Update on Strict Sequential Constructiveness

Alexander Schulz-Rosengarten
Reinhard von Hanxleden and Michael Mendler
Kiel University / Bamberg University

Constructive Sequential Constructiveness

A very much work in progress report …

Reinhard von Hanxleden
Kiel University

Based on Discussions with Marc Pouzet, Timothy Bourke (ENS)
and Michael Mendler (U Bamberg)
Funded by DFG PRETSY

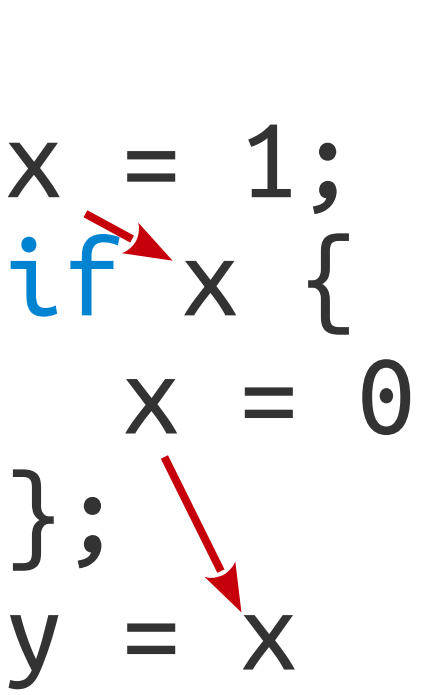
SYNCHRON Workshop, Kiel, December 2015

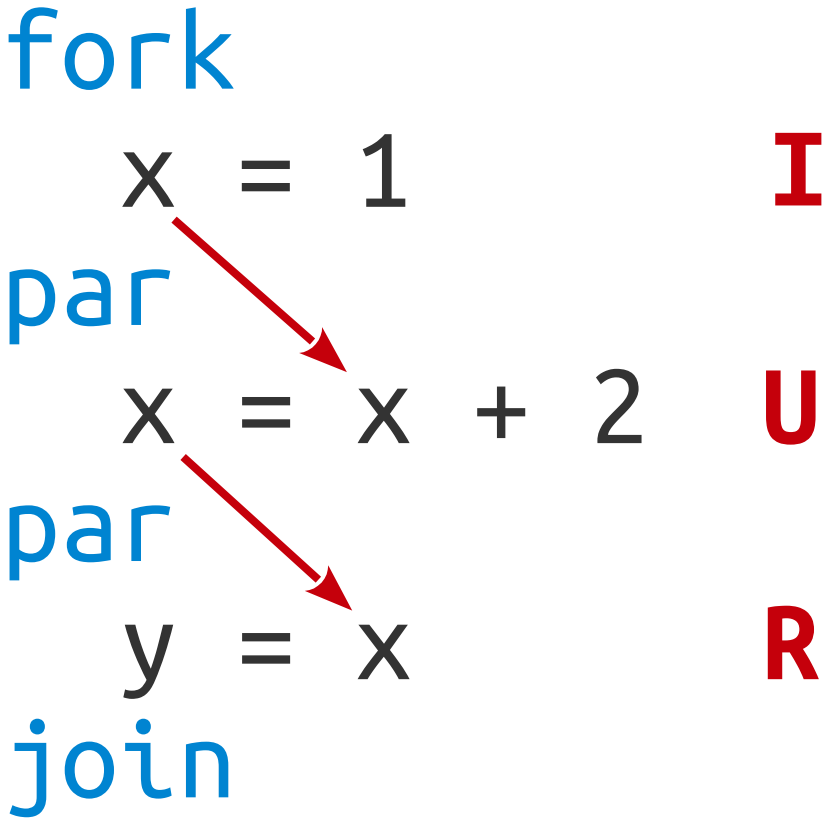
Sequential Constructiveness / SCL

```
x = 1;
if x {
   x = 0
};
y = x
```

- Variables
- Sequence
- Conditional
- Fork-par-join
- Goto
- Pause

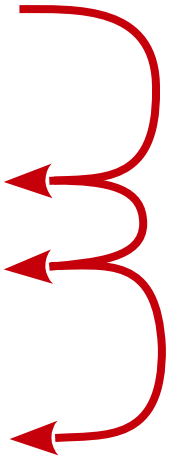
Sequential Constructiveness / SCL





Recall P10

```
int x, y;
{
   y = 0;                  //S0
   fork
      x = 1;               //S1
      y = x                //S2
   par
      if y == 0 {          //S3
         x = 0             //S4
      }
   join
}
```



→ Speculation about execution of S4

Thou shalt not Speculate

Strict Sequential Constructiveness
is
Sequential Constructiveness without speculation

Idea: Ground SC in constructiveness in the spirit of Esterel (Berry-Constructiveness)

Restricting Sequential Constructiveness

- Use Esterel for analysis and code generation
- Requires transformation from SCL to Esterel

P10

```
int x, y;
{
  y = 0;
  fork
    x = 1;
    y = x
  par
    if y == 0 {
      x = 0
    }
  join
}
```


```
int x0, x1, y0, y1;
{
  y0 = 0;
  fork
    x0 = 1;
    y1 = ?(x0, x1)
  par
    if y1 == 0 {
      x1 = 0
    }
  join
}
```

P10

```
int x, y;
{
  y = 0;
  fork
    x = 1;
    y = x
  par
    if y == 0 {
      x = 0
    }
  join
}
```


```
int x0, x1, y0, y1;
{
  y0 = 0;
  fork
    x0 = 1;
    y1 = conc(x0, x1)
  par
    if y1 == 0 {
      x1 = 0
    }
  join
}
```

Variable Encoding

SCL	Esterel
`x = true`	`emit xp;` `emit x;`
`x = false`	`emit xp;`

`x = conc(..)`	`present conflict then`
	`  emit error;`
	`else`
	`  emit xp;`
	`  present value then`
	`  emit x`
	`  end`
	`end`

P10

```
int x0, x1, y0, y1;
{
  y0 = 0;
  fork
    x0 = 1;
    y1 = conc(x0, x1)
  par
    if y1 == 0 {
      x1 = 0
    }
  join
}
```


```
signal x0p, x0, x1p, x1 in
signal y0p, y0, y1p, y1 in
signal error in
[
  emit y0p;
  [
    emit x0p;
    emit x0;
    present x0p and x1p and
            ((x0 and not x1) or
             (not x0 and x1)) then
      emit error
    else
      emit y1p;
      present (x0p and x0) or
              (x1p and x1) then
        emit y1
      end
    end
  ||
    present y1p and (not y1) then
      emit x1p
    end
  ]
||
  signal err in
    present error then
      present err else emit err end
    end
  end signal
]
```

P10

```
int x0, x1, y0, y1;
{
  y0 = 0;
  fork
    x0 = 1;
    y1 = conc(x0, x1)
  par
    if y1 == 0 {
      x1 = 0
    }
  join
}
```


```
signal x0p, x0, x1p, x1 in
signal y0p, y0, y1p, y1 in
signal error in
[
  emit y0p;
  [
    emit x0p;
    emit x0;
    present x0p and x1p and
            ((x0 and not x1) or
             (not x0 and x1)) then
      emit error
    else
      emit y1p;
      present (x0p and x0) or
              (x1p and x1) then
        emit y1
      end
    end
  ||
    present y1p and (not y1) then
      emit x1p
    end
  ]
||
  signal err in
    present error then
      present err else emit err end
    end
  end signal
]
```

P10

```
int x0, x1, y0, y1;
{
  y0 = 0;
  fork
    x0 = 1;
    y1 = conc(x0, x1)
  par
    if y1 == 0 {
      x1 = 0
    }
  join
}
```

Esterel

```
signal x0p, x0, x1p, x1 in
signal y0p, y0, y1p, y1 in
signal error in
[
  emit y0p;
  [
    emit x0p;
    emit x0;
    present x0p and x1p and
            ((x0 and not x1) or
             (not x0 and x1)) then
      emit error
    end
  ]
||
  signal err in
    present error then
      present err else emit err end
    end
  end signal
]
```

```
present x0p and x1p and
        ((x0 and not x1) or
         (not x0 and x1)) then
  emit error
else
  ...
```

P10

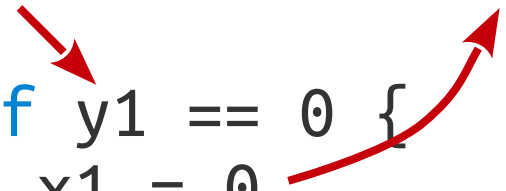
```
int x0, x1, y0, y1;
{
  y0 = 0;
  fork
    x0 = 1;
    y1 = conc(x0, x1)
  par
    if y1 == 0 {
      x1 = 0
    }
  join
}
```

Esterel

```
signal x0p, x0, x1p, x1 in
signal y0p, y0, y1p, y1 in
signal error in
[
  emit y0p;
  [
    emit x0p;
    emit x0;
    present x0p and x1p and
            ((x0 and not x1) or
             (not x0 and x1)) then
      emit error
    else
      emit y1p;
      present (x0p and x0) or
              (x1p and x1) then
        emit y1
      end
    end
  ||
    present y1p and (not y1) then
      emit x1p
    end
  ]
||
  signal err in
    present error then
      present err else emit err end
    end
  end signal
]
```

P10

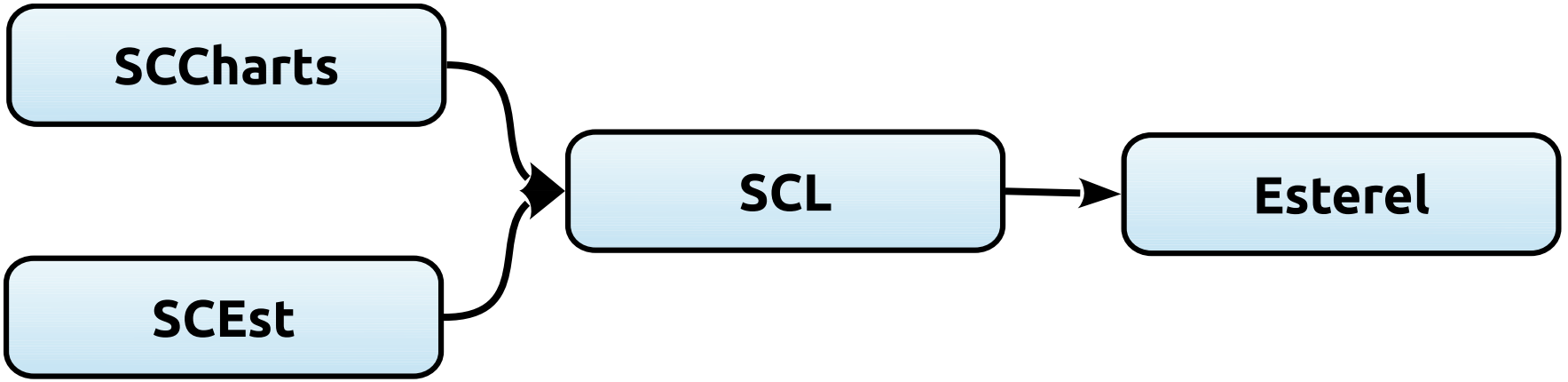
```
int x0, x1, y0, y1;
{
  y0 = 0;
  fork
    x0 = 1;
    y1 = conc(x0, x1)
  par
    if y1 == 0 {
      x1 = 0
    }
  join
}
```



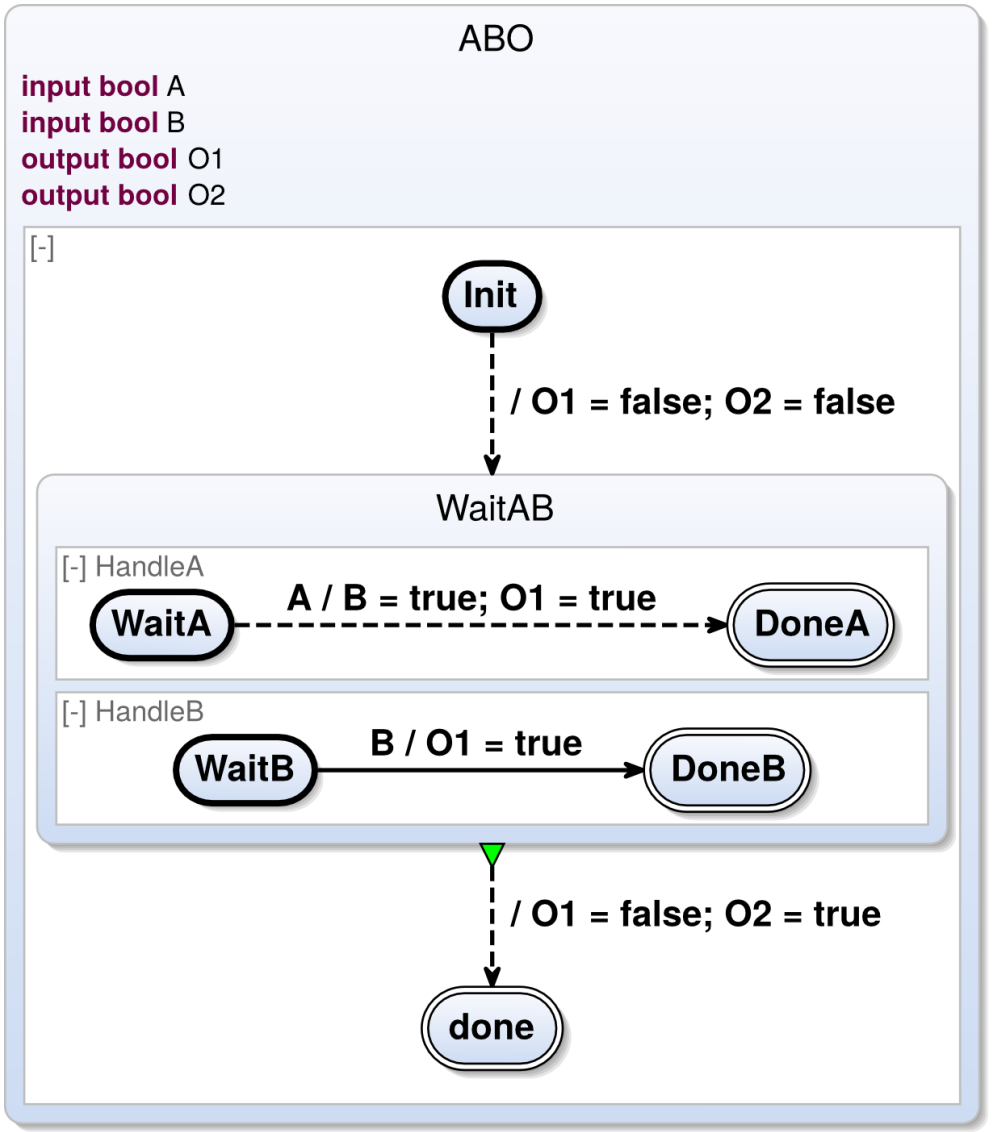
Not Berry Constructive

→ Not Strictly Sequentially Constructive

New Compile Chain



SCChart: ABO


ABO
input bool A
input bool B
output bool O1
output bool O2
[-]
Init
/ O1 = false; O2 = false
WaitAB
[-] HandleA
WaitA
A / B = true; O1 = true
DoneA
[-] HandleB
WaitB
B / O1 = true
DoneB
/ O1 = false; O2 = true
done

SCEst

```
present S then
  unemit S
else
  emit S;
end;
present S
  % do stuff
end
```

SSA Merge Functions for SC

```
seq(x0, x1)

conc(x0, x1)
```

Further issues:

- Updates
- Persistent Variables
- Interface Variables
- Loops

Update Value Merge

```
fork
   x = 1
par
   x = x + 2
par
   y = x
join
```

Update Value Merge

```
fork
   x0 = 1
par
   x1 = x0 + 2
par
   y = conc(x0, x1)
join
```

Update Value Merge

```
fork
   x0 = 1
par
   x1up = 2
par
   y = conc(x0, x1)
join
```

Update Value Merge

```
fork
   x0 = 1
par
   x1up = 2
par
   y = combine(+, x0, x1up)
join
```

Persistence of Merged Values

```
x0 = 1;
if I {
  x1 = 0
}
pause;
y = seq(x0, x1)
```

```
fork
  x0 = 1;
  if I {
    x1 = 0
  }
  pause;
  y = pre(xreg)
par
  l:
  xreg = seq(seq(
    pre(xreg), x0), x1);
  goto l //or terminate
join
```

Interface Variables

```
output int x0, x1;

x0 = 1
if I {
  x1 = 0
}
```

```
output int x;
int x0, x1;

fork
  x0 = 1;
  if I {
    x1 = 0
  }
par
  l:
  x = seq(x0, x1);
  goto l //or terminate
join
```

Loops

```
x = 0;
Loop:
  if I {
    x = 1
  }
  y = x;
  pause;
  if J {
    x = 0
  }
goto Loop
```

```
fork
  x0 = 0;
  Loop:
    if I {
      x1 = 1
    }
    y = seq(seq(seq(pre(xreg),
        x0), x2), x1);
    pause;
    if J {
      x2 = 0
    }
  goto Loop
par
  l: xreg = ...
```

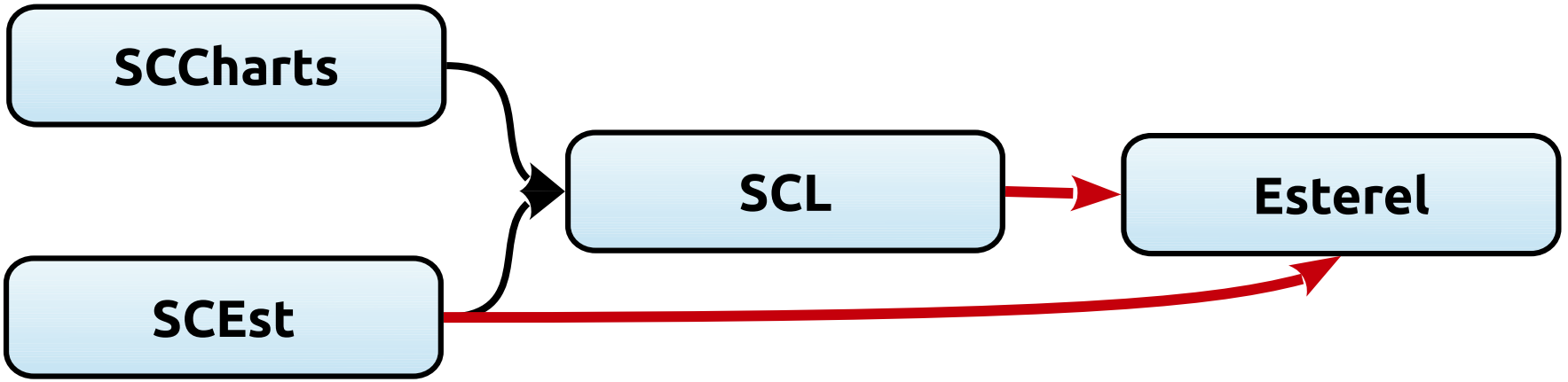
Current Limitations

- Instantaneous loops
- Goto structures
- Static scheduling of updates
- Some confluent writes

Alternative Dual-Rail Encoding

not_x \ x	present	absent
present	illegal	false
absent	true	undef

Future Work



The End