| galois |

Writing and Proving Thereoms:

Once one or more specifications have been established there are two things that should be done:

1. Develop properties the specifications should have
2. Prove equivalence between specifications and implementations

In future lessons, when the Software Analysis Workbench is discussed, the implementations will be mainly in the C language. Here, we develop and use Cryptol implementations to get the idea.

Consider a specification for testing whether two sequences are permutations of each other. This was developed in Lesson 2.4 and is the following:

```
remove lst n = drop `{2} [w.0 | w <- res]
  where
    res = [(-1,False)]#[ if (q.1 == False /\ ~(za == n))
                         then (za, q.1)
                         else (zb, True)
                       | za <- [-1]#lst | zb <- lst#[-1] | q <- res ]

member lst n = z ! 0
  where
    z = [False]#[ ss \/ (p == n) | p <- lst | ss <- z ]

perm x y = if (length x) == 0 /\ (length y) == 0 then True else
           if (length x) == (length y) then (z ! 0).1 else False
  where
    z = [ ((remove q.0 p), (member q.0 p) /\ q.1)
        | q <- [(y,True)]#z | p <- x ]
```

Observe that `perm x y` is a relation that should be an equivalence relation. An equivalence relation has properties *reflexive*, *symmetric*, and *transitive*. For reflexive, write the property like this:

```
permReflexive : ([10][32] -> [10][32] -> Bit) -> Bit
property permReflexive x y = ~(x == y) \/ perm x y
```

which says that `x` equals `y` for sequences of 32 bit numbers of length 10 implies `perm x y` is `True`. Small sequence lengths are used here so proofs are relatively fast. Prove this property like this in Cryptol (let the above be in file `perm.cry`):

```
Main> :l perm.cry
Loading module Cryptol
Loading module Main
Main> :s prover=cvc4
Main> :prove permReflexive
Q.E.D.
(Total Elapsed Time: 0.121s, using "CVC4")
```

Next set up property `permSymmetric`. Here there is a problem because the algorithm used to determine that `x` is a permutation of `y` used `-1` to fill in places as equality between elements of `x` and `y` is determined. This means the following for `permSymmetric` is not going to work:

```
permSymmetric : [10][32] -> [10][32] -> Bit
property permSymmetric x y = ~(perm x y) \/ perm y x
```

To see this, comment out property `permReflexive` and add the following type signature to `perm` (`[4][8]` is used here to get a result that is easy to see):

```
perm : [4][8] -> [4][8] -> Bit
```

Now do this in Cryptol (observe `255` is `2^^8-1`):

```
Main> :l perm.cry
Loading module Cryptol
Loading module Main
Main> perm [0,0,0,255] [0,0,0,0]
True
```

which should not be the case but happens because `255` is considered to be `-1` for width 8. What this says is that, for the particular specification chosen for perm, `2^^X-1`, where `X` is the width of the numbers in the input sequences, a test should have been added to make sure such numbers are not in the input lists. In other words, the algorithm for `perm` is wrong if all numbers up to `255` are to be allowed in input lists. Observe that without the above type signature for `perm`:

```
Main> :l perm.cry
Loading module Cryptol
Loading module Main
Main> perm [0,0,0,255] [0,0,0,0]
False
```

which illustrates an important property about Cryptol: the requirement of monomorphic type signatures for theorems finds problems that can otherwise be easily overlooked. Specification `perm` could be changed to forbid numbers `2^^X-1` but then `perm` would need a type signature to make such tests. It is left as an exercise to produce a specification for `perm` that allows all numbers in input sequences such that properties can be proved. In the meantime, the following works for sequences of 32 bit numbers of length 10:

```
permSymmetric : [10][32] -> [10][32] -> Bit
property permSymmetric x y =
   if ~(member x (2^^32-1)) /\ ~(member y (2^^32-1)) /\ perm x y
   then perm y x else True
```

Here it is tested:

```
Main> :l perm.cry
Loading module Cryptol
Loading module Main
Main> :s prover=cvc4
Main> :prove permSymmetric
Q.E.D.
(Total Elapsed Time: 29.092s, using "CVC4")
```

For `permTransitive` the property is:

```
permTransitive : [6][32] -> [6][32] -> [6][32] -> Bit
property permTransitive x y z =
   if ~(member x (2^^32-1)) /\ ~(member y (2^^32-1)) /\ ~(member z (2^^32-1)) /\
   perm x y /\ perm y z
   then perm x z else True
```

where `[6][32]` was chosen to make the proof complete in reasonable time.

```
Main> :l perm.cry
Loading module Cryptol
Loading module Main
Main> :s prover=cvc4
Main> :prove permTransitive
Q.E.D.
(Total Elapsed Time: 1m:12.672s, using "CVC4")
```

Conclude that, at least for sequences of `32` bit numbers of length `6`, where input sequences do not contain `2^^32-1`, `perm` defines an equivalence class. Specification perm will be used to help show that a given implementation of a proposed sorting algorithm is correct.

One such sorting algorithm is this:

```
bsort : {n, a} (Cmp a, Integral a, Literal 1 a, fin n) => [n]a -> [n]a
bsort lst = take `{n} (bb (lst#[-1,-1...]))
  where
    bb seq = if (seq@0) == -1 then seq else (bbl (head seq) (bb (tail seq)))
    bbl n seq =
       if ((seq@0) == -1) then [n]#[-1,-1...]   // sequence -1 … is the padding
       else if (n < (seq@0)) then [n]#seq
       else [(seq@0)]#(bbl n (tail seq))
```

Function `bbl` takes a literal `n` and an ordered sequence `seq` and places `n` in `seq` so that `seq` remains ordered. If `seq` is empty `[n]` (plus padding) is returned. If `n` is less than the first element of `seq` then `n` is added to the beginning of `seq`. Otherwise, `bbl` is called recursively and returns an ordered sequence whereupon the topmost stacked literal from `seq` is pre-pended to the result and returned. Function `bb` takes a padded sequence `seq` as input and recursively calls `bbl` where `n` is the first literal of `seq`. The recursion sets up a stack of literals to add to the output list and begins to establish an output list when `seq` is only the padding. Thus, when the stacked literals are ready to be added to the output list, what they are added to is ordered and will remain ordered as stacked literals are added to the output list. Function `bsort` just calls `bb` on the infinitely long padded input sequence. The output of `bb` is trimmed by `take`. As before, since -1 has a special role, it cannot be considered an element of the input `lst`. The following function will ensure this is the case so proofs will complete as expected:

```
valid lst = z ! 0
  where
    z = [True]#[~(x == -1) /\ y | x <- lst | y <- z ]
```

The following proves that `bsort lst` is a permutation of `lst`, one of two properties that a sorting algorithm must have, the second being that `bsort lst` must be non-decreasing.

```
isAPerm : [6][32] -> Bit
property isAPerm lst = ~(valid lst) \/ perm lst (bsort lst)
```

Here is the proof for 32 bit non-negative numbers in a sequence of 6:

```
Main> :l perm.cry
Loading module Cryptol
Loading module Main
Main> :s prover=any
Main> :prove isAPerm
Q.E.D.
(Total Elapsed Time: 1.647s, using "CVC4")
```

Since `perm` defines an equivalence class, every element in `lst` is also in `bsort lst` as many times as that element is in `lst`, and any element not in `lst` is not in `bsort lst`. The following is `True` if and only if for any two valid consecutive numbers in lst, the earlier one is no greater than the later one:

```
nondecreasing : [6][32] -> Bit
nondecreasing lst = z ! 0
  where
    z = [True]#[ if (j1 <= j2) /\ k then True else False
               | k <- z | j1 <- lst | j2 <- drop `{1} lst]
```

The second half of the proof that `bsort` sorts correctly:

```
bsortIsOrdered : [6][32] -> Bit
property bsortIsOrdered lst = ~(valid lst) \/ nondecreasing (bsort lst)
```

Proving:

```
Main> :l perm.cry
Loading module Cryptol
Loading module Main
Main> :s prover=any
Main> :prove bsortIsOrdered
Q.E.D.
(Total Elapsed Time: 0.476s, using "CVC4")
```

Thus, at least for length 6 lists of 32 bit numbers, bsort correctly sorts its input as long as -1 is not an element of the input list.

The “golden” specifications of `perm`, `non-decreasing`, and, if needed, `valid` can be used for other implementations of sorting algorithms including those where infinite padding with `-1` is not used. The next example uses a process similar to that of bsort but without the need for infinite sequences. The process makes use of a function `round` that moves greater numbers to the right and lesser numbers to the left. Running `round` as many times as there are numbers in the input sequence sorts it. Here is an outline of the operation of `round`. Let `lst` be the sequence to sort and let `acc` initially be a sequence of `-1`s of the same length as `lst`. In a `round`, elements of `lst` are compared with elements of `acc`, starting from the right (greatest elements are on or gravitate toward the right). If the `lst` element is greater than the existing `acc` element, the `acc` element moves down (to the left) by 1 position to make way for the `lst` element which is placed where the `acc` element had been. If the `lst` element is not greater then it is placed before the `acc` element it is compared with (to the left of the acc element where it replaces a `-1`) . Running a number of rounds equal to the length of `lst` sorts `lst`. The empty sequence is not allowed as input.

```
round lst acc = (func lst ((length lst)-1) acc)
  where
    func lt i ac =
      if (i == -1)
         then ac
         else if ((last ac) == 0)
              then func lt (i-1) (insert ac (lt@i) i)
              else if (lt@i) > (ac@(i+1))
                   then func lt (i-1) (insert (insert ac (ac@(i+1)) i) (lt@i) (i+1))
                   else func lt (i-1) (insert ac (lt@i) i)
```

Function `round` makes use of function `insert` which takes a sequence `lst`, a number `m`, and an index `i` and replaces the i^{th} element in `lst` with `m`.

```
insert lst m i = [ if k==i then m else x | x <- lst | k <- [0...]]
```

The following function initializes acc to a sequence of `-1`s of length lst:

```
makeNegs lst = [ -1 | i <- lst ]
```

Here is the sorting wrapper:

```
sort lst =  z ! 0
  where
    z = [ round x (makeNegs lst)  | x <- [lst]#z | i <- lst ]
```

So `x` contains the result of the previous `round` and `i` controls the comprehension to complete in length of `lst` iterations.

Permutations:

```
Loading module Cryptol
Cryptol> :l a.cry
Loading module Cryptol
Loading module Main
Main> :s prover=cvc4
Main> :prove isAPerm
Q.E.D.
(Total Elapsed Time: 3.560s, using "CVC4")
```

Non-decreasing:

```
Main> :prove sortIsOrdered
Q.E.D.
(Total Elapsed Time: 1.365s, using "CVC4")
```

The function `round` as written recursively above may be written as a comprehension like this:

```
round : {n, b} (Literal 0 b, Cmp b, fin n) => [1 + n]b -> [1 + n]b
round lst = z ! 0
  where
    z = [q]#[ if lst@i > acc@(i+1)
              then (insert (insert acc (acc@(i+1)) i) (lst@i) (i+1))
              else (insert acc (lst@i) i)
            | acc <- z | i <- (reverse (take `{n} [0...])) ];
    q = insert (makeNegs lst) (last lst) ((length lst)-1)
```

There are a few interesting things about this. First, this follows immediately from the recursive `round`: no changes were made in the `then` and `else` snippets, and `i` corresponds completely to the `i` of the recursive version. Second, note the comprehension easily

accommodates the initial state of `acc` so `acc` is not needed as an input and function `sort` looks like this for the comprehension solution:

```
sort lst =  z ! 0
  where
    z = [ round x | x <- [lst]#z | i <- lst ]
```

Third, note that Cryptol does not accept statements like `i <- [0..(length n)]` and is especially fretful about statements like `i <-[n,n-1..0]`. However, the type signature of `round` is user crafted, is added as `round` is defined, and contains `n` which relates to the range of values that `i` needs to take. Function `round` uses `(take `{n} [0...])` to get `[0..n]`, which is needed, but the `i` needs to be taken in reverse order so `i <- (reverse (take `{n} [0...]))` is used – pretty tricky. Finally, proofs appear to run faster with comprehensions than with recursive implementations even though they are doing the same thing from an abstract perspective. Consider these times:

comprehension solution

```
// isAPerm : [6][32] -> Bit
Main> :l sort1.cry
Loading module Cryptol
Loading module Main
Main> :prove isAPerm
Q.E.D.
(Total Elapsed Time: 0.235s, using "CVC4")
// sortIsOrdered : [6][32] -> Bit
Main> :l sort1.cry
Loading module Cryptol
Loading module Main
Main> :prove sortIsOrdered
Q.E.D.
(Total Elapsed Time: 0.063s, using "CVC4")
```

recursive solution

```
// isAPerm : [6][32] -> Bit
Main> :l sort.cry
Loading module Cryptol
Loading module Main
Main> :prove isAPerm
Q.E.D.
(Total Elapsed Time: 3.187s, using "CVC4")
// sortIsOrdered : [6][32] -> Bit
Main> :l sort.cry
Loading module Cryptol
Loading module Main
Main> :prove sortIsOrdered
Q.E.D.
(Total Elapsed Time: 1.395s, using "CVC4")
```

As a final example consider `mergesort`. Here is an implementation:

```
splito : {n,a} [n]a -> [inf]a
splito lst = [ lst@i | i <- [1,3...] ]
```

The input to `splito` is a sequence of numbers (as used below the sequence contains numbers to be sorted followed by infinitely many `-1`s). The output is the sequence consisting of the numbers of `lst` that are to be sorted and are located at odd indices, followed by infinitely many `-1`s.

```
splite : {n,a} [n]a -> [inf]a
splite lst = [ lst@i | i <- [0,2...] ]
```

Input to `splite` is like splito except the output is the sequence consisting of the numbers of `lst` that are to be sorted and are located at even indices, followed by infinitely many `-1`s.

```
merge : {a} (Literal 1 a, Ring a, Cmp a) => [inf]a -> [inf]a -> [inf]a
merge px py =
      if px@0 == -1 then py
      else if py@0 == -1 then px
      else if px@0 < py@0 then [px@0]#(merge (tail px) py)
      else [py@0]#(merge px (tail py))
```

Input to `merge` is two infinite sequences `px` and `py` of numbers, each in non-decreasing order, then padded with infinitely many `-1`s. The output is an infinite, non-decreasing sequence containing all of `px` and `py`, then padded with infinitely many `-1`s. The padded infinite

sequences px and py are merged, up to the padding. If px@0 is -1, meaning that px is only padding, then py is returned, if py@0 is -1, meaning py is only padding, then px is returned, otherwise if px@0 is less than py@0 then px@0 is concatenated with (merge (tail px) py) and returned, otherwise py@0 is concatenated with (merge px (tail py)) and returned.

```
mergesort : {cnt, a} (fin cnt, fin a, a >= 1, cnt >= 0) => [cnt][a] -> [cnt][a]
mergesort lst = take `{cnt} (mergesrt ax)
  where
    mergesrt p =
       if p@0 == -1 \/ p@1 == -1
       then p
       else merge (mergesrt (splite p)) (mergesrt (splito p))
    ax = lst#[-1,-1 ...]
```

Input to mergesort is a finite sequence lst of numbers (the empty sequence is allowed due to cnt >= 0 in the type signature). Output of mergesort is a finite sequence of numbers that is a permutation of lst and in non-decreasing order. Sequence ax is lst padded with -1s and becomes input p to function mergesrt which sorts p up to the padding, recursively merging the mergesrt of the even elements of p with the mergesrt of the odd elements of p. Here are some results:

```
mergesort [] is []
mergesort [16] is [16]
mergesort [2,1,2,3] is [1,2,2,3]
mergesort [8,2,1,9,5,4,6,7] is [1,2,3,4,5,6,7,8,9]
```

Function mergesort can be deemed correct for sequences of six 32 bit integers if the following two properties hold:

```
isAPerm : [6][32] -> Bit
property isAPerm lst = ~(valid lst) \/ perm lst (mergesort lst)
```

and

```
mergesortIsOrdered : [6][32] -> Bit
property mergesortIsOrdered lst = ~(valid lst) \/ nondecreasing (mergesort lst)
```

Here are the proofs:

```
mrgsrt> :l mrgsrt.cry
Loading module Cryptol
Loading module mrgsrt
mrgsrt> :s prover=cvc4
mrgsrt> :prove isAPerm
Q.E.D.
(Total Elapsed Time: 0.573s, using "CVC4")
mrgsrt> :prove mergesortIsOrdered
Q.E.D.
(Total Elapsed Time: 0.175s, using "CVC4")
```

Observations:

- Proofs of a collection of carefully chosen properties for specifications can be used to prove correctness of implementations. For example, in this case, perm was shown to be an equivalence relation which is necessary for correctness of sorting algorithms.
- A collection of carefully chosen specifications can be used as a “gold standard” for proving properties of implementations. For example, in this case, several sorting

implementations were shown correct from the same set of specifications. This is important because considerable time and effort can then be used to establish specifications and prove specification properties and those resources can be amortized over numerous implementations.

- It appears that specifications written as comprehensions will prove faster than specifications written recursively. One such example was presented but this observation applied to others that were not presented here.
- The monomorphic requirement of Cryptol for proving theorems can sometimes uncover subtle errors that would otherwise be missed. For example, in this case the specification of `perm` was wrong, as was discovered through the Cryptol typing constraints, but was compensated for by defining a function to `validate` inputs to `perm`.
- A type variable in a type signature may be used to return a portion of an infinite, padded sequence. For example, in this case mergesort, which is applied to finite sequences, was written using infinite padded sequences operated on by another procedure mergesrt. The signature for mergesort was user crafted to be:

  ```
  mergesort : {cnt, a} (fin cnt, fin a, a >= 1, cnt >= 0) => [cnt][a] -> [cnt][a]
  ```

  which allowed mergesort to be written like this:

  ```
  mergesort lst = take `{cnt} (mergesrt ax)
  ```

  which means, although `ax` is the infinite `lst#[-1,-1…]`, only the first `cnt` elements of the infinite output of `(mergesrt ax)` are presented as output by `mergesort` and are the elements of `lst` sorted.
- Similarly, suppose `cnt` needs to be passed to a function. It can be done like this:

  ```
  func a b `(cnt)
  ```

  That is, `cnt` in parens instead of curly braces.