai, Japan, November 1994.

[Rep92] J. Reppy. Concurrent ML: Design, Application and Semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, LNCS 693. Springer Verlag, 1992.

[Rey94] J. Reynolds. User Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, Cambridge, MA, 1994. MIT Press.

[Sar93] J. Sargeant. Uniting Functional and Object-Oriented Programming (invited paper). In *Object Technologies for Advanced Software*, LNCS 742, Kanazawa, Japan, November 1993.

[Sch95] E. Scholz. Four Concurrency Primitives for Haskell. In *Proc. Haskell Workshop*, pages 1–12, La Jolla, CA, 1995. Available as Yale University Research Report YALEU/DCS/RR-1075.

[Sto86] W. Stoye. Message-based Functional Operating Systems. *Science of Computer Programming*, 6:291–311, 1986.

[Tur87] D. Turner. Functional Programming and Communicating Processes. In *Conference on Parallel Languages and Architectures*, LNCS 259, 1987.

A An operational semantics for core Haskell

$$\begin{array}{lcl} a, b & ::= & x \mid \backslash x \rightarrow a \mid a\ b \mid \underline{\mathsf{let}}\ x = a\ \underline{\mathsf{in}}\ b \mid \\ & & k \mid \underline{\mathsf{case}}\ a\ \underline{\mathsf{of}}\ \overline{k \rightarrow b} \\ \\ \mathcal{E} & ::= & [\,] \mid \mathcal{E}\ b \mid \underline{\mathsf{case}}\ \mathcal{E}\ \underline{\mathsf{of}}\ \overline{k \rightarrow b} \end{array}$$

$$\begin{array}{lrcl} \textsc{Apply} & (\backslash x \rightarrow a)\ b & \mapsto & a[b/x] \\ \textsc{Case} & \underline{\mathsf{case}}\ k_j\ \overline{a}\ \underline{\mathsf{of}}\ \overline{k_i \rightarrow b_i} & \mapsto & b_j\ \overline{a} \\ \textsc{Let} & \underline{\mathsf{let}}\ x = a\ \underline{\mathsf{in}}\ b & \mapsto & b[(\underline{\mathsf{let}}\ x = a\ \underline{\mathsf{in}}\ a)/x] \\ \textsc{Cont} & \mathcal{E}[a] & \mapsto & \mathcal{E}[b] \quad \text{if } a \mapsto b \end{array}$$

B Structural congruence

$$\begin{array}{rcll} P \parallel Q & \equiv & Q \parallel P & \\ P \parallel (P' \parallel Q) & \equiv & (P \parallel P') \parallel Q & \\ P & \equiv & Q & \text{if } P =_\alpha Q \\ \nu n.\nu m.P & \equiv & \nu m.\nu n.P & \\ \nu n.(P \parallel Q) & \equiv & P \parallel \nu n.Q & \text{if } n \notin \textsc{fn}(P) \end{array}$$

C Structural reaction

$$\textsc{Par} \quad \frac{P \rightarrow P'}{P \parallel Q \rightarrow P' \parallel Q}$$

$$\textsc{Res} \quad \frac{P \rightarrow P'}{\nu n.P \rightarrow \nu n.P'}$$

$$\textsc{Equiv} \quad \frac{P \equiv Q \qquad Q \rightarrow Q' \qquad Q' \equiv P'}{P \rightarrow P'}$$

D Extending core Haskell with structures

$$\begin{array}{lcl} a, b & ::= & \ldots \mid \underline{\mathsf{struct}}\ \overline{l = a} \mid a.l \\ \mathcal{E} & ::= & \ldots \mid \mathcal{E}.l \end{array}$$

$$\textsc{Select} \quad (\underline{\mathsf{struct}}\ \overline{l_i = a_i}).l_j \quad \mapsto \quad a_j$$

E Concurrent Haskell in O'Haskell

To complement the encoding of O'Haskell given in section 7, we also provide a translation in the other direction: Concurrent Haskell's primitives implemented in our language. We consider the following IO operations:

```
returnIO     :: a -> IO a
bindIO       :: IO a -> (a -> IO b) -> IO b
forkIO       :: IO () -> IO ()
newMVar      :: IO (MVar a)
putMVar      :: MVar a -> a -> IO ()
takeMVar     :: MVar a -> IO a
```

The IO monad in implemented as a composition of a continuation monad and a reader monad, reflecting the fact that *any* expression of type IO t might involve indefinite blocking, so we must be able to produce action values for the currently executing process on the fly.

```
type IO a       = Ref () -> (a -> O () ()) -> O () ()

returnIO a      = \_ c -> c a

m `bindIO` f    = \s c -> m s (\a -> f a s c)
```

Forking off a process is interpreted as the creation of a stateless object exporting a single action-valued interface, which is immediately triggered. Note the use of the implicitly bound variable self.

```
forkIO p        = \_ c -> do o <- template
                                  with action
                                     p self return
                             o
                             c ()
```

An MVar becomes a special kind of object, with the following interface:

```
struct MVar a =
    put      :: a → Request ()
    take     :: (a → Action) → Action
```

The intention is that put updates the state of its MVar with a new item, while take announces the readiness of some process to consume an eventual item stored inside the MVar (c.f. section 5.3). A Request, rather than an Action, is necessary in the type of put in order to mimic the error-handling semantics of Concurrent Haskell.

Assuming there is a template mvartempl for creating MVar objects, the implementation of newMVar and putMVar is straightforward:

```
newMVar      = \_ c → do v ← mvartempl
                         c v

putMVar v a  = \_ c → do v.put a
                         c ()
```

Since executing takeMVar marks the end of the currently executing process fragment, the translation does not call the given continuation, but wraps it up in an action value using the Ref identity of the currently executing process. This action is sent to the MVar in question, and the calling process enters a resting state.

```
takeMVar v   = \s c → v.take (\a → act s (c a))
```

Finally we give the code for mvartempl. It is a variant of the queue encoding in section 5.3 that limits the number of stored items to at most one. Types for the state variables are included as a reading aid, but we have to put these inside comments since we do not (yet) support scoped type variables.

```
mvartempl :: Template (MVar a)
mvartempl =
    template
        val := Nothing    -- Maybe a
        takers := []      -- [a → Action]
    with struct
        put a = request
            case takers of
                []   → case val of
                           Nothing   → val := Just a
                           Just _    → error "putMVar"
                t:ts→ t a
                       takers := ts
        take t = action
            case val of
                Nothing   → takers := takers ++ [t]
                Just a    → t a
```

Judging from the sheer amount of code required, the translation of Concurrent Haskell into O'Haskell looks slightly more complex than the converse encoding. It should be borne in mind, though, that the interpretation in section 7 also involves an implementation of the abstract data type Chan, which roughly corresponds to the encoding of MVars above. We therefore conclude that neither language can be coined more 'expressive' or 'basic' than the other.