

Chapter 1

Reactive Objects for Haskell

Stefan Heimann and Matthias Neubauer¹

Abstract. Reactive objects are a convenient abstraction for writing programs which have to interact with a concurrent environment. A reactive object has two characteristics: the abandonment of all blocking operations and the unification of the concepts state and process. The former allows a reactive object to accept input from multiple sources without imposing any ordering on the input events. The latter prevents race conditions because the state of an object is only manipulated from the process belonging to the object.

Only a few programming languages exist that make the concept of reactive objects available to programmers. Unfortunately, all those systems seem to be either not actively maintained anymore or not really matured yet. In this paper we demonstrate how reactive objects can be incorporated into the existing general purpose programming language Haskell by making use of common extensions to Haskell. Our implementation is heavily inspired by the O'Haskell programming language. While O'Haskell is a standalone programming language not compatible with Haskell, our implementation comes as a Haskell library that smoothly integrates the concept of reactive objects into the world of regular functional programming.

1.1 INTRODUCTION

As noted by Nordlander and Carlsson [NC97], *objects* can be viewed as a combination of state and process, meaning that the state of the object can only be accessed from within the process belonging to the object. The absence of any blocking operation turns such an object into a *reactive object*. Reactive objects are a convenient abstraction for writing programs which have to interact with a concurrent environment where input can virtually come from anywhere and happen at any time. In a sequential setting with explicit concurrency primitives, the pro-

¹Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 079, 79110 Freiburg, Germany. Email: {heimann,neubauer}@informatik.uni-freiburg.de

grammer uses some sort of I/O multiplexing or nonblocking I/O or starts multiple threads for reading from the different input sources. Some additional synchronization mechanism is then needed so that data from different input sources can be brought together. It can become quite a burden to ensure liveness and mutual exclusion in such a setting.

Programming with reactive objects is quite different. Instead of waiting actively until input becomes available, a reactive object is notified by the environment whenever input arrives at one of the sources of interest. Because a reactive object cannot call any indefinitely blocking function, it is guaranteed (provided there is no deadlock or infinite loop) that the object can handle such notifications at any time. Besides that, a reactive object implicitly forms a critical region, which makes it easy to avoid race-conditions.

To our knowledge, only a few programming languages exist that make the concept of reactive objects available to programmers. Unfortunately, all those systems seem to be either not actively maintained anymore or not really matured yet. Writing real production code using reactive objects seems impossible up to now. Much of the work presented here is inspired by one of the programming languages featuring reactive objects called O'Haskell [NC97]. O'Haskell is a separate programming language derived from Haskell featuring reactive objects, nominal subtyping and records. Although O'Haskell is derived from Haskell, O'Haskell code is not compatible with Haskell.

We demonstrate in this paper how reactive objects can be embedded in the general purpose programming language Haskell [Has98] with common extensions, namely rank- n polymorphism, type-indexed products [KLS04a], Concurrent Haskell [JGF96] and Template Haskell [SP02]. All those extensions are available in the current version of the Glasgow Haskell Compiler [GHC]. Our implementation of reactive objects is called *rHaskell*, the source code can be downloaded from <http://www.informatik.uni-freiburg.de/~heimann/rhaskell/>.

The rest of this paper is organized as follows: In the next section we refine our definition of reactive objects and give a short introduction to O'Haskell. In Section 1.3 we show the details of our implementation of reactive objects. Section 1.4 demonstrates how rHaskell can be used to implement a simple chat server. The final sections discuss possible directions for further work, compare our approach with related work, and conclude.

1.2 BACKGROUND

In this section, we recapitulate the concept of reactive objects and related concepts as they are provided in the O'Haskell programming language. A simple calculator serves as a running example throughout the rest of the paper.

1.2.1 Reactive Objects in O'Haskell

The central ideas of reactive objects in O'Haskell [NC97, NJC⁺02] can be formulated as follows:

```

record Calculator =
  setOperands  :: (Int, Int) → Action
  setOperator   :: (Int → Int → Int) → Action
  computeResult :: Request Int

newCalculator :: Template Calculator
newCalculator =
  template
    operands := (0, 0)
    operator := (+)
  in record
    setOperands ops = action operands := ops
    setOperator op  = action operator := op
    computeResult  =
      request return (fst operands
                       'operator' snd operands)

fortyTwo :: Cmd Int
fortyTwo = do myCalculator ← newCalculator
           myCalculator.setOperator (*)
           myCalculator.setOperands (2, 21)
           x ← myCalculator.computeResult
           return x

```

FIGURE 1.1. A simple calculator in O'Haskell

- Abandon all blocking operations. Instead of waiting actively for some event, a notification is delivered whenever the event happens.
- Define an object as the combination of state and process. The state of an object can only be manipulated from within the separate process belonging to the object, so that the state is protected against race conditions.

Introducing a reactive object in O'Haskell amounts to defining the initial state of the object and specifying the communication interface of the object with respect to the rest of the world. O'Haskell objects communicate with other reactive objects by sending and receiving either asynchronous or synchronous messages. Sending an asynchronous message always returns immediately for the process of the calling object, but it also triggers the concurrent execution of the operation associated with the message in the process of the callee. On the other hand, sending a synchronous message blocks the caller until the operation associated with the message has been executed in the process of the callee and the result has been returned. Asynchronous messages are called *actions* in O'Haskell, whereas synchronous messages are called *requests*.

To exemplify the creation of an object that reacts on asynchronous as well as on synchronous messages, we present a small calculator program implemented

in O’Haskell shown in Figure 1.1². The code starts with the introduction of a *Calculator* record type that specifies the communication interface of calculator objects. The interface consists of the two asynchronous actions `setOperands` and `setOperator`, and the synchronous request `computeResult`. The reason for using an action to set the operands and the operator of the calculator is that these operations do not need to return a result; conversely, `computeResult` is a request because it should return the result of applying the operator to the operands.

A “template”³ for a new calculator is created using the **template** special form (templates are similar to class declarations in object-oriented languages in such a way that they also incorporate recipes on how to create objects). The binding group following immediately after the **template** keyword defines the initial state of objects created from this template. In our case, we introduce two state variables, `operands` and `operator`, holding a pair of operands and the arithmetical operation to be performed on the operands. The part following the **in** keyword specifies the communication interface of the object that is exposed to the outside world and also gives an implementation of it. In our case, it is a value of the previously defined record type *Calculator*. The values for the fields `setOperands` and `setOperator` are created by using the **action** keyword; the `:=` notation is used to assign a value to the corresponding state variable. The `computeResult` field is set to a value created with the **request** keyword. Note that in O’Haskell state variables are used like any other variables.

The `fortyTwo` function demonstrates the usage of the calculator template. First, a new calculator object is created by executing `newCalculator`. Conceptually, at least two processes are running from this point on: The current process, where `fortyTwo` runs in and the process belonging to the newly created calculator object. We can then use the communication interface of the calculator object to set the operands and the operator by sending the corresponding messages. Although sending these messages returns immediately and the assignments are carried out concurrently in the process of the calculator object, the `computeResult` request executed afterwards will always return the result of applying the new operator to the new operands. O’Haskell preserves the message ordering by queueing incoming message in the process of the calculator object.

1.2.2 Blocking Operations

Besides combining state and process, the abandonment of blocking operations is another important characteristic of reactive objects. Instead of waiting actively for a specific event, a reactive object waits passively until some event of interest happens. An event can be anything you would wait for actively in a sequential setting: a mouse click, arrival of data on a network socket, etc. The events of interest are specified by registering the reactive object as a callback in the runtime

²Note, that the syntax of O’Haskell code in our paper slightly differs from the syntax presented in the original O’Haskell papers [NC97, Nor98, Nor99, Nor00]. Instead, the code is adapted to run with a recent O’Haskell implementation called O’Hugs [OHu01].

³The term *template* must not be confused with templates of Template Haskell.

system. The runtime system then sends a message to the reactive object whenever one of the events the object has registered for occurs.

The main advantage of this design decision is that a reactive object can react to any event of interest at any time because the execution of an action or a request always finishes after a finite amount of time, provided there is no infinite loop and no deadlock.

1.2.3 Records and Subtyping in O'Haskell

We have already seen how records can be used to form the communication interface of a reactive object. Besides records, the underlying type theory of O'Haskell comes with a notion of subtyping. Different to other type systems with subtyping, the notion of subtyping found in O'Haskell is not a purely structural one. Instead, basic subtyping properties of a program are either built-in or declared in a nominal fashion by the programmer. In addition, structural subtyping rules also apply to propagate declared subtyping properties to compound types.

We do not go into details here (see for example [Nor98] for the technical background), instead we informally introduce what is essential for our implementation of reactive objects in Haskell, namely records, record subtyping and specific built-in subtype axioms. The built-in subtype axioms are needed to differ between stateful and stateless computations; they are discussed in Section 1.2.4. Records are important because they usually form the communication interface of a reactive object. Subtyping on records allows that user-defined reactive objects with an extended communication interface can be passed to predefined library functions, or that objects are amenable to more than one communication interface.

To introduce a new record type, the **record** keyword is used to specify both the name for the new record type and a set of record components with their associated types. Record values are also created with the **record** keyword. The calculator example demonstrates how to introduce a common calculator interface by first declaring the *Calculator* record type as the common type for calculator objects. The *newCalculator* template then returns reactive objects obeying a communication interface of type *Calculator*.

To extend an existing record type, the programmer has to state an explicit subtype declaration. The following example shows the extension of our simple calculator type to resettable calculators

```
record ResettableCalculator < Calculator =  
  reset    :: Action
```

Such a declaration introduces a new record type, *ResettableCalculator*, that is a subtype of *Calculator*. Hence the new record type has components with labels *setOperands*, *setOperator* and *computeResult* (with the same types as in the *Calculator* record) and additionally a component labeled *reset* of type *Action*. We can now use a value of type *ResettableCalculator* at places where only values of type *Calculator* are expected.

1.2.4 The O Monad

The underlying computational model of a single reactive object can be viewed as an instance of a state monad $O\ s$ with fixed state type s . Computations that operate on a reactive object with an internal state of type s and return values of type t have type $O\ s\ t$ (the type s results from combining the types of the individual state variables into a common tuple type; in our example, the state of a reactive object created by `newCalculator` would have type $((Int, Int), Int \rightarrow Int)$).

Commands are specific computations in the $O\ s$ monad which are independent of the state component s . This is reflected in the type of commands; a command returning a value of type t is assigned the type $Cmd\ t$ which is a subtype of $O\ s\ t$ for all types s . As already mentioned in Section 1.2.3, O'Haskell's notion of subtyping is induced by so-called *subtyping axioms* that form the basic subtyping theory of the language. The relationship between commands and general stateful computations is expressed by the following built-in subtype axiom:

$$Cmd\ t < O\ s\ t$$

The *Cmd* type is used in the calculator example to type the function `fortyTwo`. The function returns an *Int* value and does not access the state of any reactive object; thus, its type is *Cmd Int*.

Three other built-in subtype axioms state the connection of actions, requests and templates with respect to general commands:

$$\begin{aligned} Action &< Cmd\ () \\ Request\ t &< Cmd\ t \\ Template\ t &< Cmd\ t \end{aligned}$$

Actions can be used as commands with fixed unit return type. Requests and templates with return type t are also commands with return type t .

There is another peculiarity incorporated into O'Haskell worth mentioning: whereas assigning a value to a state variable has monadic type $O\ s\ ()$, reading a state variable has not. This can be seen from the example in Figure 1.1: The assignment operands `:= ops` has type $O\ ((Int, Int), Int \rightarrow Int)\ ()$, but in the request `computeResult` the state variables `operands` and `operator` are treated as if they were pure values.

1.3 IMPLEMENTING RHASKELL

After presenting the basic features of O'Haskell in the previous section, we will now show how to embed reactive objects in a current extension of Haskell as a library. We first explain the five primitive operations which form the core of our implementation. Next, we demonstrate how subtyping and records are simulated in Haskell. Furthermore, we present a new surface syntax which hides implementation details from the user and makes programming with the primitive operations much easier.

1.3.1 The primitive operations

Five primitive operations, namely `get_`, `set_`, `new`, `act`, and `req`, form the core of our implementation. The general scheme how to encode the primitives for reactive objects on the value level is heavily inspired by Nordlander’s work [NC97], however, there are significant differences of our encoding on the typing level. We start by defining a datatype for modeling the O monad.

```
data  $O$  s t =  $O$  (IORef s  $\rightarrow$  IO t)
```

An O monad computation is an IO computation that further depends on a state variable of type s which is passed to the IO computation by reference. It is fairly easy to make O s an instance of the monad class and so we do not show the instance declaration here. Because commands are just values of type O s t where the state argument s is irrelevant, we type commands as follows

```
type Cmd t =  $\forall$  s.  $O$  s t
```

Instead of making use of a particular notion of subtyping as the O’Haskell implementation does, we just give commands a universally quantified type. Functions that use commands are equipped with explicit rank- n polymorphic type annotations to express and handle dependencies between commands and specific O computations.

We could use similar type synonyms for actions, requests and templates, but then all types would be indistinguishable for the type checker. Therefore, we additionally wrap them into separate data types which has the additional benefit that type annotations for functions using actions, requests or templates can be omitted:

```
newtype Action      = Action (Cmd ())  
newtype Request t  = Request (Cmd t)  
newtype Template t = Template (Cmd t)
```

The `get_` and `set_` primitives read and write the state of a reactive object, respectively. We omit their implementation here because they correspond to the usual state monad operations.

```
set_ :: s  $\rightarrow$   $O$  s ()  
get_ ::  $O$  s s
```

The process belonging to a reactive object is implemented in Concurrent Haskell [JGF96] as a thread and channel transferring O s () computations. The channel is used as a unidirectional message queue to communicate with the thread of the object.

```
type ObjQueue s = Chan ( $O$  s ())
```

The job of the thread is to wait until a computation is available in the message queue. It then carries out the computation in the context of the object’s internal

```

type Calculator = ((Int, Int) → Action,
                   (Int → Int → Int) → Action,
                   Request Int)

newCalculator :: Template Calculator
newCalculator =
  new ((0, 0) {- operands -}, (+) {- operation -})
    (λself →
      ( λops → act self (do (_, op) ← get_
                           set_ (ops, op))
      , λop  → act self (do (ops, _) ← get_
                           set_ (ops, op))
      , req self (do ((left, right), op) ← get_
                    return (left 'op' right))))

```

FIGURE 1.2. The calculator implemented in rHaskell using primitives only

state and afterwards it waits for the next computation to arrive. So the effect of writing data into an object’s message queue is similar to what is usually referred to as “sending a message to an object” in the object-oriented world.

The primitive operation `new` creates a reactive object by setting up the message queue and spawning the thread belonging to the object. Its arguments are the initial state of the object and a function yielding the object’s communication interface when supplied with the message queue.

```

new :: s → (ObjQueue s → t) → Template t

```

The primitive operation `act` creates new actions; it takes the object’s message queue and the code the action should execute when triggered and returns a computation which writes the code into the message queue. The `req` primitive used for constructing requests is similar but some synchronization mechanism is needed for returning the result of the request. We use Concurrent Haskell’s *MVars* for this purpose.

```

act :: ObjQueue s → O s () → Action
req :: ObjQueue s → O s t → Request t

```

For lack of space, we do not show the implementation of `new`, `act` and `req` here; the interested reader is referred to Section 7 of the original O’Haskell paper [NC97].

Having together all core primitives to implement reactive objects, we are now able to give the first rHaskell encoding of the calculator example; the new encoding is shown in Figure 1.2. Instead of an O’Haskell record, we for now just use a triple for storing the actions and requests of the object. The function `newCalculator` constructs a new calculator object by calling the primitive operation `new` with the initial state of the calculator and a function evaluating to a triple of type *Calculator*. The primitives `act` and `req` are used together with

the message queue given to this function to construct the appropriate actions and requests.

1.3.2 Records and subtyping

As we already argued in Section 1.2.3, it is vital for an usable implementation of reactive objects that it supports a way to use objects with extended communication interfaces in places where objects with a more general communication interface are expected. O'Haskell uses its own notion of record subtyping to provide this feature. Until now, we just used tuples to pass around the actual components of an object. Unfortunately, Haskell tuples or Haskell data types with field labels are not extensible. Hence, our current encoding of objects does not allow us to write programs that show the desired flexibility.

Recently, Kiselyov et al. [KLS04a] introduced a library for Haskell, called the HList library, that uses common extensions of Haskell 98 to provide extensible records and other heterogeneous collections to the Haskell world. One of the collections you can find in their library are type-indexed products (TIPs)—extensible collections that allow access to components of the collection by using distinct index types. TIPs are a perfect candidate to encode objects with extensible communication interfaces within Haskell.

Instead of using a triple for the communication interface of our calculator, as shown in Figure 1.2, we now use a type-indexed product. To allow a type-indexed access to the different messages of the calculator, it is necessary to define separate datatypes for each message name. Those types later serve as index types to access components from the TIP representing the object. The data type definition for the *SetOperands* message together with an appropriate access functions look as follows:

```
data SetOperands =  
  SetOperands { setOpsVal :: (Int, Int) → Action }  
setOperands r = (setOpsVal . hOccurs) r
```

The data types *SetOperator* and *ComputeResult* and the corresponding access functions are defined analogously. The construction of a calculator TIP also makes use of appropriate combinator functions from the HList library:

```
(SetOperands (λops → ...) .*.  
 SetOperator (λop → ...) .*.  
 ComputeResult (...)) .*.  
 emptyTIP)
```

The `emptyTIP` function constructs a TIP without any members. The `.*.` combinator extends an existing TIP by an additional component.

Extending the communication interface for resettable calculators now just amounts to defining a new message type representing the additional reset operation:

```
data Reset = Reset { resetVal :: Action }
reset r = (resetVal . hOccurs) r
```

Objects constructed with an additional *Reset* component are still amenable at places where a regular calculator is expected. The type of a function expecting a regular calculator TIP just constraints the type of the TIP to at least supply the *SetOperands*, *SetOperator*, and *ComputeResult* components by adding the right type class constraints to the type of the function. Using a TIP handling more messages than expected always works, because the type class constraints assure that all the information needed is available.

1.3.3 Mutual exclusion and information hiding

Mutual exclusion and information hiding are two principles also found in O'Haskell that we consider very important to keep in another implementation of reactive objects. Therefore, we also provide them in our rHaskell implementation.

Our support for mutual exclusion stems from the fact that all operations accessing or manipulating the state of a reactive object form a critical region because the all are executed in the single process belonging to the object and so they cannot run in parallel.

Information hiding means that the state of a reactive object can only be accessed through messages provided in the communication interface of the object. As long as the programmer does not expose the message queue of the object to any client, the information hiding property is guaranteed. Making the message queue available in the communication interface would allow a client to create new actions and requests and would therefore provide a way of manipulating the object in an arbitrary manner. To guarantee information hiding, we need a way to prevent the programmer from accessing the object's message queue directly. The new surface syntax presented in the next section has exactly this property. A programmer only has to stick with the new surface syntax to be sure not to break information hiding by accident.

1.3.4 Syntactic sugar

Programming with just the primitive operations works but is quite cumbersome, especially when you try to manipulate the state of an object with several state variables. To make it more pleasant to work with rHaskell, we extend Haskell with additional syntactic constructs resembling those of O'Haskell. The new syntax is transformed into valid Haskell code with the help of Template Haskell [SP02]⁴, an extension to Haskell that adds compile-time macros to the language. Apart from being more convenient and making rHaskell code easier to read, the new syntax has the additional benefit of completely hiding the message queue belonging to an object from the programmer. When using the new syntax, it is impossible for a

⁴Please note that our implementation uses Template Haskell Version 2 [SP03]. Therefore, you need GHC 6.3 when trying out the transformation examples.

```

$(recordLabel "SetOperands"  [t| (Int, Int) → Action      |])
$(recordLabel "SetOperator"  [t| (Int → Int → Int) → Action |])
$(recordLabel "ComputeResult" [t| Request Int           |])

newCalculator = $(template
  [| ("operands" := (0, 0), "operator" := (+)) |]
  [| SetOperands (λops → action (set "operands" ops))
   *. SetOperator (λop → action (set "operator" op))
   *. ComputeResult (request (do
     (left, right) ← get "operands"
     op ← get "operator"
     return (left 'op' right)))
   *. emptyTIP |])

```

FIGURE 1.3. A simple calculator implemented in rHaskell using the new syntax

programmer to expose the message queue to the outside world (see Section 1.3.3 for how this would break information hiding).

The syntax transformation described in this section is inspired by the work of Nordlander [NC97]. A major difference is the treatment of state variables. Nordlander does not distinguish between reading a state variable and reading an ordinary variable. For example, in the O'Haskell term $x + y$ you cannot tell if x is bound by a `let`-binding or if x is actually a state variable. Clearly, this does not fit well in a purely functional language, because the value of a state variable depends on the state of the corresponding object whereas a `let`-bound value does not. We decided to provide a separate monadic operation for reading state variables instead of implicitly converting every read-access into a monadic operation.

Before looking at an example of the new syntax, we will shortly describe the basic features of Template Haskell. Template Haskell is a Haskell extension which lets you write code that is executed at compile-time to generate new Haskell code. Expressions enclosed in $\$(\dots)$ are evaluated at compile-time and the resulting piece of code is spliced in place of the original expression. Expressions that are surrounded by quasi-quote brackets, $[|$ and $|]$, are converted into an abstract syntax tree which can then be manipulated at compile-time. To convert types into an abstract syntax tree, you have to use the brackets $[t|$ and $|]$.

Figure 1.3 shows the calculator example using rHaskell syntax. From now on, we use extensible records as described in Section 1.3.2 for specifying the communication interface. The new syntactical form `recordLabel` declares a new record label by specifying its name and its type. It is implemented by the compile-time function `recordLabel :: String → Q Type → Q [Dec]` that generates the corresponding boilerplate code. The Q monad is Template Haskell's quotation monad hiding the details necessary of macro expansion. Figure 1.4 shows the expanded code so that you can easily see what code `recordLabel` generates.

The syntactical form `template` is the entry point of the syntax transforma-

```

data SetOperands =
  SetOperands { _setOperandsVal :: (Int, Int) → Action }
setOperands r_0 = (_setOperandsVal . hOccurs) r_0
{- The code for SetOperator and ComputeResult is not shown. -}

newCalculator = new ((0, 0), (+))
  (λself → SetOperands (λops →
    act self ((λx_2 → do (x_3, x_4) ← get_
                        set_ (x_2, x_4)) ops))
  *. SetOperator (λop →
    act self ((λx_6 → do (x_7, x_8) ← get_
                        set_ (x_7, x_6)) op))
  *. ComputeResult (req self (do
    (left, right) ← do (x_11, x_12) ← get_
                        return x_11
    op ← do (x_14, x_15) ← get_
            return x_15
    return (left 'op' right)))
  *. emptyTIP)

```

FIGURE 1.4. The example from Figure 1.3 after running the compile-time code

tion. It is implemented by the macro `template :: Q Exp → Q Exp → Q Exp`. The first argument of `template` specifies the state variables of the object being constructed, the second parameter defines the communication interface. Thus,

```
$(template [| e1 |] [| e2 |])
```

leads to the expression

```
new  $\hat{e}_1$  (λself →  $\mathcal{T}$ [[e2]]self)
```

where \hat{e}_1 is a tuple denoting the initial state of the object. \hat{e}_1 is constructed by extracting the values from the declaration of the state variables in e_1 . Normally, e_1 is a tuple as shown in the example in Figure 1.3. If an object has only one state variable, you can use the form "x" := e, for an object without any state variables, the declaration is just ().

The second parameter, e_2 , is handled by the transformation \mathcal{T} as shown in Figure 1.5 (which trivially extends to the whole expression syntax). Inside e_2 the following additional syntactic forms are available:

```

action  :: O s t → Action
request :: O s t → Request t
get     :: String → Cmd a
set     :: String → a → Cmd ()
modify  :: String → (a → a) → Cmd ()

```

The action and request forms are translated into the appropriate primitive operations by using the variable `self` which is bound by the lambda abstraction

```

 $\mathcal{T}[[\text{action}]]\text{self} = \text{act self}$ 
 $\mathcal{T}[[\text{request}]]\text{self} = \text{req self}$ 
 $\mathcal{T}[[\text{get "x"}]]\text{self} = \text{do } (x_1, \dots, x_n) \leftarrow \text{get\_}$ 
 $\quad \text{return } x_i$ 
 $\mathcal{T}[[\text{set "x"}]]\text{self} = (\lambda x \rightarrow \text{do } (x_1, \dots, x_n) \leftarrow \text{get\_}$ 
 $\quad \text{set\_ } (x_1, \dots, x_{i-1}, x, x_{i+1}, \dots, x_n))$ 
 $\mathcal{T}[[\text{modify "x"}]]\text{self} = (\lambda f \rightarrow \text{do } y \leftarrow \mathcal{T}[[\text{get "x"}]]\text{self}$ 
 $\quad \mathcal{T}[[\text{set "x"}]]\text{self } (f y))$ 

```

where "x" has index i in the state variable specification and f, x, y, x_1, \dots, x_n are fresh.

FIGURE 1.5. Definition of the syntax transformation \mathcal{T}

created by `template`. Template Haskell ensures that no other binding for `self` can break this connection. Vice versa, it is impossible for a programmer to use the value bound to `self` because of the static scoping rules imposed by Template Haskell. Hence, a programmer cannot access the message queue of the object which is important to ensure information hiding (see Section 1.3.3).

The rules for transforming `get`, `set` and `modify` are a bit more involved. The idea is to read, write or modify the appropriate component of the tuple representing the object's state. The index of the component is obtained by looking up the string used as the first argument of `get`, `set` or `modify` ("x" in our case) in the state variable specification. Because this lookup is done at compile-time, the connection between the function name and the string argument must be fixed syntactically, which means you cannot write something like `mapM get ["x", "y", "z"]`.

1.4 EXAMPLE: A SIMPLE CHAT SERVER

To demonstrate the feasibility of our approach, we will now show the implementation of a simple chat server written in rHaskell. The chat server accepts connections from clients using telnet, asks them for a nickname and then broadcasts messages between the clients.

The source code of the chat server is shown in Figure 1.6. The heart of the chat server is a client record and a peer object associated with every user connected to the server. The client record is created by the server in order to get notified whenever something important happens in the communication with the user. A client object handles the following messages:

- *ClientClose* :: *Action*: invoked from the rHaskell runtime system when the user closes the connection.
- *ClientError* :: *Exception* → *Action*: invoked in case of an error in the communication with the user.

```

1 type Nick      = String
2 type User     = (Peer, Nick)
3 type ServerState = ([User])
4
5 newServer =
6   $(template
7     [| "users" := ([] :: [User]) |]
8     [| let registerNewUser :: Peer → Nick → O ServerState ()
9         registerNewUser peer nick =
10          do users ← get "users"
11             let answer = "Welcome! Other users: ++showUsers users
12                user = (peer, nick)
13                unpack $ peerSendLine peer answer
14                set "users" (user : users)
15                broadcastMsg user (nick++" has joined the chat.")
16
17          broadcastMsg :: User → String → O ServerState ()
18          broadcastMsg user msg =
19            do users ← get "users"
20               let others = List.delete user users
21                   mapM_ (λ(p, _) → unpack $ peerSendLine p msg) others
22
23          goodbye :: Peer → O ServerState ()
24          goodbye peer =
25            do user ← getUserForPeer peer
26               case user of
27                 Nothing → return () -- user not registered yet
28                 Just user@(_, nick) →
29                   do broadcastMsg user (nick++" has left the chat")
30                      modify "users" (List.delete user)
31
32          getUserForPeer :: Peer → O ServerState (Maybe User)
33          getUserForPeer peer =
34            do users ← get "users"
35               return $ List.find ((≡) peer . fst) users
36
37          newClient peer =
38            ( ClientClose (action $ goodbye peer)
39            *. ClientError (λe → action $ goodbye peer)
40            *. ClientConnect (action $
41              unpack (peerSendLine peer "Enter your nickname:"))
42            *. ClientDeliver (λline → action $ do
43              user ← getUserForPeer peer
44              case user of
45                Nothing → registerNewUser peer (trim line)
46                Just user@(_, nick) →
47                  broadcastMsg user ("++nick++"++trim line)
48            *. emptyTIP)
49          in λpeer → request $ return (newClient peer)
50     [|]
51
52 main = with0 $ do srv ← unpack newServer
53                 waitForClients0 myPort srv myErrorHandler
54                 return ()

```

FIGURE 1.6. A simple chat server

- *ClientDeliver* :: *String* → *Action*: invoked whenever data arrives from the user (the *String* argument is the data sent).
- *ClientConnect* :: *Action*: invoked right after the user has connected to the server.

The server uses the peer object, which is created by the runtime system for every incoming connection, to send data to the user.

The main function of the chat server first creates a new server object by unpacking and executing the function `newServer` (the `unpack` function just extracts the command wrapped in a *Template*, *Action* or *Request*). It then registers the newly created server object as a callback in the runtime system using the library function `waitForClients0`. The call `waitForClients0 myPort srv myErrorHandler` causes the runtime system to listen on port `myPort` for incoming connections. As soon as a new user connects to the server, a peer object is created and passed to the function `srv` yielding a client object, which is then used for all incoming communication with the user. The net effect of passing the server object to `waitForClients0` is that the function `newClient` in line 42 generates the client records for all incoming connections.

We are now going into detail what happens when a particular user establishes a connection with the server. As discussed above, the rHaskell runtime system creates a new peer object and passes it eventually to the `newClient` function, which returns a client object. The runtime system then invokes the *ClientConnect* action of the client object. The client object reacts by asking the newly connected user for a nickname. When the user sends the nickname to the chat server, the runtime system invokes the *ClientDeliver* action. The code of *ClientDeliver* uses the function `getUserForPeer` to retrieve a *User* value matching the peer object associated with the client object. A *User* is just a pair containing the peer object and the nickname of the user. In case of a newly connected user, `getUserForPeer` yields *Nothing* and so the user is registered by calling `registerNewUser`. The functions `getUserForPeer` and `registerNewUser` use the state variable "users" for accessing and manipulating the list of users connected with the chat server.

Now that the new user is registered, he or she can participate in discussions with the other users. Whenever *ClientDelivered* is invoked for an already registered user, the `getUserForPeer` function returns a *Just* value and so the data sent by the user is broadcasted to all other users by calling the `broadcastMsg` function. In case of a disconnection or an error in the communication with the user, one of the actions *ClientClose* or *ClientError* is invoked, which in turn calls the function `goodbye`. The `goodbye` function notifies the other users about the disconnection and removes the user from the list of registered users.

1.5 FUTURE WORK

After having presented an implementation of reactive objects in Haskell, the next step is to show its usefulness and applicability by writing a non-trivial applica-

tion which uses reactive objects. We already started with such an application by writing a prototype of a mobile-agent system based on the idea of Aglets [Agl], a mobile-agent system for Java. An Aglet is an object which can move between different hosts running an Aglet Server. To move an Aglet to a different host, the server just serializes the Aglet’s state and sends it over the network together with the bytecode of the object.

Instead of Java objects, our implementation in rHaskell uses reactive objects for representing Aglets. Two simplifications ease the task of serialization significantly: Firstly, the state variables of an Aglet must all be instances of the type classes *Show* and *Read*; secondly, the code of an Aglet is transferred in binary form. Especially the second simplification is a severe restriction for a mobile-agent system. However, by using a serialization mechanism as described in [dBTL04], both restrictions could be lifted.

1.6 RELATED WORK

Reactive objects were introduced by Nordlander and Carlsson as the main concept of the O’Haskell programming language [NC97]. The underlying type system of O’Haskell [Nor98, Nor00] differs from Haskell’s type system mainly because it incorporates subtyping. Subtyping in O’Haskell stems from three sources: from built-in axioms relating built-in types, from declaration of new records or new algebraic datatypes, and from a depth subtyping rule. The differing type system is the main reason why O’Haskell implementations are not just extensions of existing Haskell systems. There is a newer descendent of O’Haskell called Timber [BCJ⁺02] which is mainly targeted at embedded real-time systems. We have concentrated on O’Haskell here because Timber does not add anything new to the idea of bringing reactive objects and Haskell together.

Our implementation of reactive objects does not adopt the type system of O’Haskell. The price we pay is that there is no explicit concept of subtyping in rHaskell. In particular, we do not encode subtyping of algebraic datatypes or the depth subtyping rule. We argue that the only place where subtyping is needed in the context of reactive objects is the possibility to extend communication interfaces. Our encoding with type-indexed products exactly captures this property.

We do not compare the concept of reactive objects presented in this paper with other approaches of concurrency and object concurrency, because our work concentrates on embedding reactive objects into an existing programming language. A detailed discussion on work related to reactive objects can be found in the original O’Haskell paper [NC97].

There is other work that studies how to employ Template Haskell to do compile-time code transformations on Haskell programs. Lynagh [Lyn03] uses Template Haskell to make simple code optimizations like simplification of arithmetic expressions or loop unrolling programmable. Seefried et al. [SCK04] demonstrate how to use Template Haskell to optimize a DSL for image manipulation by doing unboxing, inlining, or employing algebraic transformation rules.

We have used Template Haskell only for eliminating syntactic sugar and not

for optimization purpose. Nevertheless, it appeared that there is still some room for improvement to make Template Haskell more suitable as a tool to extend Haskell. Specifically, the need to use string constants for accessing state variables makes rHaskell programs look a little bit awkward. One possible solution would be the ability to splice into binding positions. This feature is already mentioned in a report on the forthcoming version of Template Haskell [SP03] but not implemented yet. Another possible solution to this problem would be the addition of non-typechecked quasi-quoted fragments as Lynagh already suggested [Lyn03].

Various proposals were made on how to add extensible records to Haskell (as for example [GJ96] or [SM01]), but none of them emerged as the accepted Haskell extension. The work of Kiselyov et al. [KLS04a] differs from other proposal because they are able to encode extensible records in Haskell with common extensions. Here, records are just library code instead of a further language extension.

Independently from our work, the same authors discovered how their HList library can be used to encode a notion of object within Haskell [KLS04b]. They demonstrate their encoding by simulating OCaml style objects in Haskell. Because they do not introduce a new syntax for objects, references to object states are exposed to the programmer in their setting.

1.7 CONCLUSION

We have presented rHaskell, an implementation of reactive objects in Haskell inspired by the ideas of O'Haskell. It makes reactive objects accessible to Haskell programmers without using yet another programming language. We have shown how the subtyping inherent to O'Haskell can be simulated with rank-n polymorphism and type-indexed products. Furthermore, we have used Template Haskell for hiding some implementation details from the programmer and for providing a more convenient syntax. The conciseness of a small client-server application, the chat server example, demonstrates the benefit of using reactive objects for writing applications which should accept input from multiple sources at the same time. Our prototype of a mobile agent system seems to indicate that rHaskell also scales well for larger programs.

REFERENCES

- [Agl] Aglets, <http://www.trl.ibm.com/aglets/>.
- [BCJ⁺02] Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. Technical report, Department of Computer Science & Engineering, Oregon Health & Science University, April 2002.
- [dBTL04] André Rauber du Bois, Phil Trinder, and Hans-Wolfgang Loidl. Implementing mobile Haskell. In *TFP 2003: Fourth Symposium on Trends in Functional Programming*. Intellect, 2004.
- [GHC] The Glasgow Haskell Compiler, <http://www.haskell.org/ghc/>.

- [GJ96] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, Department of Computer Science, 1996.
- [Has98] Haskell 98, a non-strict, purely functional language. <http://www.haskell.org/definition>, December 1998.
- [JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.
- [KLS04a] Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. Strongly typed heterogeneous collections. In *ACM SIGPLAN Workshop on Haskell*. ACM Press, September 2004.
- [KLS04b] Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. Haskell’s overlooked object system. <http://homepages.cwi.nl/~ralf/OOHaskell/>, September 2004.
- [Lyn03] Ian Lynagh. Unrolling and simplifying expressions with template haskell. <http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/>, May 2003.
- [NC97] Johan Nordlander and Magnus Carlsson. Reactive objects in a functional language - an escape from the evil I. In *Proceedings of the Third Haskell Workshop*, Amsterdam, Holland, June 1997.
- [NJC⁺02] Johan Nordlander, Mark Jones, Magnus Carlsson, Dick Kieburtz, and Andrew Black. Reactive objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.
- [Nor98] Johan Nordlander. Pragmatic subtyping in polymorphic languages. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 216–227. ACM Press, 1998.
- [Nor99] Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, 1999.
- [Nor00] Johan Nordlander. Polymorphic subtyping in O’Haskell. In *Proceedings of the APPSEm Workshop on Subtyping and Dependent Types in Programming*, 2000.
- [OHu01] O’Hugs, the O’Haskell interpreter, version 0.5. <http://www.cs.chalmers.se/~nordland/ohugs/>, January 2001.
- [SCK04] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising Embedded DSLs using Template Haskell. URL: <http://www.cse.unsw.edu.au/~sseefried/papers.html>, March 2004.
- [SM01] Mark Shields and Erik Meijer. Type-indexed rows. In Hanne Riis Nielson, editor, *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 261–275, London, January 2001. ACM Press.
- [SP02] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.
- [SP03] Tim Sheard and Simon Peyton Jones. Notes on template haskell version 2, november 7, 2003. <http://research.microsoft.com/~simonpj/tmp/notes2.ps>, 11 2003.