Standard Library of Tamed Pict Programming Language

Matej Košík

April 15, 2011
2.18.2 Channel Input/Output .................................................. 63
2.18.3 Discarding Results ..................................................... 63
2.18.4 Function Composition .................................................. 63
2.18.5 Implementation ....................................................... 64
2.19 Queue ................................................................. 65
  2.19.1 Creation ............................................................. 65
  2.19.2 Interrogation ........................................................ 65
  2.19.3 Modification ......................................................... 65
  2.19.4 Iteration .............................................................. 66
  2.19.5 Implementation ...................................................... 66
2.20 Random Numbers ....................................................... 72
  2.20.1 Operations ........................................................... 72
  2.20.2 Implementation ..................................................... 72
2.21 Revoker ................................................................. 74
  2.21.1 Implementation ...................................................... 74
2.22 Sem: Semaphores ....................................................... 75
  2.22.1 Implementation ..................................................... 75
2.23 Signal: UNIX Signals ................................................... 77
2.24 Socket ................................................................. 78
2.25 String ................................................................. 79
  2.25.1 Types ................................................................. 79
  2.25.2 Creation ............................................................. 79
  2.25.3 Interrogation ......................................................... 79
  2.25.4 Modification ......................................................... 80
  2.25.5 Comparison ........................................................ 80
  2.25.6 Conversion ........................................................ 80
  2.25.7 Iteration .............................................................. 81
  2.25.8 Implementation ...................................................... 81

3 Trusted Modules ........................................................... 88

3.1 Alarm ................................................................. 88
3.2 Args: Command-Line Arguments ...................................... 90
3.3 Fd: File-descriptor Operations ....................................... 91
3.4 Prim: Primitive Operations .......................................... 92
  3.4.1 Runtime error reporting ........................................... 92
  3.4.2 Primitive operations with booleans ................................ 92
  3.4.3 Primitive operations with integers ................................ 92
  3.4.4 Primitive operations with characters ................................ 92
  3.4.5 Primitive operations with strings .................................. 93
  3.4.6 Manipulating bytes ................................................ 93
  3.4.7 Testing pointers .................................................... 93
  3.4.8 Primitive operations with lists ..................................... 93
Copying

This is a standard library of Tamed Pict programming language.
Copyright © 2007 Matej Košík

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Many parts of this software were drawn from Pict [6, 7, 9].
Chapter 1

Standard Prelude

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

\begin{verbatim}
(lib.pi) ≡
  import "Trusted/Fd"
  import "Untrusted/Bool"
  import "Untrusted/Cmp"
  import "Untrusted/Int"
  import "Untrusted/List"
  import "Untrusted/Misc"
  import "Untrusted/String"
\end{verbatim}

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

\begin{verbatim}
(hello.pi) ≡
  (pr "Hello, world\n";
\end{verbatim}

You can compile and run it as simply as:

\begin{verbatim}
pict hello.pi
\end{verbatim}

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 \texttt{Fd} module imported by default, you can compile that module with the \texttt{-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
- 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

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

by other modules. Most of the modules introduce new types. 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:

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

2.1 Alarm

```pict
(Untrusted/Alarm.pi)
```

type AlarmAPI =

```pict
[ 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 "Trust/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 \texttt{parse} functions sets the \texttt{f} cell to \texttt{true}. Similarly for the "-g" option.

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

If you invoke the program with the "-s foo" option, then the \texttt{parse} function sets the \texttt{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 \texttt{parse} function. Thus, if you invoke the program as follows:

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

Then

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

In this case, the \texttt{parse} function will return a list of two strings:

- "foo"
- "bar"

The \texttt{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 \texttt{parse} function will print appropriate error message and it will exit the whole UNIX process.
2.2.1 Implementation

\[
\langle \text{Untrusted/Args.pi}\rangle \equiv
\]

```plaintext
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
                        - anonymous 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
                         \(\langle\text{nullaryOption:String ...:(Cell Bool)\rangle}\) 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 by integer")
      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 by string")
      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.

\[
\text{copy} = \text{\#X (Array X) / (Array X)}
\]

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

\[
\text{empty} = \text{\#X / (Array X)}
\]

(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.

\[
\text{make} = \text{\#X X Int / (Array X)}
\]

(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.

\[
\text{tabulate} = \text{\#X Int / [Int / X] / (Array X)}
\]

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

\[
\text{fromList} = \text{\#X (List X) / (Array X)}
\]

2.3.2 Interrogation

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

\[
\text{size} = \text{\#X (Array X) / Int}
\]

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

\[
\text{nth} = \text{\#X (Array X) Int / X}
\]

(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.

\[
\text{detect} = \text{\#X (Array X) / [X / Bool] / Bool}
\]

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.

\[
\text{update} = \text{\#X (Array X) Int X / []}
\]

(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

(\common\ types \#sa \equiv
  \\\value\ [#Array : (Pos Type -> Type)] = [#\X = Top]

April 15, 2011
Untrusted/Array.nw 13
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")
            end
        end
    end

    and fromList (#X l:(List X)) : (Array X) =
        ( val size = (list.size l)
            if (== size 0) then
                (empty #X){-HACK-}
            else
                (error "array.fromList: negative size")
            end
        end
    end
}
( 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 x (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.

\[
\text{copy} = \lambda [X (\text{Array2 } X) \to (\text{Array2 } X)]
\]

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

\[
\text{empty} = \lambda [X (\text{Array2 } X)]
\]

(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.

\[
\text{make} = \lambda [X \times \text{Int} \times \text{Int} \to (\text{Array2 } X)]
\]

(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 \([[0 0], [0 1], \ldots, [0 (h-1)], [1 0], [1 1], \ldots, [w h]]\).

\[
\text{tabulate} = \lambda [X \times \text{Int} \to (\text{Int} \times \text{Int} \times X) \to (\text{Array2 } X)]
\]

(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.

\[
\text{fromList} = \lambda [X (\text{List} (\text{List } X)) \to (\text{Array2 } X)]
\]

2.4.2 Interrogation

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

\[
\text{size} = \lambda [X (\text{Array2 } X) \to (\text{Int} \times \text{Int})]
\]

(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 \([x y] < (\text{size} a)\), otherwise we generate a runtime error.

\[
\text{nth} = \lambda [X (\text{Array2 } X) \times \text{Int} \times \text{Int} \to X]
\]

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

\[
\text{detect} = \lambda [X (\text{Array2 } X) \to (X \times \text{Bool} \to \text{Bool})]
\]

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.

\[
\text{update} = \lambda [X (\text{Array2 } X) \times \text{Int} \times \text{Int} \times X \to []]
\]

(reset a x) sets the value of each array element to \text{@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

(Untrusted/Array2.pi)≡

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) Int Int /X]
rotate = /[#X (Array2 X) Int Int /X]
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)
```scala
/*
  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)):[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)):[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)):[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)):[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 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.

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

(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

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

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.
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

[ && = bool.&&
  || = bool.||
  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.

\[
\text{type Bytes = [ } \text{string = String} \\
\text{   minIndex = Int} \\
\text{   maxIndex = Int} \\
\text{ ]}
\]
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 )

  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
        (error "bytes.nth: out of range")
      else
        [ string = (string.copy s)
        minIndex = 0
        maxIndex = (dec (string.size s))
      ]

    else
      (error "bytes.nth: indexes out of range")
    )
(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.

\[ \text{empty} = /[#X /(\text{Cell } X)] \]

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

\[ \text{make} = /[#X X /(\text{Cell } X)] \]

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

\[ \text{isEmpty} = /[#X (\text{Cell } X) /\text{Bool}] \]

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

\[ \text{clear} = /[#X (\text{Cell } X) /[]] \]

(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.

\[ \text{get} = /[#X (\text{Cell } X) /X] \]

(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.

\[ \text{put} = /[#X (\text{Cell } X) X /[]] \]

2.7.1 Implementation

(\text{Untrusted}/Cell.pi) \equiv

\text{import } "\text{Untrusted/Misc}"

\begin{verbatim}
val [
  #Cell : (Pos Type \to 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]
  )
\end{verbatim}
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[^1] 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 \((|| (isAlpha\ c) (isDigit\ c))\).

\[
isAlnum = \lambda[Char /Bool]
\]

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

\[
isAlpha = \lambda[Char /Bool]
\]

Checks for a digit.

\[
isDigit = \lambda[Char /Bool]
\]

Checks for a lower-case character.

\[
isLower = \lambda[Char /Bool]
\]

Checks for any whitespace character.

\[
isSpace = \lambda[Char /Bool]
\]

Checks for any printable character including space.

\[
isPrint = \lambda[Char /Bool]
\]

Checks for any punctuation character.

\[
isPunct = \lambda[Char /Bool]
\]

Checks for an uppercase letter.

\[
isUpper = \lambda[Char /Bool]
\]

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

\[
isXDigit = \lambda[Char /Bool]
\]

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.

\[
toUpper = \lambda[Char /Char]
\]
\[
toLower = \lambda[Char /Char]
\]

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

\[
fromInt = \lambda[Int /Char]
\]

[^1]: 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

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 (>= x 0) (<< x 256)) then
    (prim.intToChar x)
  else
    (error "ERROR: Integer out of range (char.fromString)."

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

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

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

inline def isDigit (c:Char) : Bool =
  (&c (<= '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 '?')))
2.9 Choice

Although the Pict language does not directly support summation [4] (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
=> = //Lock !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

```
(Untrusted/Choice.pi[35] ≡
import "Untrusted/Misc"

type Lock = /Lock

val choice: [
    choose = //Lock
    $ = //Lock //Lock
    => = /Lock !X //Lock
]

def newLock [resultOfNewLock:/Lock] =
    ( new lock: Lock
    def loop [] =
        lock?[resultOfNewLock] = (resultOfNewLock!false | loop![])
        ( resultOfNewLock!(rchan lock)
        | lock?[resultOfNewLock] = (resultOfNewLock!true | loop![])
    )
)

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

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

6The 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 $\pi$-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\footnote{In fact, they could be entirely supported by proper software.}. The third step, however, is very hard. We are going to return to this later. In \cite{5} it is possible to find a proof of a correctness and completeness of a similar choice encoding.
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 than 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.

\[
\begin{align*}
LT &= \text{Cmp} \\
EQ &= \text{Cmp} \\
GT &= \text{Cmp}
\end{align*}
\]

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.

\[
\begin{align*}
\text{lt} &= (\text{Cmp} \to \text{Bool}) \\
\text{le} &= (\text{Cmp} \to \text{Bool}) \\
\text{eq} &= (\text{Cmp} \to \text{Bool}) \\
\text{ne} &= (\text{Cmp} \to \text{Bool}) \\
\text{ge} &= (\text{Cmp} \to \text{Bool}) \\
\text{gt} &= (\text{Cmp} \to \text{Bool})
\end{align*}
\]

2.10.3 Conversion

\((\text{toString } c)\) converts \(c\) to a string.

\[
\text{toString} = (\text{Cmp} \to \text{String})
\]

2.10.4 Implementation

\[
\begin{align*}
\langle \text{Untrusted}/\text{Cmp}.pi \rangle &\equiv \\
\text{import } "\text{Untrusted/Misc}" \\
\text{val } [ \\
\#\text{Cmp} \\
\text{cmp: [} \\
\text{LT = Cmp} \\
\text{EQ = Cmp} \\
\text{GT = Cmp} \\
\text{lt = (Cmp \to Bool)} \\
\text{le = (Cmp \to Bool)} \\
\text{eq = (Cmp \to Bool)} \\
\text{ne = (Cmp \to Bool)} \\
\text{ge = (Cmp \to Bool)} \\
\text{gt = (Cmp \to Bool)} \\
\text{toString = (Cmp \to String)} \\
\text{]} = ( \\
\text{type Cmp = Int})
\end{align*}
\]
{-
  - 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 return various fields of these kind of ICMP packets.

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

2.11.3 Implementation

⟨Untrusted/ICMPPacket.pi⟩≡
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
    }
April 15, 2011

- 16-bits are significant.
-}

```scala
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)
      \(\text{wordIndex:}\) [] = (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
    []
    (1not (+ (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)));
```

Untrusted/ICMPPacket.nw 40
{- Copy the data we were provided into the ICMP packet. -}
(string.itApply data
  \(i:\text{Int} \ ch:\text{Char}) = (\text{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) up
  upperByte = (land 255 (shr checksum 8)) ]
(string.update packet 2 (char.fromInt checksum.lowerByte));
(string.update packet 3 (char.fromInt checksum.upperByte));

{- 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 and cannot be redefined.

2.12.1 Arithmetic Operations

Addition and subtraction.

\[ + = \text{[Int Int /Int]} \]
\[ - = \text{[Int Int /Int]} \]

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.

\[ * = \text{[Int Int /Int]} \]
\[ \text{div} = \text{[Int Int /Int]} \]
\[ \text{mod} = \text{[Int Int /Int]} \]

Negation, predecessor and successor.

\[ \text{neg} = \text{[Int /Int]} \]
\[ \text{dec} = \text{[Int /Int]} \]
\[ \text{inc} = \text{[Int /Int]} \]

\((\text{sgn } x)\) returns 1 if \(x\) is strictly greater than zero, 0 if \(x\) equals zero, and -1 if \(x\) is strictly less than zero.

\[ \text{sgn} = \text{[Int /Int]} \]

\((\text{abs } x)\) returns the absolute value of \(x\).

\[ \text{abs} = \text{[Int /Int]} \]

\((\text{gcd } m n)\) returns greatest common divisor of a given two natural numbers \(m\) and \(n\).

\[ \text{gcd} = \text{[Int Int /Int]} \]

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

2.12.2 Bitwise Operations

Bitwise and, or, exclusive-or and negation.

\[ \text{land} = \text{[Int Int /Int]} \]
\[ \text{lor} = \text{[Int Int /Int]} \]
\[ \text{lxor} = \text{[Int Int /Int]} \]
\[ \text{lnot} = \text{[Int /Int]} \]

\((\text{isSet bit=i x})\) returns true if the \(i\)'th bit \((0 \leq i \leq 30)\) of \(x\) is set\(^9\).

\[ \text{isSet} = \text{[bit=Int Int /Bool]} \]

\((\text{set bit=i x})\) 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:

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

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 \(0, 10\), 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

Untrusted/Int.pi ≡
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]"
\[
\begin{align*}
\text{lnot} &= \text{Int} \\
\text{isSet} &= \text{int = Int} \\
\text{set} &= \text{bit = int \ Int} \\
\text{reset} &= \text{bit = Int} \\
\text{shr} &= \text{Int} \\
\text{shl} &= \text{Int} \\
\text{max} &= \text{Int} \\
\text{min} &= \text{Int} \\
\text{minmax} &= \text{Int Int} \\
\text{==} &= \text{Int Int} \\
\text{<>} &= \text{Int Int} \\
\text{>=} &= \text{Int Int} \\
\text{<=} &= \text{Int Int} \\
\text{cmp} &= \text{Int Int} \\
\text{apply} &= \text{Int Int} \\
\text{fold} &= \text{Int Int} \\
\text{for} &= \text{Int Int} \\
\text{hash} &= \text{Int} \\
\text{upperBound} &= \text{Int} \\
\text{lowerBound} &= \text{Int} \\
\text{power} &= \text{Int Int} \\
\end{align*}
\]

\[
\begin{align*}
\text{val} + &= \text{prim.} + \\
\text{val} - &= \text{prim.} - \\
\text{val} * &= \text{prim.} * \\
\text{val div} &= \text{prim.} \text{ div} \\
\text{val mod} &= \text{prim.} \text{ mod} \\
\text{val} == &= \text{prim.} == \\
\text{val} <= &= \text{prim.} <= \\
\text{val land} &= \text{prim.} \text{ land} \\
\text{val lor} &= \text{prim.} \text{ lor} \\
\text{val lxor} &= \text{prim.} \text{ lxor} \\
\text{val lnot} &= \text{prim.} \text{ lnot} \\
\text{val shr} &= \text{prim.} \text{ shr} \\
\text{val shl} &= \text{prim.} \text{ shl} \\
\end{align*}
\]

\[
\begin{align*}
\text{inline def neg} \ (x: \text{Int}) &= \text{Int} = (- 0 \ x) \\
\text{inline def dec} \ (x: \text{Int}) &= \text{Int} = (- x 1) \\
\text{inline def inc} \ (x: \text{Int}) &= \text{Int} = (+ x 1) \\
\text{inline def <>} \ (x: \text{Int} \ y: \text{Int}) &= \text{Bool} = (\not (== x y)) \\
\text{inline def <=} \ (x: \text{Int} \ y: \text{Int}) &= \text{Bool} = (|| (== x y) (<< x y)) \\
\text{inline def =>} \ (x: \text{Int} \ y: \text{Int}) &= \text{Bool} = (\not (<< x y)) \\
\text{inline def >>} \ (x: \text{Int} \ y: \text{Int}) &= \text{Bool} = (\&\& (\not (<< x y)) (\not (== x y))) \\
\text{inline def sgn} \ (x: \text{Int}) &= \text{Int} = \\
\quad \text{if} (<< x 0) \text{ then} \\
\quad \quad -1 \\
\quad \text{else if} (== x 0) \text{ then} \\
\quad \quad 0 \\
\quad \text{else} \\
\quad \quad 1 \\
\text{inline def abs} \ (x: \text{Int}) &= \text{Int} = \\
\quad \text{if} (<< x 0) \text{ then} \\
\quad \quad (\text{neg} \ x) \\
\quad \text{else} \\
\end{align*}
\]
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 x:Int) : Bool =
  (== 1 (land (shr x i) 1))

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

inline def reset (bit=i 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)
      )

  if (>> inc 0) then
    (def loop (x: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.

```
(Untrusted/Fd.pi)
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
          fwrite: /[Fd String Int /[]]
           ) : FdAPI =
  ( inline def pr (s:String) : [] =
        (fwrite stdout s (string.size s))
  inline def prErr (s:String) : [] =
        (fwrite 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]

(serialise 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

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

(make byte0 byte1 byte2 byte3)

serialize = /[IPAddress /String]

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

(Unttrusted/IPPacket.pi)

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)
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.

\[\text{makeNeg} = \text{[/#X !X /![X Lid]]}\]

(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.

\[\text{makePos} = \text{[/#X ?X /?[X Lid]]}\]

2.16.1 Implementation

\((Untrusted/Lid.pi)\[54\]≡

\[\text{import "Untrusted/Cell"}\]

\[\text{val \[\text{#Lid}\]
  \text{lid:}\{
    \text{makeNeg} = \text{[/#X !X /![X Lid]]}
    \text{makePos} = \text{[/#X ?X /?[X Lid]]}
  \}\}
\]

\[\text{type Lid = \[\text{cover = \[/[]\]
  \text{uncover = \[/[]\]}\]}
\]

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

\[\text{[10]We have decided not to use the Caretaker name because Lids seem to be more appropriate. Lids cover channels. Lids can be covered or uncovered. Additionally, since channels have two distinct types of ends (the positive and the negative one), we had to define to define two distinct kinds of lids.}\]
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 /[] /[]]
revMap = [X (List 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 /[]]

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

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

\[
\text{sort} = \{\#X \ (\text{List X}) / [X \ X \ /\text{Cmp} \ Bool] /\text{(List X)}\}
\]

2.17.6 Conversion

Given a hash function \(\text{f}\) for values of type \(\text{X}\), \(\text{hash \ l \ f}\) returns a hash value for a list \(\text{l}\) of type \(\text{(List X)}\).

\[
\text{hash} = \{\#X \ (\text{List X}) / [X \ /\text{Int}] /\text{Int}\}
\]

2.17.7 Comparison

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

\[
\text{cmp} = \{\#X \ (\text{List X}) (\text{List X}) / [X \ X \ /\text{Cmp}] /\text{Cmp}\}
\]

2.17.8 Implementation

\[
\begin{align*}
\text{val list: [} & \text{nil = (List Bot)} \\
& \text{cons = /[#X X (List X) /\text{(List X)}]} \\
& \text{make = /[#X Int X /\text{(List X)}]} \\
& \text{tabulate = /[#X Int /\text{Int /\text{(List X)}}]} \\
& \text{null = /[#X (List X) /\text{Bool}]} \\
& \text{car = /[#X (List X) /X]} \\
& \text{cdr = /[#X (List X) /\text{(List X)}]} \\
& \text{case = /[#X #R (List X) /[/R] /\text{X (List X) /R} /\text{R}]} \\
& \text{size = /[#X (List X) /\text{Int}]} \\
& \text{nth = /[#X (List X) Int /\text{X}]} \\
& \text{apply = /[#X (List X) /\text{X /[]} /\text{[]}]} \\
& \text{revApply = /[#X (List X) /\text{X /[]} /\text{[]}]} \\
& \text{itApply = /[#X (List X) /\text{Int X /[]} /\text{[]}]} \\
& \text{revItApply = /[#X (List X) /\text{Int X /[]} /\text{[]}]} \\
& \text{fold = /[#X #R (List X) R /\text{X R /R} /\text{R}]} \\
& \text{revFold = /[#X #R (List X) R /\text{X R /R} /\text{R}]} \\
& \text{itFold = /[#X #R (List X) R /\text{Int X R /R} /\text{R}]} \\
& \text{revItFold = /[#X #R (List X) R /\text{Int X R /R} /\text{R}]} \\
& \text{map = /[#X #R (List X) /\text{X /R} /\text{List R}]} \\
& \text{revMap = /[#X #R (List X) /\text{X /R} /\text{List R}]} \\
& \text{itMap = /[#X #R (List X) /\text{Int X /R} /\text{List R}]} \\
& \text{revItMap = /[#X #R (List X) /\text{Int X /R} /\text{List R}]} \\
& \text{filter = /[#X (List X) /\text{X /Bool} /\text{(List X)}]} \\
& \text{rev = /[#X (List X) /\text{(List X)}]} \\
& \text{append = /[#X (List X) (List X) /\text{(List X)}]} \\
& \text{revAppend = /[#X (List X) (List X) /\text{(List X)}]} \\
& \text{sort = /[#X (List X) /\text{X /Cmp} /\text{Bool} /\text{(List X)}]} \\
\end{align*}
\]
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 if (null l1) then
      [l2]
    else if (null l2) then
      [l1]
    else
      (split (cons (car l2) l1) (cdr l2) (dec n))
  )
  (split list nil 0)
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

\( \text{fst, snd and thd} \) 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 \((\text{snd [3 4 5]})\) is well-typed and evaluates to 4.

\[
\begin{align*}
\text{fst} &= /[#X [X] /X] \\
\text{snd} &= /[#X #Y [X Y] /Y] \\
\text{thd} &= /[#X #Y #Z [X Y Z] /Z]
\end{align*}
\]

2.18.2 Channel Input/Output

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

\[
\begin{align*}
\text{chan} &= /[#X /^X] \\
\text{read} &= /[#X ?X /X] \\
\text{write} &= /[#X !X X /[]] \\
\text{forward} &= /[#X ?X /X] \\
\text{rchan} &= /[#X !X //X]
\end{align*}
\]

2.18.3 Discarding Results

\((\text{discard #X})\) 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 \texttt{int.pr}[5 (\text{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).

\[
\begin{align*}
\text{discard} &= /[#X //X] \\
\text{await} &= /[#X X /[]]
\end{align*}
\]

2.18.4 Function Composition

\((\text{compose g f})\) composes functions \(g\) and \(f\).

\[
\begin{align*}
\text{compose} &= /[#X #Y #Z /[Y /Z] /[X /Y] //[X /Z]] \\
\text{identity} &= /[#X X /X]
\end{align*}
\]
2.18.5 Implementation

\[\text{Untrusted/Misc.pi} \equiv\]
import "Trusted/Prim"

val misc: [
  fst = \[\text{#X} [X] /X\]
  snd = \[\text{#X} #Y [X Y] /Y\]
  thd = \[\text{#X} #Y #Z [X Y Z] /Z\]
  chan = \[\text{#X} /-X\]
  read = \[\text{#X} ?X /X\]
  write = \[\text{#X} !X X /[]\]
  forward = \[\text{#X} ?X /X\]
  rchan = \[\text{#X} !X //X\]
  discard = \[\text{#X} //X\]
  await = \[\text{#X} X /[]\]
  compose = \[\text{#X} #Y #Z /[Y /Z] /[X /Y] //[X /Z]\]
  identity = \[\text{#X} X /X\]
] = (
  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 [5] 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 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 head or tail of a queue. Generates an error if the queue is empty.

\[
\text{removeHd} = \text{\texttt{\#X \texttt{(Queue X) \texttt{[]})}}
\]
\[
\text{removeTl} = \text{\texttt{\#X \texttt{(Queue X) \texttt{[]})}}
\]

### 2.19.4 Iteration

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

\[
\text{apply} = \text{\texttt{\#X \texttt{(Queue X) \texttt{[[X /[] /[]]]}}}}
\]
\[
\text{revApply} = \text{\texttt{\#X \texttt{(Queue X) \texttt{[[X /[] /[]]]}}}}
\]

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

\[
\text{itApply} = \text{\texttt{\#X \texttt{(Queue X) \texttt{[[Int X /[] /[]]]}}}}
\]
\[
\text{revItApply} = \text{\texttt{\#X \texttt{(Queue X) \texttt{[[Int X /[] /[]]]}}}}
\]

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

\[
\text{fold} = \text{\texttt{\#X \texttt{R (Queue X) \texttt{R \texttt{[[X R /R] /R]]}}}}
\]
\[
\text{revFold} = \text{\texttt{\#X \texttt{R (Queue X) \texttt{R \texttt{[[X R /R] /R]]}}}}
\]

\((\text{itFold } q \ \text{init} \ f)\) 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 \(\text{init}\). \((\text{revItFold } q \ \text{init} \ f)\) behaves similarly, except that it traverses \(q\) in reverse order.

\[
\text{itFold} = \text{\texttt{\#X \texttt{R (Queue X) \texttt{init \texttt{[[Int X R /R] /R]]}}}}
\]
\[
\text{revItFold} = \text{\texttt{\#X \texttt{R (Queue X) \texttt{init \texttt{[[Int X R /R] /R]]}}}}
\]

### 2.19.5 Implementation

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

\((\text{Untrusted/Queue.pi})\)\[66\]

\[
\text{\texttt{\#Queue : (Pos Type -> Type)}}
\]

\[
\text{\texttt{\texttt{empty} = \texttt{\#X \texttt{(Queue X))}}}
\]
\[
\text{\texttt{make} = \texttt{\#X \texttt{Int X /(Queue X))}}
\]
\[
\text{\texttt{tabulate} = \texttt{\#X \texttt{Int /(Int /X) /(Queue X))}}
\]
\[
\text{\texttt{isEmpty} = \texttt{\#X \texttt{(Queue X) /Bool}}
\]
\[
\text{\texttt{size} = \texttt{\#X \texttt{(Queue X) /Int}}
\]
\[
\text{\texttt{head} = \texttt{\#X \texttt{(Queue X) /X}}
\]
\[
\text{\texttt{tail} = \texttt{\#X \texttt{(Queue X) /X}}
\]
\[
\text{\texttt{insertHd} = \texttt{\#X \texttt{(Queue X) X /[]}}
\]
\[
\text{\texttt{insertTl} = \texttt{\#X \texttt{(Queue X) X /[]}}
\]
\[
\text{\texttt{removeHd} = \texttt{\#X \texttt{(Queue X) /[]}}
\]
\[
\text{\texttt{removeTl} = \texttt{\#X \texttt{(Queue X) /[]}}
\]
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)
          )
        )
      (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)
          )
        )
      (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 \( x = (x \times A) \mod M \) 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

\((\text{randomize } x)\) sets the seed in the random number generator to \( x \).

\[
\text{randomize} = [\text{Int} / []]
\]

The upper limit on numbers produces by \( \text{random} \).

\[
\text{max} = \text{Int}
\]

The bottom limit on numbers produces by \( \text{random} \).

\[
\text{min} = \text{Int}
\]

Returns a random integer \( r \) such that \( \text{min} < r < \text{max} \).

\[
\text{random} = [/[\text{Int}]]
\]

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

\[
\text{randomCoin} = [/[\text{Bool}]]
\]

2.20.2 Implementation

\[
(\text{Untrusted/Random.pi}[72] ≡
\]

\[
\text{import "Untrusted/Cell"}
\text{import "Untrusted/Int"}
\]

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

\text{inline def randomize (x:\text{Int}) : [] =}
\text{ (cell.put randomSeed x)}
\]

\[
\text{def random () : \text{Int} =}
\text{ (val a = 48271
\text{ val hi = (div (cell.get randomSeed) (div a))}
\]
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 through it to the target channel until the revoke function is called. If it is called, no one will be able to receive values sent through 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 through 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 through it.

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

2.21.1 Implementation

\[(Untrusted/Revoker.pi)\] ≡ \[import "Untrusted/Choice"

val revoker: [ variations of revoker constructors]

\[\{definitions of revoker constructors\} ≡ \[(74a)\]\]

\[\{definitions of revoker constructors\} ≡ \[(74a)\] +\]

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

\[\{definitions of revoker constructors\} ≡ \[(74b)\] ≡ \[(74b)\]

\[\{makeNeg=makeNeg makePos=makePos\]
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.

\[
\text{zero} = \text{Semaphore}\]

Create a semaphore that is initialized to one.

\[
\text{one} = \text{Semaphore}\]

Create a semaphore that is initialized to a given value.

\[
\text{make} = \text{Semaphore}\]

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

\[
\text{wait} = \text{Semaphore}[]\]

The \(\text{signal s}\) function increases the value of a given semaphore.

\[
\text{signal} = \text{Semaphore}[]\]

### 2.22.1 Implementation

\[(\text{Untrusted/Semaphore.pi}\)  
\[
\text{import "Untrusted/Int"}
\]

val [  
  #Semaphore  
  semaphore:[  
    zero = Semaphore\]  
    one = Semaphore\]  
    make = 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  
          | 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

\begin{verbatim}
(\texttt{Untrusted/Signals.pl}) \equiv
  type SignalHandler = \texttt{[]}
  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
    SIGTIN   = Signal
    SIGTOU   = Signal
    SIGURG   = Signal
    SIGXCPU  = Signal
    SIGXFSZ  = Signal
    SIGVTALRM = Signal
    SIGPROF  = Signal
    SIGWINCH = Signal
    SIGIO    = Signal
    register = /\{Signal SignalHandler /[]\}
    registerPersistent = /\{Signal SignalHandler /[]\}
  ]
\end{verbatim}
2.24 Socket

Note that socket is implemented as Int but it is fresh type distinct from Int. Neither Socket < Int nor Int < Socket.

\[ \text{(Untrusted/Socket.pi)78} \equiv \]

\[
\text{import "Untrusted/IPAddress"}
\]

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.

\[
\text{copy} = /\text{String /String}
\]

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

\[
\text{make} = /\text{Char Int /String}
\]

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

\[
\text{tabulate} = /\text{Int /[Int /Char] /String}
\]

(+$ s t) concatenates s and t.

\[
+\$ = /\text{String String /String}
\]

2.25.3 Interrogation

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

\[
\text{size} = /\text{String /Int}
\]

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

\[
\text{isEmpty} = /\text{String /Bool}
\]

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

\[
\text{nth} = /\text{String Int /Char}
\]

(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.

\[
\text{update} = /\text{String Int Char /[]}
\]

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 \(\leq (\text{size s})\).

\[
\text{sub} = /\text{String Int Int /String}
\]

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

\[
\text{tokens} = /\text{String Char /([List String])}
\]

(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

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

\[
\text{apply} = /\text{[String }/\text{[Char }/[ ]} /[ ]
\]

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

\[
\text{itApply} = /\text{[#X String }/\text{[Int }/[ ]} /[ ]
\]
\[
\text{revItApply} = /\text{[#X String }/\text{[Int Char }/[ ]} /[ ]
\]

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

\[
\text{fold} = /\text{[#R String R }/\text{[Char R }/[R]} /[R]
\]
\[
\text{revFold} = /\text{[#R String R }/\text{[Char R }/[R]} /[R]
\]

\((\text{itFold } s \text{ i } f)\) 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 \(i\). \(\text{revItFold}\) behaves just like \(\text{itFold}\), except that it traverses \(s\) in reverse order.

\[
\text{itFold} = /\text{[#R String R }/\text{[Int Char R }/[R]} /[R]
\]
\[
\text{itRevFold} = /\text{[#R String R }/\text{[Int Char R }/[R]} /[R]
\]

2.25.8 Implementation

\((\text{Untrusted/} \text{String.pi})\)\[
\begin{align*}
\text{import } & "\text{Untrusted/Char}" \\
\text{import } & "\text{Untrusted/List}"
\end{align*}
\]

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

```Trusted/Alarm.pi```
```
import "Untrusted/Alarm"
```

```Trusted/Alarm.pi```
```
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);
              }
        
```

88
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 ≤ n then we generate a runtime error.

```plaintext
(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:

\[
\langle \text{Trusted/Fd.pi} \rangle \equiv
\begin{array}{c}
\text{import "Untrusted/Fd"}
\end{array}
\]

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.

val fd = (makeFd stdin stdout stderr fdwrite)

We introduce some convenient shortcuts.

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]
oxor = /[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]
xor = /[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 = /

(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 = /

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

(stringNth = /

(stringSetNth = /

3.4.6 Manipulating bytes

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

alloc = /

3.4.7 Testing pointers

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

nullPtr = /

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 = /
null = /
car = /
cdr = /

3.4.9 Primitive operations with arrays

arrayEmpty = /
arrayMake = /
arraySize = /
arrayNth = /
arrayUpdate = /

3.4.10 Support for input and output

Print a given string on the standard output.

print = /
3.4.11 Implementation

\[
\langle \text{Trusted/Prim.pi} \rangle \equiv \langle \text{common types} \rangle
\]

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

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 "(# * #)" 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(# >> #)" x y)
inline def shl (x:Int y:Int) : Int = (ccode 0 C "intInt(# << #)" 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 eliminated 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) : [] =
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\n");
       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")
  else
    (error "Array.make: negative size")
  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.
- `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.

```plaintext
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`.

  ```plaintext
  register = /[[Signal SignalHandler /[]]
  ```

- `(registerPersistent s h)` registers `h` as a persistent handler of UNIX signal `s`.

  ```plaintext
  registerPersistent = /[[Signal SignalHandler /[]]
  ```

3.5.4 Implementation

```plaintext
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>"

(val signals: SignalsAPI =
  [ SIGHUP = SIGHUP
  SIGINT = SIGINT
  SIGQUIT = SIGQUIT
```
This is our C handler for all UNIX signals. It will record (in the pendingSignals variable) what UNIX signal
happend.

Below, via ordinary UNIX system calls, we ensure that our `genericCsignalHandler` will be called whenever some signal occurs. We skipped registering the SIGHUP and SIGSTOP because these signals cannot be caught.

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 are couples of type `[handler=SignalHandler persistent=Bool]`. When our system receives any kind of signal by default it terminates.
Because our C signal handler sets up the `pendingSignals` variable, the runtime is able to recognize the event when UNIX signal happen. 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.

```scala
102a
\[
\text{def genericPictSignalHandler pendingSignals: Int } \rightarrow
\]
102b
\[
\text{if (== pendingSignals 0) then}
\]
102c
\[
\text{val handlers = (read lock)}
\]
102d
\[
\text{val persistentSignalHandlerExists =}
\]
102e
\[
\text{array.detect handlers}
\]
102f
\[
\text{\{handler=_:SignalHandler persistent=persistent:Bool\}}:Bool = persistent
\]
102g
\[
\text{if persistentSignalHandlerExists then}
\]
102h
\[
\text{\{ccode 0 E "{ sigset_t sigset;
\}
\text{sigsuspend(&sigset);}
\text{Write lock handlers}
\}
\]
102i
\[
\text{else}
\]
102j
\[
\text{\{exit 0\}}
\]
102k
\[
\text{else}
\]
102l
\[
\text{\{int.for 0 30}
\]
102m
\[
\text{\{signal:Int\}:[] =}
\]
102n
\[
\text{\{if (int.isSet bit=signal pendingSignals) then}
\]
102o
\[
\text{\{val handlers = (read lock)}
\]
102p
\[
\text{\{val [handler=handler persistent=_] = (array.nth handlers signal)}
\]
102q
\[
\text{\{write handler []\}}
\]
102r
\[
\text{\{write lock handlers\}}
\]
102s
\[
\text{\{else}
\]
102t
\[
\text{\{[]\}}
\]
102u
\[
\text{\{else\}}
\]
102v
\[
\text{\{int.for 0 30}
\]
102w
\[
\text{\{ signal:Int\}:[] =}
\]
102x
\[
\text{\{if (int.isSet bit=signal pendingSignals) then}
\]
102y
\[
\text{\{val handlers = (read lock)}
\]
102z
\[
\text{\{val [handler=handler persistent=_] = (array.nth handlers signal)}
\]
102a
\[
\text{\{write handler []\}}
\]
102b
\[
\text{\{write lock handlers\}}
\]
102c
\[
\text{\{else}
\]
102d
\[
\text{\{[]\}}
\]
102e
\[
\text{\{else\}}
\]
```

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).

```scala
102a
\[
\text{\{ccode 0 E "genericPictSignalHandler = "; genericPictSignalHandler\}}
\]
```

Register a normal handler of a given UNIX signal.

```scala
102a
\[
\text{\{signals\}:[] =}
\]
102b
\[
\text{\{ccode 0 E "register (signal:Signal handler:SignalHandler) : [] =
\}
\]
102c
\[
\text{\{val handlers = (read lock)}
\]
102d
\[
\text{\{array.update handlers signal [ handler = handler
\}
102e
\[
\text{\{persistent = false
\}
```

The runtime will invoke the `register` 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.
Register a persistent handler of a given UNIX signal.

```scala
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

⟨Trusted/Socket.pi104≡

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 = (}
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.

```
<socket 105a>≡
  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 Pict integer.\n");
        exit(1);
      } 
      s;
    })
)
```

Send a given buffer to a given address over a given socket.

```
<socket 105b>≡
  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.

```
<socket 105c>≡
  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


