Some heuristics for deriving invariant LN sections 6.7 , 6.8

Overview

- Heuristic 1: “Replace constant with counter” heuristic
- Heuristic 2: incrementally building the invariant
- Heuristic 3: tail invariant
- Other: an example of using invariant to do data refinement

Loop Reduction rule

$P \Rightarrow I$ // setting up I (Init)

$\{* g \wedge I *\}\ S\ \{* I *\}$ // invariance (IC)

$I \wedge \neg g \Rightarrow Q$ // exit cond (EC)

$\{* I \wedge g *\}\ C:=m; S\ \{* m<C *\}$ // m decreasing (TC1)

$I \wedge g \Rightarrow m > 0$ // m bounded below (TC2)

$\{* P *\}$ **while** g **do** S $\{* Q *\}$

Heuristic "replace constant with counter"

```
{* 0≤n *}

 i := 0 ; r := true ;
 while  i<n  do  {
       r := r /\ (a[i]=0)  ;  i++  }

{*  r  = (∀k : 0≤k<n : a[k]=0) *}
```

(1) The loop iterates on the counter up to some upper bound or lower bound.

- Identify this upper/lower bound. Suppose this is **n**.
- Identify the counter. Suppose this is **i**.

(2) If the post-cond is of the form **Q(n),** we try **Q(i)** /\ "range" as the invariant, where "range" is a formula stating the valid range of the counter i during the loop. Notice with this choice of invariant you get Q(i) /\ i=n ⇒ Q(n) for free.

Heuristic "replace constant with counter"

```
{* 0≤n *}

 i := 0 ; r := true ;
 while  i<n  do  {
       r := r /\ (a[i]=0)  ;  i++  }

{*  r  = (∀k : 0≤k<n : a[k]=0) *}
```

$(r = (\forall k : 0 \le k < i : a[k]=0)) \ \wedge \ 0 \le i \le n$

(we already did this example, now you know what inspired the choice of this invariant)

Let's take an example, now with multi vars

- Consider this well known recursive function to calculate the n^{th} Fibonacci number:

  fib 0 = 0
  fib 1 = 1
  fib (n + 2) = **fib** (n+1) + **fib** n

- The definition is nice but it has bad execution time. Next, we will consider an alternative, more efficient, implementation of the calculation.

Fibonacci's original problem

- Fibonacci's original problem is to calculate the number of rabbits given a certain growth model :
  1. We start with the first moth with one new pair of rabbits.
  2. A new pair needs 1 month to become adult
  3. Each month each pair of adult rabbits produces a new pair.

  We assume no rabbit dies during the experiment.

- This model can be directly mapped to an imperative implementation (next slide).

Iterative Fib, with O(n) runtime

```
RabitSim (n:int) : int  {
   nNow,nAdult,newPairs,t : int ;
   t:=1 ; nAdult:=0 ; nNow:=1 ;
   while t < n do {
      newPairs := nAdult ;
      nAdult      := nNow ;
      nNow        := nNow + newPairs ;
      t := t + 1
    } ;
return nNow }
```

pre : n > 0
post : return = fib(n)

Reducing the spec. to the statement level

{* n > 0 *}

```
t:=1 ; nAdult:=0 ; nNow:=1 ;
while t < n do {
    newPairs := nAdult ;
    nAdult      := nNow ;
    nNow        := nNow + newPairs ;
    t := t + 1
 } ;
 return := nNow
```

{* return = fib n *}

After some wp calculation

{* n > 0 *}

```
t:=1 ; nAdult:=0 ; nNow:=1 ;
while t < n do {
    newPairs := nAdult ;
    nAdult   := nNow ;
    nNow     := nNow + newPairs ;
    t := t + 1
  } ;
```

{* nNow = fib n *}

Applying the previous heuristic

```
{* n > 0 *}

t:=1 ; nAdult:=0 ; nNow:=1 ;
{* I *}
while t < n do {
    newPairs := nAdult ;
    nAdult   := nNow ;
    nNow     := nNow + newPairs ;
    t := t + 1  } ;

{* nNow = fib n  *}
```

Inv : nNow = fib t
/\ 1 ≤ t ≤ n
term. metric : n-t

Notice that with this choice of invariant, the exit condition "I /\ ¬g ⇒ Q" is quite trivially satisfied.

Let's take a look at the PIC part

- "PIC" : proof of the invariance condition. That is to prove that {* I /\ g *} S {* I *} holds, where S is the loop's body. In other words, to prove that I /\ g ⇒ **wp** S I is valid.
- Calculating **wp** S I gives:

$$nNow + nAdult = fib\ (t+1) \quad /\backslash \quad 1 \leq t+1 \leq n$$

- The proof structure is as follows:

PROOF PIC
[A1] nNow = fib t
[A2] 1≤ t ≤ n
[A3] t < n
[G1] nNow + nAdult = fib (t+1)
[G2] 1≤ t+1 ≤ n

The proof of G1

PROOF EQUATIONAL

nNow + nAdult
= { A1}
fib t + nAdult
X = { ?? cannot find the justification }
fib t + fib (t-1)
= { def. of fib and t≥1 (A2) }
fib (t+1)

The proof fails because the invariant has no information about nAdult.
Let's extend it then, with the proper information. The new invariant:

$$nNow = fib\ t \ /\backslash\ nAdult = fib\ (t-1) \ /\backslash \quad 1 \leq t \leq n$$

In general, for every variable involved in a loop, that contributes to its post-condition, we will need to capture its property in the invariant.

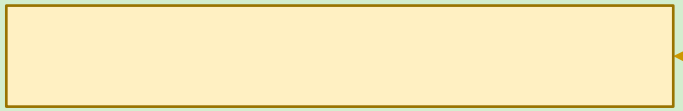
The new PIC

- Since you change the invariant, you will have to rework your proofs to reflect the change. For PIC, since you change the invariant by strengthening it with a new conjunct, this means that we can now assume more, though you also have to prove more.
- Re-calculating **wp** S I gives:

$$
\begin{aligned}
&nNow + nAdult = fib\ (t+1) \\
&/\backslash\ nNow = fib\ t \\
&/\backslash \quad 1 \leq t+1 \leq n
\end{aligned}
$$

The new PIC

PROOF PIC
[A1] nNow = fib t
[A1b] nAdult = fib (t-1)
[A2] 1≤ t ≤ n
[A3] t < n
[G1] nNow + nAdult = fib (t+1)
[G1b] nNow = fib t
[G2] 1≤ t+1 ≤ n
BEGIN
1. { see the subproof below } G1



2. { A1 } G1b
3. { follows from A2 and A3 } G2
END

PROOF EQUATIONAL
nNow + nAdult
= { A1}
fib t + nAdult
= { A1b }
fib t + fib (t-1)
= { def. of fib and t≥1 (A2) }
fib (t+1)

Tail invariant

- Consider again this previous example:

{* 0≤n *}

```
i := 0 ; r := true ;
while  i<n  do  {
      r := r /\ (a[i]=0)  ;  i++  }
```

{* r = (∀k : 0≤k<n : a[k]=0) *}

- We proved this with this as the invariant:

r = (∀k : 0≤k<**i** : a[k]=0) /\ 0≤i≤n

Tail invariant

- The invariant we used:

$$r = (\forall k : 0 \le k < i : a[k]=0) \ \wedge \ 0 \le i \le n$$

capturing the work that **has been done**

- An alternative invariant :

capturing the work that **still to be done**

$$r \wedge (\forall k : i \le k < n : a[k]=0) \ = \ (\forall k : 0 \le k < n : a[k]=0) \ \wedge \ 0 \le i \le n$$

A “tail invariant” expresses the invariant in terms of the remaining work that is still to be done. The loop works on shrinking this to-be-done part until it disappears, or become small enough it can be computed directly without a loop.

Example: iterative impl. of tail recursion

- Example of “typical recursion” :

  g x = **if** x≤0 **then** x **else** 1+g(x-1)

- A tail recursive function does not combine the result of its recursion:

  f x = **if** x≤0 **then** x **else** f(x-1)

Examples

- Get the last element of a list:

```
last [x]      =  x
last (x:y:s)  =  last (y:s)
```

- Summing the elements of a list:

```
lsum a [ ]      =  a
lsum  a (x:s)   =  lsum (x+a) s
```

- Reversing a list (more efficient in Haskell) :

```
rv t [ ]      =  t
rv t (x:s)    =  rv (x:t) s
```

Tail Recursion, general form

- A tail recursive function **F**:A→B has this general form:

```
F x  =  g x  →  base x
        |       F (Δ x)
```

// base case
// recursion

- Such a function can be optimized by implementing the recursion as loop-iterations, as the latter does not use stack space to pass around parameters.
- **Termination**. **F** terminates if we can find a function m:A→Int such that:

1. ¬ g x ⇒ m (Δ x) < m x
2. ¬ g x ⇒ m x > 0

Iterative implementation

- **Problem**: given α, calculate F α. Recall the def. of F:

$$\begin{array}{lll} F\ x & = & g\ x \rightarrow base\ x \\ & & \mid\ F\ (\Delta\ x) \end{array}$$

- Consider this simple loop to calculate F α :

{* true *}

x := α ;
while $\neg$ g x **do** x := Δ x ;
r := base x

{* r = F α *}

To prove that this works, we will use the following tail invariant:

F x = F α

For termination we use m x as the termination metric where m is the function you used to prove the termination of F (prev. slide)

Proof sketch

{* true *}

x := α ;
while $\neg$ g x **do** x := Δ x ;
r := base x

{* r = F α *}

Invariant: F x = F α

- When the loop terminates, g x holds. So by its definition F x = base x. Then the invariant implies that base x = F α.
- For the proof of invariance: **wp** (x := Δ x) inv gives F (Δx) = F α. But since $\neg$ g x holds, F (Δx) is also equal to F x, by the def. of F. So the wp can be reduced to F x = F α, which is exactly the invariant itself.

Corollary

If a problem can be expressed as computing the result of a tail recursive function, the previous implementation scheme, using the previously given invariant, provides a correct iterative program to compute the solution of the problem.

Example : GCD

- Given X,Y > 0, compute gcd X Y.

- Straight forward solution: O(X min Y).
- We can do better than that (Euclides, 320 BC)

- Approach: try to cast the problem into tail recursion.
- We will need to explore some properties of common divisors.

Some properties of gcd

- We'll assume **positive integers**! Define :

  d divides x = $\exists k : k>0 : x = k*d$

  d comdiv x,y = d divides x /\ d divides y

- Theorem. If x>y we have:

  d comdiv x,y $\Leftrightarrow$ d comdiv (x-y),y

  \\ hmm … looks like a "split"

Some properties of gcd

- We can characterize **gcd** via this equation :

$$\alpha = \textbf{gcd}\ x\ y \quad = \quad \alpha\ \underline{\text{comdiv}}\ x,y \ \wedge\ (\forall e : e\ \underline{\text{comdiv}}\ x,y : e \leq \alpha)$$

- Theorem-1. for positive integers x and y, if x>y we have: **gcd** x y = **gcd** (x-y) y

- Theorem-2. for positive integer x: **gcd** x x = x

Gcd in tail recursion

- Those properties of **gcd** leads to the following recursive relation for **gcd** . Notice that the function is tail recursive:

```
gcd  x  y   =    x = y   →   x
                 |   x > y   →   gcd (x-y)  y
                 |   /* x < y  */   gcd x  (y – x)
```

- Consider the problem of calculating **gcd** X Y, given some positive integers X and Y. The previous implementation scheme gives us a solution for this.

Iterative implementation of Gcd

- Recall the recursive definition:

```
gcd x y  =   x = y   →  x
           |  x > y   →  gcd (x-y) y
           |  /* x < y */  gcd x (y – x)
```

- **Problem**: given positive integers X,Y calculate gcd X Y. Implemented by this loop:

```
{* X>0 /\ Y>0 *}
x,y := X,Y ;
while x ≠ y do
  if x>y then x:=x-y
         else y:=y-x
{* x = gcd X Y *}
```

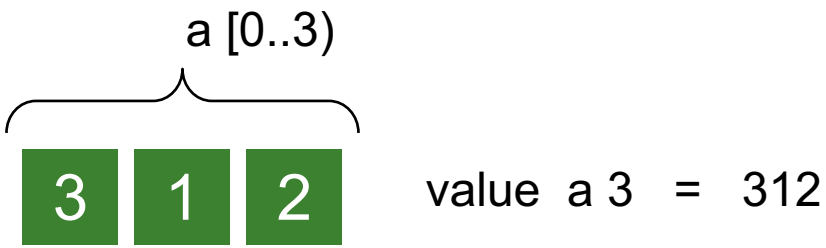
inv : **gcd** x y = **gcd** X Y

term. metric : x+y

Note: to prove that this termination metric has a lower bound 0, we need to extend the invariant with x+y ≥ 0

Another example

- Consider a list of non-negative integers representing a number.



  Write a program that computes the "value" of the number represented by the list.

- A recursive solution:

```
val x s  =  s=[ ] → x
                  | val (10*x + hd s)  (tail s)
```

  The value of a list s can be calculated by **val** 0 s.

Iterative implementation

- Recall again the tail recursive function val:

```
val x s = s=[ ] → x
                | val (10*x + hd s) (tail s)
```

- **Problem**: given an S, calculate **val** 0 S. Iterative impl.:

{* true *}

```
x , s := 0 , S ;
while s≠[ ] do {
    x := 10*x + hd s ;
    s := tail s }
```

{* x = val 0 S *}

inv : **val** x s = **val** 0 S

term. metric : #s

Data refinement, an example

- Consider again the previous program:

```
{* true *}
x , s  :=  0 , S ;
while  s≠[ ]  do  {
    x  :=  10*x + hd s   ;
    s := tail s  }
{* x = val 0 S *}
```

- What if the input list “S” is actually an array a[0..n) ? You might foresee that hd s and tail s can then be implemented more efficiently by simply shifting a cursor/counter into the array. How to justify the correctness of such a transformation?

Transformation 1

The same program, but with “S” instantiated to a[0..n):

{* 0≤n *}

```
x , s  :=  0 , a[0..n)
i := 0
while  s≠[ ]  do  {
    x  :=  10*x + hd s
    s := tail s
    i := i+1
}
```

{* x = val 0 (a[0..n)] *}

inv_1 : val x s = val 0 (a[0..n))

We will introduce a new variable i and program it so that we can maintain the following “data invariant” that captures the relation between the two data structures (the array a and the list s):

inv_2 : s = a[i..n) /\ 0≤i≤n

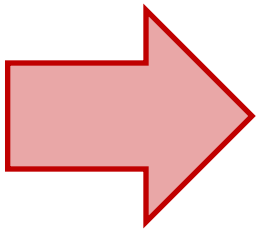
Notice that the new code does not change the behavior of the base program. So it does not break its correctness argument. This is also called “superposition” of a new variable (in this example: i).

Transformation 2

{* 0≤n *}

```
x , s  :=  0 , a[0..n)
i := 0
while  s≠[ ]  do  {
    x  :=  10*x + hd s
    s := tail s
    i := i+1
}
```

{* x = val 0 (a[0..n)] *}



{* 0≤n *}

```
x , s  :=  0 , a[0..n)
i := 0
while  i ≠ n  do  {
    x  :=  10*x + a[i]
    s := tail s
    i := i+1
}
```

{* x = val 0 (a[0..n)] *}

inv_1: **val** x s = **val** 0 (a[0..n))

inv_2 : s = a[i..n) /\ 0≤i≤n

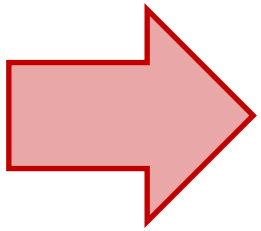
The data invariant implies that s≠[ ] in the original program is equivalent to i≠n, and can thus be replaced by the latter. Similarly, hd s can be replaced with a[i].

Transformation 3

{* 0≤n *}

x , ~~s~~ := 0 , ~~a[0..n)~~
i := 0
while i ≠ n **do** {
  x := 10*x + a[i]
  ~~s := tail s~~
  i := i+1
}

{* x = val 0 (a[0..n)] *}



{* 0≤n *}

x := 0
i := 0
while i ≠ n **do** {
  x := 10*x + a[i]
  i := i+1
}

{* x = val 0 (a[0..n)] *}

Finally, notice that we don't actually need s anymore (it has no influence towards the result x), so we can just as well drop it.