Session Types in Haskell
Updating Message Passing for the 21st Century

Matthew Sackman Susan Eisenbach

Imperial College, London

{ms, sue}@doc.ic.ac.uk

Abstract

Session Types allow plans of conversation between two concurrent processes to be treated as types. Type checking then ensures that communication between processes is safe: i.e. it obeys the protocol specified by the session type. Thus Session Types offer a means to establish conformance to protocols in both distributed applications and multi-threaded programming.

We incorporate Session Types into Haskell as a tool for concurrent programming. Our implementation, which is a standard Haskell library, presents a monadic API to the programmer. Using the library looks and feels very much like normal monadic computation and thus there is a shallow learning curve for the Haskell programmer. Our implementation lifts the invariants and properties of Session Types into Haskell's rich type system. This allows our implementation to statically verify the use of the communication primitives provided without an additional type checker, preprocessor or modification to the compiler.

Our implementation supports multiple concurrent communication channels, individual processes can interleave actions across any number of open channels, and channels themselves can be sent and received. New channels can be created between pre-existing processes as well as to newly created processes. Communication is asynchronous and fully polymorphic. To our knowledge, no other implementation of Session Types is available in any language which matches our library in terms of functionality and supported features. We describe the key aspects of our implementation and demonstrate, through a running example, its usage and flexibility.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.13 [*Software Engineering*]: Reusable Software

Keywords Session Types, Channels, Functional Programming, Type Programming, Concurrency, Message Passing, Haskell

1. Introduction

The challenge of writing reliable and safe programs has recently become significantly harder due to changes in both the ways in which we use computers and in the design of CPUs. Ever greater parallelism and distributed access to data and programs have led to an explosion in research focussing on effective ways to aid programmers to overcome these challenges. Transactional memory [13, 8, 12] has presented an alternative to traditional locking systems, whilst at the same time, message passing concurrency has received renewed study.

Whilst transactional memory solves the composability issues of traditional locking and guarantees progress, there is no temporal information nor direction: it is left to the programmer to decide upon and verify the protocols obeyed by the threads when exchanging information via shared variables within transactions. Session Types [14, 22, 11, 24, 25] provide exactly this missing information, defining channels through which two processes can communicate with a prescribed conversation plan. As such, we sought to practically investigate the use of Session Types for message passing concurrency.

Programming with Session Types adds to message passing concurrency static safety, ensuring both parties using a communication channel are speaking the same protocol; that when one party sends a value the other party receives a value; that the types of the values exchanged are agreed upon in advance and that branching and looping control-flow structures are correctly implemented by both parties. It forces the two parties sharing a communication channel to cooperate with one another and communicate in a civil and well structured way, free of shouting, interrupting one another or talking at cross-purposes.

Haskell's type system, in particular with the extensions to the Haskell'98 [20] standard available through GHC [1], present a very flexible and powerful type system. We encode Session Types into Haskell types, using Haskell's type system to enforce the invariants of Session Types and to statically check the use of Session Types. Consequently we do not require a preprocessor, external type checker or any modifications to GHC. Our implementation works with the current stable version of GHC, 6.8.3. We make use of several extensions to Haskell'98 standard, including type classes with functional dependencies [17, 9], generalised algebraic data types (GADTs) [21], overlapping and undecidable type class instances (essentially placing the burden of termination of type checking on ourselves) and recent developments including type families [4, 5, 23].

In this paper, we refer to both *Session Types* and *session types*. *Session Types* refers to the calculus as a whole: the notion of describing a protocol, whilst a *session type* refers to a particular protocol, a particular specification of communication. Thus *Session Types* are a type system, where a *session type* is a type signature.

The main contributions of this paper are as follows:

- the presentation of a rich implementation of Session Types in Haskell,

- the demonstration through example of how our implementation can be used for complex communication patterns,
- the discussion of what assurances Session Types grant and what this means to the programmer and user of our implementation,
- further evidence of how powerful Haskell's type system (with extensions) is: it is difficult to see how this work could be done in any other mainstream language without need of an external additional type checker.

This paper is structured as follows: in section 2 we introduce the n-queens problem and the design of our checker which we implement as a running example throughout much of this paper. In section 3 we describe how to define a session type and then how to write a program that conforms to a session type. Section 4 explains how we identify processes through process IDs and the extent and limitations of the information held by the type of a process ID. In section 5 we explain how a single process can have many channels open and in use at the same time, how actions can be interleaved across several channels, how new processes can be created and how new channels can be formed between existing processes. Section 6 gives a high level overview of the implementation of our library, and section 7 presents an evaluation of our library and our plans for future work. In section 8 we discuss related work and we conclude in section 9.

2. Checking N-Queens

Throughout this paper, to illustrate the functionality of our implementation of Session Types, we develop a program that takes a proposed solution to the n-queens problem and in parallel checks whether the solution is valid. The n-queens problem is to place on an n-by-n chess board, n queens such that none of them are attacking each other. There are 92 distinct solutions to this problem when $n = 8$, though if you group solutions which are rotations or reflections of each other then there are 12 unique solutions.

The design of our checker is as follows. Each of the n queens is a separate process. They are created by the *master* or *root* process which sends to each queen the coordinate of the queen. The *root* process, having created all the queens, then sends to each queen the process IDs, or *Pids*, of all the other queens. The queens group the received Pids into two sets, *before* and *after* such that for each queen q_m for $0 \leqslant m < n$ where n is the total number of queens, $\mathit{before}_m = \{q_i | i \leftarrow [0\,..\,(m-1)]\}$ and $\mathit{after}_m = \{q_i | i \leftarrow [(m+1)\,..\,(n-1)]\}$.

Each queen q_m then creates a communication channel to each Pid in after_m and sends its own coordinate to the queens in after_m. Next it creates a communication channel to each Pid in before_m and receives the coordinates of the queens in before_m. It then checks to see if its own coordinate is attacking any of the queens that it knows the coordinate of, and sends this result to the *root* process. The *root* process gathers the results from each queen and if any queen reports a conflict then the *root* process rejects the proposed solution.

To see that this algorithm is correct, consider that with n queens, the number of potential conflicts is the same as the number of edges in a fully connected graph of n nodes: there is no need to find both that queen q_i can attack q_j and that q_j can attack q_i. Then consider that each queen communicates with every other queen, thus a fully connected graph is created. Next, each communication causes just one comparison: when a queen receives a coordinate it compares its own coordinate with that coordinate and when it sends its own coordinate then the receiving queen performs a comparison, so there is one comparison per communication. Finally, we must show that all the comparisons are unique. For a comparison to be duplicated then q_i must send its coordinate to q_j and q_j must send its coordinate to q_i where $i \not\equiv j$. This never happens as if and only if $i < j$ will q_i send its coordinate to q_j and if and only if $i > j$ will q_j send its coordinate to q_i.

```
class SMonad (m :: * → * → * → *) where
  (∼≫)  :: m x y a →            m y z b  → m x z b
  (∼≫=) :: m x y a → (a → m y z b) → m x z b
  sreturn :: a → m x x a
```

Figure 1. The *SMonad* type class

We shall use session types to define the various patterns of communication that are needed, our library shall statically check that the implementations obey the session types and shall execute the n-queens checker. This example problem demonstrates how our library can be used for a much broader class of problems where work is not only farmed out to a number of workers (the queens, in this case) but how the workers must also communicate between themselves in order to complete their work before sending results back to the *root* process. Note that our example works for any number of workers: the number of queens is not known statically and so this shows how session types can work in the most general case. In our example n-queens problem, the actual work done by each queen process is trivial thus the code dealing with communication dominates. However, that code would not change significantly for other broadly similar communication patterns whilst the work performed by each worker might become significantly more extensive.

3. Simple Sessions

A session type is a plan of communication. It is a prescription for two parties communicating with each other, indicating who says what (i.e. which party sends values of what type), and when (i.e. in what order). The session type can contain control-flow structures including branches and loops. Session types cannot change dynamically: they must be fully specified statically. Having specified a session type, that session type can then be used to parameterise the type of a communication channel, which restricts operations involving that communication channel to those specified by the session type. Every session type, s, has a unique *dual*, $\overline{s}$, in which the direction of communications are reversed; the *dual* relation is self-inverting. For a communication channel between two parties, one party will see the channel parameterised by the session type s and the other party will see the same communication channel parameterised by the session type $\overline{s}$, thus reflecting that what is an output for one party is an input for the other party and vice versa. The *dual* of a session type never needs to be specified: it is always calculated statically.

To construct a session type, a domain specific language is used. This allows fragments of session types to be given labels and to refer to one another via these labels. The language supports lexical scoping and statically prevents invalid session types from being created (e.g. an attempted jump to an undefined label). The domain specific language works within an extended $Monad$ type class, $SMonad$. This is a type indexed monad [3] which models a computation transforming a *from* state to a *to* state and additionally produces a value, where the two states can have different types, shown in figure 1. Thus a value of type $(SMonad\ m) \Rightarrow m\ x\ y\ a$ represents a computation from the state x, to the state y and produces a value of type a. The functions $(\sim\!\gg)$, $(\sim\!\gg\!=)$ and $sreturn$ should all be treated as equivalents to the normal Haskell monadic functions, $(\gg)$, $(\gg\!=)$ and $return$ respectively. Sadly this means that the normal higher-level functions that work on and manipulate Haskell's normal $Monad$ cannot be used. Furthermore, there are currently no extensions to GHC which allow us to rework *do*-notation to use the methods of the $SMonad$ type class. Instances of the $SMonad$ type class are used extensively throughout this work.

There are six constructs which are used to describe behaviour in a session type: $send$, $recv$, $offer$, $select$, $jump$ and end. Labels are created through the function $newLabel$ and session type fragments are assigned to labels using the infix $(.=)$ function. The function $makeSessionType$ runs the $SMonad$ instance, producing a session type and any values returned by the expression. For example:

```
(s, (a, b)) = makeSessionType (
                  newLabel ∼≫= λa →
                  newLabel ∼≫= λb →
                  a .= send int ∼≫ recv bool ∼≫ jump b ∼≫
                  b .= recv double ∼≫ end ∼≫
                  sreturn (a, b)
                )
  where
    int    = ⊥ :: Int
    bool   = ⊥ :: Bool
    double = ⊥ :: Double
```

Thus s is a session type with two fragments referred to by the labels a and b. The labels are only visible externally because they were *sreturn*ed by the instance. The fragment at a demands that first an *Int* is sent and then a *Bool* is received and then the behaviour switches to the fragment at b which is that a *Double* is received and then the channel is closed. This is only describing behaviour and not implementing behaviour. Of course, this session type is entirely linear, and so there is no need to use two labels and two fragments and a *jump*; a single label and fragment would work perfectly well here. For recursion however, a jump is necessary.

Although s is a value, we only care about the type of s. The type is the important part because it is only the type of the session type that we can work with during type checking, so the value is completely unnecessary: it could very well just be $\perp$. However, whilst the user could define a session type using $\perp$ and an explicit type, we don't believe they would want to: the type is large, complex and quite difficult to read. Haskell has weak support for type functions and so the opportunities for making the type more readable and maintainable are few. Specifying a session type as a value through our DSL allows us to provide an easily read DSL with helpful functions and connectives and allows us to enforce invariants in the type which the user could otherwise violate.

Specifying the types that are communicated is slightly clumsy. For example, $\mathbf{let}\ \mathit{bool} = \bot :: \mathit{Bool}\ \mathbf{in}\ \mathit{send}\ \mathit{bool}$ results in the same session type as $\mathit{send}\ \mathit{True}$ which is the same as $\mathit{send}\ \mathit{False}$: i.e. the value is ignored and only the type captured. Thus for clarity, we use $\bot$ with explicit types.

The *dual* of s, or $\overline{s}$, is:

$$\begin{array}{l}
(\overline{s}, (a, b)) = \mathit{makeSessionType}\ (\\
\qquad\qquad \mathit{newLabel} \sim\!\!\gg\!\!= \lambda a \rightarrow \\
\qquad\qquad \mathit{newLabel} \sim\!\!\gg\!\!= \lambda b \rightarrow \\
\qquad\qquad a \mathrel{.=} \mathit{recv}\ \mathit{int} \sim\!\!\gg \mathit{send}\ \mathit{bool} \sim\!\!\gg \mathit{jump}\ b \sim\!\!\gg \\
\qquad\qquad b \mathrel{.=} \mathit{send}\ \mathit{double} \sim\!\!\gg \mathit{end} \sim\!\!\gg \\
\qquad\qquad \mathit{sreturn}\ (a, b) \\
\qquad\quad) \\
\quad \mathbf{where} \\
\qquad \mathit{int} \quad\;\;\; = \bot :: \mathit{Int} \\
\qquad \mathit{bool} \quad\; = \bot :: \mathit{Bool} \\
\qquad \mathit{double} = \bot :: \mathit{Double}
\end{array}$$

that is, all sends and recvs have been interchanged, whilst ends, jumps and labels remain unchanged.

With jumps, loops (including infinite loops) can specified, for example:

$$\begin{array}{l}
(s, a) = \mathit{makeSessionType}\ (\\
\qquad\qquad \mathit{newLabel} \sim\!\!\gg\!\!= \lambda a \rightarrow \\
\qquad\qquad a \mathrel{.=} \mathit{recv}\ \mathit{string} \sim\!\!\gg \mathit{jump}\ a \sim\!\!\gg
\end{array}$$

```
            sreturn a
          )
    where
      string = ⊥ :: String
```

could be the session type for a process that is forever receiving messages and logging them to file.

Choice structures are supported in session types, which can be likened to a switch statement. The choice contains a statically known list of branches. One party's implementation *offers* an implementation of every branch in the list whilst the other party *selects* which branch to take by supplying index of the of the branch in the list. The index supplied, and hence the branch taken, is dynamically chosen. The session type itself for the choice must provide session types for all branches in both the *offer* and *select* cases. In our library, the constructs *offer* and *select* take special lists of session type fragments.

```
  (s, a) = makeSessionType (
            newLabel ∼≫= λa →
            a .= offer ((recv int ∼≫ jump a)
                         ∼|∼
                        (send int ∼≫ end)
                         ∼|∼ BLNil
                       ) ∼≫
            sreturn a
          )
    where
      int = ⊥ :: Int
```

This shows a session type with a loop (via recursion) that can be exited. The *offer* provides the choice between going around the loop again (by selecting the 0th branch) or exiting the loop (by selecting the 1st branch). The (∼|∼) operator is analogous to (:) and *BLNil* is analogous to [ ] but operate on a special list type. As *send* and *recv* were interchanged by the *dual* relation, so are *offer* and *select*. The *dual* of the above session type is thus:

```
(s̄, a) = makeSessionType (
                newLabel ∼≫= λa →
                a .= select ((send int ∼≫ jump a)
                              ∼|∼
                             (recv int ∼≫ end)
                              ∼|∼ BLNil
                            ) ∼≫
                sreturn a
              )
     where
       int = ⊥ :: Int
```

Referring back to the design for the n-queens checker, we can now state the session type for the required communication, which is shown in figure 2. The initial communication from the *root* to the queens is simply sending the coordinate of the queen; this is the fragment at the label b. This is also the same fragment as is used by the queens when sending their coordinates amongst themselves. The *root* process receiving the result of the check for collisions from each of the queens is the fragment at the label c. The other fragment at the label a is the loop through which the *root* process sends to the queens all the other Pids. Note that the first two branches within the *select* construct are the same fragment and use the helper function *frag*. This gives the two paths through which the *root* process will indicate whether the Pid is *before* or *after* the current queen in question. Also note that the Pid in the fragment in the *frag* helper function is not fully specified and *sendPid* is a function provided by our library. We shall cover why Pids are treated separately and how to use them in a session type in the next section.

```
(st, _) = makeSessionType (
              newLabel ~≫= λa →
              newLabel ~≫= λb →
              newLabel ~≫= λc →
              a .= (select ((frag a) ~|~
                            (frag a) ~|~
                            end      ~|~ BLNil
                           )
                   ) ~≫
              b .= (send (int, int) ~≫ end) ~≫
              c .= (recv bool ~≫ end)
          )
  where
    frag a = sendPid ... ~≫jump a
    int    = ⊥ :: Int
    bool   = ⊥ :: Bool
```

Figure 2. The session type for the n-queens checker

Finally, see that there are several distinct and unrelated fragments all within the same session type. In our library, the session type used must contain every fragment that is ever going to be used by any process. We use the the labels identifying fragments to indicate which fragment should be used to parameterise any given communication channel. The session type does not carry any information regarding interleaving, thus it is possible for implementations to have multiple channels in use at the same time such that a deadlock occurs through a cycle of receives.

3.1 Using Session Types

To implement a session type, or a fragment of a session type, there are five functions which mirror the constructs used for specifying the session type. Only end is treated separately and will be covered later.

As with specifying a session type, manipulating a channel parameterised by a session type is achieved with instances of the $SMonad$ type class. These instances are ultimately wrappers around the IO monad. Consequently, our $SMonadIO$ type class performs the same function as the standard $MonadIO$ type class and defines a function $sliftIO :: IO\ a \rightarrow m\ x\ x\ a$, allowing normal IO actions to be used inside functions that manipulate channels.

Channels are represented by values which are instances of $SMonad$. We provide the functions $ssend$, $srecv$, $sjump$, $sselect$ and $soffer$ which all operate on the channel implicitly represented by the current *SMonad* instance. Thus to implement the session type *send int* ~>> *recv bool* ~>> *jump a* you might write:

```
ssend 42 ~>> srecv ~>>= λb →
sliftIO (print b) ~>> sjump
```

Thus no argument representing the current channel is needed. The only surprise might be that *sjump* does not take an argument indicating the target of the jump. The argument is actually supplied by the session type: there is never any choice as to what the next fragment is so *sjump* does not need the user to supply a destination. The *ssend* function is asynchronous whilst *srecv* blocks. We have a non-blocking receive test, *srecvTest* which returns *True* :: *Bool* if there is a value ready to be received and *False* otherwise. We also have a variant with a timeout, *srecvTestTimeOut*, which takes a single argument as the number of microseconds to wait. If a value becomes ready to be received at any point before the timeout, then the function will immediately return *True*. Otherwise it will block for no more than the specified number of microseconds and return *False*.

```
receivePids beforePids afterPids
 = soffer ((srecv ~>>= λbeforePid → sjump ~>>
              receivePids (beforePid : beforePids)
                          afterPids
           ) ~||~
           (srecv ~>>= λafterPid → sjump ~>>
            receivePids beforePids
```

```
                         (afterPid : afterPids)
        ) ∼||∼
        (sreturn (afterPids, reverse beforePids)
        ) ∼||∼ OfferImplsNil
       )
```

Figure 3. Receiving Pids from the *root* process

```
sendPids 0 [ ]
    = sselect (D2 E)
sendPids 0 (pid : pids)
    = sselect (D1 E) ∼≫ ssend pid ∼≫
        sjump ∼≫ sendPids 0 pids
sendPids n (pid : pids)
    = sselect (D0 E) ∼≫ ssend pid ∼≫
        sjump ∼≫ sendPids (n − 1) pids
```

Figure 4. Sending Pids to a queen process

For implementing choice, *soffer* is very similar to *offer* in that it takes a list of implementations of the branches in the same order as supplied to *offer*. There are some constraints on those implementations such as that they all return a value of the same type and they all ultimately leave the channel in the same state. These are required in order to avoid needing a fully dependent type system:

as the branch taken is not statically known, we must be able to type the channel regardless of which branch is taken. This is really no different than a normal **if** c **then** $b1$ **else** $b2$ structure where $b1$ and $b2$ must have the same type in order to successfully type the **if**-statement. To select a particular branch, *sselect* simply takes as an argument the index of the branch desired. The index is expressed as a type level number: this is required so that we can check statically that the process does go on to implement correctly the branch selected. The numbers are, for convenience, base 10, and not Peano numbers. Each digit constructor starts with a D and the number is terminated by an E. Thus $D5\ \ E$ is a value with the type $D5\ \ E$ which represents the number 5 and $D8\ \ (D1\ \ (D4\ \ E))$ is a value with the type $D8\ \ (D1\ \ (D4\ \ E))$ which represents the number 814. Like *ssend*, *sselect* is asynchronous whilst like *srecv*, *soffer* blocks. We can now implement both the queen receiving the Pids from the *root* process and the *root* process sending the Pids to a particular queen. These are shown in figures 3 and 4 respectively.

The recursion in *receivePids* is natural: *sjump* must be called in order to force the *jump* a element of the session type fragment to be followed, and then there is recursion back to *receivePids*. *OfferImplsNil* is again, the equivalent of $[\,]$ whilst $(\sim||\sim)$ is the equivalent of (:). These specialised lists are required in order to enforce the constraints mentioned above. For *sendPids*, the function receives the index of the current queen being talked to and a list of Pids, such that for queen q_m, the first $m-1$ elements of the list are Pids *before* q_m whilst the remainder are Pids *after* q_m. Thus *sendPids* implements the fragment at a of the session type shown in figure 2 whilst *receivePids* implements the *dual* of the fragment at a of the same session type.

To run an implementation of a session type, we call run, supplying it with the session type, the label of the starting fragment and the two implementations. The first implementation must be of the session type as written, and the second implementation must be of the *dual* and they both must start at the fragment indicated by the label otherwise a compile-time error will occur. The run function creates two new threads, one for each implementation and creates the channel between them. It then blocks, waiting for both threads to finish before returning the results from both implementations as a tuple.

4. Interlude: On the problem of Pids

Creating only a single channel between two processes is very limiting, and we want to cater for a much broader range of communication patterns. We therefore have a $fork$ function which creates a new process and a $createSession$ function which allows two existing processes to create a new communication channel between them. Processes are identified by Process IDs, or Pids, which are normal values, which means that we can communicate Pids between processes.

Our $createSession$ function is synchronous and requires that both processes hold the Pid of each other. They must both call $createSession$, supplying, amongst other arguments, each other's Pid. Ideally, we would like to statically know that if one process is going to call $createSession$ with a particular Pid, then the process identified by that Pid is also going to perform the reciprocal call to *createSession*. If this is not the case then the call to *createSession* will block indefinitely. We therefore parameterise Pids by both the session type and the list of the labels within the session type at which the Pid is prepared to start a new channel. This is imperfect information: the code path taken dynamically by the process will determine which sessions the process really will take part in, and cannot be known statically. Therefore the set of fragments indicated by the Pid's type is an optimistic set in that if a fragment label does not appear in the Pid's type then it is certain that the process will not take part in a session at that label, but if a label does appear in the set it does not guarantee that the process will perform the necessary call to *createSession* with that fragment label.

The type of a Pid in our library contains many items, including the complete session type of the program, and is a fairly large structure. This presents a problem when specifying a Pid in a session type as you will end up with something that looks a lot like:

```
(s, _) = makeSessionType
           (newLabel ∼≫= λa →
            a .= send (Pid s...) ∼≫ end)
```

Sadly, GHC objects to infinite types (the type of session type s contains itself due to the type of s being in the type of *Pid*) and so we must use another form of representing Pids in session types. We have two constructs: *sendPid* and *recvPid*.

For two Pids to agree to create a channel between them, they must start at the same fragment with one process performing the fragment as written and the other process performing the *dual* of the fragment. Therefore, we use type level booleans (*dual* :: *True* and *notDual* :: *False*) to indicate whether the process is going to perform the *dual* of the fragment or the fragment as written. The two functions $sendPid$ and $recvPid$ therefore take type level lists of tuples where each tuple contains the label of a fragment and a $dual$ or $notDual$ type level boolean.

For example, if we receive a Pid which is parameterised by (amongst other things) the tuple $(a, dual)$ then we know that the process may agree to participate in a channel starting at the fragment at the label a where the process is performing the *dual* of the fragment and we are performing the fragment as written. The Pid is always parameterised by what *that* process is going to do, so *we* must do the opposite.

In order to avoid confusion, Pids always carry the session type in the same form: it is never the case that one Pid carries s and the other Pid carries $\overline{s}$ (or any other session type). This is a unique property of our implementation: other implementations do not typically model process identifiers in the same way but rather use a host-name and port or URL as an identifier. This makes sense for a network and distributed focus, but does not makes sense for our focus on message passing concurrency. Having different Pids parameterised by different session types becomes very confusing very quickly and significantly complicates checking that the calls to $createSession$ from each party indicate the same (but dual) session type fragment. As such, enforcing that a single session type is used across all Pids in the system simplifies these checks considerably.

Going back to the session type for our n-queens checker, shown in figure 2, we wish to send the Pid of a queen which means we need to consider the channels that a queen is going to take part in. It will start by receiving its own coordinate from the *root* process. This is the *dual* of the fragment at the label b so $(b, dual)$. Then it will receive the Pids of the other queens from the *root* process. This is the *dual* of the fragment at a, so $(a, dual)$. It will then send its own coordinate to other queens, which is the fragment at b as written, so $(b, notDual)$ and it is also going to receive from other queens some coordinates, which is the *dual* of the fragment at b, so $(b, dual)$, which we've already got. Lastly, it must report its result to the *root* process. This is the *dual* of the fragment at c, so $(c, dual)$. The order in which these channels will be created is irrelevant as the type of the queen's Pid cannot be dynamically updated as the queen makes progress. Therefore we only care about the set of tuples. Thus sending the Pid of a queen is specified as

$$\begin{array}{lll} sendPid\ (cons\ (a, dual) & & \$ \\ \qquad\qquad cons\ (b, notDual) & & \$ \\ \qquad\qquad cons\ (b, dual) & & \$ \\ \qquad\qquad cons\ (c, dual) & & \$\ nil) \end{array}$$

The complete correct session type for our n-queens checker is shown in figure 5 (the list is pre-declared here in a **let**-clause as it is used more than once). The list is always automatically sorted so the indices can be specified in any order.

Whilst representing Pids in session types requires special treatment, sending and receiving Pids in the implementation is done as normal using the usual *ssend* and *srecv* functions. When sending a Pid, the Pid may support starting channels at a greater set of labels than is required by the session type. This is acceptable as long as it is a superset. When the Pid is received, it is converted so that it only supports the set of labels indicated by the session type - this is in effect subtyping. The process indicated by the Pid is of course unaffected: in is only the view of the Pid to the receiving process that is altered.

5. Forking and Interleaving

Actions directly manipulating a channel form instances of *SMonad* which can then be chained together. Similarly, actions to create, modify, interleave actions upon and destroy channels also are instances of *SMonad* which can similarly be combined. Whilst a few useful combinators have been defined, the primitive functions are *fork* and *createSession*.

The function *fork* takes four arguments: a fragment label; a type level boolean (*dual* or *notDual*); a list of tuples of labels and type level booleans; and an function to be called as the child process. The function *fork* creates a new thread which will execute the implementation supplied, creates a channel between the current, parent, process and the child processe and returns to the parent

```
(st, (childSessions, rootSessions
     , commPids, commCoords, commResult
     ))
     =   makeSessionType $
         newLabel ~≫= λa →
         newLabel ~≫= λb →
```

```
    newLabel ~>>= λc →
    let queenFragList = (cons (a, dual) $
                         cons (b, notDual) $
                         cons (b, dual) $
                         cons (c, dual) $ nil) in
    a .= (select ((sendAPid a queenFragList) ~|~
                  (sendAPid a queenFragList) ~|~
                  end ~|~ BLNil
                 )) ~>>
    b .= (send (int, int) ~>> end) ~>>
    c .= (recv bool ~>> end) ~>>
    sreturn (queenFragList
            , (cons (a, notDual) $
               cons (c, notDual) nil)
            , a, b, c)
  where
    sendAPid a lst = (sendPid lst ~>> jump a)
    int = ⊥ :: Int
    bool = ⊥ :: Bool
```

Figure 5. The full session type for the n-queens checker

process which called *fork* the channel specified by the first two arguments: the label and the boolean. The Pid of the new child process is also returned and is parameterised by the list supplied as the third argument. An example should make this clearer:

```
(st, a) = makeSessionType (
              newLabel ∼≫= λa →
              a .= recv int ∼≫ send bool ∼≫ end ∼≫
              sreturn a)
    where
            int = ⊥ :: Int
            bool = ⊥ :: Bool

parent = fork a dual (cons (a, notDual) nil) child
          ∼≫= λ(childCh, childPid) →
          withChannel childCh (ssend 52 ∼≫ srecv)
          ∼≫= sliftIO ∘ print

child parentCh parentPid
        = withChannel parentCh
              (srecv ∼≫= ssend ∘ (≡) 42)
```

Thus the call to *fork* produces a channel which is forced to obey the session type fragment at label a. The *parent* sees the *dual* whilst the *child* sees the *notDual* end of the channel. We also see here the use of *withChannel*. This is the main mechanism for interleaving actions across different channels. It takes the channel and the function to run on the channel. We see the *child* is parameterised by the channel to the parent and the parent's Pid.

There is no obligation to use the channel created: given that both processes have each other's Pid, they could very easily create a new channel between them. This is achieved through the function *createSession*. The first two arguments to *createSession* are the same as for *fork*: i.e. they specify the local end of the new channel. The third argument is the Pid of the other process and that must be parameterised by (amongst other things) the same fragment label

$$\begin{array}{lll}
\mathit{ssequence\_} & :: (\mathit{SMonad}\ m) \Rightarrow [m\ x\ x\ a] \rightarrow m\ x\ x\ () & \\
\mathit{ssequence\_}\ [\,] & = \mathit{sreturn}\ () & \\
\mathit{ssequence\_}\ (f : \mathit{fs}) & = f \sim\!\gg \mathit{ssequence\_}\ \mathit{fs} & \\
\mathit{ssequence} & :: (\mathit{SMonad}\ m) \Rightarrow [m\ x\ x\ a] \rightarrow m\ x\ x\ [a] & \\
\mathit{ssequence}\ \ [\,] & = \mathit{sreturn}\ [\,] & \\
\mathit{ssequence}\ \ (f : \mathit{fs}) & = f \sim\!\gg\!= \lambda r \rightarrow & \\
 & \quad \mathit{ssequence}\ \mathit{fs} \sim\!\gg\!= \lambda \mathit{rs} \rightarrow & \\
 & \quad \mathit{sreturn}\ (r : \mathit{rs}) &
\end{array}$$

Figure 6. Definitions of *ssequence* and *ssequence_* for *SMonad*

as the first argument and the opposite type level boolean (i.e. *dual* for *notDual* and vice versa). An example should help:

$$\begin{array}{l}
(\mathit{st}, a) = \mathit{makeSessionType}\ (\\
\qquad \mathit{newLabel} \sim\!\gg\!= \lambda a \rightarrow \\
\qquad a \mathrel{.{=}} \mathit{recv}\ \mathit{int} \sim\!\gg \mathit{send}\ \mathit{bool} \sim\!\gg \mathit{end} \sim\!\gg \\
\qquad \mathit{sreturn}\ a) \\
\quad \mathbf{where} \\
\qquad \mathit{int} = \bot :: \mathit{Int} \\
\qquad \mathit{bool} = \bot :: \mathit{Bool}
\end{array}$$

```
parent = fork a dual (cons (a, notDual) nil) child
           ~>>= λ(_, childPid) →
           createSession a dual childPid
           ~>>= λchildCh →
           withChannel childCh (ssend 52 ~>> srecv)
           ~>>= sliftIO ∘ print

child _ parentPid
         = createSession a notDual parentPid
           ~>>= λparentCh →
           withChannel parentCh
             (srecv ~>>= ssend ∘ (≡) 42)
```

Here, the channel created through *fork* is completely ignored and *createSession* is used to set up the channel between the *parent* and the *child*. The only piece of information missing in both of these examples is the list of tuples of fragment labels and type level bools that parameterise the Pid of the *parent*. This is supplied to the *runInterleaved* function which performs a similar role to *run* but at this higher level of encapsulation. *runInterleaved* takes the list of tuples of labels and bools, the full session type and the function that is to be executed as the *root* process. The function runs the supplied function directly, it doesn't create any new threads, in contrast to *run*. A process can access its own Pid through the *myPid* function.

Now that we can talk about channels rather than just operate implicitly upon them, we can also add the ability to close a channel. The function *scloseCh* removes the channel indicated from the set of currently available channels. Any subsequent attempt to use this channel with *withChannel* or any other function (including *scloseCh*) will result in a compile-time error.

The ability to close a channel turns out to be very useful as creating a channel, using it and then closing it allows the *to* state to remain the same as the *from* state (that is, the current SMonad instance type remains at $(SMonad\ \ m) \Rightarrow m\ \ x\ \ x\ \ a$ rather than $(SMonad\ \ m) \Rightarrow m\ \ x\ \ y\ \ a$). Therefore, we can use this pattern to construct lists of functions and then sequence them where the length of the list isn't known statically: i.e. a normal Haskell list of type $(SMonad\ \ m) \Rightarrow [m\ \ x\ \ x\ \ a]$. To be able to work with dynamically determined numbers of channels gives us many useful programming patterns. The functions $ssequence$ and $ssequence\_$ are defined which work for $SMonad$ instances and are shown in figure 6.

Without this ability to perform work without changing the type of the process we would be trying to build a list such that the first value is of type $(SMonad\ \ m) \Rightarrow m\ \ x\ \ y\ \ a$, then the second value would be of type $(SMonad\ \ m) \Rightarrow m\ \ y\ \ z\ \ a$ and so forth, thus the final *to* state would depend on the length of the list. Alternatively, we could easily build a list of type $(SMonad\ \ m) \Rightarrow [m\ \ x\ \ y\ \ a]$, but then sequencing this list would not be possible as the *to* state of the first value, y, would not be equal to the *from* state of the second value, x.

This pattern of creating, using and closing a channel is so useful that we've defined three other combinators, $withThenClose$, $forkThenClose$ and $createSessionThenClose$ that all have their obvious behaviour but close the channel as their last operation, returning the result of the rest of the actions performed on the channel.

We can now finish our implementation of the n-queens checker. Working from the simplest part upwards, the actual collision detection is straightforward: we just check to see if the two coordinates being compared at the same row, or same column or on a diagonal from each other:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{detectCollision}\ (x, y)\ (x', y')} \\
\quad = & & x \equiv x' \\
 & \vee & y \equiv y' \\
 & \vee & (\mathit{abs}\ (x' - x)) \equiv (\mathit{abs}\ (y' - y))
\end{array}$$

This is called by a queen, so we can now look at the rest of the definition of the queen:

$$\begin{array}{l}
\mathit{queen}\ \mathit{rootCoordCh}\ \mathit{rootPid} \\
\quad = \mathit{withThenClose}\ \mathit{rootCoordCh}\ \mathit{srecv} \sim\!\gg\!= \lambda(x, y) \rightarrow \\
\qquad \mathit{createSessionThenClose}\ \mathit{commPids}\ \mathit{dual}\ \mathit{rootPid} \\
\qquad\quad (\mathit{receivePids}\ [\,]\ [\,]) \sim\!\gg\!= \lambda(\mathit{beforePids}, \mathit{afterPids}) \rightarrow \\
\qquad (\mathit{smapM}_{-}\ (\lambda p \rightarrow \\
\qquad\quad \mathit{createSessionThenClose}\ \mathit{commCoords}\ \mathit{notDual}\ p \\
\qquad\qquad (\mathit{ssend}\ (x, y))) \\
\qquad\quad \$\ \mathit{afterPids}) \sim\!\gg \\
\qquad (\mathit{smapM}\ (\lambda p \rightarrow \\
\qquad\quad \mathit{createSessionThenClose}\ \mathit{commCoords}\ \mathit{dual}\ p \\
\qquad\qquad \mathit{srecv}) \\
\qquad\quad \$\ \mathit{beforePids}) \sim\!\gg\!= \lambda \mathit{beforeCoords} \rightarrow \\
\qquad \mathit{createSessionThenClose}\ \mathit{commResult}\ \mathit{dual}\ \mathit{rootPid} \\
\qquad\quad (\mathit{ssend} \circ \mathit{or}\ (\mathit{map}\ (\mathit{detectCollision}\ (x, y)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{beforeCoords}))
\end{array}$$

The queen first receives from the *root* process its own coordinates. It then creates a new channel to the *root* process and calls $\mathit{receivePids}$ as in figure 3. This produces the lists of Pids of queens, before and after the current queen. Then, to every queen after it, it sends its own coordinates, and from every queen before it it receives coordinates, producing the list $beforeCoords$. The functions $smapM$ and $smapM\_$ are the equivalents of $mapM$ and $mapM\_$ which are the compositions of $ssequence$ and $ssequence\_$ respectively with map. It then creates a channel to the *root* process and reports the result of the collision detection between itself and the coordinates it has received. The values $commPids$, $commCoords$ and $commResult$ are all labels exposed with the session type, shown in figure 5.

Given that channel creation is synchronous and both parties must perform reciprocal calls to $createSession$ for the channel to be created, it may not be apparent that the behaviour described above cannot deadlock. The last queen, q_{n-1} does not have any Pids *after* it, so it starts by receiving from the queens *before* it, q_{n-2} then q_{n-3} and so on up to q_0. The queen q_{n-2} only has one queen *after* it, q_{n-1} and so once it has sent its own coordinate to q_{n-1}, it then moves to receiving from the queens *before* it, q_{n-3} and so on. Thus the Pids must be carefully arranged to ensure that

deadlocks do not occur. This is the reason for $reverse$-ing the list of $beforePids$, shown in figure 3.

To create the queens, the *root* process calls $fork$ a number of times through a normal recursive function. The number of recursive calls is determined by the number of coordinates supplied by the user which means we need to use the channel (if we wish to) and then immediately close it in order to be able to successfully type the recursion. This then explains why a separate channel is created to send the Pids to the queens: using the channel created through the *fork* call would demand that we know the full list of Pids before the first *fork* call which we clearly cannot. We cannot wait until after all the *fork* calls to use these channels as we can't give a type to a list of channels or perform the recursion: the channels have to be closed for the recursion to be acceptable. But because every call to *fork* creates a new queen and all those queens have the same Pid type, the Pids can be placed in the same list:

```
makeQueens [] pids
  = sreturn pids
makeQueens (coord : coords) pids
  = forkThenClose commCoords notDual childSessions
        queen (ssend coord) ∼≫= λ(_, pid) →
    makeQueens coords (pid : pids)
```

Again, *commCoords* and *childSessions* are defined with the session type in figure 5, and the function *queen* is defined in the previous code example above. Having created all the queens and gathered their Pids together, we must now distribute the Pids to all the queens. This makes use of *sendPids* which has already been shown in figure 4. Effectively we cycle through the list of Pids, creating a channel to each queen and using that channel to send all the other Pids of queens appropriately:

```
distributePids 0 _ = sreturn ()
distributePids n (pid : pids)
  = createSessionThenClose commPids notDual pid
        (sendPids (n − 1) pids) ∼≫
        distributePids (n − 1) (pids ⧺ [pid])
```

The *root* process must also gather in the results from all the queens. Again, the same list of Pids of queens is iterated through and a channel is established to each queen in turn, receiving the result from that queen.

```
    gatherResults [] = sreturn False
    gatherResults (pid : pids)
      = createSessionThenClose commResult notDual
            pid srecv ~≫= λcollision →
          if collision
            then sreturn collision
            else gatherResults pids
```

The iteration stops as soon as one collision is reported. Whilst this does mean the correct result is returned to the user and you could argue that this is efficient, it does mean that queens that are after any queen that reports a collision will block indefinitely, awaiting a channel to the *root* process in order to return their result. For this reason, the following version is preferred, which does not have this deficiency, at the cost of collecting from all queens before reporting a result:

```
    gatherResults [] = sreturn False
    gatherResults (pid : pids)
      = createSessionThenClose commResult notDual
            pid srecv ~≫= λcollision →
          gatherResults pids ~≫= λcollisions →
          sreturn (collision ∨ collisions)
```

Finally, we must finish the *root* process, linking together these pieces:

```
    hasCollision coords
```

```
  = runInterleaved rootSessions st
```

```
      (makeQueens coords [] ~≫= λpids →
       distributePids (length coords) pids ~≫
       gatherResults pids
      )
```

where *st* is the full session type and *rootSessions* is defined with the session type in figure 5. And that is it: running *hasCollision* with a list of coordinates performs all the work described. As mentioned in section 2, the bulk of the code of this example is concerned with the communication and very little is actually doing the work of checking for collisions: the *detectCollision* function is trivial. However, the communication patterns involved here are entirely non-trivial and demonstrate how our Session Type library can be used to tackle such patterns. Furthermore, the guarantees of ordering, the ability to express communication patterns involving branches and loops, conformance to which is statically asserted, the ability to communicate values of any type and freedom from mistakenly attempting to receiving when should be sending (for example) are all highly desirable properties which go a long way to making message passing safer and more useful.

5.1 Multi-receive

Frequently it is the case that a particular process has several channels open and on a number of them, the next action is a *recv* of some value. The process doesn't know which value is going to be ready first and wants to block on all of the channels until one of them becomes ready. In some cases, blocking only on a single channel could result in a deadlock if the wrong channel is chosen. Our multi-receive construct supports this need. It takes a list of tuples of channels and functions such that the function associated with the first channel that becomes ready is called. This is implemented without polling.

Like with $soffer$, the functions supplied must all leave the process in the same state in order to be correctly typed. Again, were this not the case then the type of the process would be dependent on many factors not statically known including the behaviour of the scheduler. This typically means that the functions will do the same work, but not necessarily in the same order. We statically check that the channels indicated in the list passed to $multiReceive$ are due next to perform a $recv$ or $offer$ action. As this requires the channels that are to be used to be statically known and present in the type of the list, it is not possible to build the list of tuples without the length of the list being statically known. This is why we have not used this construct in our running example (we could have used it to gather the results of the collision detection but this would require that the the number of queens be known statically). The example in figure 7 demonstrates the function in use.

The $parent$ process forks two children which are parameterised by the different values $aDelay$ and $bDelay$ which control how long they sleep for before replying to the $parent$. The $parent$ uses the $multiReceive$ functionality to ensure that no matter which child replies first, it will see the replies in the order they are sent.

5.2 Higher-order Channels

It is often useful to send one channel over another, to delegate a channel to another party. This would have to be reflected in the session type which would have to capture both that a channel is to be sent or received and the type of the channel being communicated. To indicate the communication of a channel in a session type we have the constructs *sendSession* and *recvSession*, and to send and receive a channel we have the functions *sendChannel* and *recvChannel*.

```
(st, (a, b)) = makeSessionType (
      newLabel ∼≫= λa →
      newLabel ∼≫= λb →
      a .= sendSession (recv int ∼≫ end) ∼≫ end ∼≫
```

```
test aDelay bDelay = runInterleaved nil st parent
  where
      (st, x) = makeSessionType (
                        newLabel ∼≫= λx →
                        x .= send int ∼≫ end ∼≫
                        sreturn x)
        where
           int = ⊥ :: Int

      parent
         = fork x dual nil (child aDelay)
               ∼≫= λ(aCh, _) →
           fork x dual nil (child bDelay)
               ∼≫= λ(bCh, _) →
           multiReceive ((aCh, receive "A" aCh ∼≫
                              receive "B" bCh)
                         ~|||~
                         (bCh, receive "B" bCh ~>>
                              receive "A" aCh)
                         ~|||~ MultiReceiveNil
                        )
      receive str ch
        = withChannel ch
            (srecv ~>>= sliftIO ∘ putStrLn ∘
              (++) ("Parent received from child "
                   ++ str ++ ": ") ∘ show)
      child delay parentCh _
        = (sliftIO (threadDelay delay)) ~>>
            withChannel parentCh (ssend delay)
```

Figure 7. The *multiReceive* function in use

```
      b .= send int ~>> recv int ~>> end ~>>
      sreturn (a, b))
    where
      int = ⊥ :: Int
```

This shows two fragments with labels a and b and the first action in the a fragment is to send a channel such that the channel being sent is due to receive an *Int* and then stop. Note that the fragment specified within the *sendSession* or *recvSession* can refer to labels defined outside, for example *sendSession* (*recv int* ~>> *jump b*).

So now we can make a master process fork off two children and then send the master's end of the channel between the master process and the first child to the second child. The two children will thus share a channel without ever knowing of each other. The first child is completely unaware that the process at the other end of this channel has changed from the master to the second child: the delegation is transparent.

$$\begin{aligned}
master = {} & fork\ b\ notDual\ nil\ childA \leadsto\!\!\gg\!= \lambda(chA, \_) \rightarrow \\
& fork\ a\ notDual\ nil\ childB \leadsto\!\!\gg\!= \lambda(chB, \_) \rightarrow \\
& withChannel\ chA\ (ssend\ 41) \leadsto\!\!\gg \\
& sendChannel\ chA\ chB
\end{aligned}$$

$$\begin{aligned}
& childA\ chP\ \_ \\
& \qquad = withChannel\ chP\ (srecv \leadsto\!\!\gg\!= ssend \circ (+)\ 1)
\end{aligned}$$

$$\begin{aligned}
& childB\ chP\ \_ \\
& \qquad = recvChannel\ chP \leadsto\!\!\gg\!= \lambda chA \rightarrow \\
& \qquad\quad\ withChannel\ chA\ (srecv \leadsto\!\!\gg\!= sliftIO \circ print)
\end{aligned}$$

Also note the partial application: the channel is created at the label b and then first action is applied (the $send\ \ int$). This leaves the channel with the type $recv\ \ int \leadsto\!\!\gg end$ which matches fragment within the $sendSession$ construct.

6. Implementation

Our library is not particularly large, weighing in at only 3500 lines of Haskell. However, the majority of that is not runtime code, but compile time code: type classes and function signatures that allow us to enforce the requirements of Session Types and to check the validity of implementations of a given session type statically. We do not make use of any *dirty* tricks such as manual type coercions, calling IO actions unsafely or incoherent type class instances. We do however make use of most other extensions available in GHC 6.8 including type families.

Type level numbers, lists, maps and booleans with basic control flow structures have all been covered before, e.g. [18]. Whilst our particular implementations of these may differ in some details, there is nothing significantly novel about these.

6.1 Acting on a Channel

Channels between processes are bidirectional. In order to allow sending to be asynchronous, we model a channel as two unidirectional queues such that the *outgoing* queue for one process is the *incoming* queue for the other process and vice versa. Processes *enqueue* values onto their outgoing queue and *dequeue* values from their incoming queue. These queues are implemented through chains of $Cell$s where each $Cell$ contains the value being communicated and an $MVar$ pointing to the next $Cell$, as shown in figure 8. The type of the $Cell$ contains all the types of the values that are to be communicated in the current fragment for the particular chain. This requires projecting the fragment from the session type onto the *outgoing* and *incoming* queues. For example, given a fragment of $send\ int \leadsto\!\!\gg send\ bool \leadsto\!\!\gg recv\ char \leadsto\!\!\gg send\ double \leadsto\!\!\gg end$, the type of the *outgoing* queue would be based on $int \leadsto\!\!\gg bool \leadsto\!\!\gg double \leadsto\!\!\gg end$ whilst the type of the *incoming* queue would be based on $char \leadsto\!\!\gg end$. Directionality is deliberately lost as what is a $send$ for a queue when treated as *outgoing* is a $recv$ when treated as *incoming*. This projection exposes the possibility of performing actions in the wrong order as it becomes possible to send a value even though the next action according to the full session type fragment is to receive a value, and vice versa. This is solved by carrying around a full copy of the current fragment which is used to check that the action being performed really is the next action due, rather than just the next action for the relevant queue.

However, because sending is asynchronous, it is always safe to promote sends over receives: sending a value *early* can never cause a deadlock. Therefore, when calling $ssend$, it is not required that the next action according to the current fragment is a $send$, it is only required that there be zero or more $recv$ actions before the next $send$. If this is the case and the type of the value to be sent matches with the session type then the send is permitted and the current fragment rewritten, removing the promoted send.

Sending a value therefore consists of taking the value to be sent, using this with a new empty $MVar$ to create a new $Cell$ and writing this into the current $MVar$ in the *outgoing* queue. The new empty $MVar$ is then the hole for the next value to be sent to be written to. Receiving a value is simply the inverse: taking the $Cell$ from the $MVar$ that is currently the *incoming* queue and deconstructing it to reveal the value sent and the $MVar$ for the next value to be received. The structure of queues is shown in figure 9 where a box indicates an $MVar$ and the dotted lines indicate where the next queue head would be after either sending (were this an *outgoing* queue) or receiving (were this an *incoming* queue).

Selection of a branch uses a specialised $Cell$ constructor which exchanges the index taken, but as an *Int* rather than a type level number. Using a type level number here would carry that number in the type of the *Cell* and which would mean the selection of a particular branch must be statically known rather than dynamically

$$\begin{array}{l}
\textbf{data}\ \mathit{Cell} :: * \to *\ \ \textbf{where} \\
\quad \mathit{Cell} :: \mathit{val} \to \mathit{MVar}\ (\mathit{Cell}\ \mathit{nxt}) \to \mathit{Cell}\ (\mathit{Cons}\ \mathit{val}\ \mathit{nxt}) \\
\quad \mathit{SelectCell} :: \mathit{Int} \to \mathit{Cell}\ (\mathit{Cons}\ (\mathit{Choice}\ \mathit{jumps})\ \mathit{Nil}) \\
\textbf{data}\ \mathit{ProgramCell} :: * \to *\ \ \textbf{where} \\
\quad \mathit{ProgramCell} :: \mathit{MVar}\ a \to \mathit{MVar}\ (\mathit{ProgramCell}\ a) \to \\
\qquad\qquad\qquad\qquad \mathit{ProgramCell}\ a
\end{array}$$

Figure 8. *Cell*s and *ProgramCell*s

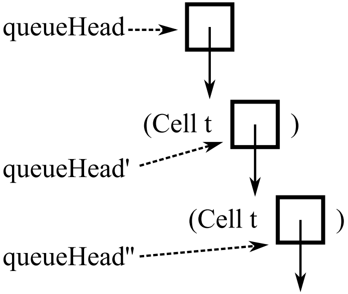

Figure 9. Queues: chains of *Cell*s

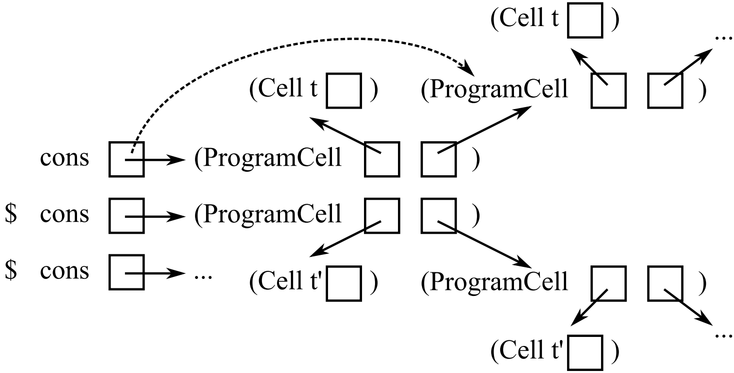

Figure 10. Program Cell structure

chosen. Our type level numbers contain the ability to be converted to Ints. Because the list of implementations of branches on the $soffer$ side of the branch all leave the channel with the same type, it is perfectly possible to iterate through the implementations, finding and executing the implementation corresponding to the received Int index.

As the type of the $Cell$ contains the types of the values exchanged, and because types must be finite, it is therefore necessary for the chains of $Cell$s to be finite. This is the motivation behind treating a session type as a list of fragments which can refer to one another as each fragment is necessarily finite. When a $jump$ is reached we must switch to a new chain of $Cell$s in such a way that both processes involved in the channel find the same next *Cell* for both their *outgoing* and *incoming* queues. This is achieved by two chains of *ProgramCell*s where each *ProgramCell* contains both an *MVar* to the starting *Cell* of a fragment, and an *MVar* to the next *ProgramCell* of the same type, also shown in figure 8. Two lists of these *ProgramCell*s are carried around, one for *outgoing* queues and one for *incoming* queues. When an *sjump* is performed, the lists are indexed by the fragment label in the jump, revealing *MVar*s to *ProgramCell*s. These *MVar*s are carefully managed to ensure that the new *Cell*s for the *outgoing* and *incoming* queues are exchanged safely and asynchronously. The new *Cell*s replace the old *Cell*s, providing the next pair of *outgoing* and *incoming* queues. This is shown pictorially in figure 10 where boxes indicate *MVar*s as before and the dotted line shows the update to the *MVar* that would occur for a jump to the first fragment index.

The chains of *Cell*s and *ProgramCell*s and other items such as the session type are all held together by a *SessionState* value. *SessionChain*s represent computations that manipulate *SessionState*s and are very similar to the normal *State* type in Haskell:

```
newtype SessionChain ... from to res
    = SessionChain { runSessionChain ::
                         (SessionState ... from) →
                         IO (res, SessionState ... to)
                     }
```

An instance of *SMonad* is then defined for *SessionChain* that allows these values to be chained together to form longer sequences of actions acting on a channel as we have seen.

6.2 Creating, Interleaving and Deleting Channels

The management of channels involves carrying the process's own Pid, several type level maps and some other details. The maps are used to access individual channels: the "channels" that are returned to the user as a result of calls to $fork$ or $createSession$ are actually just type level numbers which are used as indices into these maps. Modifying a channel through $withChannel$ uses the index to select the relevant channel, perform the indicated work on the channel and then update the map with the new channel state and the same index. Closing a channel with $scloseCh$ requires that the channel has reached an end in its current fragment and consists of removing the entry from the map.

Creating a new channel through $createSession$ is however, a little more complex. Whilst both processes know of each other as they both have the Pid of the other, they must agree on *where* to exchange values to allow them to create the new channel. The Pids contain a number of structures, one of which is a type level map, allowing Pids to pair with one another and exchange values to allow them each to build a new $SessionState$ for the new channel. This is achieved through use of $MVar$s and normal maps amongst others. The structure is fairly fine grained in that multiple processes can access and modify these structures to some extent concurrently. This pairing is always performed in the Pid which is going to perform the session type as written, rather than the *dual* of the session type. Any consistent strategy here would work equally well.

Whilst multiple Pids can have the same type (which we usefully exploit in our n-queens checker), they will always have different values, as you'd expect. The Pids contain a $RawPid$ which is really just an $[Int]$. The *root* process is always $[0]$ and its children will be $[0, 0]$ to $[0, n]$. Their children will be $[0, 0, 0]$ to $[0, n, m]$ and so on. Thus the $RawPid$ actually contains information indicating a process's ancestry.

The implementation of $multiReceive$ adds further values to $SessionState$. The $multiReceive$ function walks the set of indices of channels, extracting the channels from the type level map and modifying the $SessionState$s slightly. Both the *outgoing* and *incoming* queues within a $SessionState$ have associated with them an $MVar\ (Maybe\ (Chan\ ()))$ and these $MVar$ values are common and inverted at both ends of the channel just as with the *outgoing* and *incoming* queues themselves. The $multiReceive$ function accesses these $MVar$s for the *incoming* queue of each channel and places in all of them the same new $Chan\ ()$. The processes at the other end of the channels observe this $Chan\ ()$ on the *outgoing* queue and write to it whenever they send a value. This is sufficient to wake up the $multiReceive$ process which then identifies which channel has been written to and selects and executes the correct function. Additional code ensures that if a channel is already ready to read from, then the process does not block on the $Chan\ ()$. This same mechanism is used to implement $srecvTestTimeOut$.

7. Evaluation and Future Work

Our library coexists very peacefully with other concurrency techniques within Haskell and GHC in particular. It is perfectly possible to make direct use of traditional shared mutable structures such as $MVar$s and $Chan$s from within code that manipulates our communication channels and it is just as possible to use session types to communicate such structures: effectively communicating pointers. The same is true of GHC's support for Software Transactional Memory; all these techniques can be used as and when they are most appropriate.

We have shown through examples the use of Session Types and our implementation, and it is hopefully clear that Session Types offer different and useful properties to the programmer that traditional locking mechanisms or transactional memory do not: only Session Types are able to express temporal dependencies and allow safe heterogeneously typed communication. On the other hand, it is perfectly possible to deadlock when using message passing and our implementation does not protect against that. This is in contrast with transactional memory which can never deadlock. Incorrectly implementing a session type results in compile-time errors and the program being rejected. The mistakes that are caught are:

- sending or receiving a value of the wrong type,
- performing the wrong action (e.g. a receive when the correct action is a send) - although $ssend$s can safely be promoted over receives,
- selecting a non-existent branch or implementing non-existent branches,
- not implementing every branch of a choice,
- trying to create a channel with a Pid which is never going to try

to create the reciprocal channel end,

- trying to create a channel which is not permitted by the process's own Pid,
- trying to close a channel which has not reached its end.

The restrictions placed on the user of our library in terms of the programming patterns they can use ensure safety and are a consequence of the limitations of Haskell's type system (in that only a dependent type system would be able to overcome these issues), the properties of Session Types and the non-determined dynamic behaviour of a program.

Our library also demonstrates the extent of the flexibility of Haskell's type system, in particular with the extensions offered by GHC and shows how it can be successfully harnessed to build powerful systems which otherwise would require either a full preprocessor or modifications to the compiler.

Performance testing is tricky because of the behaviour of the GHC runtime environment. For example, a single $Chan$ between two threads where one is a producer and one is a consumer performs more than ten times worse with two OS-level threads than with one, even with all the optimisations turned up to 11. The same communication pattern using Session Types shows that with one OS-level thread we're twice as slow as the simple $Chan$ version, but with multiple OS-level threads we're more than twice as fast. These tests were performed on a dual-core Pentium, with the multi-threaded GHC runtime. Obviously, for concurrent programming in Haskell to be successful, the ability to use more than one OS-level thread and hence more than one CPU-core and achieve a performance improvement is necessary. Our conclusion is that our library does not seem to impose any substantial overhead on performance. This correlates with our runtime code being so small and lightweight: the bulk of our library is type-level programming that affects compilation rather than runtime.

7.1 Future Work

Hopefully, we have demonstrated that our library is already a viable and practically useful implementation of Session Types and can be added to the quiver of tools to tackle multi-threaded programming. However, this work is by no means finished and there are many features that we wish to add.

The most significant need is dealing with error messages from GHC. Whilst not needing to modify any compiler and being able to implement our entire library in Haskell (albeit with some recent extensions to the language that are available with GHC), it is not particularly helpful to a user of our library to receive back a thousand line error message [sic] from GHC talking in terms of our library rather than in terms of Session Types. There are two possible solutions to this problem. The first is to try and parse the error messages and detect from those what the problem could be in terms of Session Types. This could be reasonably quick to implement, but would be very fragile in that it would depend on the output format from GHC. GHC does now have an API through which it can be called and this may allow a more robust means to obtain the errors. The second solution would be to implement some form of external checker that could be run on the program prior to compilation (or interpretation). This would then allow us to analyse the session types and their implementations directly and spot errors, reporting them in more humanly accessible terms. This would obviously be a bigger undertaking, but having created such a tool, it could then be used for a variety of purposes such as creating graphs of data flow between processes, sequence diagrams, and potentially detecting race conditions and even deadlocks. Such a tool could very well be valuable and useful for development.

A variation of $createSession$ would be useful so that both parties do not already have to be aware of one another: effectively one (or both) parties stating they are prepared to take part in a particular session type with any Pid. Given the current implementation and the way in which pairing is done, this would be considerably more challenging to implement, though nevertheless useful. In a distributed setting, this becomes very challenging, especially if we wish to avoid having a single registry of Pids which would no doubt act as both a single global lock and a single point of failure!

On the subject of distributed operation, this too is an important feature we have yet to add. Our plans are currently to create threads which are proxies for the network and perform the work of serialisation and de-serialisation of values to and from the network. This raises several interesting and challenging questions such as how to verify that both ends of a network socket are really performing the same session type: this rapidly becomes a question of establishing trust.

8. Related Work

The obvious comparison to this work is the *other* implementation of Session Types in Haskell [19]. There are many differences between our work and theirs. Their implementation is focussed on the network socket, and on a single end-point of the session: they cannot verify communication between two threads, or of the system as a whole. Being focussed on network sockets which consume and produce streams of bytes, their design necessarily demands that messages to be sent and received can be converted to binary representations. As a result, their implementation demands that the user explicitly creates an instance of a type class *Command* for every unique message and that every unique message has its own data type.

This places a much heavier burden on the user of their library than does ours, though our library benefits from being able to communicate Haskell values directly and not needing to serialise such values. The problem here is that protocols tend to be specified as streams of bytes or characters whereas a typical Haskell treatment would introduce additional data types to represent different parts of the protocol. This makes specifying the session type an open-ended question: should the session type simply specify that *String*s are communicated between the parties, or should it contain the additional semantics present in the higher-level data types that a Haskell representation of the protocol would have? This is an interesting design space to explore.

Another major difference is that whilst our library focusses on a monadic approach, theirs does not, instead using explicit continuation passing. However, the most significant difference is that our library permits interleaving, the communication of Pids and the ability to create new channels between existing processes. This gives our library substantially greater expressive power and utility.

Hu et al. [16] present an implementation of Session Types for Java. Perhaps unsurprisingly, their work requires a full preprocessor to type check session types: it would appear to be infeasible to lift Session Types and their invariants into Java's type system. Their system contains a robust distributed environment, which mirrors the standard network APIs: there must be a party *accepting* a service on a given port to which another party must request a connection. Thus their use of host-names and ports as process identifiers is asymmetric in comparison to our Pids, though is clearly the cleanest abstraction on top of the standard network APIs for a network focussed implementation of Session Types. We are unaware of any other published implementations of Session Types either in functional or other programming paradigms. Currently, much of the work seems theoretical, which is a great shame given how badly tools for dealing effectively with multi-threaded programming are needed.

Dezani-Ciancaglini et al. [7] propose an object oriented language, MOOSE with Session Types. The language models sessions between objects and permits higher order sessions, though in order to ensure progress, no interleaving of sessions is permitted. This builds on earlier work [6], in which higher order session types are not permitted due to the difficulty of ensuring that only two parties are ever able to make use of any given session.

The Singularity Operating System project has also developed means to type channels and message passing [10] and their work has created ways to specify communication patterns that are equivalent to Session Types, and they do support the ability to send and receive channel endpoints themselves. Their work has focussed on performance and the demonstration that message passing need not have a high overhead or impact on performance.

Session Types themselves have become part of a W3C standard, the WSCDL [26]. Like Session Types, this work defines bidirectional two-party channels but provides the means to specify interleavings between channels as part of the session type: channel creation, destruction and modification appears in the session type itself. This offers additional guarantees though at the cost of a more complex specification and type system. Recently, multi-party session types have been proposed [15] as a typed process calculus. This extends the work of the WSCDL and in common with that work has a global session type that describes interleavings. This global session type is projected to individual processes that capture only the obligations placed upon each process.

In terms of the general ideas of message passing, Erlang [2] is the most popular *concurrency-centric* programming language based on message passing. Erlang is dynamically typed, having neither a type system for expressions nor for inter-process communication. However, the platform does feature very rich development and debugging tools which permit dynamic inspection of the system's state on live deployments including interrogating message queues. Actor languages in general seem harder to apply type systems for communication to, given that any Actor can send a message to a given mailbox and receiving messages from a mailbox is done through pattern matching.

9. Conclusions

Message Passing is an attractive model of exchanging values between concurrent processes as it is simple and has easily understood semantics. Session Types bring typing disciplines to such exchanges and provide guarantees of ordering and types of values exchanged that make message passing safer, more robust and less ad-hoc.

Our library presents a rich implementation of Session Types allowing programmers to work with complex patterns of communication amongst numerous processes. We support not only interleaved actions upon different channels, but also the ability to communicate process identifiers, the ability to use these identifiers to create new channels between existing processes and support for delegation of channels between processes. We do not make use of a preprocessor, separate compiler or modifications to an existing compiler. Instead we encode all the requirements of Session Types into Haskell's type system, demonstrating its flexibility and utility.

We have presented our library and, through a running example, demonstrated its use before discussing its design and implementation. Our library is available for download from the *Hackage* repository[1] and via our website for this work.[2] To our knowledge, no other implementation of Session Types is available in any language which matches our library in terms of functionality and supported features.

Acknowledgments

We would like to thank Tristan Allwood for getting us up to speed so rapidly with GHC's support for type families and for his input and conversation regarding this work. We would similarly like to thank Matthias Radestock for his thoughts and comments. This work is supported by funding from the European Union funded AETHER project and EPSRC.

References

[1] The Glasgow Haskell Compiler. http://www.haskell.org/ghc/.

[2] J. Armstrong. The development of Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, Amsterdam, The Netherlands, 1997. ACM Press, New York, NY, USA.

[3] R. Atkey. Parameterised notions of computation. In *Proceedings of Workshop on Mathematically Structured Functional Programming (MSFP 2006)*, BCS Electronic Workshops in Computing. BCS, July 2006.

[4] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, 2005. ISSN 0362-1340.

[5] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. *SIGPLAN Not.*, 40(1):1–13, 2005. ISSN 0362-1340.

[6] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A Distributed Object-Oriented language with Session types. In *International Symposium of Trustworthy Global Computing*, April 2005.

[7] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *20th European Conference for Object-Oriented Languages*, July 2006.

[1] http://hackage.haskell.org/cgi-bin/hackage-scripts/package/sessions

[2] http://www.wellquite.org/sessions/

[8] A. Discolo, T. Harris, S. Marlow, S. Peyton Jones, and S. Singh. Lock Free Data Structures using STMs in Haskell. In *FLOPS 2006: Eighth International Symposium on Functional and Logic Programming*, April 2006.

[9] G. J. Duck, S. L. Peyton Jones, P. J. Stuckey, and M. Sulzmann. Sound and Decidable Type Inference for Functional Dependencies. In *ESOP*, pages 49–63, 2004.

[10] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40 (4):177–190, 2006. ISSN 0163-5980.

[11] S. Gay, V. T. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, March 2003.

[12] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Object-Oriented Programming, Systems, Langauges and Applications (OOPSLA '03)*, pages 388–402, October 2003.

[13] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *PPoPP'05: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, Illinois, June 2005.

[14] K. Honda. Types for dyadic Interaction. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes In Computer Science*, pages 509–523. Springer-Verlag, 1993.

[15] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous

session types. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9.

[16] R. Hu, N. Yoshida, and K. Honda. Session-based Distributed Programming in Java. In *22nd European Conference for Object-Oriented Programming*, 2008. To appear.

[17] M. P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2000.

[18] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004. ISBN 1-58113-850-4.

[19] M. Neubauer and P. Thiemann. An implementation of session types. In *Proceedings of Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of LNCS of *Lecture Notes in Computer Science*, pages 56–70. Springer-Verlag, 2004.

[20] S. Peyton Jones. Haskell '98 Report. http://www.haskell.org/onlinereport/.

[21] S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, pages 50–61, 2006.

[22] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *Parallel Architectures and Languages Europe*, pages 398–413, 1994.

[23] M. S. Tom Schrijvers, Simon Peyton Jones and M. Chakravarty. Towards open type functions for haskell. In *Proceedings of the symposium on Implementation and Application of Functional Languages*,

Lecture Notes In Computer Science. Springer-Verlag, 2007.

[24] V. T. Vasconcelos, A. Ravara, and S. J. Gay. Session Types for Functional Multithreading. In *Proceedings of the International Conference on Concurrency Theory: CONCUR 2004*, volume 3170 of *Lecture Notes in Computer Science*, 2004.

[25] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1-2):64–87, 2006.

[26] Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.