Standard Library of Tamed Pict Programming Language

Matej Košík

April 15, 2011

Contents

Copying

Many parts of this software were drawn from Pict [6, 7, 9].

Chapter 1

Standard Prelude

Unless you compile your Pict programs with `-reset lib` options, each of them will by default import the `lib.pi` module defined below. Its purpose is to provide common definitions.

6a ⟨*lib.pi* 6a⟩≡

```
import "Trusted/Fd"
import "Untrusted/Bool"
import "Untrusted/Cmp"
import "Untrusted/Int"
import "Untrusted/List"
import "Untrusted/Misc"
import "Untrusted/String"
```

As you can see, some of the trusted modules are loaded by default. This configuration enables you to write trivial programs such as:

6b ⟨*hello.pi* 6b⟩≡

```
(pr "Hello, world\n");
```

You can compile and run it as simply as:

```
pict hello.pi
```

All this has the aim to ensure that simple things can be done easily.

In those cases when you do not want your modules to see symbols defined by the `Fd` module imported by default, you can compile that module with the `-reset lib` flag. This is the case of all the untrusted modules available in this library and it will probably be the case of all the untrusted modules that will be part of your program.

Chapter 2

Untrusted Modules

The original Pict implementation was distributed with Standard Pict Library [8]. The code in this chapter is based on that library. Some changes that we have made:

- some modules were left out because we did not need them at the moment but they can be added later
- some modules were left out because we cannot use them
- we have left out the `Ref` module and we provide the `Cell` (Section 2.7) module instead of it. This change was somewhat arbitrary.
- we have changed the interface and the implementation of the `Queue` module; now adding and removing elements from a queue takes constant time
- we have added several modules that were not present in the original Standard Pict Library (`Choice`, `Semaphore`, `Lid`, and `Revoker`)
- some `ccode` primitives were eliminated by rewriting them directly in Pict[1]
- all those `ccode` primitives that we could not rewrite in Pict were moved to the `Trusted/Prim` module. That was a significant change because from now on all these modules are regarded as untrusted. The original library contained `ccode` primitives that were scattered all over.

Each section of this chapter contains the description and the implementation of a single module. Each module is tangled into a separate file (e.g. `Untrusted/Misc.pi`) that can be imported

```
import "Untrusted/Misc"
```

by other modules. Most of the modules introduce new types[2]. All modules bind some variable to a tuple of functions provided by that module. E.g. the `Untrusted/List` module binds the `list` variable to a tuple of functions. You can use those functions if you specify the full path, i.e. `list.nil`, `list.cons`, `list.car` etc. Some modules also introduce additional bindings when there is no real threat of a name clash. E.g. you can use `nil` instead of `list.nil` because the `Untrusted/List` module creates a proper shortcut:

```
val nil = list.nil
```

2.1 Alarm

7 ⟨*Untrusted/Alarm.pi* 7⟩≡

```
type AlarmAPI =
  [ scheduleReal = /[Int Int /[]]
    scheduleVirtual = /[Int Int /[]]
    scheduleProf = /[Int Int /[]]
  ]
```

[1]This might decrease the performance but it will also minimize the TCB and that is far more valuable effect.

[2]Actually, those types existed before, we only create aliases for them to increase readability.

2.2 Args: Command-Line Arguments

This is the "guest" part of the command-line parsing functionality. The "host" part is defined in Section 3.2.

Example:

```
import "Trusted/Args"

val f = (cell.make false)
val g = (cell.make false)

val i = (cell.make 100)
val j = (cell.make 200)

val s = (cell.make "sss")
val t = (cell.make "ttt")

val anonymousOptions = (args.parse (cons > ["f" f] ["g" g] nil)
                                   (cons > ["i" i] ["j" j] nil)
                                   (cons > ["s" s] ["t" t] nil)
                       )
```

If you invoke the program with `"-f"` option, then the `parse` functions sets the `f` cell to `true`. Similarly for the `"-g"` option.

If you invoke the program with the `"-i 1234"` option, then the `parse` function sets the `i` cell to `1234`. Similarly for the `"-j"` option.

If you invoke the program with the `"-s foo"` option, then the `parse` function sets the `s` cell to `"foo"`. Similarly for the `"-t"` option.

Those options that are not mentioned within our nullary/integer/string option specifications, are collected and returned as a result of the `parse` function. Thus, if you invoke the program as follows:

```
program foo -g -j 50 bar -s s-value
```

Then

- the `f` cell remains set to `false`
- the `g` cell will be set to `true`
- the `i` cell remains set to `100`
- the `j` cell will be set to `50`
- the `s` cell will be set to `s-value`
- the `t` cell remains set to `ttt`

In this case, the `parse` function will return a list of two strings:

- `"foo"`
- `"bar"`

The `parse` function will work properly only when the following conditions are met:

- Each option name must be mentioned at most in one of the three option classes.
- At run-time, each integer option name must be followed by a string that can be converted into integer.
- At run-time, each string option name must be followed by a string.

If any of them is false, the `parse` function will print appropriate error message and it will exit the whole UNIX process.

2.2.1 Implementation

9 ⟨*Untrusted/Args.pi* 9⟩≡

```
import "Untrusted/String"
import "Untrusted/Cell"
import "Untrusted/Queue"
import "Untrusted/List"

type NullaryOptionSpec = [String (Cell Bool)]
type IntegerOptionSpec = [String (Cell Int)]
type StringOptionSpec = [String (Cell String)]

type ArgsAPI = [
  argc = Int
  argv = /[Int /String]
  parse = /[ (List NullaryOptionSpec)
             (List IntegerOptionSpec)
             (List StringOptionSpec)
             /(List String)
           ]
]

def makeArgs (argc:Int argv:/[Int /String]) : ArgsAPI =
  ( def parse ( nullaryOptionSpecs:(List NullaryOptionSpec)
                integerOptionSpecs:(List IntegerOptionSpec)
                stringOptionSpecs:(List StringOptionSpec)
              ) : (List String) =
      ( def processOption (optionIndex:Int) : (List String) =
          if (== optionIndex argc) then
            nil
          else
            ( val option = (argv optionIndex)
              if (<= (string.size option) 1) then
                {- Options that are composed from zero or only one
                 - characters are put among resulting
                 - anonymous options.
                 -}
                (cons option (processOption (inc optionIndex)))
              else if (<> '-' (string.nth option 0)) then
                {- Options that have more than one characters but
                 - do not begin with the '-' character are put among
                 - anonymouse options.
                 -}
                (cons option (processOption (inc optionIndex)))
              else
                {- Current option has more than one character and
                 - begins with the '-' character. Now set properly
                 - `isNullaryOption', `isIntegerOption' and `isStringOption'
                 - with regard whether there is proper option specification
                 - in proper list.
                 -}
                ( val option = (string.sub option 1 (dec (string.size option)))
                  val isNullaryOption =
                    (list.fold nullaryOptionSpecs false
                      \([nullaryOption:String _:(Cell Bool)] subresult:Bool):Bool =
                        if (==$ option nullaryOption) then
                          true
                        else
                          subresult
```

```
  )
val isIntegerOption =
  (list.fold integerOptionSpecs false
    \([integerOption:String _:(Cell Int)] subresult:Bool):Bool =
      if (==$ option integerOption) then
        true
      else
        subresult
  )
val isStringOption =
  (list.fold stringOptionSpecs false
    \([stringOption:String _:(Cell String)] subresult:Bool):Bool =
      if (==$ option stringOption) then
        true
      else
        subresult
  )
if (&& > isNullaryOption
         (not isIntegerOption)
         (not isStringOption)) then
  ( (list.apply nullaryOptionSpecs
      \([nullaryOption:String boolCell:(Cell Bool)]):[] =
        if (==$ nullaryOption option) then
          (cell.put boolCell true)
        else
          []
    );
    (processOption (inc optionIndex))
  )
else if (&& > (not isNullaryOption)
              isIntegerOption
              (not isStringOption)) then
  ( (list.apply integerOptionSpecs
      \([integerOption:String integerCell:(Cell Int)]):[] =
        if (==$ integerOption option) then
          if (== optionIndex (dec argc)) then
            (error "Args-Guest.makeArgs.parse: integer option name must be followed
          else
            (cell.put integerCell (string.toInt (argv (inc optionIndex))))
        else
          []
    );
    (processOption (+ optionIndex 2))
  )
else if (&& > (not isNullaryOption)
              (not isIntegerOption)
              isStringOption) then
  ( (list.apply stringOptionSpecs
      \([stringOption:String stringCell:(Cell String)]):[] =
        if (==$ stringOption option) then
          if (== optionIndex (dec argc)) then
            (error "Args-Guest.makeArgs.parse: string option name must be followed
          else
            (cell.put stringCell (argv (inc optionIndex)))
        else
          []
    );
    (processOption (+ optionIndex 2))
```

```
                  )
                else if (&& > (not isNullaryOption)
                              (not isIntegerOption)
                              (not isStringOption)) then
                  (cons (argv optionIndex) (processOption (inc optionIndex)))
                else
                  (error "Args-Guest.makeArgs.parse: Bad option specifications.")
              )
          )
      (processOption 1)
    )

  {- Check whether given option specification make sense.
   - The sets of option names in both three list must
   - be disjunct
   -}
  and checkOptions ( nullaryOptionSpecs:(List NullaryOptionSpec)
                     integerOptionSpecs:(List IntegerOptionSpec)
                     stringOptionSpecs:(List StringOptionSpec)
                   ) : [] =
    []

  [ argc=argc argv=argv parse=parse ]
)
```

2.3 Array: 1-Dimensional Array

(Array X) is the type of arrays which contain elements of type X.

```
#Array : (Type -> Type)
```

2.3.1 Creation

(copy a) returns a copy of an array a. Arrays are mutable and thus it has sense to copy them.

```
copy = /[#X (Array X) /(Array X)]
```

Return an array of length zero. This function is useful since, unlike make, it does not require an initializing element.

```
empty = /[#X /(Array X)]
```

(make x n) returns a new array containing n elements, each initialized to x. The array size n must be greater than or equal to zero.

```
make = /[#X X Int /(Array X)]
```

(tabulate n f) creates an array of size n where each array element i is initialized to (f i). The function f is called sequentially and in order. n must be greater than or equal to zero.

```
tabulate = /[#X Int /[Int /X] /(Array X)]
```

(fromList l) creates an array from the list l.

```
fromList = /[#X (List X) /(Array X)]
```

2.3.2 Interrogation

(size a) returns the size of the array a.

```
size = /[#X (Array X) /Int]
```

(nth a n) Looks up the n'th element of the array a, if $0 \leq n <$ (size a). If n is not a valid index for a then we generate a runtime error.

```
nth = /[#X (Array X) Int /X]
```

(detect a f) applies f to each element of a. The whole function returns true if function f returns true at least for one element.

```
detect = /[#X (Array X) /[X /Bool] /Bool]
```

2.3.3 Modification

(update a n x) updates the n'th element of a to be x. If n is not a valid index for a then we generate a runtime error.

```
update = /[#X (Array X) Int X /[]]
```

(reset a x) sets the value of each element in a to x.

```
reset = /[#X (Array X) X /[]]
```

`(rotate a o)` rotates `a` so that the `i`'th element of `a` contains the value which used to be at index `i+o`. For example, the expression `(rotate a -1)` rotates the contents of `a` to the right by one.

```
rotate = /[#X (Array X) Int /[]]
```

2.3.4 Iteration

`(fold a init f)` applies `f` to each element of `a` (sequentially and in order), passing `f` an accumulated result of type `R`. The initial accumulated result is `init`. `(revFold a init f)` behaves similarly, except that it traverses `a` in reverse order.

```
fold = /[#X #R (Queue X) R /[X R /R] /R]
revFold = /[#X #R (Queue X) R /[X R /R] /R]
```

`(itFold a init f)` applies `f` to each element of `a` (sequentially and in order), passing `f` the index of each element, and an accumulated result of type `R`. The initial accumulated result is `init`. `revItFold` behaves just like `itFold`, except that it traverses `a` in reverse order.

```
itFold = /[#X #R (Array X) R /[Int X R /R] /R]
revItFold = /[#X #R (Array X) R /[Int X R /R] /R]
```

`(apply a f)` applies `f` to each element of `a`. `f` is called sequentially and in order. `revApply` behaves similarly, except that it traverses the array in reverse order.

```
apply = /[#X (Array X) /[X /[]] /[]]
revApply = /[#X (Array X) /[X /[]] /[]]
```

`(itApply a f)` applies `f` to each element of `a` (sequentially and in order), passing `f` the index of each element. `revItApply` behaves just the same as `itApply`, except that it traverses `a` in reverse order.

```
itApply = /[#X (Array X) /[Int X /[]] /[]]
revItApply = /[#X (Array X) /[Int X /[]] /[]]
```

`(map a f)` applies `f` to each element of `a`, updating each element in `a` with the result of applying `f`. `f` is called sequentially and in order. `revMap` behaves similarly, except that it traverses the array in reverse order.

```
map = /[#X (Array X) /[X /X] /[]]
revMap = /[#X (Array X) /[X /X] /[]]
```

`(itMap a f)` applies `f` to each element of `a`, along with its index, updating each element in `a` with the result of applying `f`. `f` is called sequentially and in order. `revItMap` behaves similarly, except that it traverses the array in reverse order.

```
map = /[#X (Array X) /[Int X /X] /[]]
revMap = /[#X (Array X) /[Int X /X] /[]]
```

2.3.5 Implementation

13 ⟨*common types* 58a⟩≡ ◁58a

```
  val [#Array : (Pos Type -> Type)] = [#\X = Top]
```

14 ⟨*Untrusted/Array.pi* 14⟩≡
```
import "Untrusted/List"

val array:[
  empty = /[#X /(Array X)]
  make = /[#X X Int /(Array X)]
  tabulate = /[#X Int /[Int /X] /(Array X)]
  fromList = /[#X (List X) /(Array X)]
  size = /[#X (Array X) /Int]
  nth = /[#X (Array X) Int /X]
  update = /[#X (Array X) Int X /[]]
  reset = /[#X (Array X) X /[]]
  rotate = /[#X (Array X) Int /[]]
  fold = /[#X #Y (Array X) Y /[X Y /Y] /Y]
  revFold = /[#X #Y (Array X) Y /[X Y /Y] /Y]
  itFold = /[#X #Y (Array X) Y /[Int X Y /Y] /Y]
  revItFold = /[#X #Y (Array X) Y /[Int X Y /Y] /Y]
  apply = /[#X (Array X) /[X /[]] /[]]
  revApply = /[#X (Array X) /[X /[]] /[]]
  itApply = /[#X (Array X) /[Int X /[]] /[]]
  revItApply = /[#X (Array X) /[Int X /[]] /[]]
  map = /[#X (Array X) /[X /X] /[]]
  revMap = /[#X (Array X) /[X /X] /[]]
  itMap = /[#X (Array X) /[Int X /X] /[]]
  revItMap = /[#X (Array X) /[Int X /X] /[]]
  copy = /[#X (Array X) /(Array X)]
  detect = /[#X (Array X) /[X /Bool] /Bool]
] = (
  val empty = prim.arrayEmpty
  val make = prim.arrayMake
  val size = prim.arraySize
  val nth = prim.arrayNth
  val update = prim.arrayUpdate

  def tabulate (#X size:Int create:/[Int /X]) : (Array X) =
    if (<= 0 size) then
      if (== size 0) then
        (empty #X){-HACK-}
      else
        ( val array:(Array X) = (make (create 0) size)
          def loop (x:Int):[] =
            if (<< x size) then
              ( (update array x (create x));
                (loop (inc x))
              )
            else
              []
          (loop 1);
          array
        )
    else
      (error "array.tabulate: negative size")

  and fromList (#X l:(List X)) : (Array X) =
    ( val size = (list.size l)
      if (== size 0) then
        (empty #X){-HACK-}
      else
```

```
        ( val array = (make (car l) size)
          def set (x:Int v:X):[] =
            (update array x v)
          (list.itApply l set);
          array
        )
    )

  and reset (#X a:(Array X) x:X) : [] =
    ( def loop (i:Int):[] =
        if (>= i 0) then
          ( (update a i x);
            (loop (dec i))
          )
        else
          []
     (loop (dec (size a)))
    )

  and map (#X a:(Array X) f:/[X /X]) : [] =
    ( val limit = (dec (size a))
      def loop (x:Int):[] =
        if (<= x limit) then
          ( (update a x (f (nth a x)));
            (loop (inc x))
          )
       else
         []
     (loop 0)
    )

  and revMap (#X a:(Array X) f:/[X /X]) : [] =
    ( def loop (x:Int):[] =
        if (>= x 0) then
          ( (update a x (f (nth a x)));
            (loop (dec x))
          )
       else
         []
     (loop (dec (size a)))
    )

  and itMap (#X a:(Array X) f:/[Int X /X]) : [] =
    ( val limit = (dec (size a))
      def loop (x:Int):[] =
        if (<= x limit) then
          ( (update a x (f x (nth a x)));
            (loop (inc x))
          )
       else
         []
     (loop 0)
    )

  and revItMap (#X a:(Array X) f:/[Int X /X]) : [] =
    ( def loop (x:Int):[] =
        if (>= x 0) then
          ( (update a x (f x (nth a x)));
```

```
          (loop (dec x))
        )
      else
        []
    (loop (dec (size a)))
  )

and rotate (#X a:(Array X) o:Int) : [] =
  ( val sz = (size a)
    {- Ensures that -sz <= o <= sz -}
    val o = (- o (* (div o sz) sz))
    {- Ensures that 0 <= o <= sz -}
    val o = (mod (+ o sz) sz)
    if (>> o 0) then
      ( val b = (tabulate sz
                          \(x:Int):X = (nth a (mod (+ x o) sz))
                )
        (itMap a \(x:Int _:X):X = (nth b x))
      )
    else if (<< o 0) then
      ( val b = (tabulate sz
                          \(x:Int):X = (nth a (mod (+ x o) sz))
                )
        (itMap a \(x:Int _:X):X = (nth b x))
      )
   else
    []
  )

and fold (#X #Y a:(Array X) init:Y f:/[X Y /Y]) : Y =
  ( val limit = (dec (size a))
    def loop (v:Y x:Int):Y =
      if (<= x limit) then
        (loop (f (nth a x) v) (inc x))
      else
        v
    (loop init 0)
  )

and revFold (#X #Y a:(Array X) init:Y f:/[X Y /Y]) : Y =
  ( def loop (v:Y x:Int):Y =
      if (>= x 0) then
        (loop (f (nth a x) v) (dec x))
     else
      v
   (loop init (dec (size a)))
  )

and itFold (#X #Y a:(Array X) init:Y f:/[Int X Y /Y]) : Y =
  ( val limit = (dec (size a))
    def loop (v:Y x:Int):Y =
      if (<= x limit) then
        (loop (f x (nth a x) v) (inc x))
      else
        v
    (loop init 0)
  )
```

```
and revItFold (#X #Y a:(Array X) init:Y f:/[Int X Y /Y]) : Y =
  ( def loop (v:Y x:Int):Y =
      if (>= x 0) then
        (loop (f x (nth a x) v) (dec x))
     else
      v
   (loop init (dec (size a)))
  )

and apply (#X a:(Array X) f:/[X /[]]) : [] =
  ( val limit = (dec (size a))
    def loop (x:Int):[] =
      if (<= x limit) then
        ( (f (nth a x));
          (loop (inc x))
        )
      else
        []
   (loop 0)
  )

and revApply (#X a:(Array X) f:/[X /[]]) : [] =
  ( def loop (x:Int):[] =
      if (>= x 0) then
        ( (f (nth a x));
          (loop (dec x))
        )
      else
        []
   (loop (dec (size a)))
  )

and itApply (#X a:(Array X) f:/[Int X /[]]) : [] =
  ( val limit = (dec (size a))
    def loop (x:Int):[] =
      if (<= x limit) then
        ( (f x (nth a x));
          (loop (inc x))
        )
      else
        []
    (loop 0))

and revItApply (#X a:(Array X) f:/[Int X /[]]) : [] =
  ( def loop (x:Int):[] =
      if (>= x 0) then
        ( (f x (nth a x));
          (loop (dec x))
        )
      else
        []
   (loop (dec (size a)))
  )

and copy (#X a:(Array X)) : (Array X) =
  (tabulate (size a)
    \(x:Int):X = (nth a x)
  )
```

```
  and detect (#X a:(Array X) f:/[X /Bool]) : Bool =
    (fold a false
      \(element:X partialResult:Bool):Bool =
        (|| partialResult (f element))
    )

  [ empty=empty make=make tabulate=tabulate fromList=fromList size=size
    nth=nth update=update reset=reset rotate=rotate
    fold=fold revFold=revFold itFold=itFold revItFold=revItFold
    apply=apply revApply=revApply itApply=itApply revItApply=revItApply
    map=map revMap=revMap itMap=itMap revItMap=revItMap copy=copy
    detect=detect
  ]
)
```

2.4 Array2: 2-Dimensional Array

(Array2 X) is the type of two dimensional arrays which contain elements of type X.

2.4.1 Creation

(copy a) returns a copy of an 2D array a. 2D arrays are mutable and thus it has sense to copy them.

```
copy = /[#X (Array2 X) /(Array2 X)]
```

(empty) returns a 2D array of size zero. This function is useful since, unlike make, it does not require an initializing element.

```
empty = /[#X /(Array2 X)]
```

(make x w h) returns a new array of dimension [w h], with each element initialized to x. Both the width w and height h must be greater than or equal to zero.

```
make = /[#X X Int Int /(Array2 X)]
```

(tabulate [w h] f) creates an array of size [w h] where each element [x y] is initialised to (f x y). Both w and h must be greater than or equal to zero. The function f is called sequentially, and in order ([0 0], [0 1], ..., [0 (h-1)], [1 0], [1 1], ..., [w h]).

```
tabulate = /[#X [Int Int] /[Int Int /X] /(Array2 X)]
```

(fromList l) creates an array from a list of lists of elements. The sub-lists of l must all be of the same length (since Array2 only allows rectangular arrays). If this is not the case, we generate a runtime error.

```
fromList = /[#X (List (List X)) /(Array2 X)]
```

2.4.2 Interrogation

(size a) returns the width and height of a.

```
size = /[#X (Array2 X) /[Int Int]]
```

(nth a x y) looks up the element at position [x y] in a. It must be the case that [0 0] $\leq$ [x y] and [x y] $<$ (size a), otherwise we generate a runtime error.

```
nth = /[#X (Array2 X) Int Int /X]
```

(detect a f) applies f to each element of a. The whole function returns true if function f returns true at least for one element.

```
detect = /[#X (Array2 X) /[X /Bool] /Bool]
```

2.4.3 Modification

(update a x y v) updates the element at [x y] in a to be v. If [x y] is not a valid index for a then we generate a runtime error.

```
update = /[#X (Array2 X) Int Int X /[]]
```

(reset a x) sets the value of each array element to @x@.

```
reset = /[#X (Array2 X) X /[]]
```

The expression (rotate a dx dy) rotates a so that the [i j]'th element of a contains the value which used to be stored at index [(+ i dx) (+ j dy)]. For example, the expression (rotate a -1 -1) shifts the contents of a up and to the right by one.

```
rotate = /[#X (Array2 X) Int Int /[]]
```

2.4.4 Iteration

(fold a init f) applies f to each element of a, accumulating a result of type Y. f is called sequentially, and in order, with the initial accumulated argument being init. revFold behaves similarly, except that it iterates over a in reverse order.

```
fold = /[#X #Y (Array2 X) Y /[Int Int X Y /Y] /Y]
revFold = /[#X #Y (Array2 X) Y /[Int Int X Y /Y] /Y]
```

(itFold a i f) applies f to each element of a (plus its index), accumulating a result of type Y. f is called sequentially, and in order, with the initial accumulated argument being i. revItFold behaves similarly, except that it iterates over a in reverse order.

```
itFold = /[#X #Y (Array2 X) Y /[Int Int X Y /Y] /Y]
revItFold = /[#X #Y (Array2 X) Y /[Int Int X Y /Y] /Y]
```

(apply f a) applies f to each element of a. f is called sequentially, and in order. revMap behaves similarly, except that it iterates over a in reverse order.

```
apply = /[#X (Array2 X) /[X /[]] /[]]
revApply = /[#X (Array2 X) /[X /[]] /[]]
```

(itApply f a) applies f to each element of a, along with its indices. f is called sequentially, and in order. revItMap behaves similarly, except that it iterates over a in reverse order.

```
itApply = /[#X (Array2 X) /[Int Int X /[]] /[]]
revItApply = /[#X (Array2 X) /[Int Int X /[]] /[]]
```

(map a f) applies f to each element of a, updating each element in a with the result of applying f. f is called sequentially, and in order. revMap behaves similarly, except that it iterates over a in reverse order.

```
map = /[#X (Array2 X) /[X /X] /[]]
revMap = /[#X (Array2 X) /[X /X] /[]]
```

(itMap a f) applies f to each element of a, along with its index, updating each element in a with the result of applying f. f is called sequentially, and in order. revItMap behaves similarly, except that it iterates over a in reverse order.

```
itMap = /[#X (Array2 X) /[X /X] /[]]
revItMap = /[#X (Array2 X) /[X /X] /[]]
```

2.4.5 Implementation

20 ⟨*Untrusted/Array2.pi* 20⟩≡

```
import "Untrusted/Array"

val [
  #Array2 : (Pos Type -> Type)
```

```
  array2:[
    empty = /[#X /(Array2 X)]
    make = /[#X X Int Int /(Array2 X)]
    tabulate = /[#X [Int Int] /[Int Int /X] /(Array2 X)]
    fromList = /[#X (List(List X)) /(Array2 X)]
    size = /[#X (Array2 X) /[Int Int]]
    nth = /[#X (Array2 X) Int Int /X]
    update = /[#X (Array2 X) Int Int X /[]]
    reset = /[#X (Array2 X) X /[]]
    rotate = /[#X (Array2 X) Int Int /[]]
    fold = /[#X #Y (Array2 X) Y /[X Y /Y] /Y]
    revFold = /[#X #Y (Array2 X) Y /[X Y /Y] /Y]
    itFold = /[#X #Y (Array2 X) Y /[Int Int X Y /Y] /Y]
    revItFold = /[#X #Y (Array2 X) Y /[Int Int X Y /Y] /Y]
    apply = /[#X (Array2 X) /[X /[]] /[]]
    revApply = /[#X (Array2 X) /[X /[]] /[]]
    itApply = /[#X (Array2 X) /[Int Int X /[]] /[]]
    revItApply = /[#X (Array2 X) /[Int Int X /[]] /[]]
    map = /[#X (Array2 X) /[X /X] /[]]
    revMap = /[#X (Array2 X) /[X /X] /[]]
    itMap = /[#X (Array2 X) /[Int Int X /X] /[]]
    revItMap = /[#X (Array2 X) /[Int Int X /X] /[]]
    copy = /[#X (Array2 X) /(Array2 X)]
    detect = /[#X (Array2 X) /[X /Bool] /Bool]
  ]
] = (
  type (Array2 X) = (Array (Array X))

  def empty (#X) : (Array2 X) = (array.empty #(Array X)){-HACK-}

  def make (#X x:X w:Int h:Int) : (Array2 X) =
    if (&& (>= w 0) (>= h 0)) then
      (array.tabulate #(Array X) w \(_) = (array.make #X x h))
    else
      (error "array2.make: negative size")

  def tabulate (#X [w:Int h:Int] init:/[Int Int /X]) : (Array2 X) =
    if (&& (>= w 0) (>= h 0)) then
      (array.tabulate #(Array X) w
        \(x) = (array.tabulate #X h \(y) = (init x y)))
    else
      (error "array2.tabulate: negative size")

  def fromList (#X l:(List(List X))) : (Array2 X) =
    ( val width = (list.size l)
      val height = if (== width 0) then 0 else (list.size (car l))
      def makeColumn (column:(List X)) : (Array X) =
        if (== (list.size column) height) then
          (array.fromList column)
        else
          (error "array2.fromList: sub-lists have different lengths")
      (array.fromList (list.map l makeColumn))
    )

  def size (#X a:(Array2 X)) : [Int Int] =
    if (== (array.size a) 0) then
      [0 0]
    else
```

```
    [(array.size a) (array.size (array.nth a 0))]

inline def nth (#X a:(Array2 X) x:Int y:Int) : X =
  (array.nth (array.nth a x) y)

inline def update (#X a:(Array2 X) x:Int y:Int v:X) : [] =
  (array.update (array.nth a x) y v)

def reset (#X a:(Array2 X) x:X) : [] =
  (array.apply #(Array X) a \(col) = (array.reset col x))

def rotate (#X a:(Array2 X) dx:Int dy:Int) : [] =
  ( (array.apply #(Array X) a \(col) = (array.rotate col dy));
    (array.rotate a dx)
  )

def fold (#X #Y a:(Array2 X) init:Y f:/[X Y /Y]) : Y =
  (array.fold a init
    \(col:(Array X) acc:Y):Y =
      (array.fold col acc
        \(v:X acc:Y):Y = (f v acc)
      )
  )

def revFold (#X #Y a:(Array2 X) init:Y f:/[X Y /Y]) : Y =
  (array.revFold a init
    \(col:(Array X) acc:Y):Y =
      (array.revFold col acc
        \(v:X acc:Y):Y = (f v acc)
      )
  )

def itFold (#X #Y a:(Array2 X) init:Y f:/[Int Int X Y /Y]) : Y =
  (array.itFold a init
    \(x:Int col:(Array X) acc:Y):Y =
      (array.itFold col acc
        \(y:Int v:X acc:Y):Y = (f x y v acc)
      )
  )

def revItFold (#X #Y a:(Array2 X) init:Y f:/[Int Int X Y /Y]) : Y =
  (array.revItFold a init
    \(x:Int col:(Array X) acc:Y):Y =
      (array.revItFold col acc
        \(y:Int v:X acc:Y):Y = (f x y v acc)
      )
  )

def apply (#X a:(Array2 X) f:/[X /[]]) : [] =
  ( def fCol (col:(Array X)):[] =
      (array.apply col \(v:X):[] = (f v))
    (array.apply a fCol)
  )

def revApply (#X a:(Array2 X) f:/[X /[]]) : [] =
  ( def fCol (col:(Array X)):[] =
      (array.revApply col \(v:X):[] = (f v))
    (array.revApply a fCol)
```

```
  )

def itApply (#X a:(Array2 X) f:/[Int Int X /[]]) : [] =
  ( def fCol (x:Int col:(Array X)):[] =
      (array.itApply col \(y:Int v:X):[] = (f x y v))
    (array.itApply a fCol)
  )

def revItApply (#X a:(Array2 X) f:/[Int Int X /[]]) : [] =
  ( def fCol (x:Int col:(Array X)):[] =
      (array.revItApply col \(y:Int v:X):[] = (f x y v))
    (array.revItApply a fCol)
  )

def map (#X a:(Array2 X) f:/[X /X]) : [] =
  ( def fCol (col:(Array X)):[] =
      (array.map col \(v:X):X = (f v))
    (array.apply a fCol)
  )

def revMap (#X a:(Array2 X) f:/[X /X]) : [] =
  ( def fCol (col:(Array X)):[] =
      (array.revMap col \(v:X):X = (f v))
    (array.revApply a fCol)
  )

def itMap (#X a:(Array2 X) f:/[Int Int X /X]) : [] =
  ( def fCol (x:Int col:(Array X)):[] =
      (array.itMap col \(y:Int v:X):X = (f x y v))
    (array.itApply a fCol)
  )

def revItMap (#X a:(Array2 X) f:/[Int Int X /X]) : [] =
  ( def fCol (x:Int col:(Array X)):[] =
      (array.revItMap col \(y:Int v:X):X = (f x y v))
    (array.revItApply a fCol)
  )

and copy (#X a:(Array2 X)) : (Array2 X) =
  (tabulate (size a)
    \(x:Int y:Int):X = (nth a x y)
  )

and detect (#X a:(Array2 X) f:/[X /Bool]) : Bool =
  (fold a false
    \(element:X partialResult:Bool):Bool =
      (|| partialResult (f element))
  )


[ #Array2
  [ empty=empty make=make tabulate=tabulate fromList=fromList size=size
    nth=nth update=update reset=reset rotate=rotate
    fold=fold revFold=revFold itFold=itFold revItFold=revItFold
    apply=apply revApply=revApply itApply=itApply revItApply=revItApply
    map=map revMap=revMap itMap=itMap revItMap=revItMap copy=copy
    detect=detect
] ]
```

```
)
```

2.5 Bool

Type `Bool` is built in[3] and cannot be redefined. The Boolean constructors `true` and `false` are also built in, and cannot be redefined.

The code below defines the `bool` record. The definition is broken into two separate chunks (exhibited later in the text). The first chunk completes the type description of the `bool` record. The second chunk completes the complex value which defines various processes and finally returns a record of the corresponding type[4].

2.5.1 Iteration

`(while b f)` evaluates `(f)` while `(b)` is true. Similarly, `(until f b)` evaluates `(f)` until `(b)` is true.

```
while = /[/[/Bool] /[/[]] /[]]
until = /[/[/[]] /[/Bool] /[]]
```

2.5.2 Conversion

`(toString b)` converts `b` to a string.

```
toString = /[Bool /String]
```

`(fromInt x)` converts the integer `x` to a Boolean. If `x` is zero, we return `false`, otherwise we return `true`.

```
fromInt = /[Int /Bool]
```

`(toInt b)` converts `b` to the integer `1` if `b` is `true`, and `0` if `b` is `false`.

```
toInt = /[Bool /Int]
```

2.5.3 Implementation

25 ⟨*Untrusted/Bool.pi* 25⟩≡

```
import "Untrusted/Misc"

val bool: [
  &&  = /[Bool Bool /Bool]
  ||  = /[Bool Bool /Bool]
  xor = /[Bool Bool /Bool]
  not = /[Bool /Bool]
  while = /[/[/Bool] /[/[]] /[]]
  until = /[/[/[]] /[/Bool] /[]]
  toString = /[Bool /String]
  fromInt = /[Int /Bool]
  toInt = /[Bool /Int]
] = (
  val && = prim.&&
  val || = prim.||
  val xor = prim.xor
  val not = prim.not

  def while (b:/[/Bool] f:/[/[]]) : [] =
    if (b) then
```

[3] It is mapped to some built in type of the C language.

[4] Although it is not necessary to put definitions of processes into records; we could define them straightly on the top scope; it is advantageous technique to avoid process name clashes. This way `bool.toString` as well as `int.toString` can coexist in the same system.

```
      ( (f);
        (while b f)
      )
    else
      []

  def until (f:/[/[]] b:/[/Bool]) : [] =
    ( (f);
      if (not (b)) then
        (until f b)
      else
        []
    )

  inline def toString (b:Bool) : String =
    if b then
      "true"
    else
      "false"

  inline def fromInt (x:Int):Bool =
    if (prim.== x 0) then
      false
    else
      true

  inline def toInt (b:Bool) : Int = if b then 1 else 0

  [ && = && || = || xor=xor not=not while=while until=until
    toString=toString fromInt=fromInt toInt=toInt
  ]
)

val &&    = bool.&&
val ||    = bool.||
val xor   = bool.xor
val not   = bool.not
val while = bool.while
val until = bool.until
```

2.6 Bytes

2.6.1 Types

Pict representation of characters, integers, records and other kind of values has some advantages (these values can be mechanically garbage-collected) but also some disadvantages (the internal representation of characters, integers and record is different from the way how these things are represented in C and the C representation is in some case necessary).

This module provides a datatype `Bytes` with which we can construct and deconstruct raw sequences of bytes. This ability is useful for exchanging data between Pict processes and external entities. Instances of this type represent non-empty array of bytes.

2.6.2 Creation

`(make c n)` returns a new array containing `n` bytes, each initialized to `c`. The array size `n` must be greater than zero.

```
make = /[Char Int /Bytes]
```

`(fromString s)` creates a new array of bytes from a given (non-empty) string.

```
fromString = /[String /Bytes]
```

`(sub b minIndexOffset newSize)` function returns sub-array of a given array. The beginning of the new sub-array is specified by the `minIndexOffset` parameter and its size is given by the `newSize` parameter. Indexes of elements in the new array will be again relative to the beginning of that new array.

```
sub = /[Bytes Int Int /Bytes]
```

2.6.3 Interrogation

`(size b)` returns the size (in bytes) of `b`.

```
size = /[Bytes /Int]
```

`(nth b n)` looks up the `n`-th element of `b`.

```
nth = /[Bytes Int /Char]
```

2.6.4 Modification

`(update b n c)` updates the `n`-th element of `b` to be `c`.

```
update = /[Bytes Int Char /[]]
```

2.6.5 Implementation

The following record is used for representation of array of bytes. The `string` field refers to some string. The `minIndex` and `maxIndex` determine which portion of the string belongs to the byte array. Indexes (used in `sub`, `nth` and `update` functions) are always relative to the `minIndex`. The advantage of this representation is that we can create more byte arrays over one and the same string.

27 ⟨*internal representation of Bytes type* 27⟩≡ (28)
```
  type Bytes = [ string = String
                 minIndex = Int
                 maxIndex = Int
               ]
```

28 ⟨*Untrusted/Bytes.pi* 28⟩≡

```
import "Untrusted/String"

val [
  #Bytes
  bytes : [
    make = /[Char Int /Bytes]
    sub = /[Bytes Int Int /Bytes]
    fromString = /[String /Bytes]
    size = /[Bytes /Int]
    nth = /[Bytes Int /Char]
    update = /[Bytes Int Char /[]]
  ]
] = (
  ⟨internal representation of Bytes type 27⟩

  def make (c:Char n:Int) : Bytes =
    if (== 0 n) then
      (error "bytes.make: cannot create empty array of bytes")
    else
      [ string = (string.make c n)
        minIndex = 0
        maxIndex = (dec n)
      ]

  def sub (b:Bytes minIndexOffset:Int newSize:Int) : Bytes =
    ( val newMinIndex = (+ b.minIndex minIndexOffset)
      val newMaxIndex = (+ newMinIndex newSize)
      if (&& > (<= b.minIndex newMinIndex)
               (<= newMinIndex newMaxIndex)
               (<= newMaxIndex b.maxIndex)
         )
      then
        [ string=b.string
          minIndex=newMinIndex
          maxIndex=newMaxIndex
        ]
      else
        (error "bytes.sub: indexes out of range")
    )

  def size (b:Bytes) : Int =
    (inc (- b.maxIndex b.minIndex))

  def fromString (s:String) : Bytes =
    if (string.isEmpty s) then
      (error "bytes.fromString: cannot create empty array of bytes")
    else
      [ string = (string.copy s)
        minIndex = 0
        maxIndex = (dec (string.size s))
      ]

  def nth (b:Bytes n:Int) : Char =
    if (&& (<= 0 n)
           (<< n (size b))
       )
    then
```

```
      (string.nth b.string (+ b.minIndex n))
    else
      (error "bytes.nth: index out of range")

  def update (b:Bytes n:Int c:Char) : [] =
    if (&& (<= 0 n)
           (<< n (size b))
       )
    then
      (string.update b.string (+ b.minIndex n) c)
    else
      (error "bytes.update: index out of range")

  [ #Bytes
    [ make=make
      sub=sub
      fromString=fromString
      size=size
      nth=nth
      update=update
    ]
  ]
)
```

2.7 Cell

Cells represent persistent storage for channel names (and thus also values). Cells might but need not to contain a value.

`(empty)` creates an empty cell.

```
empty = /[#X /(Cell X)]
```

`(make value)` creates a new cell with a given initial `value`.

```
make = /[#X X /(Cell X)]
```

`(isEmpty cell)` determines whether a given `cell` contains a value or if it is empty.

```
isEmpty = /[#X (Cell X) /Bool]
```

`(clear cell)` clears a given `cell` if it contains a value. If it is already empty then it has no effect.

```
clear = /[#X (Cell X) /[]]
```

`(get cell)` retrieves value from a given `cell`. This operation is non destructive, i.e. the value stored in the `cell` might be retrieved as many times as necessary. If the cell is empty then this operation will block the sender until it gets some value.

```
get = /[#X (Cell X) /X]
```

`(put cell value)` puts a given `value` to a given `cell`. This operation is always non-blocking and works regardless of the fact whether the original `cell` were empty or not.

```
put = /[#X (Cell X) X /[]]
```

2.7.1 Implementation

30 ⟨*Untrusted/Cell.pi* 30⟩≡

```
import "Untrusted/Misc"

val [
  #Cell : (Pos Type -> Type)

  cell : [
    empty = /[#X /(Cell X)]
    make = /[#X X /(Cell X)]
    isEmpty = /[#X (Cell X) /Bool]
    clear = /[#X (Cell X) /[]]
    get = /[#X (Cell X) /X]
    put = /[#X (Cell X) X /[]]
  ]
] = (
  type (Cell X) = [value=^X isEmpty=^Bool]

  def empty (#X) : (Cell X) =
    ( new value:^X
      new isEmpty:^Bool
      run isEmpty!true
      [value=value isEmpty=isEmpty]
    )
```

```
  def cellIsEmpty [#X cell:(Cell X) r:/Bool] =
    cell.isEmpty?b = (cell.isEmpty!b | r!b)

  def clear [#X cell:(Cell X) r:/[]] =
    cell.isEmpty?b =
      if b then (cell.isEmpty!true | r![])
      else cell.value?_ = (cell.isEmpty!true | r![])

  def get [#X cell:(Cell X) r:/X] =
    cell.value?value = (cell.value!value | r!value)

  def put [#X cell:(Cell X) value:X r:/[]] =
    cell.isEmpty?isEmpty =
      if isEmpty then (cell.isEmpty!false | cell.value!value | r![])
      else cell.value?_ = (cell.isEmpty!false | cell.value!value | r![])

  def cellMake (#X value:X) : (Cell X) =
    ( val cell = (empty #X)
      (put cell value);
      cell
    )

  [ #Cell

    [ empty=empty make=cellMake isEmpty=cellIsEmpty clear=clear
      get=get put=put
  ] ]
)

val put = cell.put
val get = cell.get
```

2.8 Char

The type `Char` is built in[5] and is a subtype of `Int`. This means that integer arithmetic operations and comparisons will also accept characters as arguments. The integer value of each character is its ASCII code.

2.8.1 Character Classification

Checks for an alphanumeric character; it is equivalent to `(|| (isAlpha c) (isDigit c))`.

```
isAlnum = /[Char /Bool]
```

Checks for an alphabetic character; it is equivalent to `(|| (isUpper c) (isLower c))`.

```
isAlpha = /[Char /Bool]
```

Checks for a digit.

```
isDigit = /[Char /Bool]
```

Checks for a lower-case character.

```
isLower = /[Char /Bool]
```

Checks for any whitespace character.

```
isSpace = /[Char /Bool]
```

Checks for any printable character including space.

```
isPrint = /[Char /Bool]
```

Checks for any punctuation character.

```
isPunct = /[Char /Bool]
```

Checks for an uppercase letter.

```
isUpper = /[Char /Bool]
```

Checks for a hexadecimal digits, i.e. one of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `0`, `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, `F`.

```
isXDigit = /[Char /Bool]
```

2.8.2 Conversion

`toUpper` converts `c` to upper case, if possible. `toLower` converts `c` to lower case, if possible. The character `c` is returned unchanged if the conversion was not possible.

```
toUpper = /[Char /Char]
toLower = /[Char /Char]
```

Convert an ASCII code to a character. If `x` is not a valid ASCII code, then we generate a runtime error.

```
fromInt = /[Int /Char]
```

[5]It is mapped to some built in type of the C language.

`(toString c)` creates a string of size one, containing the character `c`.

```
toString = /[Char /String]
```

`(hash c)` returns a hash value for `c`.

```
hash = /[Char /Int]
```

2.8.3 Implementation

33 ⟨*Untrusted/Char.pi* 33⟩≡

```
import "Untrusted/Int"

val char: [
  isAlnum = /[Char /Bool]
  isAlpha = /[Char /Bool]
  isDigit = /[Char /Bool]
  isLower = /[Char /Bool]
  isSpace = /[Char /Bool]
  isPrint = /[Char /Bool]
  isPunct = /[Char /Bool]
  isUpper = /[Char /Bool]
  isXDigit = /[Char /Bool]
  toUpper = /[Char /Char]
  toLower = /[Char /Char]
  fromInt = /[Int /Char]
  toString = /[Char /String]
  hash = /[Char /Int]
] = (
  val toString=prim.charToString

  inline def fromInt (x:Int) : Char =
    if (&& (>= x 0) (<< x 256)) then
      (prim.intToChar x)
    else
      (error "ERROR: Integer out of range (char.fromInt).")

  inline def isLower (c:Char) : Bool =
    (&& (<= 'a' c) (<= c 'z'))

  inline def isUpper (c:Char) : Bool =
    (&& (<= 'A' c) (<= c 'Z'))

  inline def isAlpha (c:Char) : Bool =
    (|| (isUpper c) (isLower c))

  inline def isDigit (c:Char) : Bool =
    (&& (<= '0' c) (<= c '9'))

  inline def isAlnum (c:Char) : Bool =
    (|| (isAlpha c) (isDigit c))

  inline def isXDigit (c:Char) : Bool =
    (|| > (&& (<= '0' c) (<= c '9'))
          (&& (<= 'a' c) (<= c 'f'))
          (&& (<= 'A' c) (<= c 'F')) )

  inline def toUpper (c:Char) : Char =
```

```
    if (isLower c) then (prim.intToChar (+ (- c 'a') 'A'))
    else c

  inline def toLower (c:Char) : Char =
    if (isUpper c) then (prim.intToChar (+ (- c 'A') 'a'))
    else c

  inline def isSpace (c:Char) : Bool =
    (&& (<= 9 c)
        (<= c 13))

  inline def isPunct (c:Char) : Bool =
    (|| (&& (<= '!' c)
            (<= c '/'))
        (&& (<= ':' c)
            (<= c '?')))

  inline def isPrint (c:Char) : Bool =
    (|| > (isLower c)
          (isUpper c)
          (isDigit c)
          (isPunct c))

  inline def hash (c:Char) : Int = c

  [ isAlnum=isAlnum isAlpha=isAlpha isDigit=isDigit
    isLower=isLower isSpace=isSpace isPrint=isPrint isPunct=isPunct
    isUpper=isUpper isXDigit=isXDigit toUpper=toUpper
    toLower=toLower fromInt=fromInt toString=toString hash=hash
  ]
)
```

2.9 Choice

Although the Pict language does not directly support *summation* [4, page 87] (or in other words *choice*) it does not mean that Pict would be less expressive than the original π-calculus because of this. As it was shown in [**?**] the choice can be considered as a syntactic sugar expressible via core language constructs. This section contains implementation of the choice construct along the lines as suggested by the [**?**] article[6].

Although the doubts concerning availability of the choice construct were dispelled, the tax for not including the choice construct into the core language is that its usage—expressed indirectly via `choose`, `$` and `=>` functions defined below—is slightly more awkward.

Section 2.9.2 contains a proof of the correctness of this choice encoding.

Processes of the `Lock` type behave as functions with no parameters and a `Bool` return value.

These are the functions via which we can express the choice. They are used for example to implement the revoker, see Section 2.21.

```
choose = //Lock
$ = /[/Lock /Lock //Lock]
=> = /[#X ^X !X //Lock]
```

The auxiliary `(newLock)` function returns a function that returns a `Bool` value. The first time we evaluate it, it returns `true`. On every other evaluation it returns `false`. Example:

```
val lock = (newLock)
val b1:Bool = (lock)
val b2:Bool = (lock)
val b3:Bool = (lock)
val b4:Bool = (lock)
```

the `b1` will be set to `true` while `b2`, `b2` and `b3` will be set to false.

2.9.1 Implementation

35 ⟨*Untrusted/Choice.pi* 35⟩≡

```
import "Untrusted/Misc"

type Lock = /[/Bool]

val choice: [
  choose = //Lock
  $ = /[/Lock /Lock //Lock]
  => = /[#X ^X !X //Lock]
] = (
  def newLock [resultOfNewLock:/Lock] =
    ( new lock: ^[/Bool]
      def loop [] =
        lock?[resultOfLock] = (resultOfLock!false | loop![])
      ( resultOfNewLock!(rchan lock)
      | lock?[resultOfLock] = (resultOfLock!true | loop![])
    ) )

  and $ (e1:/Lock e2:/Lock) : /Lock =
    \lock:Lock = (e1!lock | e2!lock)

  and => (#X c:^X receiver:!X) : /Lock =
    \lock:Lock = c?v = if (lock) then receiver!v else c!v
```

[6] The code samples in the article seem to be written in a different (probably older) Pict version.

```
    and choose e:/Lock =
      e!(newLock)

    [ choose = choose
      $ = $
      => = =>
    ]
  )

  val choose = choice.choose
  val $ = choice.$
  val => = choice.=>
```

2.9.2 Few Remarks Concerning Correctness

The fact that Pict has formally defined semantics originally inspired us to try to prove the correctness of our choice encoding with respect to the original summation operator available in the π-calculus. To do that, we would have to:

- unfold simple derived forms used in our encoding to the core language
- unfold the continuation passing constructs to the core language
- examine capabilities of the unfolded program

The first two steps are easy[7]. The third step, however, is very hard. We are going to return to this later. In [5] it is possible to find a proof of a correctness and completeness of a similar choice encoding.

[7] In fact, they could be entirely supported by proper software.

2.10 `Cmp`: Result Comparison

Values of type `Cmp` are usually returned by comparison functions. They are particularly useful in the case of comparisons of strings or lists, since a single comparison operation, which returns a value of type `Cmp`, can be much more efficient that making two calls to an ordering predicate.

2.10.1 Construction

The values `LT`, `EQ` and `GT` indicate that a comparison operator found that its first argument was (respectively) strictly less than, equal to, or strictly greater than its second argument.

```
LT = Cmp
EQ = Cmp
GT = Cmp
```

2.10.2 Operations

The above functions test whether the result of a comparison was strictly less than, less than or equal to, equal to, not equal to, greater than or equal to, or strictly greater than.

```
lt = /[Cmp /Bool]
le = /[Cmp /Bool]
eq = /[Cmp /Bool]
ne = /[Cmp /Bool]
ge = /[Cmp /Bool]
gt = /[Cmp /Bool]
```

2.10.3 Conversion

`(toString c)` converts `c` to a string.

```
toString = /[Cmp /String]
```

2.10.4 Implementation

37 ⟨*Untrusted/Cmp.pi* 37⟩≡

```
import "Untrusted/Misc"

val [
  #Cmp

  cmp:[
    LT = Cmp
    EQ = Cmp
    GT = Cmp
    lt = /[Cmp /Bool]
    le = /[Cmp /Bool]
    eq = /[Cmp /Bool]
    ne = /[Cmp /Bool]
    ge = /[Cmp /Bool]
    gt = /[Cmp /Bool]
    toString = /[Cmp /String]
  ]
] = (
  type Cmp = Int
```

```
  {-
   - Note that we interpret ANY strictly positive number as meaning GT,
   - 0 as meaning EQ and ANY strictly negative number as meaning LT.
   -}

  val == = prim.==
  val << = prim.<<
  val not = prim.not
  val || = prim.||

  inline def <> (x:Int y:Int) : Bool = (not (prim.== x y))
  inline def <= (x:Int y:Int) : Bool = (|| (== x y) (<< x y))
  inline def >= (x:Int y:Int) : Bool = (not (<< x y))
  inline def >> (x:Int y:Int) : Bool = true
    {- (and (not (<< x y)) (not (== x y))) -}

  inline def toString (c:Cmp) : String =
    if (prim.<< c 0) then "LT"
                     else if (>> c 0) then "GT" else "EQ"


  [ #Cmp
    [ LT = -1
      EQ = 0
      GT = 1
      lt = \(c:Cmp):Bool = (prim.<< c 0)
      le = \(c:Cmp):Bool = (<= c 0)
      eq = \(c:Cmp):Bool = (prim.== c 0)
      ne = \(c:Cmp):Bool = (<> c 0)
      ge = \(c:Cmp):Bool = (>= c 0)
      gt = \(c:Cmp):Bool = (>> c 0)
      toString = toString
  ] ]
)
```

2.11 ICMP Packet

ICMP packets are described in RFC 792 [2]. Computation of the Internet checksum is explained in RFC 1071 [1].

2.11.1 Creation

`(makeEchoRequest identifier sequenceNumber data)` returns a new ICMP packet that represents "ICMP echo request" type with given values.

```
makeEchoRequest = /[Int Int String /ICMPPacket]
```

When string `s` contains encoding of an ICMP packet in the same form as they travel in the Internet then `(fromString s)` converts it into `ICMPPacket` value.

```
fromString = /[String /ICMPPacket]
```

2.11.2 Interrogation

`(isEchoRequest icmpPacket)` returns `true` if a given ICMP packet represents "ICMP echo request". If not, it returns `false`.

```
isEchoRequest = /[ICMPPacket /Bool]
```

`(isEchoReply icmpPacket)` returns `true` if a given ICMP packet represents "ICMP reply". If not, it returns `false`.

```
isEchoReply = /[ICMPPacket /Bool]
```

The following functions can be used for ICMP packet of proper types (either "ICMP echo request" or "ICMP reply"). They returns various fields of these kind of ICMP packets.

```
identifier = /[ICMPPacket /Int]
sequenceNumber = /[ICMPPacket /Int]
fromString = /[String /ICMPPacket]
```

2.11.3 Implementation

39 ⟨*Untrusted/ICMPPacket.pi* 39⟩≡

```
  import "Untrusted/Cell"
  import "Untrusted/String"

  val [
    #ICMPPacket < String
    icmpPacket: [
      makeEchoRequest = /[Int Int String /ICMPPacket]
      isEchoRequest = /[ICMPPacket /Bool]
      isEchoReply = /[ICMPPacket /Bool]
      identifier = /[ICMPPacket /Int]
      sequenceNumber = /[ICMPPacket /Int]
      fromString = /[String /ICMPPacket]
    ]
  ] = (
    type ICMPPacket = String

    {- This function returns 16-bit internet checksum
     - of a given string. In the result, only lower
```

```
 - 16-bits are significant.
 -}
def computeChecksum (s:String) : Int =
  ( {- Return n-th word (2 bytes) of a given string. -}
    def getNthWyde (s:String n:Int) : Int =
      (lor (string.nth s (* n 2))
           (shl (string.nth s (inc (* n 2))) 8)
      )

    val stringSize = (string.size s)
    val numberOfWords = (div stringSize 2)
    val sum = (cell.make 0)
    (int.for 0 (dec numberOfWords)
      \(wordIndex:Int):[] = (cell.put sum
                                      (+ (cell.get sum)
                                         (getNthWyde s wordIndex)
                            )         )
    );

    {- Handle a case when there is odd number of bytes. -}
    if (== (mod stringSize 2) 1) then
      (cell.put sum (+ (cell.get sum)
                       (string.nth s (dec stringSize))
      )             )
    else
      [];

    (lnot (+ (land 65535 (cell.get sum))
             (land 65535 (shr (cell.get sum) 16))
    )     )
  )

{- Create an ICMP echo request packet. It will have only 8 bytes.
 - The "type" field (1 byte) is 8
 - The "code" field (1 byte) is 0
 - The "checksum" field (2 bytes) will be computed
 - The "identifier" field (2 bytes) is hardcoded to 100.
 - The "sequence number" field (2 bytes) is specified by the caller
 -}
def makeEchoRequest (identifier:Int sequenceNumber:Int data:String) : String =
  ( val dataSize = (string.size data)
    val packet = (string.make '\000' (+ 8 dataSize))

    {- Set the ‘type’ (byte) field. -}
    (string.update packet 0 '\008');

    {- Set the ‘code’ (byte) field. -}
    (string.update packet 1 '\000');

    {- The checksum field is already set to '\000' as it is required. -}

    {- Set the ‘identifier’ (2 bigendian bytes) field. -}
    (string.update packet 4 (char.fromInt (land 255 (shr identifier 8))));
    (string.update packet 5 (char.fromInt (land 255 identifier)));

    {- Set the ‘sequence number’ (2 bigendian bytes) field. -}
    (string.update packet 6 (char.fromInt (land 255 (shr sequenceNumber 8))));
    (string.update packet 7 (char.fromInt (land 255 sequenceNumber)));
```

```
    {- Copy the data we were provided into the ICMP packet. -}
    (string.itApply data
      \(i:Int ch:Char) = (string.update packet (+ 8 i) ch)
    );

    {- Compute the checksum and set the ‘checksum’ (2 bytes) field.
     - The result is already in ‘bigendian’ encoding because
     - this is how words in the packet should be interpreted so we
     - copy the checksum into the packet without swapping bytes.
     -}
    val checksum = (computeChecksum packet)
    val checksum = [ lowerByte = (land 255 checksum)
                     upperByte = (land 255 (shr checksum 8))
                   ]
    (string.update packet 2 (char.fromInt checksum.lowerByte));
    (string.update packet 3 (char.fromInt checksum.upperByte));

    packet
  )

{- Return ‘true’ if a given packet represents echo request. -}
def isEchoRequest (packet:ICMPPacket) : Bool =
  (&& (== 0 (string.nth packet 0))
      (== 8 (string.nth packet 1))
  )

{- Return ‘true’ if a given packet represents echo reply. -}
def isEchoReply (packet:ICMPPacket) : Bool =
  (&& (== 0 (string.nth packet 0))
      (== 0 (string.nth packet 1))
  )

{- Return the identifier (encoded as big endian) -}
def identifier (packet:ICMPPacket) : Int =
  (lor (shl (string.nth packet 4) 8)
       (string.nth packet 5)
  )

{- Return the sequence number (encoded as big endian) -}
def sequenceNumber (packet:ICMPPacket) : Int =
  (lor (shl (string.nth packet 6) 8)
       (string.nth packet 7)
  )

{- The ‘fromString’ function can be applied at the result
 - of the ‘ipPacket.data’ in case if IP packet carries
 - ICMP packet
 -}
def fromString (s:String) : ICMPPacket =
  (string.copy s)

[ #ICMPPacket
  [ makeEchoRequest=makeEchoRequest
    isEchoRequest=isEchoRequest
    isEchoReply=isEchoReply
    identifier=identifier
    sequenceNumber=sequenceNumber
```

```
        fromString=fromString
      ]
    ]
  )
```

2.12 Int: Integers

The type `Int` is built in[8], and cannot be redefined.

2.12.1 Arithmetic Operations

Addition and subtraction.

```
+ = /[Int Int /Int]
- = /[Int Int /Int]
```

Product, quotient and modulus. `div` and `mod` generate a runtime error if the divisor is zero. Note that, unlike the standard mathematical operations, `div` and `mod` round towards zero, so be careful if you intend to pass `div` and `mod` negative arguments.

```
* = /[Int Int /Int]
div = /[Int Int /Int]
mod = /[Int Int /Int]
```

Negation, predecessor and successor.

```
neg = /[Int /Int]
dec = /[Int /Int]
inc = /[Int /Int]
```

`(sgn x)` returns `1` if `x` is strictly greater than zero, `0` if `x` equals zero, and `-1` if `x` is strictly less than zero.

```
sgn = /[Int /Int]
```

`(abs x)` returns the absolute value of `x`.

```
abs = /[Int /Int]
```

`(gcd m n)` returns greatest common divisor of a given two natural numbers `m` and `n`.

```
gcd = /[Int Int /Int]
```

`(power a b)` returns $\mathtt{a}^{\mathtt{b}}$. The `b` value must be non-negative.

2.12.2 Bitwise Operations

Bitwise and, or, exclusive-or and negation.

```
land = /[Int Int /Int]
lor = /[Int Int /Int]
lxor = /[Int Int /Int]
lnot = /[Int /Int]
```

`(isSet bit=i x)` returns `true` if the `i`'th bit ($0 \leq \mathtt{i} \leq 30$) of `x` is set[9].

```
isSet = /[bit=Int Int /Bool]
```

`(set bit=i x)` returns an integer which is a copy of the original integer whose `i`-th bit is set.

[8] It is mapped to some built in type of the C language.

[9] Representation of Pict integers is shown in Definition 8.5 in [9]. The least significant bit of the raw integer representation is invisible to the programmer. When he refers to the 0-th bit, he will actually reveal the 1-st bit of the raw representation. When he refers to the 30-th bit of the integer, he will actually reveal the 31-st bit.

```
set = /[bit=Int Int /Int]
```

`(reset bit=i x)` returns an integer which is a copy of the original integer whose `i`-th bit is reset.

```
reset = /[bit=Int Int /Int]
```

Functions `(shr x y)` and `(shl x y)` can be used to shift a given integer `x` given `y` bits to the right or to the left respectively.

```
shr = /[Int Int /Int]
shl = /[Int Int /Int]
```

2.12.3 Comparison

Maximum and minimum.

```
max = /[Int Int /Int]
min = /[Int Int /Int]
```

The `minmax` function resembles the `min:max:` method in Smalltalk. It behaves as follows:

$$minmax(x, y, z) = \begin{cases} x & \text{if } y < x \\ y & \text{if } x \leq y \leq z \\ z & \text{if } z < y \end{cases}$$

```
minmax = /[Int Int Int /Int]
```

Equality, inequality.

```
== = /[Int Int /Bool]
<> = /[Int Int /Bool]
```

Strictly greater than, and strictly less than.

```
>> = /[Int Int /Bool]
\begin{verbatim}
```

Greater than or equal to, and less than or equal to.

```
>= = /[Int Int /Bool]
<= = /[Int Int /Bool]
```

`(cmp x y)` returns a single value indicating the ordering of `x` and `y` (cf. Section 2.10)

```
cmp = /[Int Int /Cmp]
```

2.12.4 Iteration

`(apply start inc finish f)` applies `f` to the sequence of integers `start`, `start+inc`, … until it reaches an integer strictly greater than `finish`, or strictly less than `finish` if `inc` is negative.

For example, the following code:

```
(apply 0 3 10
           \(n) = ( (pr ($$ n));
                    (pr "\n")
                  )
);
```

will print

```
0
3
6
9
```

on the screen.

```
apply = /[Int Int Int /[Int /[]] /[]]
```

`(fold start inc finish f init)` applies `f` to the sequence of integers `start`, `start+inc`, ... until it reaches an integer strictly greater than `finish`, or strictly less than `finish` if `inc` is negative. Successive applications of `f` accumulate a result of type `R`, with `init` being the initial accumulated result.

For example, the following expression:

```
(fold 0 1 100
      \(n:Int s:Int):Int = (+ n s)
      0
)
```

evaluates to the sum of all integers in the interval $\langle 0, 10\rangle$, i.e. 5050.

```
fold = /[#R Int Int Int /[Int R /R] R /R]
```

The `for` function is specialization of the `apply` function. The expression `(for min max f)` applies `f` to the integers `min`, ..., `max`.

```
for = /[Int Int /[Int /[]] /[]]
```

2.12.5 Conversion

`(hash x)` returns a hash value for `x`.

```
hash = /[Int /Int]
```

2.12.6 Implementation

45 ⟨*Untrusted/Int.pi* 45⟩≡

```
import "Untrusted/Cmp"
import "Untrusted/Bool"

val int: [
  + = /[Int Int /Int]
  - = /[Int Int /Int]
  * = /[Int Int /Int]
  div = /[Int Int /Int]
  mod = /[Int Int /Int]
  neg = /[Int /Int]
  dec = /[Int /Int]
  inc = /[Int /Int]
  sgn = /[Int /Int]
  abs = /[Int /Int]
  gcd = /[Int Int /Int]
  land = /[Int Int /Int]
  lor = /[Int Int /Int]
  lxor = /[Int Int /Int]
```

```
    lnot = /[Int /Int]
    isSet = /[bit=Int Int /Bool]
    set = /[bit=Int Int /Int]
    reset = /[bit=Int Int /Int]
    shr = /[Int Int /Int]
    shl = /[Int Int /Int]
    max = /[Int Int /Int]
    min = /[Int Int /Int]
    minmax = /[Int Int Int /Int]
    == = /[Int Int /Bool]
    <> = /[Int Int /Bool]
    >> = /[Int Int /Bool]
    << = /[Int Int /Bool]
    >= = /[Int Int /Bool]
    <= = /[Int Int /Bool]
    cmp = /[Int Int /Cmp]
    apply = /[Int Int Int /[Int /[]] /[]]
    fold = /[#R Int Int Int /[Int R /R] R /R]
    for = /[Int Int /[Int /[]] /[]]
    hash = /[Int /Int]
    upperBound = Int
    lowerBound = Int
    power = /[Int Int /Int]
  ] = (
    val + = prim.+
    val - = prim.-
    val * = prim.*
    val div = prim.div
    val mod = prim.mod
    val == = prim.==
    val << = prim.<<
    val land = prim.land
    val lor = prim.lor
    val lxor = prim.lxor
    val lnot = prim.lnot
    val shr = prim.shr
    val shl = prim.shl

    inline def neg (x:Int) : Int = (- 0 x)
    inline def dec (x:Int) : Int = (- x 1)
    inline def inc (x:Int) : Int = (+ x 1)
    inline def <> (x:Int y:Int) : Bool = (not (== x y))
    inline def <= (x:Int y:Int) : Bool = (|| (== x y) (<< x y))
    inline def >= (x:Int y:Int) : Bool = (not (<< x y))
    inline def >> (x:Int y:Int) : Bool = (&& (not (<< x y)) (not (== x y)))

    inline def sgn (x:Int) : Int =
      if (<< x 0) then
        -1
      else if (== x 0) then
        0
      else
        1

    inline def abs (x:Int) : Int =
      if (<< x 0) then
        (neg x)
      else
```

```
      x

  def gcd (m:Int n:Int) : Int =
    ( val r = (mod m n)
      if (== r 0) then
        n
      else
        (gcd n r)
    )

  inline def max (x:Int y:Int) : Int =
    if (<< x y) then
      y
    else
      x

  inline def min (x:Int y:Int) : Int =
    if (<< x y) then
      x
    else
      y

  inline def minmax (x:Int y:Int z:Int) : Int = (max x (min y z))

  inline def isSet (bit=i:Int x:Int) : Bool =
    (== 1 (land (shr x i) 1))

  inline def set (bit=i:Int x:Int) : Int =
    (lor (shl 1 i) x)

  inline def reset (bit=i:Int x:Int) :Int =
    (land (lnot (shl 1 i)) x)

  inline def cmp (x:Int y:Int) : Cmp =
    if (<< x y) then
      cmp.LT
    else if (== x y) then
      cmp.EQ
    else
      cmp.GT

  def apply (start:Int inc:Int finish:Int f:/[Int /[]]) : [] =
    if (>> inc 0) then
      ( def loop (x:Int): [] =
          if (<= x finish) then ((f x); (loop (+ x inc))) else []
        (loop start)
      )
    else
      ( def loop (x:Int): [] =
          if (>= x finish) then ((f x); (loop(+ x inc))) else []
        (loop start)
      )

  def fold (#R start:Int inc:Int finish:Int f:/[Int R /R] init:R) : R =
    if (>> inc 0) then
      ( def loop (x:Int v:R) : R =
          if (<= x finish) then (loop (+ x inc) (f x v)) else v
        (loop start init)
```

```
      )
    else
      ( def loop (x:Int v:R) : R =
          if (>= x finish) then (loop (+ x inc) (f x v)) else v
        (loop start init)
      )

  inline def for (min:Int max:Int f:/[Int /[]]) : [] = (apply min 1 max f)

  inline def hash (x:Int) : Int = x

  def power (a:Int b:Int) : Int =
    if (<< b 0) then
      (error "int.power: negative exponent")
    else if (== b 0) then
      1
    else
      (* a (power a (dec b)))

  [ + = +  - = -  * = *  div=div mod=mod neg=neg dec=dec inc=inc sgn=sgn
    abs=abs gcd=gcd land=land lor=lor lxor=lxor lnot=lnot isSet=isSet set=set
    reset=reset shr=shr shl=shl max=max min=min minmax=minmax == = ==
    <> = <> >> = >>  << = << >= = >=  <= = <=  cmp=cmp apply=apply fold=fold
    for=for hash=hash upperBound = 536870911 lowerBound = -536870912
    power=power
  ]
)

val + = int.+
val - = int.-
val inc = int.inc
val dec = int.dec
val neg = int.neg
val * = int.*
val div = int.div
val mod = int.mod
val == = int.==
val <> = int.<>
val >> = int.>>
val << = int.<<
val >= = int.>=
val <= = int.<=
val land = int.land
val lor = int.lor
val lnot = int.lnot
val shl = int.shl
val shr = int.shr
```

2.13 Fd: File-descriptor Operations

This is the “guest” part of the Fd functionality. The “host” part is defined in Section 3.3.

49 ⟨*Untrusted/Fd.pi* 49⟩≡

```
import "Untrusted/String"

type FdAPI = [
  pr = /[String /[]]
  prErr = /[String /[]]
  nl = /[/[]]
  prNL = /[String /[]]
  print = /String
  printi = /Int
]

def makeFd (#Fd stdin:Fd stdout:Fd stderr:Fd
                fdwrite:/[Fd String Int /[]]
           ) : FdAPI =
( inline def pr (s:String) : [] =
    (fdwrite stdout s (string.size s))

  inline def prErr (s:String) : [] =
    (fdwrite stderr s (string.size s))

  inline def nl () : [] = (pr "\n")

  inline def prNL (s:String) : [] =
    ( (pr s);
      (nl)
    )

  inline def print s:String = ((pr (+$ s "\n")); ())

  inline def printi i:Int = print!($$ i)

  [ pr=pr prErr=prErr nl=nl prNL=prNL print=print printi=printi ]
)
```

2.14 IP Address

Values of `IPAddress` type represent IPv4 addresses. Internally we represent them as a quadruple of integers `byte3 byte2 byte1 byte0` where `byte0` is the least significant byte and `byte3` is the most significant byte.

`(make byte0 byte1 byte2 byte3)` creates a new value of `IPAddress` type.

```
make = /[Int Int Int Int /IPAddress]
```

`(serialize ipAddress)` returns a string that contains encoding of a given `ipAddress`. `ipAddress` encoded this way can be embedded into network packets of various types.

```
serialize = /[IPAddress /String]
```

`(toString ipAddress)` returns a string that represents human readable form of a given `ipAddress`.

2.14.1 Implementation

50 ⟨*Untrusted/IPAddress.pi* 50⟩≡

```
import "Untrusted/String"

val [
  #IPAddress < [Int Int Int Int]
  ipAddress : [
    make = /[Int Int Int Int /IPAddress]
    serialize = /[IPAddress /String]
    toString = /[IPAddress /String]
  ]
] = (
  type IPAddress = [Int Int Int Int]

  def make (byte0:Int byte1:Int byte2:Int byte3:Int) : IPAddress =
    if ( && > (<= 0 byte0)
              (<= byte0 255)
              (<= 0 byte1)
              (<= byte1 255)
              (<= 0 byte2)
              (<= byte2 255)
              (<= 0 byte3)
              (<= byte3 255)
       ) then
      [byte0 byte1 byte2 byte3]
    else
      (error "ipAddress.make: nonsense parameters")

    def serialize (ipAddress:IPAddress) : String =
      ( val [byte0 byte1 byte2 byte3] = ipAddress
        (+$ > ($$ byte3) "." ($$ byte2) "."
              ($$ byte1) "." ($$ byte0))
      )

    def toString (ipAddress:IPAddress) : String =
      ( val [byte0 byte1 byte2 byte3] = ipAddress
        (+$ > ($$ byte3) "." ($$ byte2) "." ($$ byte1) "." ($$ byte0))
      )

  [ #IPAddress
    [ make=make serialize=serialize toString=toString ]
  ]
```

```
)
```

2.15 IP Packet

Values of `IPPacket` type represent IP packets.

2.15.1 Constants

`protocols.icmp` is a value of "protocol" field of all IP packets with netsted ICMP packets.

2.15.2 Creation

If a string `s` contains encoding of an IP packet in that form in which they travel in the Internet, then `(fromString s)` converts it into `IPPacket` value.

```
fromString = /[String /IPPacket]
```

2.15.3 Interrogation

The following functions extract various fields of a given IP packet.

```
sourceAddress = /[IPPacket /IPAddress]
destinationAddress = /[IPPacket /IPAddress]
protocol = /[IPPacket /Int]
size = /[IPPacket /Int]
data = /[IPPacket /String]
headerSize = /[IPPacket /Int]
```

2.15.4 Implementation

52 ⟨*Untrusted/IPPacket.pi* 52⟩≡

```
import "Untrusted/IPAddress"

val [
  #IPPacket < String

  ipPacket: [
    protocols = [ icmp = Int ]
    fromString = /[String /IPPacket]
    sourceAddress = /[IPPacket /IPAddress]
    destinationAddress = /[IPPacket /IPAddress]
    protocol = /[IPPacket /Int]
    size = /[IPPacket /Int]
    data = /[IPPacket /String]
    headerSize = /[IPPacket /Int]
  ]
] = (
  type IPPacket = String

  def fromString (s:String) : IPPacket =
    (string.copy s)

  def sourceAddress (packet:IPPacket) : IPAddress =
    ( val byte0 = (string.nth packet 15)
      val byte1 = (string.nth packet 14)
      val byte2 = (string.nth packet 13)
      val byte3 = (string.nth packet 12)
```

```
      (ipAddress.make byte0 byte1 byte2 byte3)
    )

  def destinationAddress (packet:IPPacket) : IPAddress =
    ( val byte0 = (string.nth packet 19)
      val byte1 = (string.nth packet 18)
      val byte2 = (string.nth packet 17)
      val byte3 = (string.nth packet 16)

      (ipAddress.make byte0 byte1 byte2 byte3)
    )

  def protocol (packet:IPPacket) : Int =
    (string.nth packet 9)

  def size (packet:IPPacket) : Int =
    (lor (shl (string.nth packet 2) 8)
         (string.nth packet 3)
    )

  def headerSize (packet:IPPacket) : Int =
    (shl (land (string.nth packet 0) 15) 2)

  def data (packet:IPPacket) : String =
    (string.sub packet (headerSize packet)
                       (- (size packet) (headerSize packet))
    )

  [ #IPPacket

    [ protocols = [ icmp=1 ]

      fromString=fromString sourceAddress=sourceAddress
      destinationAddress=destinationAddress protocol=protocol
      size=size data=data headerSize=headerSize
    ]
  ]
)
```

2.16 Lid

Our lid idiom is loosely based on the Caretaker [3, §9.3][10].

The revoker, defined in Section 2.21, provides slightly simpler functionality.

Values of the `Lid` type are returned by `makeNeg` and `makePos` lid constructors. These values can be used to uncover or cover the respective channel.

The `(makeNeg destination)` enable us to issue a revocable capability to send messages through the `destination` channel.

`(makeNeg target)` can be used to create a lid for a negative channel name. This function returns a couple `[proxy lid]`. The `proxy` can be handed out to the client. When a client sends a value to this channel, it will be forwarded to the `target` channel. The `lid` can be used by the lid creator to stop forwarding of values from the `proxy` to the `target`

```
makeNeg = /[#X !X /[!X Lid]]
```

`(makePos source)` can be used to create a lid for a positive channel name. This function returns a couple `[proxy lid]`. The `proxy` can be handed out to the client. Values sent to the `proxy` channel will be forwarded to the proxy channel and thus made available to the client holding the `proxy` capability. The `lid` can be used by the lid creator to stop forwarding values from the `source` to the `proxy` channel.

```
makePos = /[#X ?X /[?X Lid]]
```

2.16.1 Implementation

54 ⟨*Untrusted/Lid.pi* 54⟩≡

```
import "Untrusted/Cell"

val [
  #Lid

  lid:[
    makeNeg = /[#X !X /[!X Lid]]
    makePos = /[#X ?X /[?X Lid]]
  ]
] = (
  type Lid =
    [ cover = /[/[]]
      uncover = /[/[]]
    ]


  def makeNeg (#X destination:!X) : [!X Lid] =
  ( val covered = (cell.make false)
    new proxy: ^X
    def loop [proxy:?X destination:!X] =
      proxy?v = if (get covered) then
                  loop![proxy destination]
                else
                  ( destination!v | loop![proxy destination] )
    run loop![proxy destination]
    val lid:Lid = [ cover = \() = (put covered true)
                    uncover = \() = (put covered false)]
    [proxy lid]
```

[10]We have decided not to use the Caretaker name because Lid seems to be more appropriate. Lids cover channels. Lids can be covered or uncovered. Additionally, since channels have two distinct types of ends (the positive and the negative one), we had to define to define two distinct kinds of lids.

```
  )

  def makePos (#X source:?X) : [?X Lid] =
  ( val covered = (cell.make false)
    new proxy: ^X
    def loop [proxy:!X source:?X] =
      source?v = if (get covered) then
                   loop![proxy source]
                 else
                   ( proxy!v | loop![proxy source] )
    run loop![proxy source]
    val lid:Lid = [ cover = \() = (put covered true)
                    uncover = \() = (put covered false)]
    [proxy lid]
  )

  [ #Lid
    [makeNeg=makeNeg makePos=makePos]
  ]
)
```

2.17 List

(List X) is the type of lists of elements of type X.

2.17.1 Creation

nil is the empty list.

```
nil = (List Bot)
```

(cons x l) creates a new cons cell.

```
cons = /[#X X (List X) /(List X)]
```

(make n x) creates a list of size n, with each cell containing x.

```
make = /[#X Int X /(List X)]
```

(tabulate n f) creates a list of size n, such that the element with index i is initialized to (f i).

```
tabulate = /[#X Int /[Int /X] /(List X)]
```

2.17.2 Interrogation

(null l) returns a boolean indicating whether l is the empty list.

```
null = /[#X (List X) /Bool]
```

(car l) returns the head of the list l, if l is a cons cell. Causes a runtime failure if l is the empty list.

```
car = /[#X (List X) /X]
```

(cdr l) returns the tail of the list l, if l is a cons cell. Causes a runtime failure if l is the empty list.

```
cdr = /[#X (List X) /(List X)]
```

(case l n c) returns the result of evaluating (n) if l is the empty list, otherwise it returns the result of evaluating (c hd tl) where hd and tl are the head and tail of the list l.

```
case = /[#X #R (List X) /[/R] /[X (List X) /R] /R]
```

(size l) returns the size of l.

```
size = /[#X (List X) /Int]
```

(nth l n) finds the n-th element of the list l. Causes a

```
nth = /[#X (List X) Int /X]
```

(detect l f) applies f to each element of l. The whole function returns true if function f returns true at least for one element.

```
detect = /[#X (List X) /[X /Bool] /Bool]
```

2.17.3 Iteration

`(apply l f)` applies `f` to each element of `l` (sequentially and in order). `revApply` behaves just like `apply`, except that it traverses `l` in reverse order.

```
apply = /[#X (List X) /[X /[]] /[]]
revApply = /[#X (List X) /[X /[]] /[]]
```

`(itApply l f)` applies `f` to each element of `l` (sequentially and in order), passing `f` the index of each element. `revItApply` behaves just the same as `itApply`, except that it traverses `l` in reverse order.

```
itApply = /[#X (List X) /[Int X /[]] /[]]
revItApply = /[#X (List X) /[Int X /[]] /[]]
```

`(fold l init f)` applies `f` to each element of `l` (sequentially and in order), passing `f` an accumulated result of type `R`. The initial accumulated result is `init`. `revFold` behaves just like `fold`, except that it traverses `l` in reverse order.

```
fold = /[#X #R (List X) R /[X R /R] /R]
revFold = /[#X #R (List X) R /[X R /R] /R]
```

`(itFold l i f)` applies `f` to each element of `l` (sequentially and in order), passing `f` the index of each element, and an accumulated result of type `R`. The initial accumulated result is `i`. `revItFold` behaves just like `itFold`, except that it traverses `l` in reverse order.

```
itFold = /[#X #R (List X) R /[Int X R /R] /R]
revItFold = /[#X #R (List X) R /[Int X R /R] /R]
```

`(map l f)` applies `f` to each element of `l`, updating each element in `l` with the result of applying `f`. `f` is called sequentially and in order. `revMap` behaves similarly, except that it traverses `l` in reverse order.

```
map = /[#X (List X) /[X /X] /[]]
revMap = /[#X (List X) /[X /X] /[]]
```

`(itMap l f)` applies `f` to each element of `l`, along with its index, updating each element in `l` with the result of applying `f`. `f` is called sequentially and in order. `revItMap` behaves similarly, except that it traverses `l` in reverse order.

```
itMap = /[#X (List X) /[Int X /X] /[]]
revItMap = /[#X (List X) /[Int X /X] /[]]
```

`(filter l f)` returns a list containing those elements of `l` for which `f` returns `true`.

```
filter = /[#X (List X) /[X /Bool] /(List X)]
```

2.17.4 Combination

`(rev l)` reverses the list `l`.

```
rev = /[#X (List X) /(List X)]
```

`(append l1 l2)` appends the lists `l1` and `l2`. `(revAppend l1 l2)` reverses `l1` and appends it to `l2`.

```
append = /[#X (List X) (List X) /(List X)]
revAppend = /[#X (List X) (List X) /(List X)]
```

2.17.5 Sorting

`(sort l f d)` sorts `l` according to the comparison function `f`. If `d` is `true`, then all but one of each set of elements of `l` that are judged equal by `f` will be dropped.

```
sort = /[#X (List X) /[X X /Cmp] Bool /(List X)]
```

2.17.6 Conversion

Given a hash function `f` for values of type `X`, `(hash l f)` returns a hash value for a list `l` of type `(List X)`.

```
hash = /[#X (List X) /[X /Int] /Int]
```

2.17.7 Comparison

Given an comparison function `f` for values of type `X`, `(cmp l1 l2 f)` returns a single value indicating the ordering of `l1` and `l2` (cf. Section 2.10).

```
cmp = /[#X (List X) (List X) /[X X /Cmp] /Cmp]
```

2.17.8 Implementation

58a ⟨*common types* 58a⟩≡ (94) 13 ▷

```
val [#List : (Pos Type -> Type)] = [#\X = Top]
```

58b ⟨*Untrusted/List.pi* 58b⟩≡

```
import "Untrusted/Bool"
import "Untrusted/Int"

val list: [
  nil = (List Bot)
  cons = /[#X X (List X) /(List X)]
  make = /[#X Int X /(List X)]
  tabulate = /[#X Int /[Int /X] /(List X)]
  null = /[#X (List X) /Bool]
  car = /[#X (List X) /X]
  cdr = /[#X (List X) /(List X)]
  case = /[#X #R (List X) /[/R] /[X (List X) /R] /R]
  size = /[#X (List X) /Int]
  nth = /[#X (List X) Int /X]
  apply = /[#X (List X) /[X /[]] /[]]
  revApply = /[#X (List X) /[X /[]] /[]]
  itApply = /[#X (List X) /[Int X /[]] /[]]
  revItApply = /[#X (List X) /[Int X /[]] /[]]
  fold = /[#X #R (List X) R /[X R /R] /R]
  revFold = /[#X #R (List X) R /[X R /R] /R]
  itFold = /[#X #R (List X) R /[Int X R /R] /R]
  revItFold = /[#X #R (List X) R /[Int X R /R] /R]
  map = /[#X #R (List X) /[X /R] /(List R)]
  revMap = /[#X #R (List X) /[X /R] /(List R)]
  itMap = /[#X #R (List X) /[Int X /R] /(List R)]
  revItMap = /[#X #R (List X) /[Int X /R] /(List R)]
  filter = /[#X (List X) /[X /Bool] /(List X)]
  rev = /[#X (List X) /(List X)]
  append = /[#X (List X) (List X) /(List X)]
  revAppend = /[#X (List X) (List X) /(List X)]
  sort = /[#X (List X) /[X X /Cmp] Bool /(List X)]
```

```
    hash = /[#X (List X) /[X /Int] /Int]
    cmp = /[#X (List X) (List X) /[X X /Cmp] /Cmp]
    detect = /[#X (List X) /[X /Bool] /Bool]
  ] = (
    val nil = prim.nil
    val cons = prim.cons
    val null = prim.null
    val car = prim.car
    val cdr = prim.cdr

    def make (#X n:Int x:X) : (List X) =
      if (== n 0) then
        nil
      else
        (cons x (make (dec n) x))

    def make (#X n:Int x:X) : (List X) =
      ( def loop (n:Int l:(List X)) : (List X) =
          if (== n 0) then l else (loop (dec n) (cons x l))
        (loop n nil)
      )

    def tabulate (#X n:Int f:/[Int /X]) : (List X) =
      ( def loop (i:Int) : (List X) =
          if (== i n) then nil else (cons (f i) (loop (inc i)))
        (loop 0)
      )

    inline def case (#X #R l:(List X) n:/[/R] c:/[X (List X) /R]) : R =
      if (null l) then
        (n)
      else
        (c (car l) (cdr l))

    def nth (#X l:(List X) n:Int) : X =
      if (null l) then
        (error "Bad index in list.nth: ")
      else if (== n 0) then
        (car l)
      else
        (nth (cdr l) (dec n))

    def size (#X l:(List X)) : Int =
      ( def loop (x:Int l:(List X)):Int =
          if (null l) then x else (loop (inc x) (cdr l))
        (loop 0 l)
      )

    def apply (#X l:(List X) f:/[X /[]]) : [] =
      if (not (null l)) then
        ((f (car l)); (apply (cdr l) f))
      else
        []

    def revApply (#X l:(List X) f:/[X /[]]) : [] =
      if (not (null l)) then
        ((revApply (cdr l) f); (f (car l)))
      else
```

```
      []

  def itApply (#X l:(List X) f:/[Int X /[]]) : [] =
    ( def loop (x:Int l:(List X)) : [] =
        if (not (null l)) then
          ( (f x (car l));
            (loop (inc x) (cdr l))
          )
       else
         []
      (loop 0 l)
    )

  def revItApply (#X l:(List X) f:/[Int X /[]]) : [] =
    ( def loop (index:Int l:(List X)) : [] =
        if (not (null l)) then
          ( (loop (inc index) (cdr l));
            (f index (car l))
          )
       else
         []
      (loop 0 l)
    )

  def fold (#X #R l:(List X) init:R f:/[X R /R]) : R =
    if (null l) then init else (fold (cdr l) (f (car l) init) f)

  def revFold (#X #R l:(List X) init:R f:/[X R /R]) : R =
    if (null l) then init else (f (car l) (revFold (cdr l) init f))

  def itFold (#X #R l:(List X) init:R f:/[Int X R /R]) : R =
    ( def loop (x:Int l:(List X) v:R) : R =
        if (null l) then v
        else (loop (inc x) (cdr l) (f x (car l) v))
      (loop 0 l init)
    )

  def revItFold (#X #R l:(List X) init:R f:/[Int X R /R]) : R =
    ( def loop (x:Int l:(List X) v:R) : R =
        if (null l) then
          v
        else
          (f x (car l) (loop (inc x) (cdr l) v))
      (loop 0 l init)
    )

  def map (#X #R l:(List X) f:/[X /R]) : (List R) =
    if (null l) then
      nil
    else
      (cons (f (car l)) (map (cdr l) f))

  def revMap (#X #R l:(List X) f:/[X /R]) : (List R) =
    if (null l) then
      nil
    else
      (val r = (revMap (cdr l) f) (cons (f (car l)) r))
```

```
def itMap (#X #R l:(List X) f:/[Int X /R]) : (List R) =
  ( def loop (l:(List X) index:Int) : (List R) =
      if (null l) then
        nil
      else
        (cons (f index (car l))
              (loop (cdr l) (inc index))
        )
    (loop l 0)
  )

def revItMap (#X #R l:(List X) f:/[Int X /R]) : (List R) =
  ( def loop (l:(List X) index:Int) : (List R) =
      if (null l) then
        nil
      else
        ( val r = (loop (cdr l) (inc index))
          (cons (f index (car l))
                r
        ) )
    (loop l 0)
  )

def filter (#X l:(List X) f:/[X /Bool]) : (List X) =
  ( case #X #(List X) l
      \() = nil
      \(hd tl) = if (f hd) then (cons hd (filter tl f)) else (filter tl f)
  )

def rev (#X l:(List X)) : (List X) =
  (fold #X #(List X) l nil \(hd tl) = (cons hd tl))

def append (#X l1:(List X) l2:(List X)) : (List X) =
  (revFold #X #(List X) l1 l2 \(hd tl) = (cons hd tl))

def revAppend (#X l1:(List X) l2:(List X)) : (List X) =
  (fold #X #(List X) l1 l2 \(hd tl) = (cons hd tl))

def hash (#X l:(List X) f:/[X /Int]) : Int =
  (fold #X #Int l 0 \(x h) = (+ (* (f x) 19) h))

def cmpList (#X l1:(List X) l2:(List X) f:/[X X /Cmp]) : Cmp =
  if (null l1) then
    if (null l2) then cmp.EQ else cmp.LT
  else if (null l2) then
    cmp.GT
  else
    ( val c = (f (car l1) (car l2))
      if (cmp.eq c) then (cmpList (cdr l1) (cdr l2) f)
      else c
    )

def sort (#X list:(List X) f:/[X X /Cmp] removeDuplicates:Bool) : (List X) =
  ( def split (l1:(List X) l2:(List X) n:Int) : [(List X) (List X)] =
      if (== n 0) then
        [(rev l1) l2]
      else
        (split (cons (car l2) l1) (cdr l2) (dec n))
```

```
      def merge (rest:(List X) l1:(List X) l2:(List X)) : (List X) =
        if (null l1) then
          (revAppend rest l2)
        else if (null l2) then
          (revAppend rest l1)
        else
          ( val h1 = (car l1)
            val h2 = (car l2)
            val c = (f h1 h2)
            if (&& (cmp.eq c) removeDuplicates) then
              (merge rest (cdr l1) l2)
            else if (cmp.le c) then
              (merge (cons h1 rest) (cdr l1) l2)
            else
              (merge (cons h2 rest) l1 (cdr l2))
          )
      def s (l:(List X) n:Int) : (List X) =
        if (<= n 1) then
          l
        else
          ( val middle = (div n 2)
            val [l1 l2] = (split nil l middle)
            (merge nil (s l1 middle) (s l2 (- n middle)))
          )
      (s list (size list))
    )

  def detect (#X l:(List X) f:/[X /Bool]) : Bool =
    (fold l false
      \(element:X partialResult:Bool):Bool =
        (|| partialResult (f element))
    )

  [ nil=nil cons=cons make=make tabulate=tabulate null=null car=car
    cdr=cdr case=case size=size
    nth=nth apply=apply revApply=revApply itApply=itApply
    revItApply=revItApply fold=fold revFold=revFold itFold=itFold
    revItFold=revItFold map=map revMap=revMap itMap=itMap revItMap=revItMap
    filter=filter rev=rev append=append revAppend=revAppend sort=sort
    hash=hash cmp=cmpList detect=detect
  ]
)

val nil  = list.nil
val cons = list.cons
val null = list.null
val car  = list.car
val cdr  = list.cdr
```

2.18 Misc: Miscellaneous Useful Functions

2.18.1 Tuple Operations

fst, snd and thd are functions for projecting the first, second and third components of a tuple. Note that record subtyping allows these functions to be applied to longer tuples. For example, the expression (snd [3 4 5]) is well-typed and evaluates to 4.

```
fst = /[#X [X] /X]
snd = /[#X #Y [X Y] /Y]
thd = /[#X #Y #Z [X Y Z] /Z]
```

2.18.2 Channel Input/Output

The (chan #X) is a wrapper around the core-language new operation. It creates a fresh channel of a give type. It makes this operation syntactically convenient in the functional context.

```
chan = /[#X /^X]
```

The expression (read c) returns a value which has been read from channel c.

```
read = /[#X ?X /X]
```

The expression (write c v) asynchronously writes the value v on c.

```
write = /[#X !X X /[]]
```

The expression (forward c r) repeatedly reads values from c and forwards them to the responsive channel r.

```
forward = /[#X ?X /X]
```

The expression (rchan r) creates and returns a responsive channel and forward values from it to ordinary channel r. It is used in those situations when we hold a value of !X type and we need its /X variant.

```
rchan = /[#X !X //X]
```

2.18.3 Discarding Results

(discard #X) returns a process abstraction that accepts a value of type X and throws it away. Useful for calling functions that expect result channels when the result is actually not needed. For example writing int.pr![5 (discard #[])] creates a process that prints 5 but does not do anything special when it finishes (the [] sent by the int.pr is thrown away).

```
discard = /[#X //X]
```

The expression (await v) evaluates expression v and then returns [].

```
await = /[#X X /[]]
```

2.18.4 Function Composition

(compose g f) composes functions g and f.

```
compose = /[#X #Y #Z /[Y /Z] /[X /Y] //[X /Z]]
```

(identity x) always returns x

```
identity = /[#X X /X]
```

2.18.5 Implementation

64 ⟨*Untrusted/Misc.pi* 64⟩≡

```
import "Trusted/Prim"

val misc: [
  fst = /[#X [X] /X]
  snd = /[#X #Y [X Y] /Y]
  thd = /[#X #Y #Z [X Y Z] /Z]
  chan = /[#X /^X]
  read = /[#X ?X /X]
  write = /[#X !X X /[]]
  forward = /[#X ?X /X]
  rchan = /[#X !X //X]
  discard = /[#X //X]
  await = /[#X X /[]]
  compose = /[#X #Y #Z /[Y /Z] /[X /Y] //[X /Z]]
  identity = /[#X X /X]
] = (
  inline def fst (#X [x:X]) : X = x
  inline def snd (#X #Y [_:X y:Y]) : Y = y
  inline def thd (#X #Y #Z [_:X _:Y z:Z]) : Z = z
  inline def chan (#X) : ^X = (new x:^X x)
  inline def read [#X c:?X res:/X] = c?v = res!v
  def forward [#X c:?X to:/X] = c?v = (to!v | forward![c to])
  inline def write [#X c:!X v:X res:/[]] = ( c!v | res![] )

  def rchan (#X r:!X) : /X = \v:X = r!v
  inline def discard (#X): /X = \_ = ()

  inline def await (#X _:X) : [] = []
  inline def compose (#X #Y #Z g:/[Y /Z] f:/[X /Y]) : /[X /Z] = \(x) = (g (f x))
  inline def identity (#X x:X) : X = x

  [ fst=fst snd=snd thd=thd chan=chan read=read write=write forward=forward
    rchan=rchan discard=discard await=await compose=compose
    identity=identity
  ]
)

val fst = misc.fst
val snd = misc.snd
val thd = misc.thd
val chan = misc.chan
val read = misc.read
val write = misc.write
val forward = misc.forward
val rchan = misc.rchan
val discard = misc.discard
val await = misc.await
val compose = misc.compose
val identity = misc.identity
```

2.19 Queue

The original standard Pict library [8] also contains definition of a `Queue` data type. We have decided to completely change the definition of this datatype to make it more effective. Our queues are realized as bidirectional linked list of queue items. This makes it possible to add/remove elements from the front/tail in constant time. The original implementation is correct too but not as effective as it could be.

`(Queue X)` represents a bidirectionally linked list whose elements can be added and removed at either end. The first item is called *head*. The last item is called *tail*. The `lock` is used internally for serializing operations with the queue.

2.19.1 Creation

`(copy q)` returns a copy of a queue `q`. Queues are mutable and thus it has sense to copy them.

```
copy = /[#X (Queue X) /(Queue X)]
```

Create and return an empty queue.

```
empty = /[#X /(Queue X)]
```

`(make n x)` creates a queue of size `n`, with each element containing `x`.

```
make = /[#X Int X /(Queue X)]
```

`(tabulate n f)` creates a queue of size `n` such that the element with index `i` is initialized to `(f i)`.

```
tabulate = /[#X Int /[Int /X] /(Queue X)]
```

2.19.2 Interrogation

`(isEmpty q)` tests whether `q` is empty.

```
isEmpty = /[#X (Queue X) /Bool]
```

`(size q)` returns the size of `q`.

```
size = /[#X (Queue X) /Int]
```

`(head q)` and `(tail q)` return the head or tail of `q` respectively. Both will report an error if a given queue is empty.

```
head = /[#X (Queue X) /X]
tail = /[#X (Queue X) /X]
```

`(detect q f)` applies `f` to each element of `q`. The whole function returns `true` if function `f` returns `true` at least for one element.

```
detect = /[#X (Queue X) /[X /Bool] /Bool]
```

2.19.3 Modification

Insert an element at the head or tail of a queue.

```
insertHd = /[#X (Queue X) X /[]]
insertTl = /[#X (Queue X) X /[]]
```

Remove an element from the hear or tail of a queue. Generates an error if the queue is empty.

```
removeHd = /[#X (Queue X) /[]]
removeTl = /[#X (Queue X) /[]]
```

2.19.4 Iteration

`(apply q f)` applies `f` to each element of `q` (sequentially and in order). `(revApply f q)` behaves similarly, except that it traverses `q` in reverse order.

```
apply = /[#X (Queue X) /[X /[]] /[]]
revApply = /[#X (Queue X) /[X /[]] /[]]
```

`(itApply q f)` applies `f` to each element of `q` (sequentially and in order), passing `f` the index of each element. `(revItApply f q)` behaves similarly, except that it traverses `q` in reverse order.

```
itApply = /[#X (Queue X) /[Int X /[]] /[]]
revItApply = /[#X (Queue X) /[Int X /[]] /[]]
```

`(fold q init f)` applies `f` to each element of `q` (sequentially and in order), passing `f` an accumulated result of type `R`. The initial accumulated result is `init`. `(revFold q init f)` behaves similarly, except that it traverses `q` in reverse order.

```
fold = /[#X #R (Queue X) R /[X R /R] /R]
revFold = /[#X #R (Queue X) R /[X R /R] /R]
```

`(itFold q init f)` applies `f` to each element of `q` (sequentially and in order), passing `f` the index of each element, and an accumulated result of type `R`. The initial accumulated result is `init`. `(revItFold q init f)` behaves similarly, except that it traverses `q` in reverse order.

```
itFold = /[#X #R (Queue X) init /[Int X R /R] /R]
revItFold = /[#X #R (Queue X) init /[Int X R /R] /R]
```

2.19.5 Implementation

`(QueueItem X)` below represents a single item of the queue. Each item knows about the `value` it holds, the `previous` item and the `next` item.

66 ⟨*Untrusted/Queue.pi* 66⟩≡

```
import "Untrusted/Cell"
import "Untrusted/Int"

val [
  #Queue : (Pos Type -> Type)

  queue: [
    empty = /[#X /(Queue X)]
    make = /[#X Int X /(Queue X)]
    tabulate = /[#X Int /[Int /X] /(Queue X)]
    isEmpty = /[#X (Queue X) /Bool]
    size = /[#X (Queue X) /Int]
    head = /[#X (Queue X) /X]
    tail = /[#X (Queue X) /X]
    insertHd = /[#X (Queue X) X /[]]
    insertTl = /[#X (Queue X) X /[]]
    removeHd = /[#X (Queue X) /[]]
    removeTl = /[#X (Queue X) /[]]
```

```
      apply = /[#X (Queue X) /[X /[]] /[]]
      revApply = /[#X (Queue X) /[X /[]] /[]]
      itApply = /[#X (Queue X) /[Int X /[]] /[]]
      revItApply = /[#X (Queue X) /[Int X /[]] /[]]
      fold = /[#X #R (Queue X) R /[X R /R] /R]
      revFold = /[#X #R (Queue X) R /[X R /R] /R]
      itFold = /[#X #R (Queue X) R /[Int X R /R] /R]
      revItFold = /[#X #R (Queue X) R /[Int X R /R] /R]
      copy = /[#X (Queue X) /(Queue X)]
      detect = /[#X (Queue X) /[X /Bool] /Bool]
    ]
  ] = (
    type (QueueItem X) =
      (rec QI =
        [value=(Cell X) prev=(Cell QI) next=(Cell QI)]
      )

    type (Queue X) = [head=(Cell (QueueItem X)) tail=(Cell (QueueItem X))]

    def empty (#X) : (Queue X) =
      [ head=(cell.empty #(QueueItem X))
        tail=(cell.empty #(QueueItem X))
      ]

    and make (#X n:Int x:X) : (Queue X) =
      ( val queue = (empty #X)
        (int.for 1 n
                 \(i) = (insertTl queue x)
        );
        queue
      )

    and tabulate (#X n:Int f:/[Int /X]) : (Queue X) =
      ( val queue = (empty #X)
        (int.for 0 (dec n)
                 \(i) = (insertTl queue (f i))
        );
        queue
      )

    and isEmpty (#X queue:(Queue X)) : Bool =
      (cell.isEmpty queue.head)

    and size (#X queue:(Queue X)) : Int =
      ( def size (#X item:(QueueItem X)):Int =
          ( val (rec unfoldedItem) = item
            val next = unfoldedItem.next
            if (cell.isEmpty next) then
              1
            else
              (inc (size (get next)))
          )
        if (cell.isEmpty queue.head) then 0
        else (size (get queue.head))
      )

    and head (#X queue:(Queue X)) : X =
      if (isEmpty queue) then
```

```
      (error "Expected a non-empty queue in queue.head")
    else
      ( val (rec head) = (get queue.head)
        (get head.value)
      )

  and tail (#X queue:(Queue X)) : X =
    if (isEmpty queue) then
      (error "Expected a non-empty queue in queue.tail")
    else
      ( val (rec tail) = (get queue.tail)
        (get tail.value)
      )

  and insertHd (#X queue:(Queue X) x:X) : [] =
    if (isEmpty queue) then
      ( val newHead:(QueueItem X) =
          (rec
            [ value=(cell.make x)
              prev=(cell.empty #(QueueItem X))
              next=(cell.empty #(QueueItem X))
            ]
          )
        (put queue.head newHead);
        (put queue.tail newHead)
      )
    else
      ( val newHead:(QueueItem X) =
          (rec
            [ value=(cell.make x)
              prev=(cell.empty #(QueueItem X))
              next=(cell.make (get queue.head))
            ]
          )
        val (rec oldHead) = (get queue.head)
        (put oldHead.prev newHead);
        (put queue.head newHead)
      )

  and insertTl (#X queue:(Queue X) x:X) : [] =
    if (isEmpty queue) then (
      val newTail:(QueueItem X) =
        ( rec
            [ value=(cell.make x)
              prev=(cell.empty #(QueueItem X))
              next=(cell.empty #(QueueItem X))
            ]
        )
        (put queue.head newTail);
        (put queue.tail newTail)
      )
    else
      ( val newTail:(QueueItem X) =
          ( rec
            [ value=(cell.make x)
              prev=(cell.make (get queue.tail))
              next=(cell.empty #(QueueItem X))
            ]
```

```
        )
      val (rec oldTail) = (get queue.tail)
      (put oldTail.next newTail);
      (put queue.tail newTail)
    )

and removeHd (#X queue:(Queue X)) : [] =
  if (isEmpty queue) then
    (error "Empty queue in queue.removeHd.")
  else
    ( val (rec oldHead) = (get queue.head)
      if (cell.isEmpty oldHead.next) then
        ( (cell.clear queue.head);
          (cell.clear queue.tail)
          )
      else
        ( val newHead = (get oldHead.next)
          val (rec unfoldedNewHead) = newHead
          (cell.clear unfoldedNewHead.prev);
          (cell.clear oldHead.next);
          (put queue.head newHead)
        )
    )

and removeTl (#X queue:(Queue X)) : [] =
  if (isEmpty queue) then (error "Empty queue in queue.removeTl.")
  else
    ( val (rec oldTail) = (get queue.tail)
      if (cell.isEmpty oldTail.prev) then
        ( (cell.clear queue.head);
          (cell.clear queue.tail)
        )
      else
        ( val newTail = (get oldTail.prev)
          val (rec unfoldedNewTail) = newTail
          (cell.clear unfoldedNewTail.next);
          (cell.clear oldTail.prev);
          (put queue.tail newTail)
        )
    )

and apply (#X queue:(Queue X) f:/[X /[]]) : [] =
  ( def loop (c:(Cell (QueueItem X)) f:/[X /[]]) : [] =
      if (cell.isEmpty c) then
        []
      else
        ( val queueItem = (get c)
          val (rec unfoldedQueueItem) = queueItem
          (f (get unfoldedQueueItem.value));
          (loop unfoldedQueueItem.next f)
        )
    (loop queue.head f)
  )

and revApply (#X queue:(Queue X) f:/[X /[]]) : [] =
  ( def loop (c:(Cell (QueueItem X)) f:/[X /[]]) : [] =
      if (cell.isEmpty c) then
        []
```

```
        else
          ( val queueItem = (get c)
            val (rec unfoldedQueueItem) = queueItem
            (f (get unfoldedQueueItem.value));
            (loop unfoldedQueueItem.prev f)
          )
      (loop queue.tail f)
    )

  and itApply (#X queue:(Queue X) f:/[Int X /[]]) : [] =
    ( def loop (i:Int c:(Cell (QueueItem X)) f:/[Int X /[]]) : [] =
        if (cell.isEmpty c) then
          []
        else
          ( val queueItem = (get c)
            val (rec unfoldedQueueItem) = queueItem
            (f i (get unfoldedQueueItem.value));
            (loop (inc i) unfoldedQueueItem.next f)
          )
      (loop 0 queue.head f)
    )

  and revItApply (#X queue:(Queue X) f:/[Int X /[]]) : [] =
    ( def loop (i:Int c:(Cell (QueueItem X)) f:/[Int X /[]]) : [] =
        if (cell.isEmpty c) then
          []
        else
          ( val queueItem = (get c)
            val (rec unfoldedQueueItem) = queueItem
            (f i (get unfoldedQueueItem.value));
            (loop (dec i) unfoldedQueueItem.prev f)
          )
      (loop (dec (size queue)) queue.tail f)
    )

  and fold (#X #R queue:(Queue X) init:R f:/[X R /R]) : R =
    ( def loop (c:(Cell (QueueItem X)) init:R f:/[X R /R]) : R =
        if (cell.isEmpty c) then init
        else
          ( val queueItem = (get c)
            val (rec unfoldedQueueItem) = queueItem
            (loop unfoldedQueueItem.next
                  (f (get unfoldedQueueItem.value) init)
                  f
          ) )
      (loop queue.head init f)
    )

  and revFold (#X #R queue:(Queue X) init:R f:/[X R /R]) : R =
    ( def loop (c:(Cell (QueueItem X)) init:R f:/[X R /R]) : R =
        if (cell.isEmpty c) then
          init
        else
          ( val queueItem = (get c)
            val (rec unfoldedQueueItem) = queueItem
            (loop unfoldedQueueItem.prev
                  (f (get unfoldedQueueItem.value) init)
                  f
```

```
        ) )
      (loop queue.tail init f)
    )

  and itFold (#X #R queue:(Queue X) init:R f:/[Int X R /R]) : R =
    ( def loop (i:Int c:(Cell (QueueItem X)) init:R f:/[Int X R /R]):R =
        if (cell.isEmpty c) then init
        else
          ( val queueItem = (get c)
            val (rec unfoldedQueueItem) = queueItem
            (loop (inc i)
                  unfoldedQueueItem.next
                  (f i (get unfoldedQueueItem.value) init)
                  f
          ) )
      (loop 0 queue.head init f)
    )

  and revItFold (#X #R queue:(Queue X) init:R f:/[Int X R /R]) : R =
    ( def loop (i:Int c:(Cell (QueueItem X)) init:R f:/[Int X R /R]) : R =
        if (cell.isEmpty c) then init
        else
          ( val queueItem = (get c)
            val (rec unfoldedQueueItem) = queueItem
            (loop (dec i)
                  unfoldedQueueItem.prev
                  (f i (get unfoldedQueueItem.value) init)
                  f
          ) )
      (loop (dec (size queue)) queue.tail init f)
    )

  and copy (#X q:(Queue X)) : (Queue X) =
    ( val result = (empty #X)
      (apply q
        \(element:X):[] = (insertTl result element)
      );
      result
    )

  and detect (#X q:(Queue X) f:/[X /Bool]) : Bool =
    (fold q false
      \(element:X partialResult:Bool):Bool =
        (|| partialResult (f element))
    )

  [ #Queue
    [ empty=empty make=make tabulate=tabulate isEmpty=isEmpty size=size
      head=head tail=tail insertHd=insertHd insertTl=insertTl removeHd=removeHd
      removeTl=removeTl apply=apply revApply=revApply itApply=itApply
      revItApply=revItApply fold=fold revFold=revFold itFold=itFold
      revItFold=revItFold copy=copy detect=detect
    ]
  ]
)
```

2.20 Random Numbers

The random number generator used in this library is the PMMMLCG (Prime Modulus M Multiplicative Linear Congruential Generator), a modified version of the random number generator proposed by Park and Miller in "Random Number Generators: Good Ones Are Hard to Find", CACM October 1988, Vol 31, No. 10. It includes modifications proposed by Park to provide better statistical properties (i.e. more 'random' — less correlation between sets of generated numbers). It was developed by John Burton (jcburt@cs.wm.edu) of G & A Technical Software, Inc 28 Research Drive Hampton, Va. 23666.

The generator is of the form $x = (x * A)\%M$ and has a period of $2^{30} - 1$, with numbers generated in the range of $0 < x < M$. The generator can run on any machine with a 32-bit integer, without overflow. The choice of A and M can radically modify the properties of the generator.

2.20.1 Operations

`(randomize x)` sets the seed in the random number generator to `x`.

```
randomize = /[Int /[]]
```

The upper limit on numbers produces by `random`.

```
max = Int
```

The bottom limit on numbers produces by `random`.

```
min = Int
```

Returns a random integer `r` such that `min` < `r` < `max`.

```
random = /[/Int]
```

A special case of the random number generator that returns a random boolean.

```
randomCoin = /[/Bool]
```

2.20.2 Implementation

72 ⟨*Untrusted/Random.pi* 72⟩≡

```
import "Untrusted/Cell"
import "Untrusted/Int"

val random: [
  randomize = /[Int /[]]
  min = Int
  max = Int
  random = /[/Int]
  randomCoin = /[/Bool]
] = (
  val randomSeed = (cell.make 123456789)
  val max = int.upperBound

  inline def randomize (x:Int) : [] =
    (cell.put randomSeed x)

  def random () : Int =
    ( val a = 48271
      val hi = (div (cell.get randomSeed) (div max a))
```

```
        val lo = (mod (cell.get randomSeed) (div max a))
        val test = (- (* a lo) (* (mod max a) hi))
        if (<< 0 test) then
          (cell.put randomSeed test)
        else
          (cell.put randomSeed (+ test max));
        (cell.get randomSeed)
      )

    inline def randomCoin () : Bool =
      (<< (random) (div max 2))

    [ randomize=randomize min=1 max=max random=random
      randomCoin=randomCoin
    ]
  )
```

2.21 Revoker

Revokers are somewhat similar to lids (see Section 2.16) in that they can also be used for creation of revocable capabilities. The revokers however create such capabilities which—once revoked—cannot be reenabled again.

If we have a negative `target` capability and we want to create a revocable variant of this capability, then this can be accomplished with the `(makeNeg target)` function. It will return the `[proxy revoke]` couple. The `proxy` channel can be given to the partner. The `proxy` channel will forward all the values passing though it to the `target` channel until the `revoke` function is called. If it is called, no one will be able to receive values sent though the `proxy` channel.

```
makeNeg = /[#X !X /[!X /[/[]]]]
```

If we have a positive `source` capability and we want to create a revocable variant of this capability, then this can be accomplished with the `(makePos source)` function. It will return the `[proxy revoke]` couple. The `proxy` channel can be given to the partner. All the values passing though the `source` channel will be forwarded into the `proxy` channel until the `revoke` function is called. If it is called, then anyone reading from the proxy channel will be blocked forever because no further message ever comes though it.

```
makePos = /[#X ^X /[?X /[/[]]]]
```

2.21.1 Implementation

74a ⟨*Untrusted/Revoker.pi* 74a⟩≡

```
import "Untrusted/Choice"

val revoker: [
  makeNeg = /[#X !X /[!X /[/[]]]]
  makePos = /[#X ^X /[?X /[/[]]]]
] = (
  ⟨definitions of revoker constructors 74b⟩
)
```

74b ⟨*definitions of revoker constructors* 74b⟩≡ (74a) 74c ▷

```
def makeNeg (#X target:!X) : [!X /[/[]]] =
  ( new proxy:^X
    new revoke:^[/[]]
    def loop [] =
      choose!( $ (=> proxy \v:X = ( target!v | loop![] ))
                 (=> revoke \[r:/[]] = r![]) )
    run loop![]
    [proxy (rchan revoke)]
  )
```

Unfortunately, the `source` capability must be both, readable and writable because of the way how our choice operator (see Section 2.9) is implemented.

74c ⟨*definitions of revoker constructors* 74b⟩+≡ (74a) ◁ 74b

```
def makePos (#X source:^X) : [?X /[/[]]] =
  ( new proxy:^X
    new revoke:^[/[]]
    def loop [] =
      choose!( $ (=> source \v:X = ())
                 (=> revoke \[r:/[]] = r![]) )
    run loop![]
    [proxy (rchan revoke)]
  )

[makeNeg=makeNeg makePos=makePos]
```

2.22 `Sem`: Semaphores

This module implements semaphores that can be used for usual synchronization purposes. This lock interface is friendly to “functional programming syntax”.

Create a semaphore with a zero value.

```
zero = /[/Semaphore]
```

Create a semaphore that is initialized to one.

```
one = /[/Semaphore]
```

Create a semaphore that is initialized to a given value.

```
make = /[Int /Semaphore]
```

The `(wait s)` function blocks the caller until a give semaphore can be acquired and its value can be decreased.

```
wait = /[Semaphore /[]]
```

The `(signal s)` function increases the value of a given semaphore.

```
signal = /[Semaphore /[]]
```

2.22.1 Implementation

75 ⟨*Untrusted/Semaphore.pi* 75⟩≡

```
import "Untrusted/Int"

val [
  #Semaphore

  semaphore:[
    zero = /[/Semaphore]
    one = /[/Semaphore]
    make = /[Int /Semaphore]
    wait = /[Semaphore /[]]
    signal = /[Semaphore /[]]
  ]
] = (
  type Semaphore = ^[]

  def zero [r:/Semaphore] =
    ( new lock: ^[]
      r!lock
    )

  def one [r:/Semaphore] =
    ( new lock: ^[]
      ( r!lock
      | lock![]
      )
    )

  def make [i:Int r:/Semaphore] =
    ( new lock: ^[]
      ( r!lock
```

```
      | (discard #[])!(int.for 1 i \(j) = (write lock []))
    ) )

  def wait [s:Semaphore r:/[]] =
    s?_ = r![]

  def signal [s:Semaphore r:/[]] =
    ( s![]
    | r![]
    )

  [ #Semaphore
    [zero=zero one=one make=make wait=wait signal=signal]
  ]
)
```

2.23 Signal: UNIX Signals

77 ⟨*Untrusted/Signals.pi* 77⟩≡
```
type SignalHandler = ![]

val [#Signal < Int] = [#Int]

type SignalsAPI =
  [ SIGHUP    = Signal
    SIGINT    = Signal
    SIGQUIT   = Signal
    SIGILL    = Signal
    SIGTRAP   = Signal
    SIGABRT   = Signal
    SIGBUS    = Signal
    SIGFPE    = Signal
    SIGUSR1   = Signal
    SIGSEGV   = Signal
    SIGUSR2   = Signal
    SIGPIPE   = Signal
    SIGALRM   = Signal
    SIGTERM   = Signal
    SIGSTKFLT = Signal
    SIGCHLD   = Signal
    SIGCONT   = Signal
    SIGTSTP   = Signal
    SIGTTIN   = Signal
    SIGTTOU   = Signal
    SIGURG    = Signal
    SIGXCPU   = Signal
    SIGXFSZ   = Signal
    SIGVTALRM = Signal
    SIGPROF   = Signal
    SIGWINCH  = Signal
    SIGIO     = Signal
    register = /[Signal SignalHandler /[]]
    registerPersistent = /[Signal SignalHandler /[]]
  ]
```

2.24 Socket

Note that socket is implemented as Int but it is fresh type distinct from Int. Neigher Socket $<$ Int nor Int $<$ Socket.

78 ⟨*Untrusted/Socket.pi* 78⟩≡

```
import "Untrusted/IPAddress"

val [#Socket] = [#Int]

type SocketAPI =
  [ openRaw = /[/Socket]
    sendTo = /[Socket IPAddress String /[]]
    recv = /[Socket String /Int]
  ]
```

2.25 String

2.25.1 Types

The type String is built in and cannot be redefined.

2.25.2 Creation

(copy s) returns a copy of a string s. Strings are mutable and thus it has sense to copy them.

```
copy = /[String /String]
```

(make c n) creates a string of size n, containing the character c.

```
make = /[Char Int /String]
```

(tabulate n f) creates a string of size n, such that the character with index i is initialised to (f i).

```
tabulate = /[Int /[Int /Char] /String]
```

(+$ s t) concatenates s and t.

```
+$ = /[String String /String]
```

2.25.3 Interrogation

(size s) returns the number of characters in s.

```
size = /[String /Int]
```

(isEmpty s) determines whether a given string is empty. If yes, it returns true. If not, it returns false.

```
isEmpty = /[String /Bool]
```

(nth s n) returns the n-th character of s. If n is not in the range $0 \leq n <$ (size s) then we generate a runtime error.

```
nth = /[String Int /Char]
```

(update s n ch sets the n-th character in string s to ch. The procedure will fail if you try to change characters that are not part of a given string. Due to the representation of literal strings (they are located in a constant segment), it is not possible to modify strings that were expressed as literals.

```
update = /[String Int Char /[]]
```

Extract the substring of s, starting at start and of length len. It must be the case that len $\geq$ 0, start $\geq$ 0 and start + len $\leq$ (size s).

```
sub = /[String Int Int /String]
```

(tokens s ch) returns tokens of string s separated by a given character ch.

```
tokens = /[String Char /(List String)]
```

(detect s f) applies f to each character of s. The whole function returns true if function f returns true at least for one character.

```
detect = /[String /[Char /Bool] /Bool]
```

`(indexOf s f)` applies `f` to each character of `s`. The whole function returns index of the first character (starting from zero) for which function `f` returns `true`. If a given function `f` does not return true for any character of a given string `s`, this function returns `-1`.

```
indexOf = /[String /[Char /Bool] /Int]
```

2.25.4 Modification

`(update s n c)` changes `n`'th character of string `s` to `c`.

```
update = /[String Int Char /[]]
```

2.25.5 Comparison

Returns a single value indicating the ordering of `s` and `t` (cf. Section **??**).

```
cmp = /[String String /Cmp]
```

`s` equal to `t`, and `s` not equal to `t`.

```
==$ = /[String String /Bool]
<>$ = /[String String /Bool]
```

`(prefix s t)` returns `true` whenever `s` occurs as a prefix of `t`.

```
prefix = /[String String /Bool]
```

`(indexIn c s h)` returns the index of the first occurrence of `c` in `s`. If `c` does not occur in `s`, run-time error occurs.

```
indexIn = /[Char String /Int]
```

`(occursIn c s)` returns `true` if the character `c` occurs in the string `s`.

```
occursIn = /[Char String /Bool]
```

2.25.6 Conversion

`(hash s)` returns a hash value for `s`.

```
hash = /[String /Int]
```

`(fromList l)` converts the list of characters `l` to a string.

```
fromList = /[(List Char) /String]
```

`($$ i)` converts a given integer to the corresponding string representation.

```
$$ = /[Int /String]
```

`(toInt s)` converts a string `s` to an integer. If `s` is not a valid string (a non-empty sequence of digits, possible prefixed by a negation symbol `-`) then `toInt` it reports an error.

```
toInt = /[String /Int]
```

2.25.7 Iteration

`(apply s f)` applies `f` to each element of `s` (sequentially and in order). `revApply` behaves just like `apply`, except that it traverses `s` in reverse order.

```
apply = /[String /[Char /[]] /[]]
```

`(itApply s f)` applies `f` to each element of `s` (sequentially and in order), passing `f` the index of each element. `revItApply` behaves just the same as `itApply`, except that it traverses `s` in reverse order.

```
itApply = /[#X String /[Int Char /[]] /[]]
revItApply = /[#X String /[Int Char /[]] /[]]
```

`(fold s init f)` applies `f` to each character of `s` (sequentially and in order), passing `f` an accumulated result of type `R`. The initial accumulated result is `init`. `revFold` behaves just like `fold`, except that it traverses `s` in reverse order.

```
fold = /[#R String R /[Char R /R] /R]
revFold = /[#R String R /[Char R /R] /R]
```

`(itFold s i f)` applies `f` to each element of `s` (sequentially and in order), passing `f` the index of each element, and an accumulated result of type `R`. The initial accumulated result is `i`. `revItFold` behaves just like `itFold`, except that it traverses `s` in reverse order.

```
itFold = /[#R String R /[Int Char R /R] /R]
itRevFold = /[#R String R /[Int Char R /R] /R]
```

2.25.8 Implementation

81 ⟨*Untrusted/String.pi* 81⟩≡

```
import "Untrusted/Char"
import "Untrusted/List"

val string : [
  copy = /[String /String]
  make = /[Char Int /String]
  tabulate = /[Int /[Int /Char] /String]
  +$ = /[String String /String]
  size = /[String /Int]
  nth = /[String Int /Char]
  update = /[String Int Char /[]]
  sub = /[String Int Int /String]
  cmp = /[String String /Cmp]
  ==$ = /[String String /Bool]
  <>$ = /[String String /Bool]
  hash = /[String /Int]
  fromList = /[(List Char) /String]
  $$ = /[Int /String]
  fromCString = /[CString /String]
  fold = /[#R String R /[Char R /R] /R]
  revFold = /[#R String R /[Char R /R] /R]
  itFold = /[#R String R /[Int Char R /R] /R]
  itRevFold = /[#R String R /[Int Char R /R] /R]
  apply = /[String /[Char /[]] /[]]
  revApply = /[String /[Char /[]] /[]]
  itApply = /[String /[Int Char /[]] /[]]
  revItApply = /[String /[Int Char /[]] /[]]
  prefix = /[String String /Bool]
```

```
    indexIn = /[Char String /Int]
    occursIn = /[Char String /Bool]
    toInt = /[String /Int]
    detect = /[String /[Char /Bool] /Bool]
    indexOf = /[String /[Char /Bool] /Int]
    tokens = /[String Char /(List String)]
    isEmpty = /[String /Bool]
  ] = (
    val shorten = prim.stringShorten
    val make = prim.stringMake
    val nth = prim.stringNth
    val update = prim.stringSetNth
    val fromCString = prim.stringFromCString
    val size = prim.stringSize

    inline def stringCmp (s1:String s2:String) : Cmp =
      ( val sz1 = (size s1)  val sz2 = (size s2)
        if (<< sz1 sz2) then
          cmp.LT
        else if (>> sz1 sz2) then
          cmp.GT
        else
          ( def loop (x:Int maxIndex:Int) : Cmp =
              if (<< maxIndex x) then
                cmp.EQ
              else if (<< (nth s1 x) (nth s2 x)) then
                cmp.LT
              else if (>> (nth s1 x) (nth s2 x)) then
                cmp.GT
              else if (== (nth s1 x) (nth s2 x)) then
                cmp.EQ
              else
                (loop (inc x) maxIndex)
            (loop 0 sz1)
          )
      )

    inline def <>$ (s:String t:String) : Bool = (cmp.ne (stringCmp s t))
    inline def >>$ (s:String t:String) : Bool = (cmp.gt (stringCmp s t))
    inline def <<$ (s:String t:String) : Bool = (cmp.lt (stringCmp s t))
    inline def >=$ (s:String t:String) : Bool = (cmp.ge (stringCmp s t))
    inline def <=$ (s:String t:String) : Bool = (cmp.le (stringCmp s t))

    def tabulate (n:Int f:/[Int /Char]) : String =
      ( val s = (prim.alloc (+ n 1))
        (int.for 0 (- n 1) \(x) = (update s x (f x)));
        (update s n '\000');
        s
      )

    def fromList (l:(List Char)): String =
      (val sz = (list.size l)
       val s = (prim.alloc (inc sz))
       (list.itApply l \(x:Int c:Char):[] = (update s x c));
       (update s sz '\000');
       s)

    def +$ (s1:String s2:String) : String =
```

```
    ( val sz1 = (size s1)
      val sz2 = (size s2)
      val r = (prim.alloc (+ > sz1 sz2 1))
      (int.for 0 (dec sz1) \(x) = (update r x (nth s1 x)));
      (int.for 0 (dec sz2) \(x) = (update r (+ sz1 x) (nth s2 x)));
      (update r (+ sz1 sz2) '\000');
      r
    )

  def $$ (i:Int) : String =
    if (== i 0) then
      "0"
    else if (<< i 0) then
      (+$ "-" ($$ (neg i)))
    else
      ( def loop (i:Int):String =
          if (== i 0) then
            ""
          else
            (+$ (loop (div i 10))
                (char.toString (char.fromInt (+ 48 (mod i 10))))
            )
        (loop i)
      )

  def fold (#R s:String init:R f:/[Char R /R]) : R =
    ( val limit = (size s)
      def loop (accum:R index:Int) : R =
        if (<< index limit) then
          (loop (f (nth s index) accum) (inc index))
        else
          accum
      (loop init 0)
    )

  def revFold (#R s:String init:R f:/[Char R /R]) : R =
    ( def loop (accum:R index:Int) : R =
        if (<< index 0) then
          accum
        else
          (loop (f (nth s index) accum) (dec index))
      (loop init (dec (size s)))
    )

  def itFold (#R s:String init:R f:/[Int Char R /R]) : R =
    ( val limit = (size s)
      def loop (accum:R index:Int) : R =
        if (<< index limit) then
          (loop (f index (nth s index) accum) (inc index))
        else
          accum
      (loop init 0)
    )

  def itRevFold (#R s:String init:R f:/[Int Char R /R]) : R =
    ( def loop (accum:R index:Int) : R =
        if (<< index 0) then
          accum
```

```
      else
        (loop (f index (nth s index) accum) (dec index))
    (loop init (dec (size s)))
  )

inline def hash (s:String) : Int =
  (fold s 0
        \(c:Char hash:Int):Int = (+ (* hash 19) c)
  )

def apply (s:String f:/[Char /[]]) : [] =
  ( val limit = (size s)
    def loop (index:Int) : [] =
      if (<< index limit) then
        ( (f (nth s index));
          (loop (inc index))
        )
      else
        []
    (loop 0)
  )

def revApply (s:String f:/[Char /[]]) : [] =
  ( def loop (index:Int) : [] =
      if (<< index 0) then
        []
      else
        ( (f (nth s index));
          (loop (dec index))
        )
    (loop (dec (size s)))
  )

def itApply (s:String f:/[Int Char /[]]) : [] =
  ( val limit = (size s)
    def loop (index:Int) : [] =
      if (<< index limit) then
        ( (f index (nth s index));
          (loop (inc index))
        )
      else
        []
    (loop 0)
  )

def revItApply (s:String f:/[Int Char /[]]) : [] =
  ( def loop (index:Int) : [] =
      if (<< index 0) then
        []
      else
        ( (f index (nth s index));
          (loop (dec index))
        )
    (loop (dec (size s)))
  )

inline def prefix (s:String t:String) : Bool =
  ( val limit = (size s)
```

```
    if (<= limit (size t)) then
      ( def loop (i:Int) : Bool =
          if (<< i 0) then
            true
          else if (== (nth s i)
                      (nth t i)) then
            (loop (dec i))
          else
            false
        (loop (dec limit))
      )
    else
      false
  )

inline def indexIn (c:Char s:String) : Int =
  ( def loop (i:Int) : Int =
      if (<< i 0) then
        (error "ERROR: Given character does not occur in a string (string.indexIn)")
      else if (== c (nth s i)) then
        i
      else
        (loop (dec i))

    (loop (dec (size s)))
  )

inline def occursIn (c:Char s:String) : Bool =
  ( def loop (i:Int) : Bool =
      if (<< i 0) then
        false
      else if (== c (nth s i)) then
        true
      else
        (loop (dec i))

    (loop (dec (size s)))
  )

inline def ==$ (s:String t:String) : Bool =
  ( val limit = (size s)
    val sizeT = (size t)
    if (== limit sizeT) then
      ( def loop (i:Int):Bool =
          if (<< i 0) then
            true
          else if (== (nth s i) (nth t i)) then
            (loop (dec i))
          else
            false
        (loop (dec limit))
      )
    else
      false
  )

inline def <>$ (s:String t:String) : Bool =
  (not (==$ s t))
```

```
def sub (s:String start:Int len:Int) : String =
  if (&& > (>= len 0)
           (>= start 0)
           (<= (+ start len)
           (size s))) then
    ( val sub = (prim.alloc (inc len))
      (int.for 0 (dec len) \(i) = (update sub i (nth s (+ start i))));
      (update sub len '\000');
      sub
    )
  else
    (error "string.sub: Integer out of range")

def toInt (s:String) : Int =
  if (== (size s) 0) then
    (error "string.toInt: empty string cannot be converted to integers")
  else if (== '-' (nth s 0)) then
    (neg (toInt (sub s 1 (dec (size s)))))
  else
    ( def loop (index:Int significance:Int) : Int =
        if (== index -1) then
          0
        else
          ( val ch = (nth s index)
            if (char.isDigit ch) then
              (+ (* (- ch '0') (int.power 10 significance))
                 (loop (dec index) (inc significance))
              )
            else
              (error "string.toInt: given string contains non-digit characters")
          )
      (loop (dec (size s)) 0)
    )

def copy (original:String) : String =
  (tabulate (size original)
            \(x:Int):Char = (nth original x)
  )

and detect (s:String f:/[Char /Bool]) : Bool =
  (fold s false
    \(element:Char partialResult:Bool) : Bool =
      (|| partialResult (f element))
  )

and indexOf (s:String f:/[Char /Bool]) : Int =
  (itRevFold s -1
    \(index:Int element:Char partialResult:Int) : Int =
      if (f element) then
        index
      else
        partialResult
  )

def tokens (s:String separator:Char) : (List String) =
  ( val stringSize = (size s)
    if (== stringSize 0) then
```

```
        {- the string is empty -}
        nil
      else
        ( {- string is not empty -}
          val separatorIndex = (indexOf s \(ch:Char):Bool = (== ch separator))
          val lastIndex = (dec stringSize)
          if (== -1 separatorIndex) then
            {- separator does not occur in string -}
            (cons s nil)
          else if (&& (== 0 separatorIndex) (== separatorIndex lastIndex)) then
            {- separator is the only character of string -}
            nil
          else if (&& (<> 0 separatorIndex) (== separatorIndex lastIndex)) then
            {- string is last but not the only character of string -}
            (cons (sub s 0 separatorIndex) nil)
          else if (&& (== 0 separatorIndex) (<> separatorIndex lastIndex)) then
            {- separator is the first but neither last and nor the only character in string -}
            (tokens (sub s 1 lastIndex) separator)
          else
            {- separator is neither first nor last nor the only character in string -}
            (cons (sub s 0 separatorIndex)
                  (tokens (sub s (inc separatorIndex) (- lastIndex separatorIndex)) separator)
    )    )  )

  inline def isEmpty (s:String) : Bool =
    (== 0 (size s))

  [ copy=copy make=make tabulate=tabulate +$ = +$ size=size nth=nth
    update=update sub=sub cmp=stringCmp ==$ = ==$ <>$ = <>$ hash=hash
    fromList=fromList $$ = $$ fromCString=fromCString
    fold=fold revFold=revFold itFold=itFold itRevFold=itRevFold
    apply=apply revApply=revApply itApply=itApply revItApply=revItApply
    prefix=prefix indexIn=indexIn occursIn=occursIn toInt=toInt
    detect=detect indexOf=indexOf tokens=tokens isEmpty=isEmpty
  ]
)

val +$ = string.+$
val ==$ = string.==$
val <>$ = string.<>$
val $$ = string.$$
```

Chapter 3

Trusted Modules

3.1 Alarm

This module provides each system with three interval timers, each decrementing in a distinct time domain. When any timer expires, a signal is sent to the process, and the timer (potentially) restarts.

With `(scheduleReal seconds microSeconds)` you can schedule occurrence of the `SIGALRM` after a specified number of seconds and microseconds of real time. By real time we mean time shown by wall clock.

```
scheduleReal = /[Int Int /[]]
```

With `(scheduleVirtual seconds microSeconds)` you can schedule occurrence of the `SIGVTALRM` after a specified number of seconds and microseconds of time during which our system is given CPU (in user space).

```
scheduleVirtual = /[Int Int /[]]
```

With `(scheduleProf seconds microSeconds)` you can schedule occurrence of the `SIGPROF` signal after a given number of seconds and microseconds during which our system is given CPU (in user space) and the operating system is acting on behalf of our system (in kernel space).

```
scheduleProf = /[Int Int /[]]
```

88 ⟨*Trusted/Alarm.pi* 88⟩≡

```
import "Untrusted/Alarm"

(ccode 0 I "##include <sys/time.h>");

val alarm : AlarmAPI = (

  def scheduleReal (seconds:Int microSeconds:Int) : [] =
    (ccode 0 E
      " struct itimerval it;
        int rvl;

        it.it_interval.tv_usec = I(#);
        it.it_interval.tv_sec = I(#);
        it.it_value.tv_usec = I(#);
        it.it_value.tv_sec = I(#);
        rvl = setitimer(ITIMER_REAL, &it, NULL);
        if (rvl == -1) {
          printf(\"The ‘setitimer’ call failed. Exiting.\\n\");
          exit(1);
        }
      "
```

```
      microSeconds seconds microSeconds seconds
    )

  def scheduleVirtual (seconds:Int microSeconds:Int) : [] =
    (ccode 0 E
      " struct itimerval it;
        int rvl;

        it.it_interval.tv_usec = I(#);
        it.it_interval.tv_sec = I(#);
        it.it_value.tv_usec = I(#);
        it.it_value.tv_sec = I(#);
        rvl = setitimer(ITIMER_VIRTUAL, &it, NULL);
        if (rvl == -1) {
          printf(\"The ‘setitimer’ call failed. Exiting.\\n\");
          exit(1);
        }
      "
      microSeconds seconds microSeconds seconds
    )

  def scheduleProf (seconds:Int microSeconds:Int) : [] =
    (ccode 0 E
      " struct itimerval it;
        int rvl;

        it.it_interval.tv_usec = I(#);
        it.it_interval.tv_sec = I(#);
        it.it_value.tv_usec = I(#);
        it.it_value.tv_sec = I(#);
        rvl = setitimer(ITIMER_PROF, &it, NULL);
        if (rvl == -1) {
          printf(\"The ‘setitimer’ call failed. Exiting.\\n\");
          exit(1);
        }
      "
      microSeconds seconds microSeconds seconds
    )

  [ scheduleReal=scheduleReal
    scheduleVirtual=scheduleVirtual
    scheduleProf=scheduleProf
  ]
)
```

3.2 Args: Command-Line Arguments

This is the "host" part of the command-line parsing functionality. The guest part is defined in Section 2.2.

The `argc` contains the number of command line parameters used to invoke this UNIX process.

```
argc: Int
```

`(argv n)` returns the `n`'th argument used to invoke the current UNIX process. If `n` < 0 or `argc` $\leq$ `n` then we generate a runtime error.

90 ⟨*Trusted/Args.pi* 90⟩≡

```
import "Untrusted/Bool"
import "Untrusted/Int"
import "Untrusted/String"
import "Untrusted/Args"

val argc : Int = (ccode 0 P "intInt(ArgC)")

def argv (n:Int) : String =
  if (|| (<< n 0) (<= argc n)) then
    (error "argv: index out of range\n")
  else
    (string.fromCString (ccode 0 P "(Val)ArgV[I(#)]" n))

val args = (makeArgs argc argv)
```

3.3 Fd: File-descriptor Operations

This is the "host" part of the Fd functionality. The guest part is defined in Section 2.13.

The primitives we need:

91a ⟨*Trusted/Fd.pi* 91a⟩≡ 91b▷

```
import "Untrusted/Fd"

val [#Fd stdin:Fd stdout:Fd stderr:Fd] = [#Int 0 1 2]

inline def fdwrite (fd:Fd str:String count:Int) : [] =
  (ccode 0 E "write(I(#), S(#), I(#));" fd str count)
```

The Fd type represents UNIX file descriptor. Concerning its definition notice that we did not use the "equality constraint" so Fd (where it is visible) is not an alias to Int. Neither we used "subtype constraint" so Fd (where it is visible) is not subtype of Int. Which means that values of type Fd are not mutually interchangeable with values of type Int. That is, the following operation:

```
val stderr = (+ stdin stdin)
```

will not be compilable even if values stdin are internally represented as integer 1 and stderr is internally represented as integer 2.

Here we create powerboxed I/O functions.

91b ⟨*Trusted/Fd.pi* 91a⟩+≡ ◁91a 91c▷

```
val fd = (makeFd stdin stdout stderr fdwrite)
```

We introduce some convenient shortcuts.

91c ⟨*Trusted/Fd.pi* 91a⟩+≡ ◁91b

```
val pr = fd.pr
val nl = fd.nl
val prNL = fd.prNL
val print = fd.print
val printi = fd.printi
```

3.4 Prim: Primitive Operations

3.4.1 Runtime error reporting

In case of a run time error (such as division by zero) we do not let the faulty component to bring the whole system down. However, for debugging purposes, we currently want to be notified when such even occurs. Runtime errors are reported with the:

```
error = /[String /Bot]
```

procedure. It prints out a given string on the screen and does not return. This ensures that faulty (sequential) process will be blocked forever (and thus garbage-collected) and will not try to continue because its actions would not make sense.

3.4.2 Primitive operations with booleans

Conjunction, disjunction, exclusive-or and negation of booleans.

```
&&  = /[Bool Bool /Bool]
||  = /[Bool Bool /Bool]
xor = /[Bool Bool /Bool]
not = /[Bool /Bool]
```

3.4.3 Primitive operations with integers

Addition, subtraction, multiplication, integer division and remainder after integer division.

```
+ = /[Int Int /Int]
- = /[Int Int /Int]
* = /[Int Int /Int]
div = /[Int Int /Int]
mod = /[Int Int /Int]
```

Comparison of integers.

```
== = /[Int Int /Bool]
<< = /[Int Int /Bool]
```

Bitwise operations.

```
land = /[Int Int /Int]
lor = /[Int Int /Int]
lxor = /[Int Int /Int]
lnot = /[Int /Int]
shr = /[Int Int /Int]
shl = /[Int Int /Int]
```

Conversion of integer to a character.

```
intToChar = /[Int /Char]
```

3.4.4 Primitive operations with characters

(`charToString c`) creates a string of size one, containing the character `c`.

```
charToString = /[Char /String]
```

3.4.5 Primitive operations with strings

(stringMake c n returns a new string that has n characters each of which is equal to c.

(stringSize s) return the size of the string s.

```
stringSize = /[String /Int]
```

(stringShorten s) counts the number of characters in s before the first null character, and then sets the length of s to match. Thus, whenever s is garbage collected it will be shortened.

```
stringShorten = /[String /[]]
```

The following two functions let use get and set the i-th character of a given string s.

```
stringNth = /[String Int /Char]
stringSetNth = /[String Int Char /[]]
```

3.4.6 Manipulating bytes

(alloc n) allocates enough memory to store a string consisting of n characters.

```
alloc = /[Int /String]
```

3.4.7 Testing pointers

(nullPtr x) checks if x is a null pointer.

```
nullPtr = /[#X X /Bool]
```

3.4.8 Primitive operations with lists

Nil refers to the empty list. Cons constructs a new list from a given head and tail. Null tests whether a given list is empty. UnsafeCar returns the head of a given list. Cdr returns the tail of a given list. If car or cdr are given an empty list, run-time failure will occur.

```
nil = (List Bot)
cons = /[#X X (List X) /(List X)]
null = /[#X (List X) /Bool]
car = /[#X (List X) /X]
cdr = /[#X (List X) /(List X)]
```

3.4.9 Primitive operations with arrays

```
arrayEmpty = /[#X /(Array X)]
arrayMake = /[#X X Int /(Array X)]
arraySize = /[#X (Array X) /Int]
arrayNth = /[#X (Array X) Int /X]
arrayUpdate = /[#X (Array X) Int X /[]]
```

3.4.10 Support for input and output

Print a given string on the standard output.

```
print = /String
```

3.4.11 Implementation

94 ⟨*Trusted/Prim.pi* 94⟩≡

```
  ⟨common types 58a⟩

 val [
   #CString
   prim : [
     error = /[String /Bot]
     + = /[Int Int /Int]
     - = /[Int Int /Int]
     * = /[Int Int /Int]
     div = /[Int Int /Int]
     mod = /[Int Int /Int]
     == = /[Int Int /Bool]
     << = /[Int Int /Bool]
     land = /[Int Int /Int]
     lor = /[Int Int /Int]
     lxor = /[Int Int /Int]
     lnot = /[Int /Int]
     shr = /[Int Int /Int]
     shl = /[Int Int /Int]
     intToChar = /[Int /Char]
     charToString = /[Char /String]
     stringShorten = /[String /[]]
     stringMake = /[Char Int /String]
     stringNth = /[String Int /Char]
     stringSetNth = /[String Int Char /[]]
     stringFromCString = /[CString /String]
     stringSize = /[String /Int]
     alloc = /[Int /String]
     nullPtr = /[#X X /Bool]
     &&  = /[Bool Bool /Bool]
     ||  = /[Bool Bool /Bool]
     xor = /[Bool Bool /Bool]
     not = /[Bool /Bool]
     nil = (List Bot)
     cons = /[#X X (List X) /(List X)]
     null = /[#X (List X) /Bool]
     car = /[#X (List X) /X]
     cdr = /[#X (List X) /(List X)]
     arrayEmpty = /[#X /(Array X)]
     arrayMake = /[#X X Int /(Array X)]
     arraySize = /[#X (Array X) /Int]
     arrayNth = /[#X (Array X) Int /X]
     arrayUpdate = /[#X (Array X) Int X /[]]
     exit = /[Int /Bot]
   ]
 ] = (
   val [#CString] = [#Top]

   def error [s:String r:/Bot] = ((ccode 0 E "puts(S(#)); exit(1);" s); ())

   inline def && (a:Bool b:Bool) : Bool = (ccode 0 C "(# & #)" a b)
   inline def || (a:Bool b:Bool) : Bool = (ccode 0 C "(# | #)" a b)
   inline def xor (a:Bool b:Bool) : Bool = (ccode 0 C "(# ^ #)" a b)
   inline def not (b:Bool) : Bool = (ccode 0 C "(# ^ 1)" b)

   inline def + (x:Int y:Int) : Int = (ccode 0 C "(# + #)" x y)
```

```
inline def - (x:Int y:Int) : Int = (ccode 0 C "(# - #)" x y)
inline def * (x:Int y:Int) : Int = (ccode 0 C "(# * I(#))" x y)

inline def == (x:Int y:Int) : Bool = (ccode 0 C "(# == #)" x y)
inline def << (x:Int y:Int) : Bool = (ccode 0 C "(# < #)" x y)

{- This following two functions is redefined in the Int module.
   Those function are not visible here and since we need them we
   have to define it here too.
-}
inline def <= (x:Int y:Int) : Bool = (|| (== x y) (<< x y))
inline def >= (x:Int y:Int) : Bool = (not (<< x y))

inline def land (x:Int y:Int) : Int = (ccode 0 C "(# & #)" x y)
inline def lor (x:Int y:Int) : Int = (ccode 0 C "(# | #)" x y)
inline def lxor (x:Int y:Int) : Int = (ccode 0 C "(# ^ #)" x y)
inline def lnot (x:Int) : Int = (ccode 0 C "((~#) - 1)" x)
inline def shr (x:Int y:Int) : Int = (ccode 0 C "intInt(I(#) >> I(#))" x y)
inline def shl (x:Int y:Int) : Int = (ccode 0 C "intInt(I(#) << I(#))" x y)

inline def <> (x:Int y:Int) : Bool = (not (== x y))

inline def div (x:Int y:Int) : Int =
  if (<> y 0) then
    (ccode 0 C "intInt(# / #)" x y)
  else
    (error "ERROR: Division by zero (int.div).")

inline def mod (x:Int y:Int) : Int =
  if (<> y 0) then
    (ccode 0 C "(# % #)" x y)
  else
    (error "ERROR: Division by zero (int.mod).")

inline def alloc (n:Int) : String =
  if (<< n 32000) then
    (ccode 8192 P "({
              Val string = TAG(Free); int bytes = I(#);
              long sz = (bytes + sizeof(Val) + sizeof(Val) - 1) / sizeof(Val);
              STATUS(string) = STRING(bytes); Free += sz;
              string;
          })" n)
  else
    (error "ERROR: no support for large objects yet (prim.alloc).")

{- This primitive could be elliminated by using given integer as
   an index to some array of proper characters. The problem is that
   in the 'Char' module function defined in 'Array' are not visible. -}
inline def charToString (c:Char) : String =
  ( val s = (alloc 2)
    (ccode 0 E "S(#)[0] = I(#); S(#)[1] = 0;" s c s);
    s
  )

inline def nullPtr (#X x:X) : Bool =
  (ccode 0 C "(((void *)#) == NULL)" x)

inline def stringShorten (s:String) : [] =
```

```
  ( val sz = (+ (ccode 0 P "intInt(strlen((char*)S(#)))" s) 1)
    (ccode 0 E "STATUS(#) = STRING(I(#));" s sz)
  )

inline def stringSize (s:String) : Int =
  (ccode 0 P "intInt(SIZE(STATUS(#))-1)" s)

inline def stringSetNth (s:String x:Int c:Char):[] =
  ( val sz = (stringSize s)
    if (|| (<< x 0) (<< sz x)) then
      (error "prim.stringSetNth: index outside bounds")
    else if (&& (== x sz) (<> c 0)) then
      (error "prim.stringSetNth: last character can only be set to zero character")
    else
      ( ccode 0 E "if ( (fromSpace <= (Val *)# && (Val *)# <= fromLimit) ||
                       (toSpace <= (Val *)# && (Val *)# <= toLimit) ) {
                    S(#)[I(#)] = I(#);
                  } else {
                    printf(\"prim.stringSetNth: cannot be applied to string located outside heap\\
                    exit(1);
                  }
                 "
        s s s s s x c
      )
  )

inline def stringMake (c:Char n:Int) : String =
  ( val s = (alloc (+ n 1))
    (ccode 0 E "memset(S(#),I(#),I(#));" s c n);
    (stringSetNth s n '\000');
    s
  )

inline def stringUnsafeNth (s:String x:Int) : Char =
  (ccode 0 P "intInt((Val)((unsigned char *)S(#))[I(#)])" s x)

inline def stringNth (s:String x:Int) : Char =
  if (&& (<= 0 x) (<< x (stringSize s))) then
    (stringUnsafeNth s x)
  else
    (error "prim.stringNth: index outside bounds")

inline def intToChar (x:Int) : Char = (ccode 0 C "#" x)

val nil:(List Bot) = (ccode 0 C "#" [])
inline def cons (#X hd:X tl:(List X)) : (List X) = (ccode 0 C "#" [hd tl])

inline def null (#X l:(List X)) : Bool = (ccode 0 C "(# == Zero)" l)
inline def unsafeCar (#X l:(List X)) : X = (ccode 0 P "OFFSET(#,1)" l)
inline def unsafeCdr (#X l:(List X)) : (List X) = (ccode 0 P "OFFSET(#,2)" l)
inline def car (#X l:(List X)) : X =
  if (not (null l)) then
    (unsafeCar l)
  else
    (error "car: expected a non-empty list")

inline def cdr (#X l:(List X)) : (List X) =
  if (not (null l)) then
```

```
      (unsafeCdr l)
    else
      (error "cdr: expected a non-empty list")

{-
 - We have to implement zero-length arrays as pointers to an object
 - outside the heap, since we cannot put zero-length objects in the heap.
 -}

(ccode 0 I "extern const Val emptyArray[1];");
(ccode 0 S "const Val emptyArray[1] = {TUPLE(1)};");

inline def arrayEmpty (#X) : (Array X) =
  (ccode 0 C "TAG(emptyArray)")

{-HACK-}
inline def mkArray (#X x:X n:Int) : (Array X) =
  if (<< n 8191) then
   (ccode 8192 P "({
      Val array = TAG(Free); int sz = I(#);
      STATUS(array) = TUPLE(sz+1); Free += sz+1;
      while (sz) OFFSET(array, sz--) = #;
      array;
    })" n x)
  else
    (error "Array.make: no support for large objects yet")

inline def arrayMake (#X x:X n:Int) : (Array X) =
  if (<= 0 n) then
    if (== n 0) then
      (arrayEmpty #X)
    else
      (mkArray x n)
  else
    (error "array.make: negative size")

inline def arraySize (#X a:(Array X)) : Int =
  (ccode 0 P "intInt(SIZE(STATUS(#))-1)" a)

inline def arrayNth (#X a:(Array X) n:Int) : X =
  if (&& (<= 0 n) (<< n (arraySize a))) then
    (ccode 0 R "OFFSET(#,I(#)+1)" a n)
  else
    (error "array.nth: index out of range")

inline def arrayUpdate (#X a:(Array X) n:Int x:X) : [] =
  if (&& (<= 0 n) (<< n (arraySize a))) then
    (ccode 0 E "OFFSET(#,I(#)+1) = #;" a n x)
  else
    (error "array.update: index out of range")

inline def stringFromCString (c:CString) : String =
  ( val sz = (+ (ccode 0 P "intInt(strlen((char*)#))" c) 1)
    val s = (alloc sz)
    (ccode 0 E "COPY(S(#),((char*)#),I(#));" s c sz);
    s
  )
```

```
  inline def exit [status:Int r:/Bot] =
    ( (ccode 0 E "exit(I(#));" status);
      ()
    )

  [ [ error=error + = + - = - * = * div=div mod=mod == = == << = <<
      land=land lor=lor lxor=lxor lnot=lnot shr=shr
      shl=shl intToChar=intToChar charToString=charToString
      stringShorten=stringShorten stringMake=stringMake stringNth=stringNth
      stringSetNth=stringSetNth stringFromCString=stringFromCString
      stringSize=stringSize
      alloc=alloc nullPtr=nullPtr && = && || = || xor=xor not=not
      nil=nil cons=cons null=null car=car cdr=cdr
      arrayEmpty=arrayEmpty arrayMake=arrayMake arraySize=arraySize
      arrayNth=arrayNth arrayUpdate=arrayUpdate exit=exit
  ] ]
)

val error = prim.error
val exit = prim.exit
```

3.5 Signal: UNIX Signals

Those who import this module are able to register handlers of traditional UNIX signals. In C signal handlers are set up with `signal (2)` and `sigaction (2)` system calls.

3.5.1 Types

This module introduces two types.

`Signal` is a (fresh) type of values that represent particular UNIX signals.

The `SignalHandler` is an alias to type of capabilities that can be registered as UNIX signal observers. Whenever proper UNIX signal occurs, this module will send value `[]` over those capabilities.

```
type SignalHandler = /[]
```

3.5.2 Constants

This module defines constants that identify particular kinds of UNIX signals, for which we can register handlers:

`SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGBUS`, `SIGFPE`, `SIGUSR1`, `SIGSEGV`, `SIGUSR2`, `SIGPIPE`, `SIGALRM`, `SIGTERM`, `SIGSTKFLT`, `SIGCHLD`, `SIGCONT`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`, `SIGURG`, `SIGXCPU`, `SIGXFSZ`, `SIGVTALRM`, `SIGPROF`, `SIGWINCH`, `SIGIO`.

All are of type `Signal`.

3.5.3 API

It is possible to install handlers of UNIX signals either as "normal" or as "persistent". If there is at least one persistent handler install, the system will not terminate even though its run-queue becomes empty. Instead, it will be suspend (and consume no CPU bandwidth) and blocked until the next signal.

`(register s h)` registers `h` as a (normal, non-persistent) handler of UNIX signal `s`.

```
register = /[Signal SignalHandler /[]]
```

`(registerPersistent s h)` registers `h` as a persistent handler of UNIX signal `s`.

```
registerPersistent = /[Signal SignalHandler /[]]
```

3.5.4 Implementation

99a ⟨*Trusted/Signals.pi* 99a⟩≡ 99b ▷

```
import "Untrusted/Array"
import "Untrusted/Signals"

(ccode 0 I "extern Val genericPictSignalHandler;");
(ccode 0 I "extern int pendingSignals;");

(ccode 0 I "##include <signal.h>");
(ccode 0 I "##include <stdlib.h>");
```

99b ⟨*Trusted/Signals.pi* 99a⟩+≡ ◁ 99a

```
val signals: SignalsAPI =
( ⟨signals 100⟩

  [ SIGHUP    = SIGHUP
    SIGINT    = SIGINT
    SIGQUIT   = SIGQUIT
```

```
      SIGILL    = SIGILL
      SIGTRAP   = SIGTRAP
      SIGABRT   = SIGABRT
      SIGBUS    = SIGBUS
      SIGFPE    = SIGFPE
      SIGUSR1   = SIGUSR1
      SIGSEGV   = SIGSEGV
      SIGUSR2   = SIGUSR2
      SIGPIPE   = SIGPIPE
      SIGALRM   = SIGALRM
      SIGTERM   = SIGTERM
      SIGSTKFLT = SIGSTKFLT
      SIGCHLD   = SIGCHLD
      SIGCONT   = SIGCONT
      SIGTSTP   = SIGTSTP
      SIGTTIN   = SIGTTIN
      SIGTTOU   = SIGTTOU
      SIGURG    = SIGURG
      SIGXCPU   = SIGXCPU
      SIGXFSZ   = SIGXFSZ
      SIGVTALRM = SIGVTALRM
      SIGPROF   = SIGPROF
      SIGWINCH  = SIGWINCH
      SIGIO     = SIGIO

      register=register registerPersistent=registerPersistent
    ]
  )
```

100 ⟨*signals* 100⟩≡ (99b) 101a▷

```
  val SIGHUP    : Signal = (ccode 0 C "intInt(SIGHUP)")
  val SIGINT    : Signal = (ccode 0 C "intInt(SIGINT)")
  val SIGQUIT   : Signal = (ccode 0 C "intInt(SIGQUIT)")
  val SIGILL    : Signal = (ccode 0 C "intInt(SIGILL)")
  val SIGTRAP   : Signal = (ccode 0 C "intInt(SIGTRAP)")
  val SIGABRT   : Signal = (ccode 0 C "intInt(SIGABRT)")
  val SIGBUS    : Signal = (ccode 0 C "intInt(SIGBUS)")
  val SIGFPE    : Signal = (ccode 0 C "intInt(SIGFPE)")
  val SIGUSR1   : Signal = (ccode 0 C "intInt(SIGUSR1)")
  val SIGSEGV   : Signal = (ccode 0 C "intInt(SIGSEGV)")
  val SIGUSR2   : Signal = (ccode 0 C "intInt(SIGUSR2)")
  val SIGPIPE   : Signal = (ccode 0 C "intInt(SIGPIPE)")
  val SIGALRM   : Signal = (ccode 0 C "intInt(SIGALRM)")
  val SIGTERM   : Signal = (ccode 0 C "intInt(SIGTERM)")
  val SIGSTKFLT : Signal = (ccode 0 C "intInt(SIGSTKFLT)")
  val SIGCHLD   : Signal = (ccode 0 C "intInt(SIGCHLD)")
  val SIGCONT   : Signal = (ccode 0 C "intInt(SIGCONT)")
  val SIGTSTP   : Signal = (ccode 0 C "intInt(SIGTSTP)")
  val SIGTTIN   : Signal = (ccode 0 C "intInt(SIGTTIN)")
  val SIGTTOU   : Signal = (ccode 0 C "intInt(SIGTTOU)")
  val SIGURG    : Signal = (ccode 0 C "intInt(SIGURG)")
  val SIGXCPU   : Signal = (ccode 0 C "intInt(SIGXCPU)")
  val SIGXFSZ   : Signal = (ccode 0 C "intInt(SIGXFSZ)")
  val SIGVTALRM : Signal = (ccode 0 C "intInt(SIGVTALRM)")
  val SIGPROF   : Signal = (ccode 0 C "intInt(SIGPROF)")
  val SIGWINCH  : Signal = (ccode 0 C "intInt(SIGWINCH)")
  val SIGIO     : Signal = (ccode 0 C "intInt(SIGIO)")
```

This is our C handler for all UNIX signals. It will record (in the `pendingSignals` variable) what UNIX signal happend.

101a ⟨*signals* 100⟩+≡ (99b) ◁100 101b▷

```
(ccode 0 E "
  void genericCsignalHandler(int signal)
  {
    sigset_t sigset;
    sigfillset(&sigset);
    sigprocmask(SIG_BLOCK, &sigset, NULL);
    pendingSignals |= 1 << signal;
    sigprocmask(SIG_UNBLOCK, &sigset, NULL);
  }"
);
```

Below, via ordinary UNIX system calls, we ensure that our `genericCsignalHandler` will be called whenever some signal occurs. We skipt registering the `SIGKILL` and `SIGSTOP` because these signals cannot be caught.

101b ⟨*signals* 100⟩+≡ (99b) ◁101a 101c▷

```
(ccode 0 E "
  { struct sigaction sa;
    sa.sa_handler = genericCsignalHandler;
    sigfillset(&sa.sa_mask);
    sa.sa_flags = 0;

    int signal;
    for (signal=1; signal < 32; signal++) {
      if (signal != SIGKILL && signal != SIGSTOP) {
        if (sigaction(signal, &sa, NULL) < 0) {
          printf(\"Signals.nw: sigaction failed: %d\\n\", signal);
          exit(1);
        }
      }
    }
  }
");
```

The `lock` channel has two purposes.

- It contains useful information related to what to do when particular signal occurs.
- It is used for synchronization among functions
  - genericPictSignalHandler
  - register
  - registerPersistent

The `lock` channel always contains at most one value. This value is an array with one element per UNIX signal. Array elements of are couples of type `[handler=SignalHandler persistent=Bool]`. When our system receives any kind of signal by default it terminates.

101c ⟨*signals* 100⟩+≡ (99b) ◁101b 102a▷

```
new lock: ^(Array [handler=SignalHandler persistent=Bool])
val handlers =
  (array.tabulate
    32
    \(x:Int):[handler=SignalHandler persistent=Bool] =
      [ handler = \[] =
          ( (ccode 0 E "printf(\"UNIX signal %ld was received. Terminating by default.\\n\", I(#));"
            (exit 1);
            ()
          )
```

```
        persistent = false
      ]
  )
  run lock ! handlers
```

Because our C signal handler sets up the `pendingSignals` variable, the runtime is able to recognize the event when UNIX signal happend. It will send the value of the `pendingSignals` variable to the `genericPictSignalHandler` process defined below.

Runtime also invokes this function whenever there is nothing more to do (the run queue is empty). Then it is responsibility of this function to decide whether the whole system should exit (when there are no persistent Pict signal handlers) or whether the whole system should be suspended and wait the occurrence of the next UNIX signal.

102a ⟨*signals* 100⟩+≡ (99b) ◁101c 102b▷

```
  def genericPictSignalHandler pendingSignals:Int =
    ( if (== pendingSignals 0) then
        ( val handlers = (read lock)
          val persitentSignalHandlerExists =
            (array.detect handlers
              \([handler=_:SignalHandler persistent=persistent:Bool]):Bool = persistent
            )
          if persitentSignalHandlerExists then
            ( (ccode 0 E "{ sigset_t sigset;
                           sigemptyset(&sigset);
                           sigsuspend(&sigset);
                         }");
              (write lock handlers)
            )
          else
            (exit 0)
        )
      else
        (int.for 0 30
            \(signal:Int):[] =
              if (int.isSet bit=signal pendingSignals) then
                ( val handlers = (read lock)
                  val [handler=handler persistent=_] = (array.nth handlers signal)
                  (write handler []);
                  (write lock handlers)
                )
              else
                []
        );
      ()
    )
```

The address of the `genericPictSignalHandler` function/process is stored into the `genericPictSignalHandler` C variable. This will enable runtime to send this process a message (the value of the `pendingSignals` C variable).

102b ⟨*signals* 100⟩+≡ (99b) ◁102a 102c▷

```
  (ccode 0 E "genericPictSignalHandler = #;" genericPictSignalHandler);
```

Register a normal handler of a given UNIX signal.

102c ⟨*signals* 100⟩+≡ (99b) ◁102b 103▷

```
  def register (signal:Signal handler:SignalHandler) : [] =
    ( val handlers = (read lock)
      (array.update handlers signal [ handler = handler
                                      persistent = false
                                    ]
```

```
    );
    (write lock handlers)
  )
```

Register a persistent handler of a given UNIX signal.

103 ⟨*signals* 100⟩+≡ (99b) ◁102c

```
 def registerPersistent (signal:Signal handler:SignalHandler) : [] =
   ( val handlers = (read lock)
     (array.update handlers signal [ handler = handler
                                     persistent = true
                                   ]
     );
     (write lock handlers)
   )
```

3.6 Socket

This module provides basic functions for opening a socket in a raw mode, receiving and sending data through it. This module provides necessary functionality required by the `ping` program but it cannot be regarded yet as feature complete.

3.6.1 Types

Values of the `Socket` type represent individual sockets.

3.6.2 Constants

Several constants related to opening sockets.

These constants: `domain.inet`, `typ.raw`, `protocol.icmp` are useful if you want to open a given socket in a raw mode.

3.6.3 API

`(open domain typ protocol)` creates a new socket opened in raw mode. Note that if some Pict program uses this function, it will only succeed if the whole program is run with root privileges.

```
openRaw = /[/Socket]
```

`(send s ip buffer)` sends given `buffer` over toward given `ip` address over a given socket `s`. The meaning of the data depends on the mode in which a given socket was created. When the socket was opened in a raw mode, the `buffer` represents some kind of packet used in protocols above the IP layer, for example `ICMPPacket < String`.

```
sendTo = /[Socket IPAddress String /[]]
```

`(recv s buffer` receives data from a given socket `s` and stores them into a given buffer. The meaning of that data depends on the mode in which socket `s` was created. In case of sockets opened in the raw mode, the `buffer` will hold `IPPacket < String`. When the return value is negative, it means that reception of packet from a given socket failed. Otherwise it tells the number of received bytes.

```
recv = /[Socket String Int]
```

3.6.4 Implementation

104 ⟨*Trusted/Socket.pi* 104⟩≡

```
import "Untrusted/IPAddress"
import "Untrusted/Socket"

(ccode 0 I "##include <sys/poll.h>");
(ccode 0 I "##include <sys/types.h>");
(ccode 0 I "##include <sys/socket.h>");
(ccode 0 I "##include <netinet/in.h>");
(ccode 0 I "##include <netinet/ip.h>");
(ccode 0 I "##include <netinet/ip_icmp.h>");
(ccode 0 I "##include <arpa/inet.h>");
(ccode 0 I "##include <errno.h>");

val socket: SocketAPI = (

  ⟨socket 105a⟩
```

```
    [ openRaw=openRaw
      sendTo=sendTo
      recv=recv
    ]
  )
```

With respect to particular values of parameters, in order to succeed, the whole system might be required to run as a superuser. This happens if you try to open a socket in the raw mode.

105a ⟨*socket* 105a⟩≡ (104) 105b ▷

```
  def openRaw () : Socket =
    (ccode 0 W "intInt({ int s = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
                         if (s == -1) {
                           printf(\"socket.open: failed\\n\");
                           exit(1);
                         } else if (1000000 < s) {
                           printf(\"socket.open: too big filedescriptor; it cannot be represented as Pi
                           exit(1);
                         }
                         s;
                       })"
    )
```

Send a given buffer to a given address over a given socket.

105b ⟨*socket* 105a⟩+≡ (104) ◁ 105a 105c ▷

```
  def sendTo (socket:Socket ip:IPAddress buffer:String) : [] =
    ( val [ipByte0 ipByte1 ipByte2 ipByte3] = ip
      val bufferSize = (string.size buffer)
      (ccode 0 E "struct sockaddr_in pingaddr;
                  memset(&pingaddr, 0, sizeof(struct sockaddr_in));
                  pingaddr.sin_family = AF_INET;

                  u_int32_t ip_address;
                  ip_address = htonl((I(#) << 24) | (I(#) << 16) | (I(#) << 8) | I(#));
                  pingaddr.sin_family = AF_INET;
                  pingaddr.sin_addr.s_addr = ip_address;

                  int rvl = sendto( I(#), S(#), I(#), 0,
                                    (struct sockaddr *)&pingaddr,
                                    sizeof(struct sockaddr_in)
                                  );
                  if (rvl != I(#)) {
                    printf(\"socket.sendTo: failed\\n\");
                    exit(1);
                  }
                 "
        ipByte0 ipByte1 ipByte2 ipByte3 socket buffer bufferSize bufferSize
      )
    )
```

Receive a packet from a given socket.

105c ⟨*socket* 105a⟩+≡ (104) ◁ 105b

```
  def recv (socket:Socket buffer:String) : Int =
    ( val bufferSize = (string.size buffer)
      (ccode 0 W "({ int rvl;
                     rvl = recv(I(#), S(#), I(#), 0);
                     intInt(0 <= rvl ? rvl : -errno);
                  })"
```

```
      socket buffer bufferSize
    )
  )
```

Bibliography

[1] Network Working Group. RFC 1071: Computing the Internet Checksum.

[2] Network Working Group. RFC 792: Internet Control Message Protocol.

[3] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[4] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[5] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119, pages 179–194, Pisa, Italy, 1996. Springer-Verlag.

[6] Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. 1997.

[7] Benjamin C. Pierce and David N. Turner. Pict language definition. `http://citeseer.ist.psu.edu/article/pierce96pict.html`, 1997.

[8] Benjamin C. Pierce and David N. Turner. Pict libraries manual. Available electronically, 1997.

[9] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.