Working Note on a Type Checking Algorithm for Behavioral Separation Types

Luís Caires João C. Seco

(DRAFT of January 22, 2013)

1 Overview

In this draft note, we preview ongoing work on the development of a type checking algorithm for the system of behavioral separation types introduced in [1]. This note is not to be regarded as an appendix to [1]; the algorithmic aspects of behavioral separation type checking, which are quite interesting, will be eventually presented in a separate publication. Several issues are still under investigation, e.g., the concrete syntax of source programs and types, the amount of annotations required from the programmer, a more flexible treatment of sum types, and many others. Nevertheless, our current prototype (implemented in OCaml) is already useful to type-check sample programs, and play with behavioral separation typings of programs on our core language. In particular, the goal of this note is to illustrate how it behaves on all examples presented in [1] (and a few others). The reader is referred to [1] for motivation, further details and explanations about the examples themselves.

The algorithm implements the following procedure: given a type assertion A, a program e and a type U, check whether there is an active type context $\mathcal{E}[-]$ such that $A \vdash_x e :: \mathcal{E}[x{:}U]$ is derivable. Intuitively, A defines the type "environment" under which e is to be typed, U the expected type of the value x returned by e, and $\mathcal{E}[x{:}U]$, the residual type assertion (the post-condition), in which the behavior$x : U$ of the return value appears embedded.

For a first simple example, let

$$A = (a{:}op_1{:}\mathtt{0} \mid b{:}op_2{:}\mathtt{0})\,;a{:}op_1{:}\mathtt{0}$$
$$e = a.op_1$$
$$U = \mathtt{0}$$

The corresponding input to the type checker is given thus

```
type
(a:(op1:stop) | b:(op2:stop)) ; a:(op1:stop)
|-
a.op1
::
stop
;;
```

In this case, we know that

$$(a{:}op_1{:}\mathtt{0} \mid b{:}op_2{:}\mathtt{0})\,;a{:}op_1{:}\mathtt{0} \vdash_x a.op_1 :: (x{:}\mathtt{0} \mid b{:}op_2{:}\mathtt{0})\,;a{:}op_1{:}\mathtt{0}$$

is derivable in the type system. The answer by the type checker is as follows:

```
post-condition:
stop -- (*00 | b:op2:stop) ; a:op1:stop
```

It is composed of two parts:

- a type (`stop`, the concrete syntax for `0`),

- a context `(*00 | b:op2:stop) ; a:op1:stop`, the concrete syntax for $(\Box \,|\, b{:}op_2{:}0)\,;a{:}op_1{:}0$.

So notice that the context hole in the postcondition is represented by `*00`. In some cases, as seen below, there will be multiple occurrences of the hole `*00`, for technical reasons that need not concern us at this point (that is just an approximation of a set of contexts where any occurrence of the hole may be used in the continuation, important in intermediate steps of the algorithm). The key fact to take note is that the algorithm checks whether there is a type derivation for e in the given pre-condition and return type.

We now consider a second simple example, in the concrete syntax of the type checker.

```
def t1;;
def t2;;

type
(a:(t1->t2) | b:t1) ; a:t2
|-
(a b)
::
t2
;;
post-condition:
t2 -- *00 ; a:t2
```

Notice the definition of two atomic (opaque) types `t1` and `t2`, and the concrete syntax for the functional type `t1->t2`. This example proves

$$(a{:}(t_1 \mapsto t_2) \,|\, b{:}t_1)\,;a{:}t_2 \vdash_x (a\ b) :: x{:}t_2\,;a{:}t_2$$

On the other hand, consider the trace

```
type
(a:(t1->t2) | b:t2) ; c:t1
|-
(a c)
::
t2
;;
No
Error: cannot extract c:t1 from (*10 | b:t2) ; c:t1 at nesting level 0 skiping
```

The type checker issued a typing error, signaling that there is no B such that the following judgment is typable.

$$(a{:}(t_1 \mapsto t_2) \,|\, b{:}t_2)\,;c{:}t_1 \vdash_x (a\ c) :: B$$

Our last introductory example, using a heap allocated variable in the context of higher order store, and a "strong" update.

```
def t1;;
def t2;;

type
 (x:var) | f:(t1->t2) | y:(t1;t2)
|-
x:(t1->t2) := f ;
x:t2 := (x y)
::
stop
```

```
;;
post-condition:
stop -- (*00 ; x:rd(t2) ; x: var) ; (*00 | y:t2)
```

The example proves

$$x{:}\mathtt{var} \,|\, f{:}(t_1 \mapsto t_2) \,|\, y{:}(t_1\,;t_2) \vdash_z x := f; x := (x\ y) :: x{:}\mathtt{rd}(t_2)\,;x{:}\mathtt{var}\,;y{:}t_2$$

2 Examples

We consider a sequence of examples, mostly from [1]. We use the following definitions, to simulate base types NAT and STR

```
def str;;
def nat;;
def NAT = !(iso (nat));;
def STR = !(iso (str));;
```

A type of the form $!\circ U$ encodes an "intuitionistic" type, both shared and isolated. As noted in [1], we have NAT <: NAT | NAT and NAT <: ∘NAT. So we have

```
type
x:NAT | f: (! iso (NAT -> NAT))
|-
(f x)
::
NAT
;;
post-condition:
NAT -- (*00 | x:NAT) | !iso f:NAT->NAT | *00
```

```
type
x:NAT | f: (! iso (NAT -> NAT))
|-
(f (f x))
::
NAT
;;
post-condition:
NAT -- (*00 | x:NAT) | (!iso f:NAT->NAT | *00) | *00
```

Compare, e.g., with

```
type
x:nat | f:(nat->nat)
|-
(f x)
::
nat
;;
type (x:nat | f:(nat->nat)) |- (f x) :: nat
solution:
nat -- *00
```

```
type
x:nat | f:(nat->nat)
|-
(f (f x))
::
nat
;;
type (x:nat | f:(nat->nat)) |- (f (f x)) :: nat
No

Error: cannot type f : any->any in x:nat | *39 at nesting level 0
```

2.1 Collection ADT

```
def initt = init:(STR->stop);;
def Node = InitNode; !iso GNode;;
def InitNode = (setElt:(NAT->stop)) ; (setNext:((!iso PNode)->stop));;
def GNode = !getNext:PNode | !getElt:NAT;;
def PNode = !SumNode;;
def SumNode = {NULL:stop , NODE:GNode};;
def SColUse =  stop & ((((!scan:stop); add:(NAT->stop)); SColUse);;
def SCol = initt; (!getId:STR | SColUse);;

type stop
|-
fun x:stop =>
var next, elt in
 [
    setElt = fun e:NAT => elt:NAT := e,
    getElt = elt,
    setNext = fun p:!iso PNode => next:!iso PNode := p,
    getNext = next
 ]{Node}
::
!iso(stop->Node)
;;
post-condition:
!iso (stop->((setElt:NAT->stop; setNext:!iso PNode->stop); !iso (!(getNext:PNode) | !(getElt:N

type
  newNode:!iso(stop->Node)
|-
fun x:stop =>
 var hd, id in
  [
   init = fun i:STR =>
            hd:!(iso (PNode)) := NULL{nil} ;
            id:!(iso (STR)) := i,
   getId = (id:!iso STR),
   add = fun e:NAT =>
            let n = (newNode nil) in
              (n.setElt e) ; (n.setNext hd) ;
               hd:!(iso (PNode)) := NODE{n},
   scan = var s in s:PNode := hd ;
```

```
            rec(L) (case s of
                      | NULL(x) -> nil
                      | NODE(n) -> s:PNode := n.getNext ; L)
  ]{SCol}
::
(stop->SCol)
;;
post-condition:
stop->(init:STR->stop; (!(getId:!iso STR) | stop & ((!(scan:stop); add:NAT->stop); SColUse)))
```

We have separately type checked the node generating function and the collection generating function. Note that we have added certain annotations to guide the type checker, and document the code. It is not the case that all annotations shown are required by the type checker, but we include them all here. because they help to figure out what is going on. On particular notice that in an assignment $a := e$ we often write $a{:}U := e$ to highlight the behavioral separation type U which is actually extracted from e and assigned to heap variable a. We also explicitly type function arguments, and records. Namely for a record $[\ldots]$ we write $[\ldots]\{U\}$ where u is the expected type of the record value (see, e.g., $[\ldots]\{Node\}$ above.

The use of type abbreviations `def U = T` should be clear. Notice that we also use type abbreviations to introduce (guarded) recursion in type definitions, rather than the abstract syntax $\texttt{rec}(X)T$ adopted in the technical development. For example, see the (recursive) definition of `SColUse` above.

The concrete syntax of types and programs should also be pretty obvious. We write `iso` T for $\circ T$ (isolated type). The concrete syntax for a sum type $\oplus_{l\in I} l{:}T_l$ is $\{\ l_1{:}T_1, l_2{:}T_2, \ldots, l_n{:}T_n\ \}$, and for a variant value $l(v)$ the concrete syntax is `l{v}`.

To simplify the treatment of sum types at this stage, we are requiring, as in ML like languages, the sets of labels of the various declared sum types to be disjoint (or hide previously declared ones). This makes it easier for the type checker to infer the sum type for a given labeled variant value. Of course, this restriction is not assumed in the general typings presented in [1], and we are investigating how it may be lifted, so to get more flexibility. Anyway, this familiar issue does not hinder the ability of the type checker, at the expense of some redundancy in type declarations.

Notice also that our pretty printer is not fully streamlined yet, and inserts some extra unneeded parenthesis around type subexpressions, noticeable in the answers of the two typing problems above.

2.2 Usage of Collection ADT (1)

```
type
c:SCol | my:STR | one:NAT
|-
(c.init my) ; c.scan ; (c.add one)
::
stop
;;
post-condition:
stop -- (*00 ; (!c:getId:STR | *00 ; c:SColUse) | *00 | my:STR) | *00 | one:NAT
```

2.3 Usage of Collection ADT (2)

```
type
c:SCol | my:STR | one:NAT
|-
(c.init my) ; (c.add one); c.getId ; c.scan
::
stop
;;
```

```
post-condition:
stop -- (*00 ; (!c:getId:STR | *00 ; ((!c:scan:stop | *00) ;
                c:add:NAT->stop) ; c:SColUse) | *00 | my:STR) | *00 | one:NAT
```

2.4 Usage of Collection ADT (3)

```
type
c:SCol | your:STR | two:NAT
|-
let f = fun x:initt => x.init your in ((f c) ; (c.add two))
::
stop
;;
post-condition:
stop -- ((*00 ; (!c:getId:STR | *00 ; c:SColUse) | your:STR) | *00 | two:NAT) | *00
```

2.5 Usage of Collection ADT (4)

```
type
c:SCol | my:STR | two:NAT
|-
let
  g:!((scan:stop)->stop) = (fun x:(scan:stop) => x.scan)
  in
     ((c.init my); (g c); c.scan; (c.add two); (g c))
::
stop
;;
post-condition:
stop -- ((*00 ; (!c:getId:STR | *00 ; ((!c:scan:stop | *00) ;
                 c:add:NAT->stop) ; c:SColUse) | *00 | my:STR) | *00 | two:NAT) | *00
```

2.6 Usage of Collection ADT (5)

```
type
c:SCol | your:STR | two:NAT
|-
let h = fun x:initt => x.init your in ((c.add two); (h c))
::
stop
;;

No
Error: cannot type c : add:any->any in ((c:SCol | your:STR) | two:NAT) | h_78:initt->stop at n
```

2.7 Usage of Collection ADT (6)

A sequence of examples illustrating borrowing through the store.

```
def ICol = (initt; add:NAT->stop); add:NAT->stop;;
type
c:SCol | my:STR | one:NAT
|-
var a in (
   a:ICol := c ;
```

```
   (a.init my);(a.add one);(a.add one);c.scan )
::
stop
;;
post-condition:
stop -- ((!c:getId:STR | ((!c:scan:stop | *00) ; c:add:NAT->stop) ; c:SColUse) | *00 | my:STR)
```

2.8 Usage of Collection ADT (7)

```
type
c:SCol | my:STR | one:NAT
|-
  (c.init my);
   var a in (a:(NAT->stop):= c.add ; (a one); c.scan)
::
stop
;;
post-condition:
stop -- (*00 ; (!c:getId:STR | ((!c:scan:stop | *00) ; c:add:NAT->stop) ; c:SColUse) | *00 | m
```

2.9 Usage of Collection ADT (8)

```
type
c:SCol
|-
let m = c.init in c.scan
::
stop
;;
No
Error: cannot type c : scan:stop in m_12:STR->stop ; (!c:getId:STR | c:SColUse) at nesting lev
```

2.10 Usage of Collection ADT (9)

```
type
newColl:(stop->SCol) | hi:STR | one:NAT
|-
var s in (
  s:STR := hi;
  let F:(STR->SColUse) = (fun x:STR => let c = (newColl nil) in (c.init x; c)) in
  let u = (F s) in (u.add one))
::
stop
;;
post-condition:
stop -- *00 | (*00 | hi:STR) | *00 | one:NAT
```

2.11 Usage of Collection ADT (10)

A sequence of examples illustrating basic concurrency.

```
type
c:SCol | my:STR | one:NAT
|-
(c.init my) ; (c.add one) ;
```

```
let t1 = fork (c.scan) in
  let t2 = fork (c.scan) in
     (wait(t1);wait(t2))
::
stop
;;
post-condition:
stop -- (*00 ; (!c:getId:STR | *00 ; (((!c:scan:stop | *00) | *00) ;
                c:add:NAT->stop) ; c:SColUse) | *00 | my:STR) | *00 | one:NAT
```

2.12 Usage of Collection ADT (11)

```
type
c:SCol | my:STR | one:NAT | two:NAT
|-
let f:!((!scan:stop)->stop) = fun x:(!scan:stop) => x.scan
   in ( (c.init my) ;
           let t1 = fork(f c) in (c.scan; wait(t1));
           let t3 = fork(c.getId) in
         (c.add two; wait(t3));
     (f c)
   )
::
stop
;;
post-condition:
stop -- (((*00 ; (!c:getId:STR | ((!c:scan:stop | *00) ;
                  c:add:NAT->stop) ; c:SColUse) | *00 | my:STR) | one:NAT) | two:NAT) | *00
```

2.13 Usage of Collection ADT (12)

```
type
c:SCol | my:STR | one:NAT | two:NAT
|-
c.init my;
let t1 = fork(c.add one) in (c.scan; wait(t1))
::
stop
;;
No

Error: cannot type c : scan:any in
     ((*195 ; (!c:getId:STR | t1_41: th(stop) ; c:SColUse)
     | *195 | my:STR) | one:NAT) | two:NAT at nesting level 0
```

2.14 Usage of Collection ADT (13)

```
type
c:SCol | my:STR | one:NAT | two:NAT
|-
let f = fun x:SColUse => (let t = fork(x.add one) in (x.add two ; wait(t)))
 in (
(c.init my);
(f c)
)
```

```
::
stop
;;
No

Error: cannot type x_44 : add:any->any in
  (((c:SCol | my:STR) | one:NAT) | two:NAT) |
  t_45: th(stop) ; x_44:SColUse at nesting level 0
```

2.15 Usage of Collection ADT (14)

```
type
c:SCol | my:STR | one:NAT | two:NAT
|-
let f = (fun x:(!scan:stop) => let t = fork(x.scan) in x.scan; wait(t))
in (
 (c.init my);
 (c.add one);
 (f c)
)
::
stop
;;

post-condition:
stop -- (((*00 ; (!c:getId:STR | *00 ; ((!c:scan:stop | *00) ;
          c:add:NAT->stop) ; c:SColUse) | *00 | my:STR) | *00 | one:NAT) | two:NAT) | *00
```

2.16 Usage of Collection ADT (15)

```
type
c:SCol | my:STR | one:NAT | two:NAT
|-
c.init my;
var a in (
  a:th(stop) := fork(c.scan);
  (c.add one);
  wait(a)
)
::
stop
;;
No

Error: cannot type c : add:any->any in
  ((*231 ; (!c:getId:STR | ((!c:scan:stop | *237 ; a_54:rd(th(stop)) ; a_54: var) ;
   c:add:NAT->stop) ; c:SColUse) | *231 | my:STR) | one:NAT) | two:NAT at nesting level 0
```

2.17 Variable Cell Example (1)

The "atomic cell", usable by concurrent clients. Notice the explicit declaration of the lock invariant in the variable declaration $s : \texttt{rd(NAT);var}$.

```
def CELL = !set:(NAT->stop) | !get:NAT;;
def ATOM = NAT -> CELL;;
```

```
type
stop
|-
fun v:NAT =>
  var s in (s:NAT := v;
    var lock (s:(rd(NAT);var)) in
    [
       set = fun x:NAT => sync(lock) (s:NAT := x),
       get = sync(lock)(s)
    ]{CELL})
::
ATOM
;;
post-condition:
NAT->(!(set:NAT->stop) | !(get:!iso nat)) -- *00
```

2.18 Variable Cell Example (2)

The following code risks a race on s and is not typeable as requested.

```
def CELL = !set:(NAT->stop) | !get:NAT;;
def ATOM = NAT -> CELL;;

type
stop
|-
fun v:NAT =>
  var s in s:NAT := v;
    [
      set = fun x:NAT => (s:NAT:=x),
      get = (s)
     ]{CELL}
::
ATOM
;;
No

Error: cannot type s_66 : var in ((*272 ; s_66:rd(NAT)) ; s_66: var |
   *272 | v_65:NAT) | insh(0,*278 | x_68:NAT) at nesting level !0
```

2.19 Variable Cell Example (3)

Variable cell typed with serialized set / get protocol, allowing sharing by readers while in the read phase (requires coordination among clients, since it is not lock protected).

```
def SCELL = stop & ( !get:NAT; set:(NAT->stop);SCELL );;
def SATOM = NAT -> CELL;;

type
one:NAT
|-
fun v:NAT =>
  var s in s:NAT := v;
    [
```

```
      set = fun x:NAT => (s:NAT:=x),
      get = (s)
    ]{SCELL}
::
NAT -> SCELL
;;
post-condition:
NAT->(stop &
      ((!(get:NAT); set:NAT->stop); stop &
      ((!(get:NAT); set:NAT->stop); SCELL)))
-- *00 | (*00 | (*00 | one:NAT | *00) | *00) | *00
```

2.20 Variable Cell Example (4)

```
def BADCELL = stop & ( get:NAT; !set:(NAT->stop);SCELL ) ;;

type
one:NAT
|-
fun v:NAT =>
  var s in s:NAT := v;
    [
      set = fun x:NAT => (s:NAT:=x),
      get = (s)
    ]{BADCELL}
::
NAT -> BADCELL
;;
No

Error: cannot type s_85 : var in ((*516 ; (*522 | s_85:rd(NAT))) ;
       s_85: var | one:NAT | *516 | v_84:NAT) | insh(0,*525 | x_87:NAT) at nesting level !0
```

2.21 Linked Queue (List Node)

```
def SHeadT = {NULL:stop, VAL:iso HeadT};;
def HeadT = iso (unlink: (iso SHeadT));;
def TailT = iso (link: ((iso SHeadT) -> stop));;
def Node = HeadT | TailT ;;

type
stop
|-
fun x:stop =>
  var next in
    next:(iso SHeadT) := NULL{nil};
    var lock(next:(rd(iso SHeadT);var)) in
    [
      unlink = sync(lock) (
         let x = (next:iso SHeadT) in
           next:iso SHeadT := NULL{nil}; x),
      link = fun x:iso SHeadT => sync(lock)(next:iso SHeadT := x)
    ]{Node}
::
```

```
!(stop->iso Node)
;;
post-condition:
!(stop->iso (iso (unlink:iso SHeadT) | iso (link:iso SHeadT->stop))) -- *00
```

2.22 Linked List Queue

```
def STailT = {NULLT:stop, VALT:iso TailT};;
def SQueue = (stop & ((enq:stop & deq:stop);SQueue));;

type
!new:(stop->iso Node)
|-
  var head, tail in
    head:iso SHeadT := NULL{nil};
    tail:iso STailT := NULLT{nil};
    [
      enq =
        let n = (new nil) in
          case tail of
            | NULLT(x) ->
                 head: iso SHeadT := VAL{n};
                 tail: iso STailT := VALT{n}
            | VALT(y) ->
                 y.link VAL{n};
                 tail:iso STailT := VALT{n},
      deq = case head of
            | NULL(z) -> head:iso SHeadT := NULL{nil}
            | VAL(y) ->
                 head:iso SHeadT  := y.unlink;
                 case head of
                    | NULL(z) ->
                        tail:iso STailT := NULLT{nil};
                        head:iso SHeadT := NULL{nil}
                    | VAL(y) -> head:iso SHeadT := VAL{y}
     ]{SQueue}
::
SQueue;;
post-condition:
stop & (enq:stop & deq:stop; stop & (enq:stop & deq:stop; SQueue)) --
*00 | *00 | *00 | *00 | *00 | *00 | *00 | *00 | !new:stop->iso Node
```

2.23 Concurrent Linked List Queue

```
def CQueue = !enq:stop | !deq:stop;;

type
!new:(stop->iso Node)
|-
  var head, tail in
    head:iso SHeadT := NULL{nil};
    tail:iso STailT := NULLT{nil};
    var lock (head:(rd(iso SHeadT);var) |
        tail:(rd(iso STailT);var)) in
```

```
    [
       enq = sync(lock)(
               let n = (new nil) in
                 case tail of
                  | NULLT(x) ->
                      head: iso SHeadT := VAL{n};
                      tail: iso STailT := VALT{n}
                  | VALT(y) ->
                      y.link VAL{n};
                      tail:iso STailT := VALT{n}),
       deq = sync(lock)(
               case head of
                 | NULL(z) -> head:iso SHeadT := NULL{nil}
                 | VAL(y) ->
                     head:iso SHeadT  := y.unlink;
                     case head of
                       | NULL(z) ->
                           tail:iso STailT := NULLT{nil};
                           head:iso SHeadT := NULL{nil}
                       | VAL(y) ->
                           head:iso SHeadT := VAL{y})
]{CQueue}
::
CQueue;;
post-condition:
!(enq:stop) | !(deq:stop) -- *00 | !new:stop->iso Node | insh(0,!new:stop->iso Node)
```

2.24 Using the Concurrent Queue

```
type
q:CQueue
|-
let t = fork (q.enq; q.enq)
  in (
    q.deq;
    q.deq;
    wait(t)
  )
::
stop
;;
post-condition:
stop -- (!q:enq:stop | *00) | (!q:deq:stop | *00) | *00
```

2.25 Landin's Knot

Please refer to [1], for a discussion on the usage of invariant-based separation to type this example.

```
def t = stop->stop;;
def I = !iso (t);;

type
  stop
|-
var a in (
```

```
  a:I := (fun i:stop => i);
  var s ( a:(rd(I);var) ) in
  let f:I = (fun i:stop =>( (sync(s)(a) i) ))
  in ( sync(s)(a:I := f);
       (f nil)
     )
)
::
stop
;;
post-condition:
stop -- *00
```

2.26 Double Linked List Queue with Concurrent Iterator

```
def nextt = getNext:(!optnextrec);;
def optnextrec = {NULLNR:stop, ValNR:(nextt)};;

def opttl = {NULLT:stop, ValT:iso tailt};;
def tailt = getPrev:(iso opttl);;

def headt = (nextt ; iso headt) & (setPrev:((iso opttl)->stop);!iso nextt);;
def opthd = {NULLH:stop, ValH:(iso headt)};;

def celltype = iso tailt | setNext:((!iso optnextrec)->stop); iso headt;;

type
stop
|-
fun z:stop =>
  var next, prev in
    prev:iso opttl := NULLT{nil};
    var lock (prev:(rd(iso opttl);var)) in
      [
        getNext = next,
        setNext = fun x:!iso optnextrec => next:!iso optnextrec := x,
        getPrev = sync(lock) (let x = (prev:iso opttl) in
           prev:(iso opttl) := NULLT{nil}; x),
        setPrev = fun x:(iso opttl) => sync(lock) (prev:(iso opttl) := x)
      ]{celltype}
::
!(stop -> iso celltype)
;;

post-condition:
!(stop->iso (iso (getPrev:iso {NULLT:stop, ValT:iso tailt}) |
                  setNext:!iso optnextrec->stop; iso ((getNext:!optnextrec; iso headt) &
                  (setPrev:iso opttl->stop; !iso (getNext:!optnextrec)))))
 -- *00

def sc = (!scn:stop) ; (ok:stop) ;;
def opts = { NULLS:stop, ValS:sc };;
def queuetype = (enq:stop & deq:stop & iter:opts) ; queuetype ;;
```

```
type
!newCell:(stop->iso celltype)
|-
  var head, tail in
    head:iso opthd := NULLH{nil}; tail:iso opttl := NULLT{nil};
      [
         enq = let new = newCell(nil) in
                 case head of
                 | NULLH(x) ->
                     new.setNext(NULLNR{nil}) ;
                     head:iso opthd := ValH{new} ;
                     tail:iso opttl := ValT{new}
                 | ValH(y) ->
                     y.setPrev(ValT{new});
                     new.setNext(ValNR{y});
                     head:iso opthd := ValH{new},
         deq = case tail of
               | NULLT(z) -> tail:iso opttl := NULLT{nil}
               | ValT(y) ->
                   tail:(iso opttl) := y.getPrev ;
                   case tail of
                   | NULLT(z) -> tail:iso opttl := NULLT{nil};
                                 head:iso opthd := NULLH{nil}
                   | ValT(y) -> tail:iso opttl := ValT{y},
         iter = case head of
                | NULLH(x) -> head:iso opthd := NULLH{nil} ; NULLS{nil}
                | ValH(n) ->  head:stop:=nil;
                              let s:!optnextrec = n.getNext  in
                                let v =
                                  [
                                    scn = var a in ( a:!optnextrec := s ;
                                            rec(L)
                                              (case ((a:!optnextrec)) of
                                               | NULLNR(z) -> nil
                                                | ValNR(m) ->
                                                     a:!optnextrec := m.getNext;L)),
                                    ok = head:iso opthd := ValH{n}
                                  ]{sc} in ValS{v}
             ]{queuetype}
::
queuetype;;

post-condition:
((enq:stop & deq:stop) & iter:{NULLS:stop, ValS:sc}); queuetype --
*00 | !newCell:stop->iso celltype
```

2.27 Using Queue with Concurrent Iterator (1)

```
type
q:queuetype
|-
q.enq ; q.deq ;
let i:opts = q.iter
in (
```

```
     case i of
     | NULLS(z) -> nil
     | ValS(s) ->  ( let f1  = fork(s.scn) in s.scn ; wait(f1)); s.ok
    )
::
stop
;;

post-condition:
stop -- *00 ; q:queuetype
```

2.28 Using Queue with Concurrent Iterator (2)

```
type
q:queuetype
|-
q.enq ; q.deq ;
let i:opts = q.iter
in (
     q.deq;
     case i of
     | NULLS(z) -> nil
     | ValS(s) -> s.ok
    )
::
stop;;
No

Error: cannot type q : deq:any in *3654 ; *3656 ; i_333:opts ; q:queuetype at nesting level 0
```

2.29 Future

A "future" receives a thunk producing a `NAT` and spawns it concurrently, the client can then poll for the result. We type both the "future" object and some client code.

```
def ov = {NONE:stop, SOME:NAT};;
def vopt = !iso ov;;
def thunktype = (iso (stop -> NAT));;
def future = spawn:(thunktype -> stop) ; !get:vopt;;
```

```
type
  stop
|-
var s in (
 s:vopt:=NONE{nil} ;
var l ( s:(rd(vopt);var)  )
 in (
 [
   spawn = fun f:thunktype =>
      (let a:(iso th(stop)) =
                    fork (let v:NAT = (f nil) in sync(l)( s:vopt:= SOME{v} )) in nil ),
   get = sync(l)(s:vopt)
 ]{future} ))
::
future
;;
post-condition:
spawn:thunktype->stop; !(get:vopt) -- *00

type
  f:future | lib:(!thunktype) | print:(NAT->stop)
|-
let t:thunktype = (fun x:stop => (lib x))
in ( (f.spawn t) ;
      rec(L)( let r:vopt = (f.get) in
                    case r of
                       |  NONE(z) -> L
                       |  SOME(z) -> (print z))
    )
::
stop
;;

post-condition:
stop -- (*00 ; (!f:get:vopt | *00) | !lib:thunktype | *00) | *00
```

2.30 Thread pool

```
clear;;
def str;;
def nat;;
def NAT = !(iso (nat));;
def STR = !(iso (str));;

def ov = {NONE:stop, SOME:NAT};;
def vopt = !iso ov;;
def thunktype = (iso (stop -> NAT));;
def future = spawn:(thunktype -> stop) ; !get:vopt;;

clear;;
def str;;
def nat;;
def NAT = !(iso (nat));;
def STR = !(iso (str));;
```

```
def ov = {NONE:stop, SOME:NAT};;
def vopt = !iso ov;;
def Queue = !enq:NAT->stop | !deq:ov;;


type
  !iso (values:Queue) | f:! iso (NAT->NAT)
|-
  let spawn:!(NAT->stop) =
    fun x:NAT =>
      (let t:iso th(stop) = fork (let v:NAT = f x in values.enq v) in nil)

  in let waitresult:!(stop -> NAT) =
     (fun x:stop =>
      rec(L)(case values.deq of
      | NONE(n) -> L
      | SOME(n) -> n)) in

  fun x:NAT => let w = spawn x in (waitresult nil)
::
!(NAT->NAT)
;;

post-condition:
!(NAT->!iso nat) -- *00 | (*00 | ((!iso values:Queue |
                    !values:enq:NAT->stop | !values:deq:ov | *00) |
                    insh(0,!iso values:Queue)) | !iso f:NAT->NAT | insh(0,!iso f:NAT->NAT))
```

3 Conclusions

This note describes work in progress and will be updated frequently, with further examples and extensions.

References

[1] L. Caires and J. C. Seco. The Type Discipline of Behavioral Separation. Technical report, TR-UNL, in author's web site, 2012.