Algorithms correct by construction
à la JRA
using EB2Algo

Dominique Cansell

Neeraj Kumar Singh

June 2024

- JRA proposed many examples of algorithms construction in 2001 with his 2 merges events (algo)

- JRA gave lecture "Analysing and Constructing Computer Programs" from 2014 to 2019

- No many other case study

- No tool to apply JRA's rules except an interactive one but not complete in Shanghai (a student of Li Quin)

- JRA asked me to work again on this topics

- more EBRP algorithms: compute the n first prime numbers (many versions), bubbelsort , Dutch Flag, Better decomposition, quick sort, merge sort.

- instantiation plugin was used for prime numbers ($\neg finite(Prime)$) quicksort (variant), Better decomposition

- After these Rodin developments, I asked Neeraj to develop a tool to produce an algorithm

```
p[0]= 0; p[1]= n; t = 1;
for(i=0;i<n;i=i+1) g[i]=f[i];
while (t!=0){
   if (p[t-1]==p[t]-1){ t=t-1;}
   else {
            a=p[t-1]; b=p[t]-1; c=g[(a+b)/2];
            while (a<b)
               if (g[a]<c) {a=a+1;}
               else if (g[b]>c) {b=b-1;}
               else {v=g[a];g[a]=g[b];g[b]=v;a=a+1;b=b-1;}
            if (b<a) {k=b;}
            else if (g[a]<=c) {k=b;} else {k=a-1;};
            p[t+1]=p[t];
            p[t]=k+1;
            t=t+1;
     }
};
}
```

What do this program

- It's a sorting program
- at end g contains all values of f but in the order
- a permutation on index of f gives the order
- not easy to see this

It’s a sequential version
of the famous quicksort

final
any
PI
where
$PI \in 0..n-1 \rightarrowtail\!\!\!\twoheadrightarrow 0..n-1$
$\forall i, j \cdot$
$i \in 0..n-1 \land j \in i..n-1$
$\Rightarrow$
$f(PI(i)) \leq f(PI(j))$
then
$g := PI; f$
end

After seven or eight refinements: 10 events

final
when
$topI = 0$
end

choiceI
when
$boolchI = 0$
$topI \neq 0$
$stack(topI - 1)$
$\neq stack(topI) - 1$
then
$boolchI := 1$
$binf := stack(topI - 1)$
$bsup := stack(topI) - 1$
$pv := g((stack(topI - 1)$
$+stack(topI) - 1)/2)$
end

progress_singI
when
$topI \neq 0$
$stack(topI - 1)$
$= stack(topI) - 1$
then
$topI := topI - 1$
end

progress_binf
when
$boolchI \neq 0$
$binf < bsup$
$g(binf) < pv$
then
$binf := binf + 1$
end

progress_bsup
when
$boolchI \neq 0$
$binf < sup$
$g(binf) \geq pv$
$g(bsup) > pv$
then
$bsup := bsup - 1$
end

swap
when
$boolchI \neq 0$
$binf < bsup$
$g(binf) \geq pv$
$g(bsup) \leq pv$
then
$g := g \Leftarrow \{binf \mapsto g(bsup)$
$, bsup \mapsto g(binf)\}$
$binf := binf + 1$
$bsup := bsup - 1$
end

compute_K1
when
$boolchI \neq 0$
$boolk = 0$
$bsup < binf$
then
$K := bsup$
$boolk := 1$
end

compute_K2
when
$boolchI \neq 0$
$boolk = 0$
$bsup = binf$
$g(binf) \leq pv$
then
$K := bsup$
$boolk := 1$
end

compute_K3
when
$boolchI \neq 0$
$boolk = 0$
$bsup = binf$
$g(binf) > pv$
then
$K := binf - 1$
$boolk := 1$
end

progress
when
$boolk \neq 0$ $boolchI \neq 0$ $bsup \leq binf$
then
$stack := stack \lhd\!\!- \{topI \mapsto K + 1 \,,\, topI + 1 \mapsto stack(topI)\}$
$boolchI := 0$
$topI := topI + 1$
$boolk := 0$
end

$$boolchI \neq 0 \Rightarrow topI \neq 0 \wedge stack(topI - 1) \neq stack(topI) - 1$$

What about algorithm construction ?

- At the end of the refinement process we have many small guarded events
- JRA proposed to merge two events (or guarded algorithms) using two merging rules and an additional rule for init.
- when we merge two events we get a guarded algorithm
- some side condition are necessary

Merging Rule M_IF

when
P
Q
then
S
end
	when
	P
	$\neg\, Q$
	then
	T
	end
		$\leadsto$	**when**
			P
			then
			if Q **then**
			S
			else
			T
			end
			end
				M_IF

- Side Conditions:
  - Both events must have been introduced at the same refinement.

- Special Case: If P is missing the resulting "event" has no guard

Merging Rule M_WHILE


```
when          when
  P             P
  Q             ¬Q
then          then
  S             T
end           end
```

$\rightsquigarrow$

```
when
  P
then
  while Q do
    S
  end;
  T
end
```

M_WHILE

- Side Conditions:

  - P must be invariant under S

  - The first event must have been introduced at one refinement step below the second one.

- Special Case: If P is missing the resulting "event" has no guard

- when $\boldsymbol{P}$ is not invariant under $\boldsymbol{S}$ (or if you cannot prove $\boldsymbol{P}$) JRA said always use the **M_IF** rule.

- level of the merge event: it's the less one if the abstract one is the small one

- rules defined in 2001 before anticipated events (2005) then I have corrected a little level definition.

Merging Rule : level

- convergent level = level where the event is convergent convlevel(evt)

- definition level = level where the event is defined defevel(evt)

- level(evt)=convlevel(evt) $\mapsto$ defevel(evt)

- lexicographies order

- progress is defined and convergent at level 1, swap is defined at level $\mathbf{0}$ (g $:=$) and convergent at level $\mathbf{2}$

Merge compute_K2 and compute_K3

compute_K1
when
$boolchI \neq 0$
$boolk = 0$
$bsup < binf$
then
$K := bsup$
$boolk := 1$
end

compute_K2
when
$boolchI \neq 0$
$boolk = 0$
$bsup = binf$
$g(binf) \leq pv$
then
$K := bsup$
$boolk := 1$
end

compute_K3
when
$boolchI \neq 0$
$boolk = 0$
$bsup = binf$
$g(binf) > pv$
then
$K := binf - 1$
$boolk := 1$
end

Merge compute_K2 and compute_K3

compute_K1	compute_K2	compute_K3
when	**when**	**when**
$boolchI \neq 0$	$boolchI \neq 0$	$boolchI \neq 0$
$boolk = 0$	$boolk = 0$	$boolk = 0$
$bsup < binf$	$bsup = binf$	$bsup = binf$
$bsup \leq binf$	$bsup \leq binf$	$bsup \leq binf$
then	$g(binf) \leq pv$	$g(binf) > pv$
$K := bsup$	**then**	**then**
$boolk := 1$	$K := bsup$	$K := binf - 1$
end	$boolk := 1$	$boolk := 1$
	end	**end**

Merge compute_K2 and compute_K3

```
compute_K2_K3
  when
    boolchI ≠ 0
    boolk = 0
    bsup ≤ binf
    bsup = binf
  then
    if g(binf) ≤ pv then
      K := bsup
      boolk := 1
    else
      K := binf − 1
      boolk := 1
    end
  end
```

compute_K1_K2_K3

when

$boolchI \neq 0 \quad bsup \leq binf \quad boolk = 0$

then

if $bsup < binf$ **then**

$K := bsup$

$boolk := 1$

else

if $g(binf) \leq pv$ **then**

$K := bsup \,||\, boolk := 1$

else

$K := binf - 1 \,||\, boolk := 1$

end

end

end

Merge compute_K1_K2_K3 and progress

```
compute_K1_K2_K3_progress
  when
    boolchI ≠ 0  bsup ≤ binf
  then
    while  boolk = 0  do
      if  bsup < binf  thenK := bsup|| boolk := 1
      else  if  g(binf) ≤ pv  thenK := bsup|| boolk := 1
            elseK := binf − 1|| boolk := 1
            end
      end
    end ;
    stack := stack <+ {topI ↦ K + 1 , topI + 1 ↦ stack(topI)}
    boolchI := 0
    topI := topI + 1
  end
```

Merge compute_K1_K2_K3 and progress

A while with one turn in the loop ! $boolk$ can disapear.

compute_K1_K2_K3_progress
  when
    $boolchI \neq 0 \;\; bsup \leq binf$
  then
    if $bsup < binf$ **then** $K := bsup$
    else **if** $g(binf) \leq pv$ **then** $K := bsup$
        else $K := binf - 1$
        end
    end ;
    $stack := stack \Lleftarrow \{topI \mapsto K + 1\,,\, topI + 1 \mapsto stack(topI)\}$
    $boolchI := 0$
    $topI := topI + 1$
  end

- Neeraj's constraints: the tool need to be automatic. No interaction. No proof obligations can be generated

- I proposed to compute a guard tree

- guards need to be completed (RED GUARDS)

- the tool try to split a subset of events in two non empty subsets using a guard G.

- one subset with all events using G as guard and the other one using $\neg G$ as guard. Both subsets are splited using the same algorithm (recursive) without guards G and $\neg G$. All events are in the union of both subset (partition).

- when there is only one event all guards of this event are consumed.

- It's a backtracking algorithm but G is not choiced arbitrary. We take a guard inside an event which has the minimum number of guard (less tentatives). If G doesn't work we choice the next guard of the event.

- if the split abord using all guard of the event: A guard misses or there is a deadlock. It's the difficult part for a beginner.

Applying JRA's rules

On the guard tree we compute

- the subset of events
- A level: it's the minimum of all events levels of the subset. then we can decide to apply M_IF or perhaps M_WHILE
- the set of guards (above) which are preserved by all events in the subset (for a while)
- With these three attributs we can apply JRA's rules (down-top)

Applying JRA's rules

On the guard tree we compute

- the subset of events
- A level: it's the minimum of all events levels of the subset. then we can decide to apply M_IF or perhaps M_WHILE
- We compute the variables which doesn't change by all events in the algorithm (NO substitution on this variable $x := \ldots$) then a guard which uses only these variables is preserved. Other can change, we apply a M_IF
- With these four attributs we can apply JRA's rules (down-top)

EB2Algo: A guard tree

$$(topI = 0)[final]$$
$$(topI \neq 0)[progress; choiceI; progress\_binf; progress\_bsup; swap; progress\_singI; computeK1; computeK2; computeK3]$$

EB2Algo: A guard tree

$(topI = 0)$ $[final]$

$(topI \neq 0)$ $[progress; choiceI; progress\_binf; progress\_bsup; swap; progress\_singI; computeK1; computeK2; computeK3]$

$(stack(topI - 1) \neq stack(topI) - 1)[progress; choiceI; progress\_binf; progress\_bsup; swap; computeK1; computeK2; computeK3]$

$(stack(topI - 1) = stack(topI) - 1)[progress\_singl]$

EB2Algo: A guard tree

$(topI = 0)$ $[final]$

$(topI \neq 0)$ $[progress; choiceI; progress\_binf; progress\_bsup; swap; progress\_singI; computeK1; computeK2; computeK3]$

$(stack(topI-1) \neq stack(topI)-1)[progress; choiceI; progress\_binf; progress\_bsup; swap; computeK1; computeK2; computeK3]$

$(boolchI \neq 0)[progress; progress\_binf; progress\_bsup; swap; computeK1; computeK2; computeK3]$

$(boolchI = 0)[choiceI]$

$(stack(topI-1) = stack(topI)-1)[progress\_singl]$

EB2Algo: A guard tree

$(topI = 0)$ $[final]$

$(topI \neq 0)$ $[progress; choiceI; progress\_binf; progress\_bsup; swap;$
$progress\_singI; computeK1; computeK2; computeK3]$

$(stack(topI - 1) \neq stack(topI) - 1)[progress; choiceI; progress\_binf; progress\_bsup;$
$swap; computeK1; computeK2; computeK3]$

$(boolchI \neq 0)[progress; progress\_binf; progress\_bsup; swap;$
$computeK1; computeK2; computeK3]$

$(boolchI = 0)[choiceI]$

$\neg(binf < bsup)[progress; computeK1; computeK2; computeK3]$

$(boolk \neq 0)[progress]$

$(boolk = 0)[computeK1; computeK2; computeK3]$

$\neg(binf = bsup)[computeK1]$

$(binf = bsup)[computeK2; computeK3]$

$(g(binf)pv)[computeK2]$

$\neg(g(binf)pv)[computeK3]$

$(binf < bsup)[progress\_binf; progress\_bsup; swap]$

$(g(binf) < pv)[progress\_binf]$

$\neg(g(binf) < pv)[progress\_bsup; swap]$

$(g(bsup) > pv)[progress\_bsup]$

$\neg(g(bsup) > pv)[swap]$

$(stack(topI - 1) = stack(topI) - 1)[progress\_singl]$

$(topI = 0)\ \ [final]$

$(topI \neq 0)\ \ [progress; choiceI; progress\_binf; progress\_bsup; swap;$
$progress\_singI; computeK1; computeK2; computeK3]$

$(stack(topI - 1) \neq stack(topI) - 1)[progress; choiceI; progress\_binf; progress\_bsup;$
$swap; computeK1; computeK2; computeK3]$

$(boolchI \neq 0)[progress; progress\_binf; progress\_bsup; swap;$
$computeK1; computeK2; computeK3]$

$(boolchI = 0)[choiceI]$

$\neg(binf < bsup)[progress; computeK1; computeK2; computeK3]$

$(boolk \neq 0)[progress]$

$(boolk = 0)[computeK1; computeK2; computeK3]$

$\neg(binf = bsup)[computeK1]$

$(binf = bsup)[computeK2; computeK3]$

$(g(binf)pv)[computeK2]$

$\neg(g(binf)pv)[computeK3]$

$(binf < bsup)[progress\_binf; progress\_bsup; swap]$

$(g(binf) < pv)[progress\_binf]$

$\neg(g(binf) < pv)[progress\_bsup; swap]$

$(g(bsup) > pv)[progress\_bsup]$

$\neg(g(bsup) > pv)[swap]$

$(stack(topI - 1) = stack(topI) - 1)[progress\_singl]$

EB2Algo: A guard tree

$(topI = 0)$ $[final]$
$(topI \neq 0)$ $[progress; choiceI; progress\_binf; progress\_bsup; swap; progress\_singI; computeK1; computeK2; computeK3]$
$(stack(topI - 1) \neq stack(topI) - 1)[progress; choiceI; progress\_binf; progress\_bsup; swap; computeK1; computeK2; computeK3]$
$(boolchI \neq 0)[progress; progress\_binf; progress\_bsup; swap; computeK1; computeK2; computeK3]$
$(boolchI = 0)[choiceI]$
$\neg(binf < bsup)[progress; computeK1; computeK2; computeK3]$
$(boolk \neq 0)[progress]$
$(boolk = 0)[computeK1; computeK2; computeK3]$
$\neg(binf = bsup)[computeK1]$
$(binf = bsup)[computeK2; computeK3]$
$(g(binf)pv)[computeK2]$
$\neg(g(binf)pv)[computeK3]$
$(binf < bsup)[progress\_binf; progress\_bsup; swap]$
$(g(binf) < pv)[progress\_binf]$
$\neg(g(binf) < pv)[progress\_bsup; swap]$
$(g(bsup) > pv)[progress\_bsup]$
$\neg(g(bsup) > pv)[swap]$
$(stack(topI - 1) = stack(topI) - 1)[progress\_singl]$

EB2Algo: A guard tree

$(topI = 0)$ $[final]$

$(topI \neq 0)$ $[progress; choiceI; progress\_binf; progress\_bsup; swap;$
$progress\_singI; computeK1; computeK2; computeK3]$

$(stack(topI - 1) \neq stack(topI) - 1)[progress; choiceI; progress\_binf; progress\_bsup;$
$swap; computeK1; computeK2; computeK3]$

$(boolchI \neq 0)[progress; progress\_binf; progress\_bsup; swap;$
$computeK1; computeK2; computeK3]$

$(boolchI = 0)[choiceI]$

$\neg(binf < bsup)[progress; computeK1; computeK2; computeK3]$

$(boolk \neq 0)[progress]$

$(boolk = 0)[computeK1; computeK2; computeK3]$

$\neg(binf = bsup)[computeK1]$

$(binf = bsup)[computeK2; computeK3]$

$(g(binf)pv)[computeK2]$

$\neg(g(binf)pv)[computeK3]$

$(binf < bsup)[progress\_binf; progress\_bsup; swap]$

$(g(binf) < pv)[progress\_binf]$

$\neg(g(binf) < pv)[progress\_bsup; swap]$

$(g(bsup) > pv)[progress\_bsup]$

$\neg(g(bsup) > pv)[swap]$

$(stack(topI - 1) = stack(topI) - 1)[progress\_singl]$

$INITIALISATION$

$(topI = 0)\ \ [final]$

$(topI \neq 0)\ \ [progress; choiceI; progress\_binf; progress\_bsup; swap;$
$progress\_singI; computeK1; computeK2; computeK3]$

$(stack(topI - 1) \neq stack(topI) - 1)[progress; choiceI; progress\_binf; progress\_bsup;$
$swap; computeK1; computeK2; computeK3]$

$(boolchI \neq 0)[progress; progress\_binf; progress\_bsup; swap;$
$computeK1; computeK2; computeK3]$

$(boolchI = 0)[choiceI]$

$\neg(binf < bsup)[progress; computeK1; computeK2; computeK3]$

$(boolk \neq 0)[progress]$

$(boolk = 0)[computeK1; computeK2; computeK3]$

$\neg(binf = bsup)[computeK1]$

$(binf = bsup)[computeK2; computeK3]$

$(g(binf)pv)[computeK2]$

$\neg(g(binf)pv)[computeK3]$

$(binf < bsup)[progress\_binf; progress\_bsup; swap]$

$(g(binf) < pv)[progress\_binf]$

$\neg(g(binf) < pv)[progress\_bsup; swap]$

$(g(bsup) > pv)[progress\_bsup]$

$\neg(g(bsup) > pv)[swap]$

$(stack(topI - 1) = stack(topI) - 1)[progress\_singl]$

Conclusion

- EB2Algo was developed by Neeraj under the umbrella of EB2ALL
- EB2Algo works on all JRA’s examples (RED GUARDS are added using a refinement)
- EB2Algo works on all my EBRP models
- ask Neeraj for a demo
- https://www.irit.fr/EBRP/software