Lecture Notes on Session-Typed Concurrency

15-411: Compiler Design
Frank Pfenning

Lecture 25
November 18, 2014

1 Introduction

Generally speaking, it is difficult to add concurrency to a language and retain the same kind of strong guarantees that static typing in a language like C0 gives us. For a sequential language, type safety usually includes preservation (that program remains well-typed as it executes) and progress (there is always a well-defined action to take). In the presence of concurrency, we would like to add deadlock-freedom (which is an analogue to progress) and the absence of race conditions (to guarantee the result is well-defined).

In order achieve these properties, we work under the following conditions:

- Communication between processes is by message-passing rather then via shared memory. In a concrete implementation the message-passing might be accomplished using shared memory, but the computational model itself is at a higher level of abstraction.
- Processes communicate with each other along channels with just two end-points, one process on each end.
- A process offers a service along a unique channel and uses services along possibly many other channels. This allows us to identify a process with the channel along which it offers a service.

Under these assumptions we have designed a type system that guarantees preservation and progress, including the absence of deadlock and race conditions [CP10]. It uses the idea of *linear typing*, which is closely related to the concept of *linear inference* we used to specify program transformations. It is a particular instantiation of the idea of *session types* [Hon93] that prescribe interactions between processes along private channels. The settings for the prior work was process calculi [CPT13]

and functional languages [TCP13]; here we import some of the ideas in the (first-order) imperative setting. It should be emphasized that we have not proven anything about this extension of C0, so all the above properties may be false for this language instance.

2 Process Definitions

We have to extend our language of types by *session types* σ. We continue to use τ for ordinary value types. We will introduce various forms of session types incrementally and summarize them at the end.

Session type σ are used to prescribe communication behavior along *channels*, which are written as $\$c$. A channel declaration is therefore of the form $\sigma\ \$c$ in the concrete syntax and $\$c : \sigma$ in the typing judgments.

A process that *offers* service σ along channel $\$c$ and *uses* services $\sigma_1, \ldots, \sigma_n$ along channels $\$d_1, \ldots, \$d_n$ is spawned by invoking a process definition p with prototype

$$\sigma\ \$c\ p(\sigma_1\ \$d_1, \ldots, \sigma_n\ \$d_n);$$

A process can additionally take value arguments of *primitive types*, a feature we will exploit shortly. The body of a process definition contains the computation and communications to be performed by the process when spawned.

As a first example we consider a process producing a (potentially infinite) stream of integers. The protocol requires that the consumer request a new integer by sending the label 'next', to which the process responds with the next integers. In addition, the consumer can stop the process by sending the label 'stop'. This behavior is expressed by the type

```
choice natstream {
  int /\ choice natstream next;
  void stop;
};
```

The keyword `choice` indicate that the client chooses the operation to be performed by sending of the labels. *natstream* is the name for this particular choice. Syntactically, this is analogous to `struct s`, where s is the name of the struct.

The session type $\tau \wedge \sigma$ (here $\tau = \mathsf{int}$ and $\sigma = \mathsf{choice}\ natstream$ means that the process hand to send a value of type τ and then follow the session type σ. next is the label of the first alternative, stop the label of the second alternative. The session type void indicates that the process should terminate without further interactions.

Instead of writing out `choice natstream` we create `nats` as a synonym for it:

```
typedef choice natstream nats;
```

Next we would like to define a process that outputs a stream of integers according to the above protocol, starting at an integer n.

```
nats $c from(int n) {           /* $c : nats */
  switch ($c) {                 /* receive label along $c */
  case next:                    /* $c : int /\ nats */
    send($c, n);                /* $c : nats */
    $c = from(n+1);             /* tail call, continue in current process */
  case stop:
    close($c);                  /* $c : void */
  }
}
```

The construct

$$\text{switch } (\$c) \ \{\text{case } l_i : seq_i\}_i$$

waits to receive a label l_i along channel $\$c$ and then executes the corresponding case in the body of the switch statement. For this to be correct, the channel $\$c$ must be a choice among the labels l_i. In case the label next is received, we have the command send($\$c, n$) which has the general form

$$\text{send}(\$c, e);$$

which sends the value of e along channel $\$c$. For this to be correct, the channel $\$c$ must have type $\tau \wedge \sigma$, for some session type σ, and e must have type τ. Some restrictions may also be imposed on the type of e, so we do not have potentially complex marshaling and unmarshaling operations to be performed. The primitive types int, bool, char and string seem to be a reasonable choice.

What is curious here is that the switch statement requires $\$c$ to present a choice, while the subsequent send command requires $\$c$ to be a conjunction. This reflects the fact that the types of channels must change throughout the interactions of a process with its environment. If a channel $\$c$ has type choice $name \ \{\sigma_i : l_i\}_i$ then after receiving label l_i it will have type σ_i. Similarly, if a channel $\$c$ has type $\tau \wedge \sigma$, then after sending τ the channel will have type σ.

Next we come to

```
$c = from(n+1);
```

where the current process offers along $\$c$ and from$(n+1)$ is a process invocation. This is the process analogue of a tail-call for a function, and means the current process continues by executing from$(n+1)$. This is also the last statement along this branch of the switch statement, since this process invocation will never return. Notice that the value argument n to a process essentially functions as a variable local to the process.

In the stop branch of the switch, channel $\$c$ has type void. This means we have to close the channel, which is the last action the from process will take.

3 Typing

Because channels change their type during communcations, the typing judgment is a bit unusual. In addition to the context Γ that types value variables, we have a second context

$$\Delta = (c_1 : \sigma_1, \ldots, c_n : \sigma_n)$$

of channel typings. Since channels are distinguished by their position in the judgment, we drop the $-prefix for the names. The general form of the judgment then is

$$\Gamma \; ; \Delta \vdash P :: (c : \sigma)$$

which says that P is a process expression that offers service of session type σ along channel c, using value variables in Γ and channels in Δ. As usual, we do not care about the order of declarations in Γ or Δ.

$$\frac{\{\Gamma \; ; \Delta \vdash P_i :: c : \sigma_i\}_i}{\Gamma \; ; \Delta \vdash \mathsf{switch}(c, \{l_i : P_i\}_i) :: c : \mathsf{choice}\{l_i : \sigma_i\}_i}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \; ; \Delta \vdash P :: c : \sigma}{\Gamma \; ; \Delta \vdash \mathsf{send}(c, e) \; ; P :: c : \tau \wedge \sigma}$$

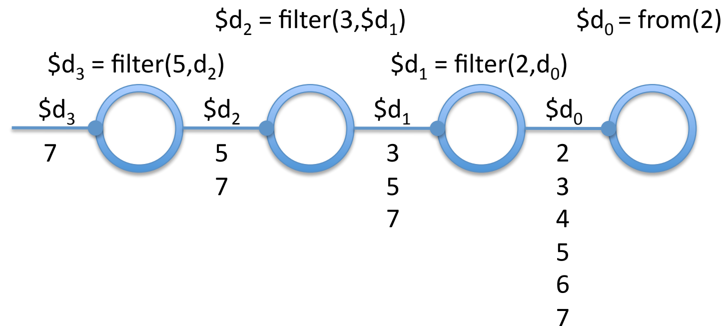
$$\frac{}{\Gamma \; ; \cdot \vdash \mathsf{close}(c) :: c : \mathsf{void}}$$

$$\frac{p : (\sigma_1, \ldots, \sigma_n \vdash \sigma) \quad \Delta \sim (d_1{:}\sigma_1, \ldots, d_n{:}\sigma_n)}{\Gamma \; ; \Delta \vdash c = p(d_1, \ldots, d_n) :: c : \sigma}$$

In the last rule, $\Delta \sim \Delta'$ means that Δ' is a permutation of Δ. We therefore have a precise account for all channels: they are used exactly once (even if that use might be in a recursive procedure invocation). For simplicity, we omitted value arguments, but they would just be expressions e_j of ordinary types τ_j, matching the process declaration. For example, when we close a channel c there may not be any unused channels left in the context. In a switch construct we check every branch in the same context Δ. This is correct, because exactly one of the branches will be chosen when the program is executed.

4 Stream Transducers

Our overall goal of this lecture will be to write a code for the prime sieve (or Sieve of Eratosthenes) that produces a stream of prime numbers. The sieve is implemented as a sequence of filters, each of which filters out all multiples of one particular prime. This is illustrated in the following picture.



Each circle represents a process, where the bullet indicates the channel along which it offers. Below each channel is the sequence of integers flowing from right to left, eliding the request labels next flowing from left to right.

We have already implemented the from process on the far right. Next we implement the filter processes. filter uses one process ($\$d$) and offers along another ($\$c$). It drops all multiples of p from $\$d$ and forwards the rest ot $\$c$. It begins by receiving a label along $\$c$, which must be one of next and stop.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    ...
  case stop:
    ...
  }
}
```

If it is next, we now need to request the next integer along $\$d$. This means we have to *send* the label next along $\$d$. We are using the channel $\$d$, and $\$d$ has type choice $natstream$, so this follows the protocol correctly. Sending a label l along a channel $\$d$ has the syntax $\$d.l$, analogous to the field selection in a struct.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    $d.next;
    ...
  case stop:
    ...
  }
}
```

After sending next, $\$d$ has evolved to type int $\wedge$ choice $natstream$. This means that the process providing $\$d$ will *send* an integer, and we have to receive it. The syntax is $x = \mathsf{recv}(\$d)$, where x must be declared if it hasn't already.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int k = recv($d);
    /* $d : nats |- $c : int /\ nats */
    ...
  case stop:
  }
}
```

We have indicated the types of $\$d$ and $\$c$ at this point during the computation. We now need to check whether p divides k. If so, we keep requesting integers from $\$d$ until we receive a value that is not a multiple of p. We then send the first such value along $\$c$ and recurse. We could accomplish this with two mututally recursive function, or with a loop. For illustration purposes we use a loop.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int k = recv($d);
    /* $d : nats |- $c : int /\ nats */
    while (k % p == 0) {
      $d.next;
      k = recv($d);
    }
    /* $d : nats |- $c : int /\ nats */
    send($c, k);
    $c = filter(p, $d);
  case stop:
    ...
  }
}
```

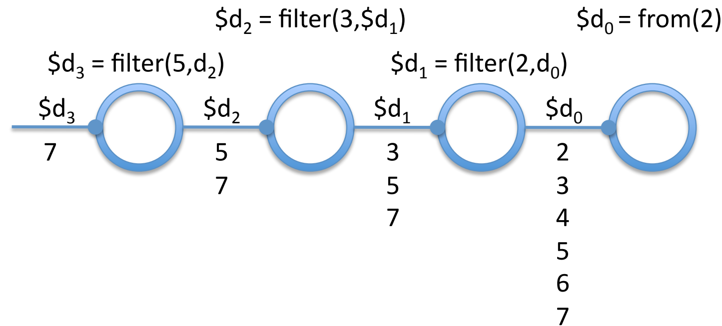
Because we don't know how often the loop will be traversed, the channel types must remain invariant throughout the loop. We have indicated those types in a comment before the loop. This will also be the type of the channels immediately after the loop.

In the case we receive the stop label, we cannot just close the channel $\$c$, because the channel $\$d$ would be left unaccounted for. Instead, we send it in turn a stop label and wait until it finishes. Of course, waiting wouldn't be strictly necessary if we trust it to complete, but for typing purposes we would like to consume that channel.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int k = recv($d);
    /* $d : nats |- $c : int /\ nats */
    while (k % p == 0) {
      $d.next;
      k = recv($d);
    }
    /* $d : nats |- $c : int /\ nats */
    send($c, k);
    $c = filter(p, $d);
  case stop:
    $d.stop;
    wait($d);
    close($c);
  }
}
```

5 Spawning New Processes

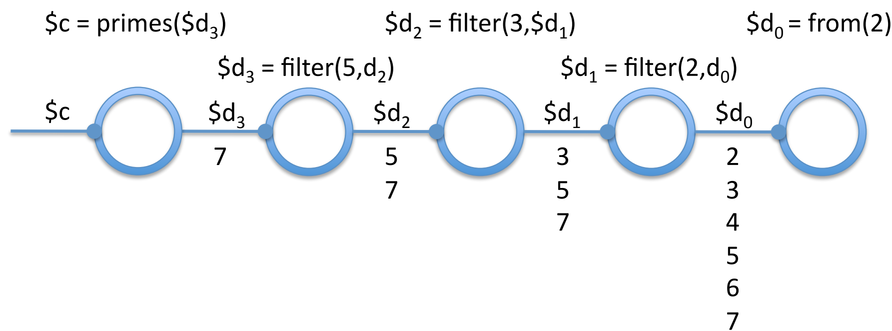
We have now written from and filter. Assume we would like to write the enclosing process primes which is supposed to produce the increasing sequence of prime numbers on channel $\$c$. Let's refer back to the diagram.



We want to write an outermost process

```
nats $c primes(nats $d);
```

which *uses* $\$d_3$ (in the diagram above) and produces along $\$c$ as in the extended diagram below.



If we are requested to produce a number (by receiving next along $\$c$), we ask $\$d$ for the next integer. Since this has already been filtered by all smaller primes, it should be a prime number and we can send it along $\$c$.

```
nats $c primes(nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int p = recv($d);
    send($c, p);
    ...
```

```
  case stop:
    ...
  }
}
```

Before we can recurse we need to spawn a new process, which filters the multiples of p. We do this simply by invoking the process definition and assigning a (new!) channel to the output. We then recurse, using this new channel. It's easy to see that there is exactly one filtering channel for each prime number that we send along $\$c$. In the case we are asked to stop, we just stop the process we use (which will cascade to the right).

```
nats $c primes(nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int p = recv($d);
    send($c, p);
    nats $e = filter(p, $d);    /* spawn new process */
    $c = primes($e);
  case stop:
    $d.stop; wait($d);
    close($c);
  }
}
```

We could also reuse the channel name $\$d$, since $\$d$ is no longer in the context since it is used in the invocation of filter.

```
   $d = filter(p, $d);          /* spawn new process */
   $c = primes($d);
```

At the risk of blurring the line between process invocation and function call, we might also abbreviate these two lines to

```
   $c = primes(filter(p, $d));  /* spawn new process */
```

which would desugar to the first phrasing.

We can now create a process that just produces a stream of primes by creating the process on the far right (producing $2, 3, 4, 5, 6, \ldots$) and supplying it to primes.

```
nats $c prime_stream() {
  nats $d = from(2);
  $c = primes($d);
}
```

Here the tail call preserves the process (identified from the channel $\$c$) even though it is not a recursive call.

We can now embed this in a top-level *function* which creates a fresh stream of prime numbers and requests the first n before terminating the stream.

```
void print_primes(int n) {
  nats $c = prime_stream();
  /* $c : nats |- */
  for (int i = 0; i < n; i++) {
    $c.next;
    int p = recv($c);
    printint(p);
  }
  $c.stop; wait($c);
  return;
}
```

6 Additional Typing Rules

We show another few sample rules to supplement the ones shown earlier. The first is process invocation.

$$\frac{p : (\sigma_1, \ldots, \sigma_n \vdash \sigma') \quad \Delta \sim (d_1{:}\sigma_1, \ldots, d_n{:}\sigma_n) \quad \Delta', e{:}\sigma' \vdash P :: c : \sigma}{\Gamma \; ; \; \Delta, \Delta' \vdash e = p(d_1, \ldots, d_n); P :: c : \sigma}$$

Implicit here is that e is different from c and all the channels declared in Δ'. If e is a variable previously declared, it must have been used (not in Δ') and its new type σ' should be consistent (which is something we do not track explicitly in these rules).

$$\frac{\Gamma \; ; \; \Delta, d{:}\sigma_i \vdash P :: c : \sigma}{\Gamma \; ; \; \Delta, d{:}\mathsf{choice}\{l_i : \sigma_i\}_i \vdash c.l_i \; ; \; P :: c : \sigma}$$

$$\frac{\Gamma, x{:}\tau \; ; \; \Delta, d{:}\sigma' \vdash P :: c : \sigma}{\Gamma \; ; \; \Delta, d{:}\tau \wedge \sigma' \vdash x = \mathsf{recv}(d) \; ; \; P :: c : \sigma}$$

$$\frac{\Gamma \; ; \; \Delta \vdash P :: c : \sigma}{\Gamma \; ; \; \Delta, d{:}\mathsf{void} \vdash \mathsf{wait}(d) \; ; \; P :: c : \sigma}$$

We see that each type construct has a rule when it types the channel we are offering and when it types a channel we are using. One corresponds to a send, while the other corresponds to a receive, reflecting the complementary roles of the processes on the two ends of a channel.

7 Internal Choice and Forwarding

We refer to the choice construct we introduced so far as *external choice*. This is because if we offer $\mathsf{choice}\{l_i : \sigma_i\}_i$ along channel $\$c$, then the client can choose the label. We write

$$\mathsf{branch}\{l_i : \sigma_i\}_i$$

for the opposite *internal choice*, where the provider can choose the label and the consumer has to account for all possibilities.

An example of this is a simple implementaton of stacks, where each process holds an element of the stack. In the absence of further operations on the stack elements, this does not exhibit any concurrency, but it illustrates both internal choice and forwarding (explained later).

The operations on a stack are push, pop, and deallocation. When the client indicates he want to push an element onto the stack, we then have to *receive* an element along the same channel. For this we have the type

$$\tau \Rightarrow \sigma$$

(receive a value of type τ and then behave according to σ) which is symmetric to $\tau \wedge \sigma$ (send a value of type τ and then behave according to σ).

When the client would like to *pop* an element from the stack, we send one of two labels: none if there is no element on the stack, and some if there is one. In the latter case we follow it up with the element itself. This is an example of the internal choice we mentioned above. We define:

```
choice stack_node {
  int => choice stack_node push;
  branch opt_int pop;
  void dealloc;
};

/* Optional result from pop */
branch opt_int {
  int /\ choice stack some;
  choice stack_node none;
};

typedef choice stack_node stack;
```

We now represent the stack constructors empty and node simply as processes. We start with the empty stack. In case of a push, we call spawn a new empty process and recurse as a node with the new element. In case of a pop we indicate that the stack is empty and recurse. Deallocation just closes the channel and terminates the process.

```
stack $c empty();                /* empty stack */
stack $c node(int k, stack $d); /* stack with top element k */

stack $c empty() {
  switch ($c) {
  case push:
    int k = recv($c);
    stack $d = empty();
    $c = node(k, $d);            /* tail call: continue as nonempty */
  case pop:
    $c.none;                     /* no element available */
    $c = empty();                /* tail call: continue as empty */
  case dealloc:
    close($c);
  }
}
```

Second, the case of a nonempty stack with top element k. When we receive a push, we just spawn a new process and recursive. The interesting operation is that of pop. We first send the label some, indicating that the stack is non-empty, then we send the top element.

```
stack $c node(int k, stack $d) {
  switch ($c) {
  case push:
    int n = recv($c);
    stack $e = node(k, $d);      /* spawn new process */
    $c = node(n, $e);            /* new top of stack in current process */
  case pop:
    $c.some; send($c, k);        /* send current element */
    ...
 case dealloc:
    ...
  }
}
```

At this point, we would like to terminate the current process and hand off any interactions along $\$c$ to $\$d$. This is an example of *channel forwarding*. We write this as

```
$c = $d;
```

which identifies $\$c$ and $\$d$.

This could be implemented in several ways. Perhaps the cleanest way is for the current process to send $\$d$ to its client, essentially telling it "*communicate along* $\$d$

from now on instead of $\$c$". Then the process identified with $\$c$ can terminate since the channel $\$c$ is effectively no longer in use.

At a great cost of efficiency, we could also keep the process identified with $\$c$ alive, continuously forwarding messages in both directions until the channel is closed.

Deallocation is straightforward, leading to the following final process definition.

```
/* nonempty stack */
stack $c node(int k, stack $d) {
  switch ($c) {
  case push:
    int n = recv($c);
    stack $e = node(k, $d);     /* spawn new process */
    $c = node(n, $e);           /* new top of stack in current process */
  case pop:
    $c.some; send($c, k);       /* send current element */
    $c = $d;                    /* identify $c and $d; or: forward $d along $c */
  case dealloc:
    $d.dealloc; wait($d);
    close($c);
  }
}
```

We do not show any additional typing rules for the branch construct, since they are symmetric to the choice construct, just swapping offer and use. For forwarding, we just need to keep in mind that there is no continuation after forwarding, so the forwarded channel must be the only one in the context.

$$\frac{}{\Gamma\ ;\ d{:}\sigma \vdash c = d :: c : \sigma}$$

8 Further Constructs

The type language in [TCP13] contains further important constructs. One allows the sending and receiving of channels along channels. Another allows *persistent channels* that do not change their type, but allow new instances of a persistent service to be spawned. We elide persistent channels entirely and just briefly show the rules for sending or receiving channels along channels, since they are used in the next example. We have type $\sigma_1 \otimes \sigma_2$ (concrete syntax `s1 ** s2`) for sending and $\sigma_1 \multimap \sigma_2$ (concrete syntax `s1 -o s2`) for receiving a channel.

$$\frac{\Gamma \; ; \; \Delta \vdash P :: c : \sigma}{\Gamma \; ; \; \Delta, d{:}\sigma' \vdash \mathsf{send}(c, d) \; ; \; P :: c : \sigma' \otimes \sigma}$$

$$\frac{\Gamma \; ; \; \Delta, d{:}\sigma' \vdash P :: d : \sigma}{\Gamma \; ; \; \Delta \vdash d = \mathsf{recv}(c) \; ; \; P :: c : \sigma' \multimap \sigma}$$

9 Further Example: Mergesort

Sending and receiving channels is exemplified in the mergesort program below.[1] The main complication in this implementation is setting up and tearing down the processes to, eventually, achieve an in-place sort. For the sake of brevity, we present it here without further comment.

```
/* Mergesort */
/* Henry DeYoung, transcribed from SILL by fp */

branch list {
  int /\ branch list cons;
  void nil;
};
typedef branch list list;

branch forest {
  int /\ (list ** branch forest) cons; /* int nl = num. of elems in list l */
  void nil;
};
typedef branch forest forest;

/* $c = nil() */
list $c nil() {
  $c.nil;
  close($c);
}

/* $c = cons(k, $d) */
list $c cons(int k, list $d) {
  $c.cons;
  send($c, k);
  $c = $d;
}

/* $c = merge($l, $r) offers sorted merge of $l and $r along $c */
```

[1] requested by a student in lecture

```
list $c merge(list $l, list $r) {
  switch ($l) {
  case cons:
    int x = recv($l);
    switch ($r) {
    case cons:
      int y = recv($r);
      if (x < y) {
        list $r2 = cons(y, $r); /* push y back onto r */
        list $d = merge($l, $r2); /* spawn new process */
        $c = cons(x, $d);
      } else {
        list $l2 = cons(x, $l); /* push x back onto l */
        list $d = merge($l2, $r); /* spawn new process */
        $c = cons(y, $d);
      }
    case nil:                   /* $r is empty */
      wait($r);
      $c = cons(x, $l);         /* push x back onto l */
    }
  case nil:                     /* $l is empty */
    wait($l);
    $c = $r;                    /* forward $r to $c */
  }
}

/* $f = fnil() */
forest $f fnil() {
  $f.nil;
  close($f);
}

/* $g = fcons(nl, $l, $f), adjoins list to forest */
/* invariant: nl = num of elements in $l */
forest $g fcons(int nl, $l list, $f forest) {
  $g.cons;
  send($g, nl);
  send($g, $l);                 /* send channel $l along $g */
  $g = $f;
}

/* $g = join(nl, $l, $f), adjoins list to forest */
/* rebalances if necessary */
/* invariant: nl = num of elements in $l */
forest $g join(int nl, list $l, forest $f) {
  switch ($f) {
  case cons:
    int nr = recv($f);
```

```
    list $r = recv($f);
    if (nl >= nr) {             /* $l is bigger than $r, first list in $f */
      list $m = merge($l, $r);  /* merge $l and $r */
      $g = join(nl+nr, $m, $f); /* adjoin merged list to forest */
    } else {
      forest $f1 = fcons(nr, $r, $f); /* push $r back onto $f */
      $g = fcons(nl, $l, $f1);        /* adjoin $l */
    }
  case nil:
    wait($f);
    $f1 = fnil();               /* recreate empty forest */
    $g = fcons(nl, $l, $f1);
  }
}

/* $m = compress($f), linearizes $f to obtain list $m */
list $m compress(forest $f) {
  switch ($f) {
  case cons:
    int nl = recv($f);
    list $l = recv($f);
    list $r = compress($f);     /* spawn new process */
    $m = merge($l,$r);
  case nil:
    wait($f);
    $m = nil();
  }
}

/* $g = load(A, n) */
/* \length(A) = n, load A[0..n) into forest $g */
/* A should be read-only here; perhaps this should
 * be inlined in sort instead */
forest $g load(int[] A, int n) {
  forest $f = fnil();
  for (int i = 0; i < n; i++) {
    list $l = nil();
    list $l1 = cons(A[i], $l);
    $f = join(1, $l1, $f);
  }
  $g = $f;
}

/* unload($f, A, n) */
/* \length(A) = n, load $f onto A[0..n) */
/* We can write A here, since unload is a function, not process */
void unload(forest $f, int[] A, int n) {
  list $m = compress($f);
```

```
  for (int i = 0, i < n; i++)
    switch ($m) {
    case cons:
      int k = recv($m);
      A[i] = k;
    /* case nil should be impossible */
    }
  switch ($m) {
  /* case cons should be impossible */
  case nil:
    wait($m);
  }
  return;
}

/* sort(A, n), sort A[0..n) in ascending order */
void sort(int[] A, int n) {
  forest $f = load(A, n);       /* create $f */
  unload($f, A, n);             /* consume $f */
  return;
}
```

References

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

[CPT13] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 2013. To appear. Special Issue on Behavioural Types.

[Hon93] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.

[TCP13] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 350–369, Rome, Italy, March 2013. Springer LNCS 7792.