Costing by construction

Greg Michaelson
School of Mathematical & Computer Sciences
Heriot-Watt University

Summary

- how can we use cost information (e.g. WCET, space) to guide software construction?
- mini-Hume
  - compilation to stack machine
  - cost model
- box calculus
- augmenting box calculus with costs
- costulator

Overview

- mature WCET/space costs models for many languages
- analysis tools are whole program
  - apply *after* not *during* initial software construction
- can we see cost implications at every stage as we construct software?

Hume

- with Kevin Hammond (St Andrews)
- formally motivated language for resource aware system construction
- concurrent finite state *boxes* joined by *wires*
- box transitions from *patterns* to *expressions*
- rich polymorphic types

Hume

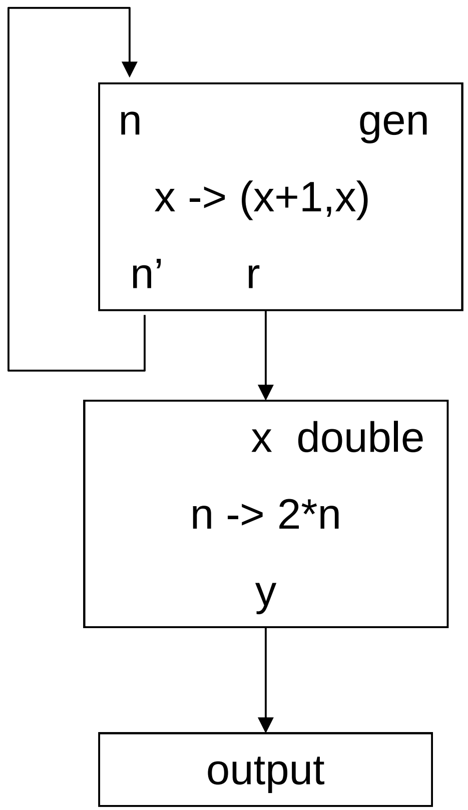
- strong separation of:
  - *coordination*: between boxes & environment
  - *expression*: within boxes
- strong formal foundations
  - semantics + type system
- tool chain via abstract machine to code
- amortised cost models instantiated for concrete platforms

Hume

- Turing complete - too big for this presentation
- mini-Hume
  - integers types only
  - no functions
  - no if/case
  - no * (ignore) pattern or expression

mini-Hume

```
box gen
in (n)
out (n’,r)
match (x) -> (x+1,x);

box double
in (x)
out (y)
match (n) -> (2*n);

stream output to “std_out”;

wire gen
(gen.n’initially 0)
(gen.n,double.x);

wire double
 (gen.r) (output);
```



mini-Hume

prog -> coord ; [prog]

coord -> box | wire | stream

box -> `box` id `in` (vars) `out` (vars)

  `match` (patt) `->` (exp) | ...

vars -> var| var , vars

patt -> int | var | patt , patt

exp -> int | var | exp op exp | exp , exp

wire -> `wire` id (ins) (outs)

ins -> var | var.var [`initially` int] | ins , ins

outs -> var | var.var | outs , outs

stream -> `stream` id [`from`/`to`] “path”

Execution model

forever

for each box - *execute*

find pattern matching inputs

bind pattern variables

evaluate expression to

produce outputs

for each box - *super step*

copy outputs to associated inputs

Stack machine

```
PUSHI integer           stack[sp++] = integer
VAR identifier          allocate next memory address
PUSHM identifier        stack[sp++] = mem[addr(identifier)]
POPM identifier         mem[addr(identifier)] = stack[--sp]
POP                     sp--
ADD              stack[sp-2] = stack[sp-2]+stack[sp-1]; sp--
SUB              stack[sp-2] = stack[sp-2]-stack[sp-1]; sp--
MULT             stack[sp-2] = stack[sp-2]*stack[sp-1]; sp--
DIV              stack[sp-2] = stack[sp-2]/stack[sp-2]; sp--
LABEL label
JNEG label              if(stack[sp--]<0) goto label
JZERO label             if(stack[sp--]==0) goto label
JPOS label              if(stack[sp--]>0) goto label
JNZERO label            if(stack[sp--]!=0) goto label
JMP label               goto label
```

Compilation - memory

box -> `box` id `in (vars) out (vars)` ... ==>

`VAR` id`I1`

`VAR` id`I2`

`...`

`VAR` id`O1`

`VAR` id`O2`

`...`

stream `-> stream` id [`from/to`] “path`” ==>`

`VAR` id

Compilation - box

box -> `box` id `in (vars) out (vars)`

      `match (patt) -> (exp)` | ... ==>

```
LABEL id1:
<<patt_1>>
<<exp_1>>
JMP idEND
...
LABEL idN+1:
LABEL idEND:
```

Compilation - pattern

- for box id

$patt_i$ -> int ==>
PUSHM idIi
PUSHI int
SUB
JNZERO idi+1

$patt_i$ -> $patt_1$, $patt_2$ ==>
<<$patt_1$>>
<<$patt_2$>>

Compilation - expression

- for box id

exp_i -> int ==> `PUSH` int

exp_i -> var_i ==> `PUSHM` id`Ii`

exp_i -> exp_1 op exp_2 ==>
<<exp1>>
<<exp2>>
<<op>>

Compilation - expression

<<+>> ==> `ADD`

<<->> ==> `SUB`

<<*>> ==> `MULT`

<</>> ==> `DIV`

- for box id

exp$_i$ -> exp$_1$, exp$_2$ ==>
<<exp$_1$>>
`POPM` id0i
<<exp$_2$>>

Compilation - wire super step

wire -> `wire` id `(`ins`)` `(`outs`)` `==>`

- for out_i

$var_1.var_2$ ==>
`PUSHM` id0i
`POPM` *input wired to* $var_1.var_2$

var (of stream) ==>
`PUSHM` id0i
`POPM` var

Compilation - initially

- for wire id, out_i

var$_1$.var $_2$ `initially` int ==>

`PUSHI` int

`POPM` id`I`i

Compilation - program

memory

inititially

`LABEL _MAIN`

box

wire superstep

`SHOW`

`GOTO _MAIN`

Compilation

```
box incd
in (s,n)
out (s',n',r)
match
 (0,x) -> (1,x+1,x+1) |
 (1,x) -> (0,x+1,2*x);

stream output to
  "std_out";

wire incd
(incd.s' initially 0,
 incd.n' initially 0)
(incd.s,incd.n,output);
```

```
-  links
VAR incdI0 - s
VAR incdI1 - n
VAR incdO0 – s’
VAR incdO1 – n’
VAR incdO2 - r
VAR output

- initially
PUSHI 0
POPM incdI0
PUSHI 0
POPM incdI1

- execute
LABEL _MAIN
```

```
LABEL incd0
-  box/patt 0
PUSHM incdI0
PUSHI 0
SUB
JNZERO incd1
-  exp 0
PUSHI 1
POPM incdO0
PUSHM incdI1
PUSHI 1
ADD
POPM incdO1
PUSHM incdI1
PUSHI 1
ADD
POPM incdO2
JMP incdEND
```

```
-  pattern 1
LABEL incd1
...
- (pattern 2)
LABEL incd2

- super step
LABEL incdEND
PUSHM incdO0
POPM incdI0
PUSHM incdO1
POPM incdI1
PUSHM incdO2
POPM output

- loop
SHOW
JMP _MAIN
```

Cost model

- box costs in execution

```
box -> box id
     in (vars) out (vars)
     match
     (patts) -> (exps) | ...
```

- max $((\Sigma cost(patt_i)) + cost (exp_i)) + 1$ - JMP

patt ->	int	- 4 – PUSHM,PUSHI,SUB,JNZ
	var	- 0
	$patt_1$, $patt_2$	- $cost(patt_1) + cost(patt_2)$
exp ->	int	- 1 - PUSHI
	var	- 1 - PUSHM
	exp_1 op exp_2	- $cost(exp_1)+cost(exp_2)+1$ - op
	exp_1 , exp_2	- $cost(exp_1)+1+cost(exp_2)$ - POPM

Cost model

- wire costs in super step

wire -> `wire` id `(`ins`)` `(`outs`)`		- cost(ins)+cost(outs)
ins ->	var	- 2 – PUSHM,POPM
	var.var [`initially` int]	- 2[+2] – PUSHM,POPM
[+PUSHI,POPM]		
	ins_1 `,` ins_2	- cost(ins_1)+cost(ins_2)
outs ->	var	- 2 – PUSHM,POPM
	var`.`var	- 2 – PUSHM,POPM
	$outs_1$ `,` $outs_2$	- cost($outs_1$)+cost($outs_2$)
stream -> `stream` id `to` “path”		- 1 - POPM

Example

```
box gen
in (n)
out (n’,r)
match (x) -> (x+1,x);

box double
in (x)
out (y)
match (n) -> (2*n);

stream output to
 “std_out”;

wire gen
(gen.n’initially 0)
(gen.n,double.x);

wire double
(gen.r)(output);
```

```
VAR genI0
VAR genO0
VAR genO1
VAR doubleI0
VAR doubleO0
VAR output

PUSHI 0
POPM genI0

LABEL _MAIN

LABEL gen0

PUSHM genI0
PUSHI 1
ADD
POPM genO0
PUSHM genI0
POPM genO1
JMP genEND
LABEL gen1
LABEL genEND
```

```
LABEL double0

PUSHI 2
PUSHM doubleI0
MULT
POPM doubleO0
JMP doubleEND
LABEL double1
LABEL doubleEND

PUSHM genO0
POPM genI0
PUSHM genO1

POPM doubleI0
PUSHM doubleO0

POPM output

SHOW
JMP _MAIN
```

gen: space 3
pattern 0 exp 7
total cost 7

double: space 2
pattern 0 exp 5
total cost 5

output: space 1
superstep 1

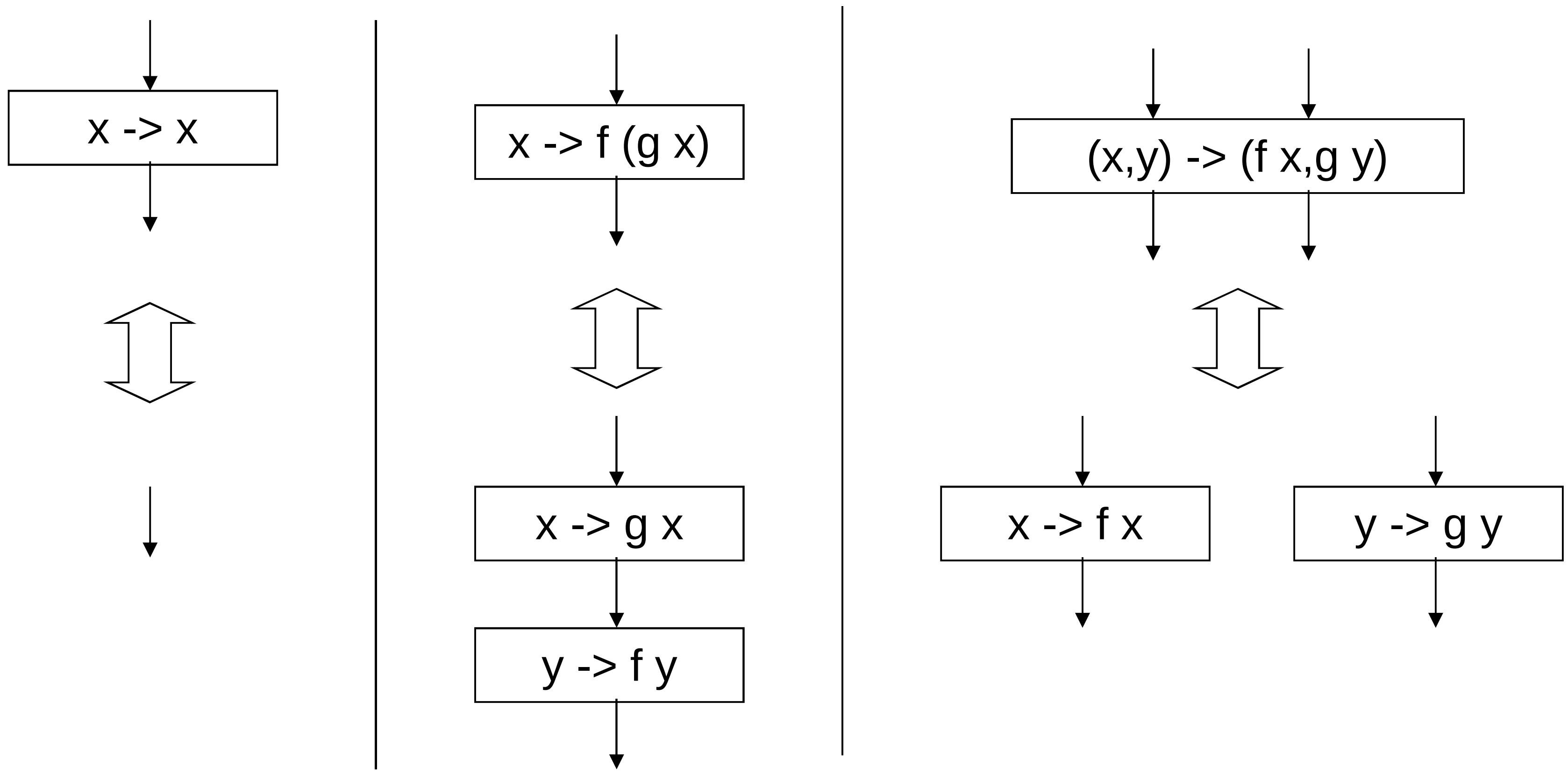
gen: initially 2
superstep 4

double: initially 0
superstep 2

Box calculus

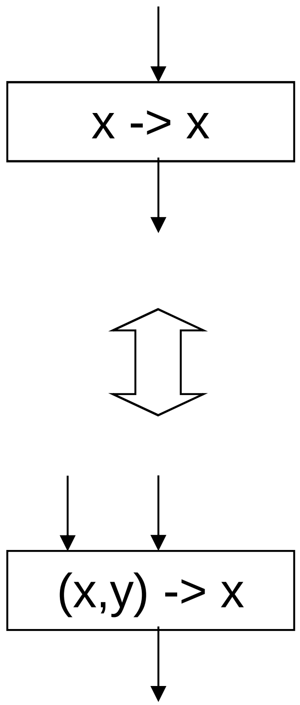
- with Gudmund Grov (Heriot-Watt)
- based on BMF, fold/unfold, FP etc
- rules to:
  – introduce/eliminate boxes/wires
  – split/join boxes horizontally/vertically
- NB rules affect coordination and expressions layers

Box calculus

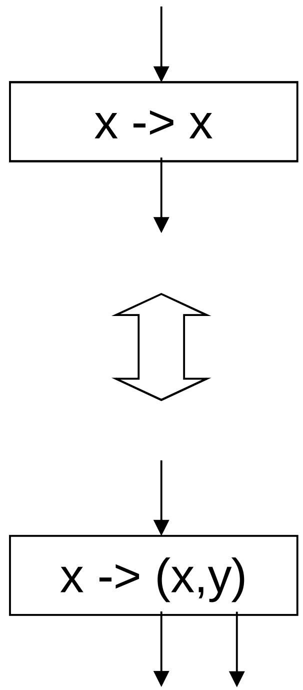


identity vertical split/join horizontal split/join

Box calculus



input introduction/elimination



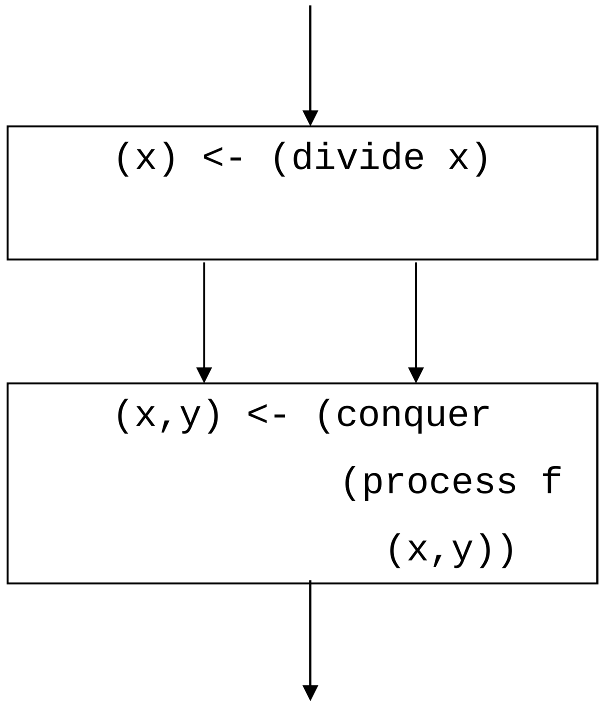
output introduction/elimination

Example: divide & conquer

```
(x) <- (conquer
        (process f
          (divide x)))
```

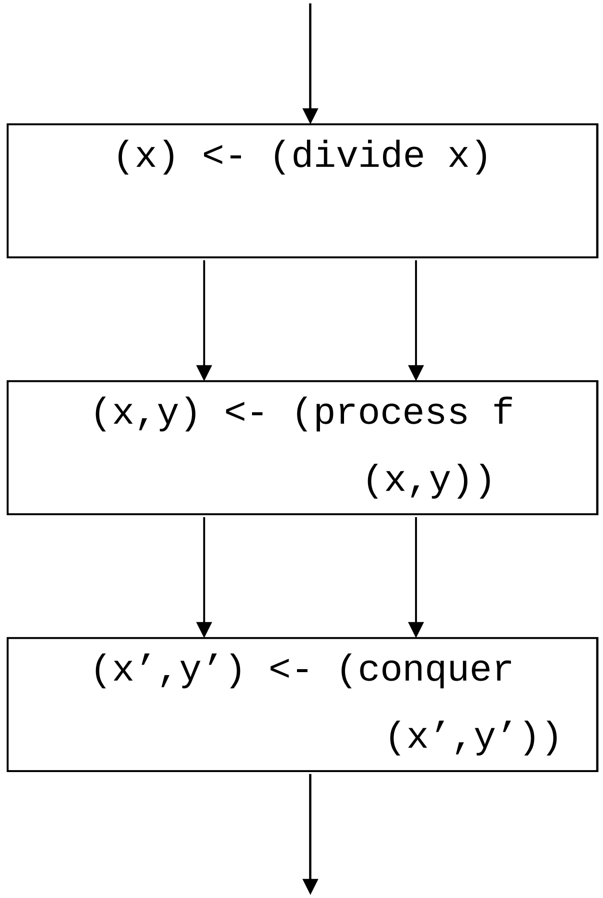
```
process f x y = (f x,f y)
```

Example: divide & conquer



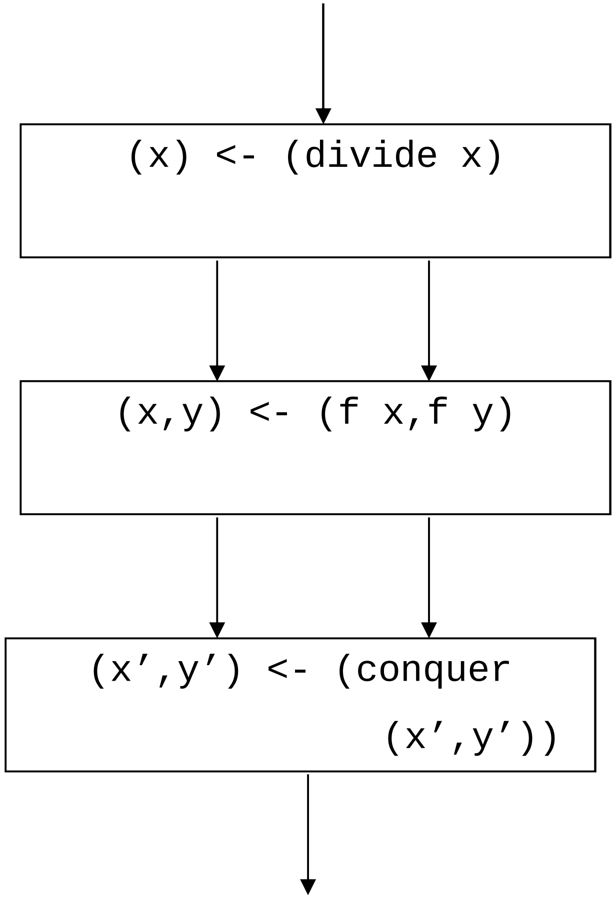
- vertical split

Example: divide & conquer



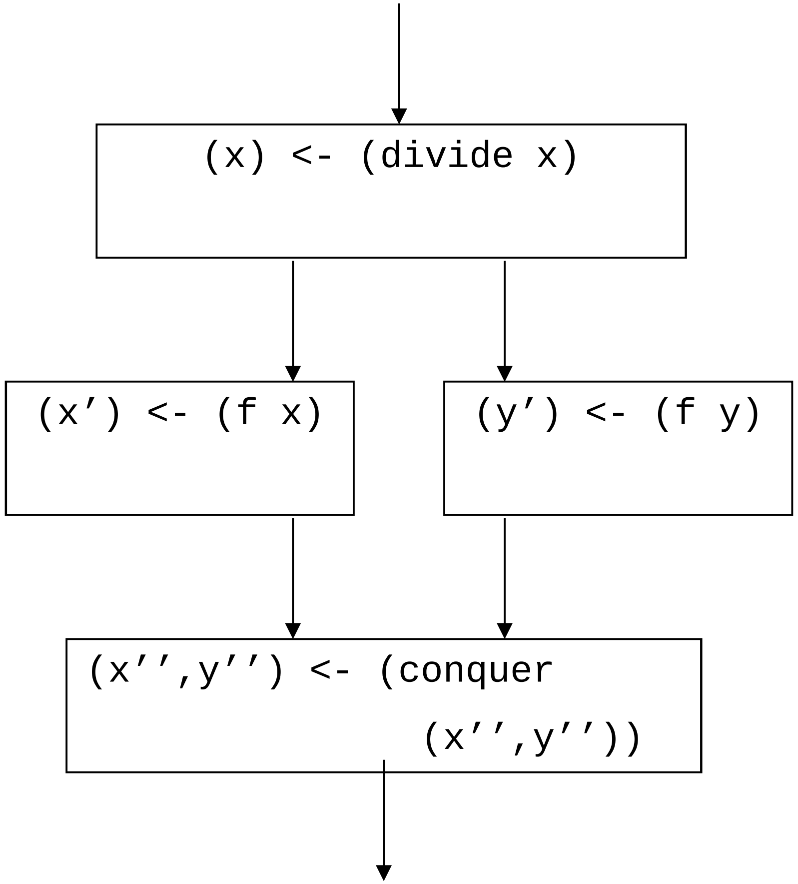
- vertical split

Example: divide & conquer



- unfold

Example: divide & conquer

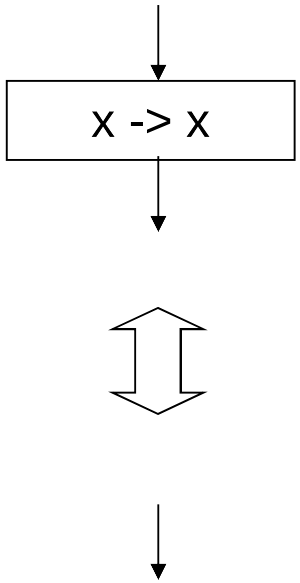


- horizontal split

Costing by construction

- augment rules with cost judgements
- construct software from scratch
  – use rules to justify each step
  – show cost impact of each rule application

Box calculus + costs



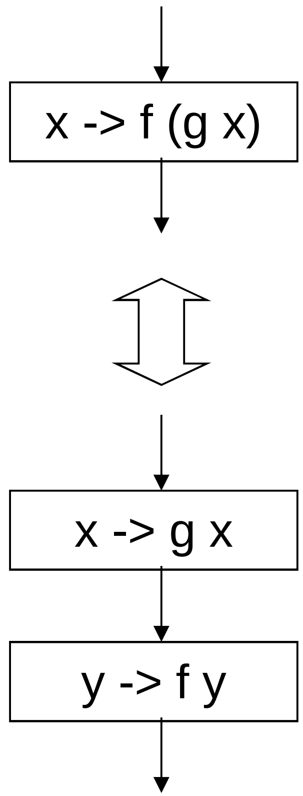
+/- cost patt x

+/- cost exp x

+/- 2 - super step from x

+/- 2 - super step to x

identity



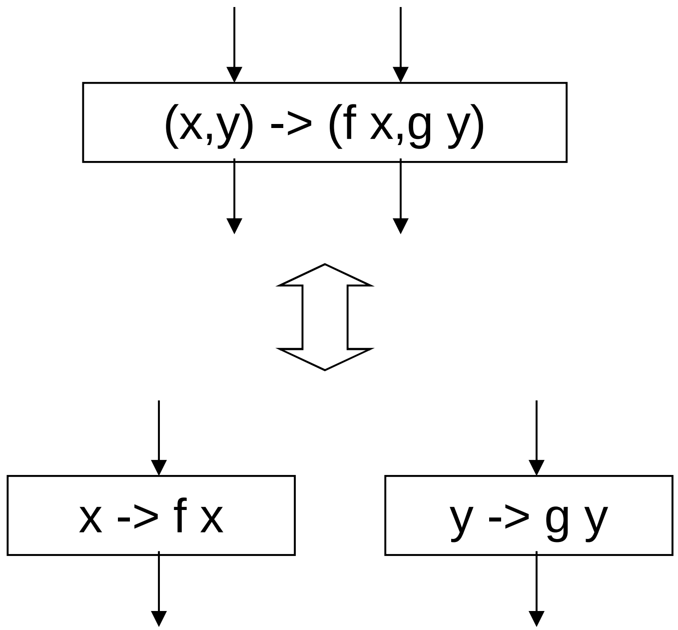
+/- cost patt y

+/- 2 - super step from g x

+/- 2 - super step to y

vertical split/join

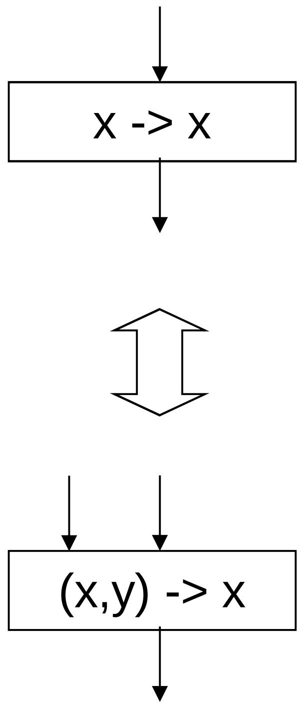
Box calculus + costs



no additional cost

horizontal split/join

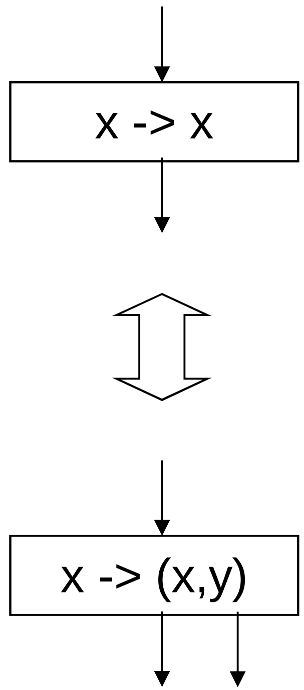
Box calculus + costs



+/- cost patt y

+/- 2 - super
step to y

input introduction/elimination



+/- cost exp y

+/- 2 - super
step from y

output introduction/elimination

Example: divide & conquer

```
(x) <- (conquer
        (process f
          (divide x)))
```

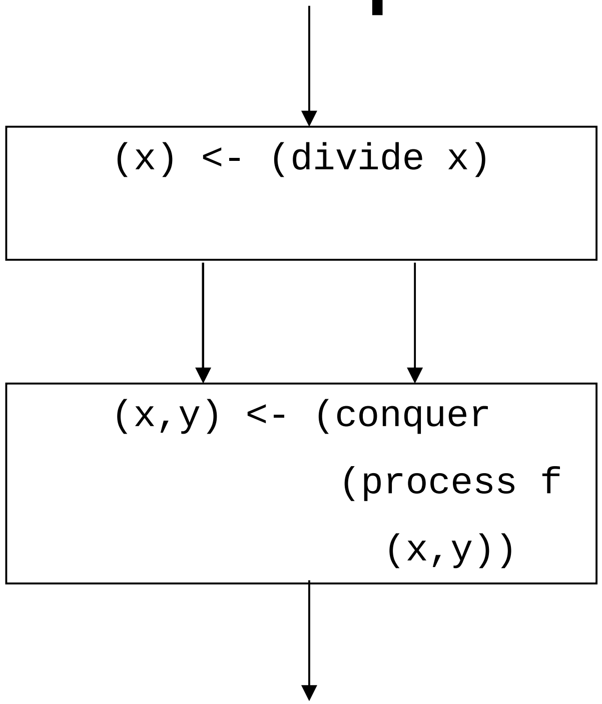
```
box sumsq1 in (x) out (s)
match
(x) ->
 ((x div 2)*(x div 2)+
  (x div 3)*(x div 3));

wire sumsq1 (input) (output);
```

sumsq1: space 2 pattern 0 exp 17
total cost 17

sumsq1: initially 0 superstep 2

```
process f x y = (f x,f y)
```

Example: divide & conquer



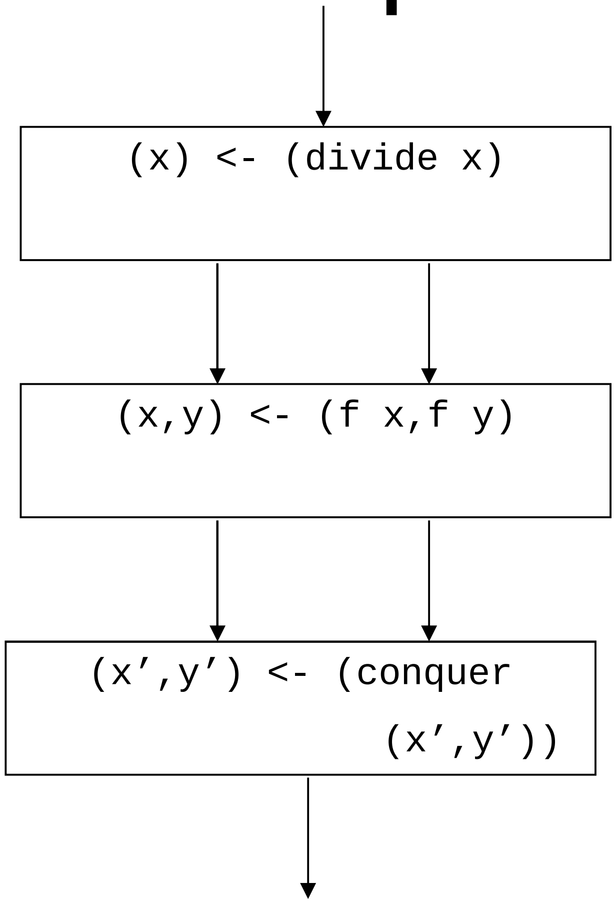

```
box sumsq2 in (x) out (x1,x2)
match (x) -> (x div 2,x div 3);

box conq in (x1,x2) out (s)
match (x1,x2) -> (x1*x1+x2*x2);
```

sumsq2: space 3 pattern 0 exp 9
total cost 9
sumsq2: initially 0 superstep 4

conq: space 3 pattern 0 exp 9
total cost 9
conq: initially 0 superstep 2

- vertical split

Example: divide & conquer



- vertical split/unfold

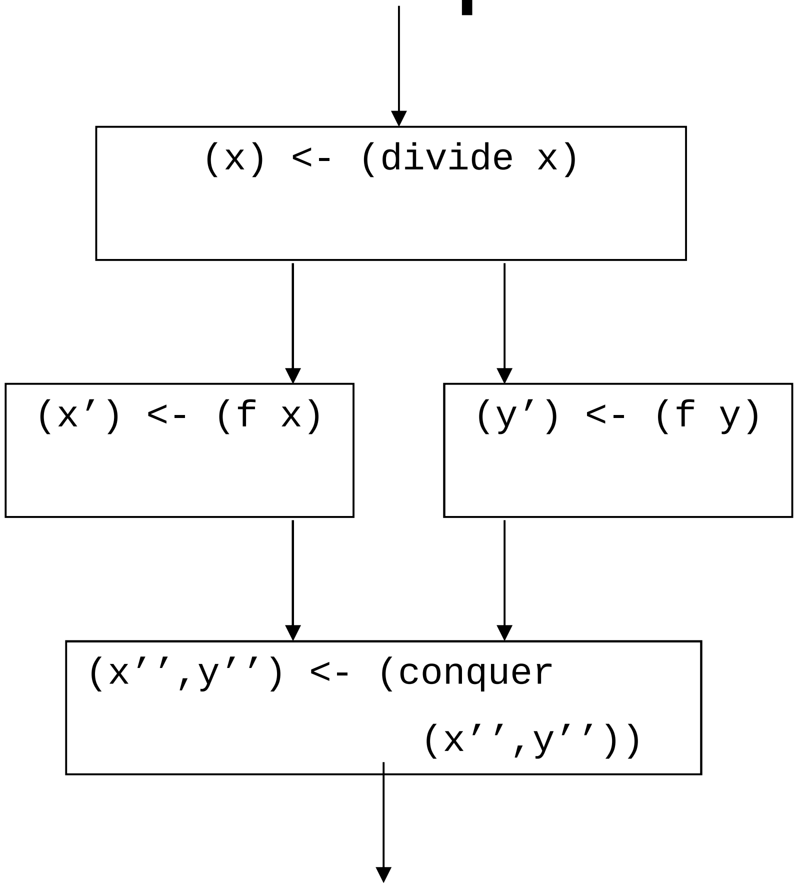
```
box sumsq3 in (x) out (x1,x2)
match (x) -> (x div 2,x div 3);

box process in (x1,x2) out (x1',x2')
match (x1,x2) -> (x1*x1,x2*x2);

box conq in (x1,x2) out (s)
match (x1,x2) -> (x1+x2);
```

sumsq3: space 3 pattern 0 exp 9 total cost 9
sumsq3: initially 0 superstep 3

process: space 4 pattern 0 exp 9 total cost 9
process: initially 0 superstep 4

conq: space 3 pattern 0 exp 5 total cost 5
conq: initially 0 superstep 2

Example: divide & conquer



- horizontal split

```
box sumsq4 in (x) out (x1,x2)
match (x) -> (x div 2,x div 3);

box process1 in (x1) out (x1')
match (x1) -> (x1*x1);

box process2 in (x2) out (x2')
match (x2) -> (x2*x2);

box conq in (x1,x2) out (s)
match (x1,x2) -> (x1+x2);
```

sumsq4: space 3 pattern 0 exp 9 total cost 9
sumsq4: initially 0 superstep 4

process1: space 2 pattern 0 exp 5 total cost 5
process1: initially 0 superstep 2

process2: space 2 pattern 0 exp 5 total cost 5
process2: initially 0 superstep 2

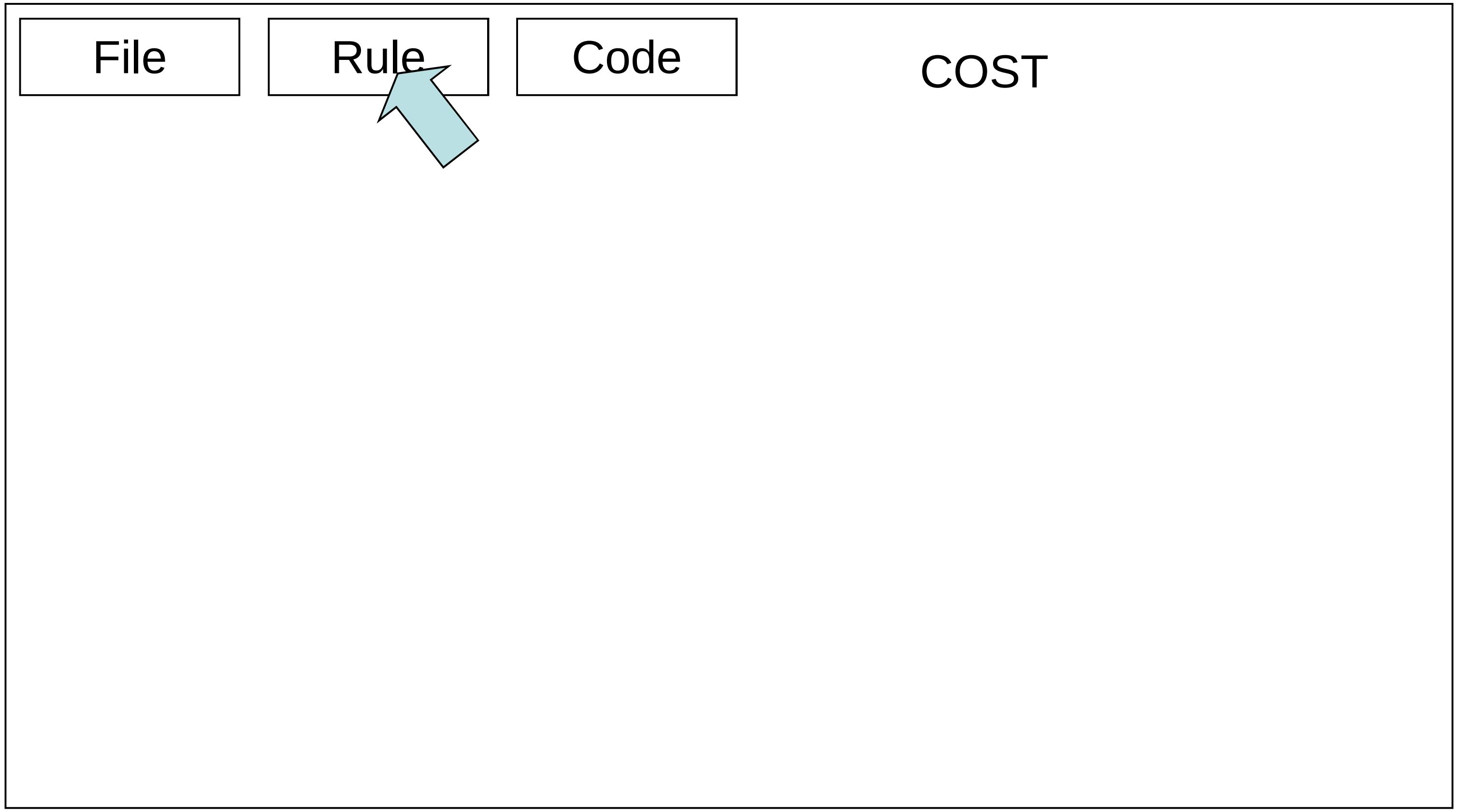
conq: space 3 pattern 0 exp 5 total cost 5
conq: initially 0 superstep2

Costulalator

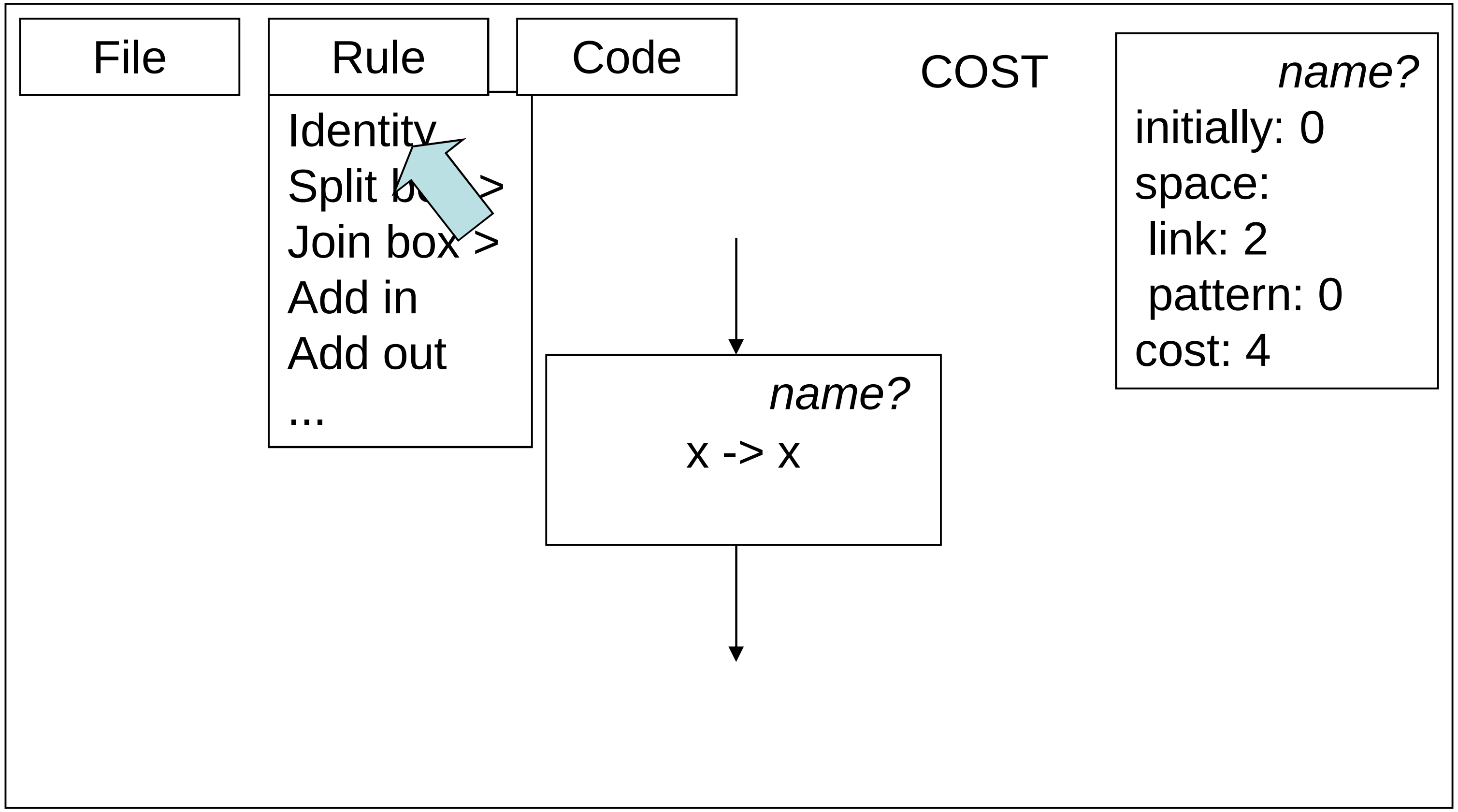
- IDE for costing by construction
- draw boxes
- fill in box details
- costulator displays imputed costs stage by stage

Costulator

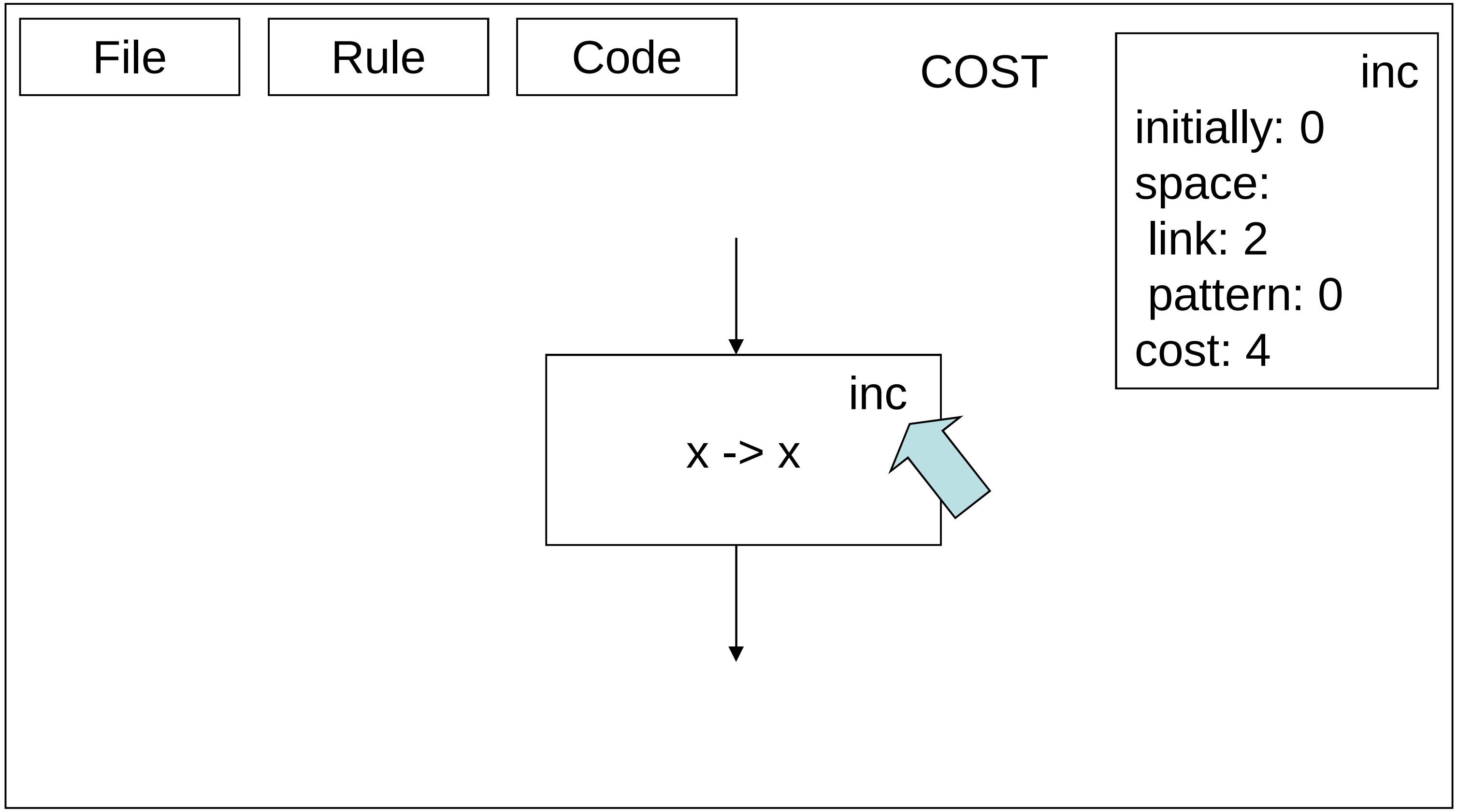
Costulator



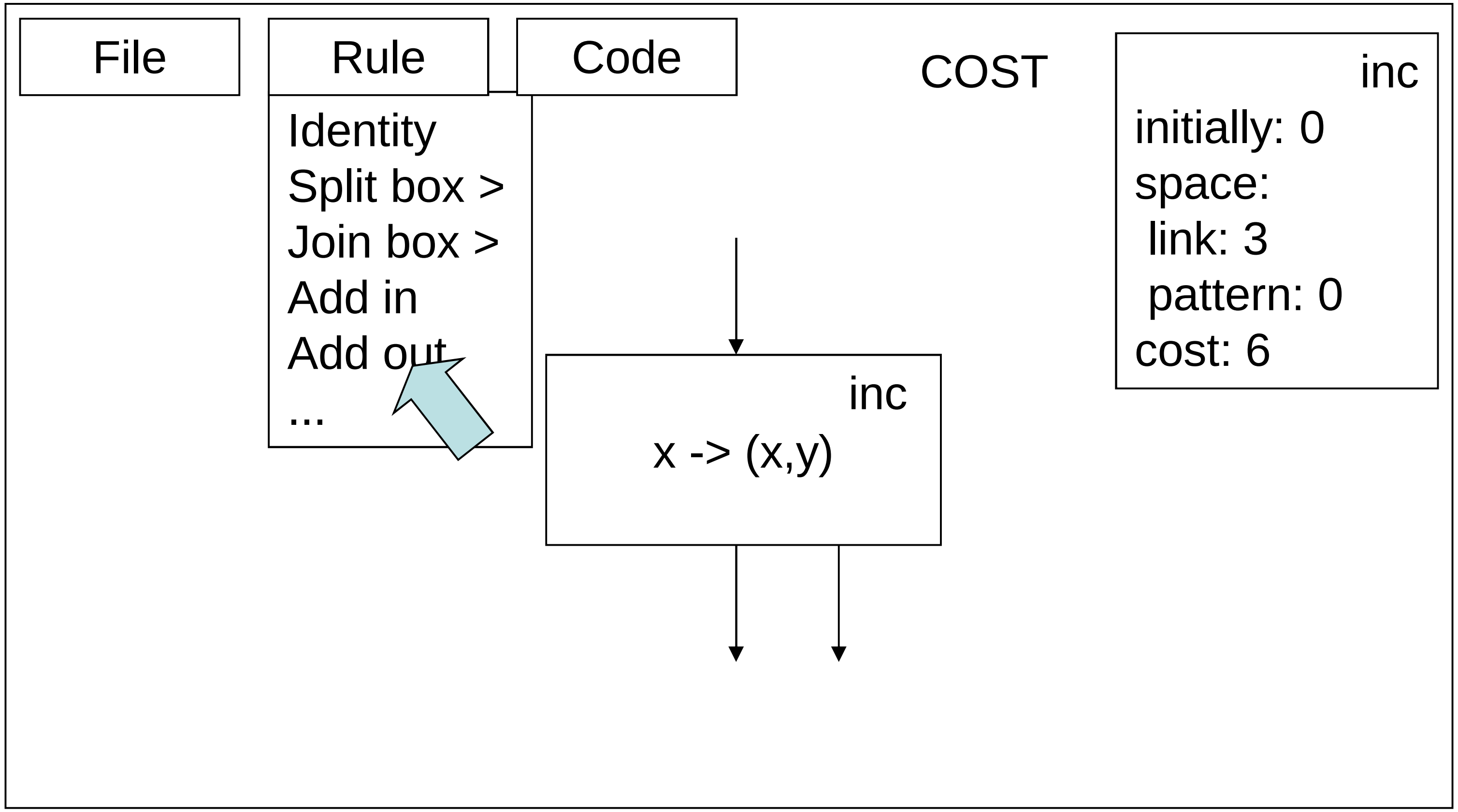
Costulator


File
Rule
Code
COST
name?
initially: 0
space:
link: 2
pattern: 0
cost: 4
Identity
Split b >
Join box >
Add in
Add out
...
name?
x -> x

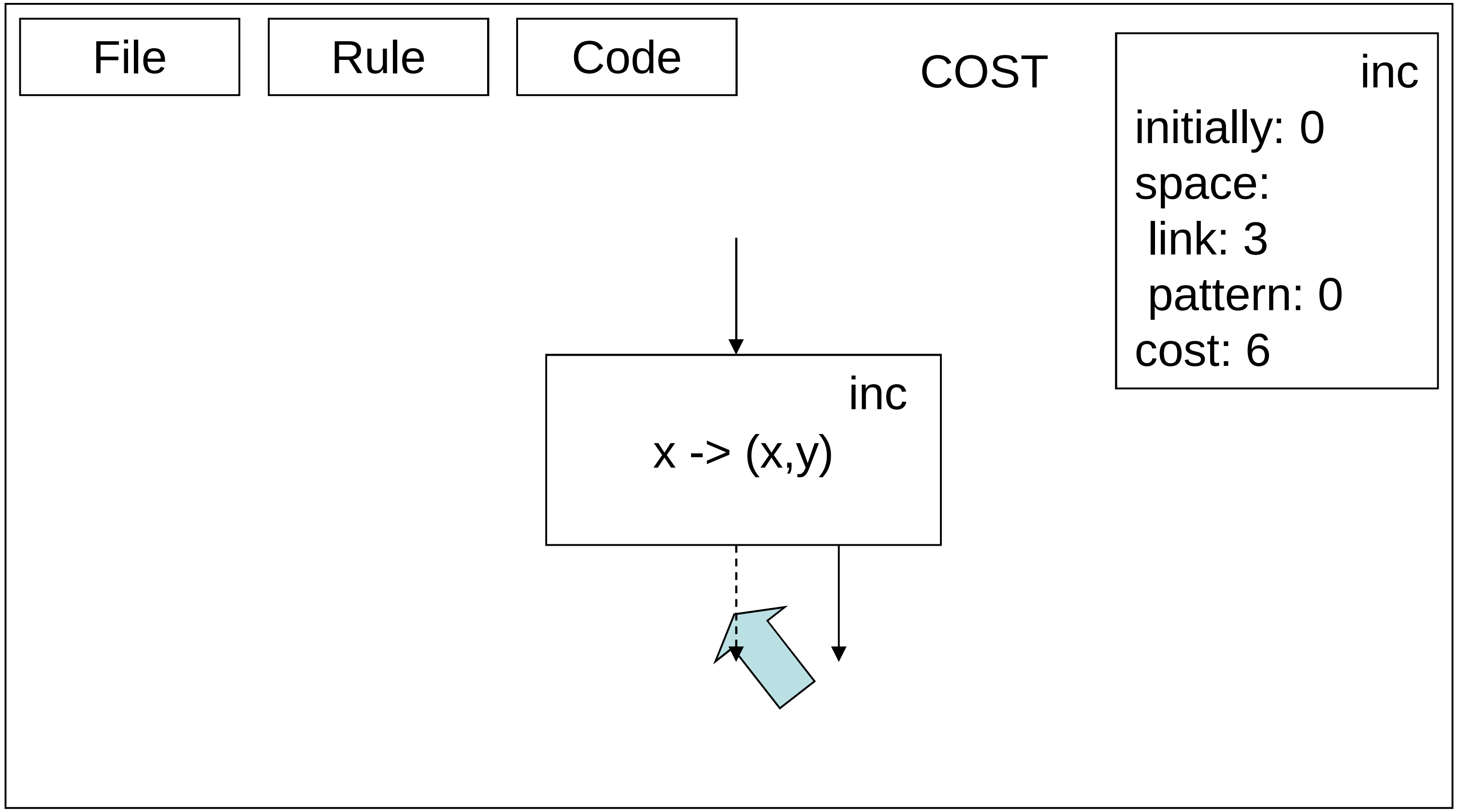
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 2
pattern: 0
cost: 4
inc
x -> x

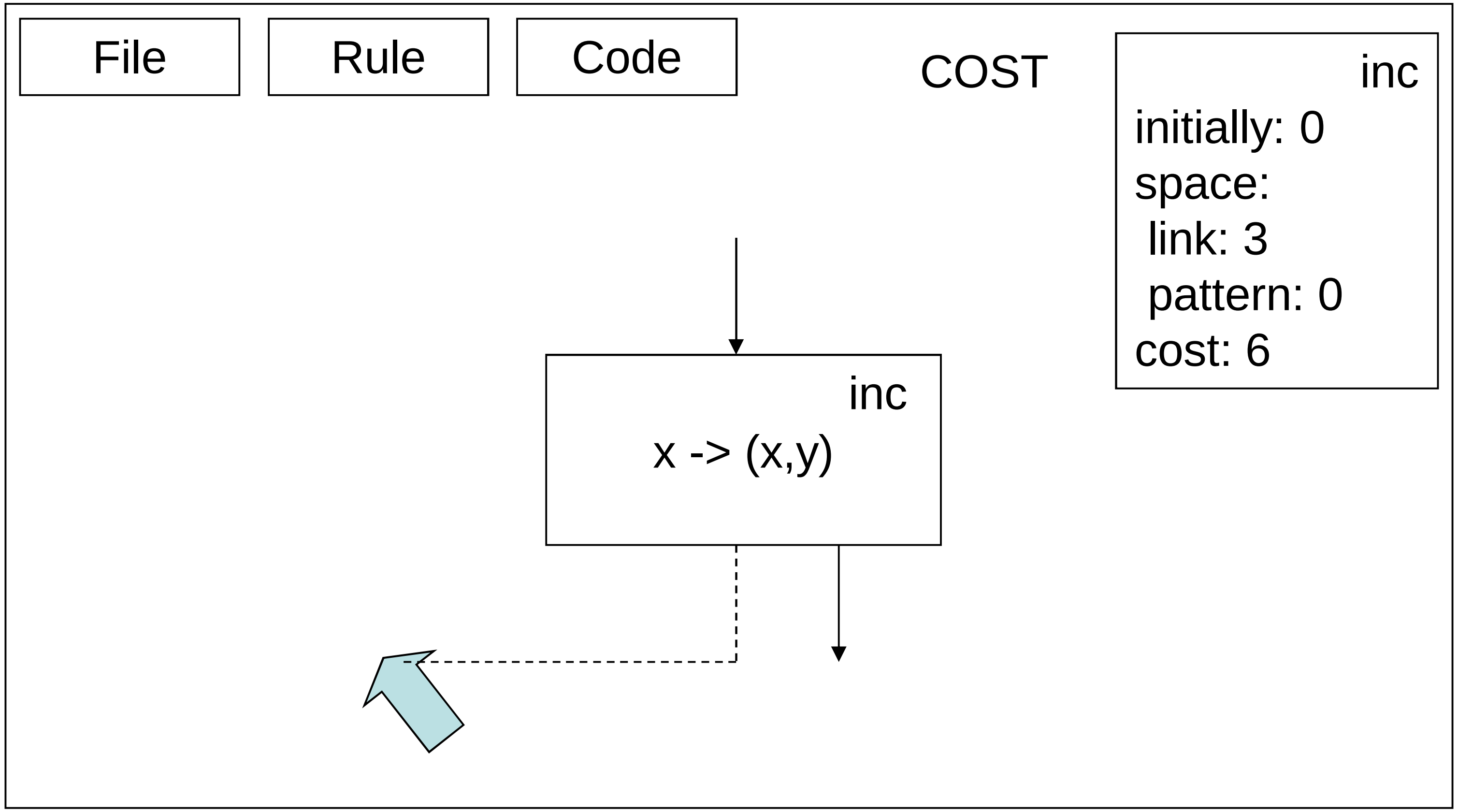
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 3
pattern: 0
cost: 6
Identity
Split box >
Join box >
Add in
Add out
...
inc
x -> (x,y)

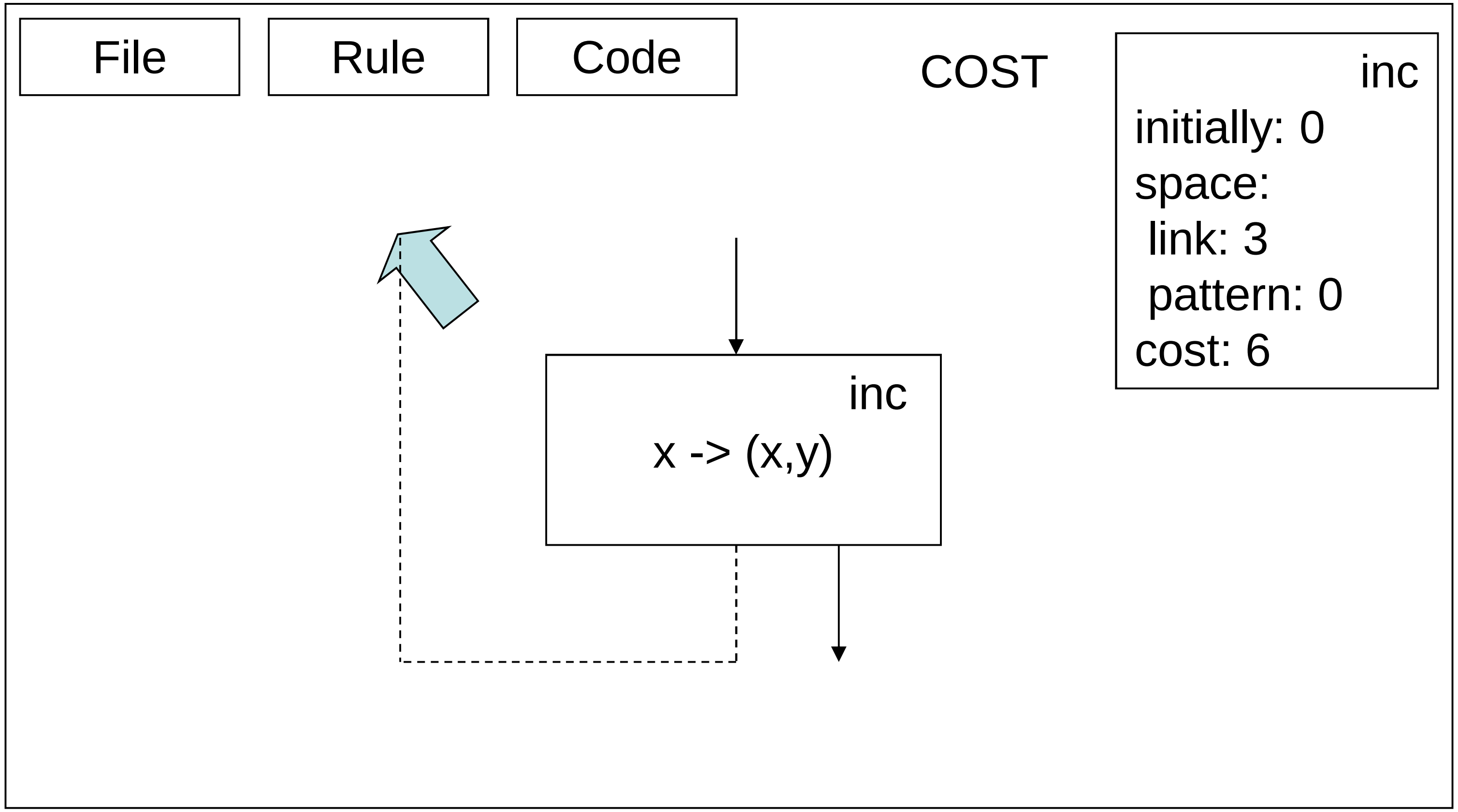
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 3
pattern: 0
cost: 6
inc
x -> (x,y)

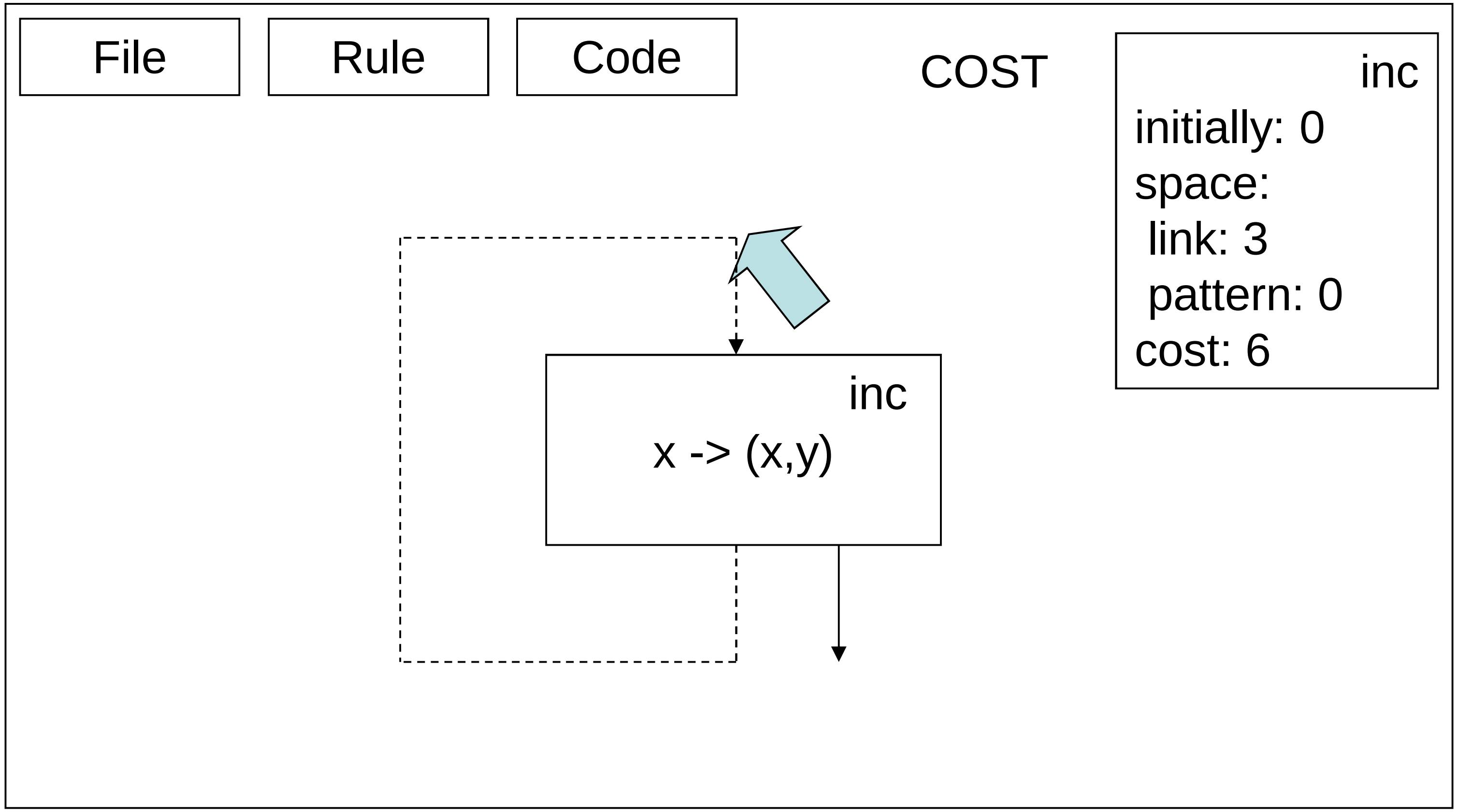
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 3
pattern: 0
cost: 6
inc
x -> (x,y)

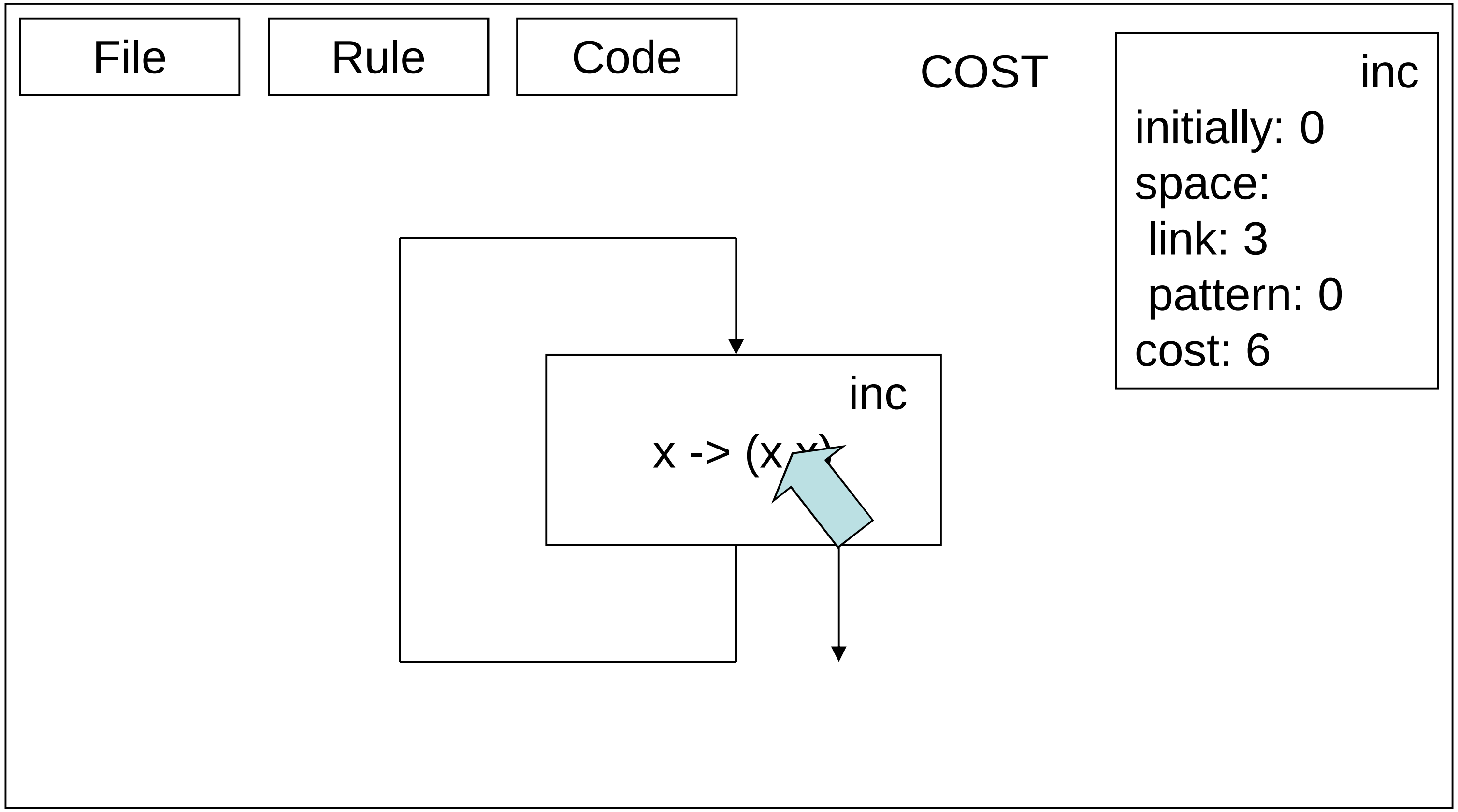
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 3
pattern: 0
cost: 6
inc
x -> (x,y)

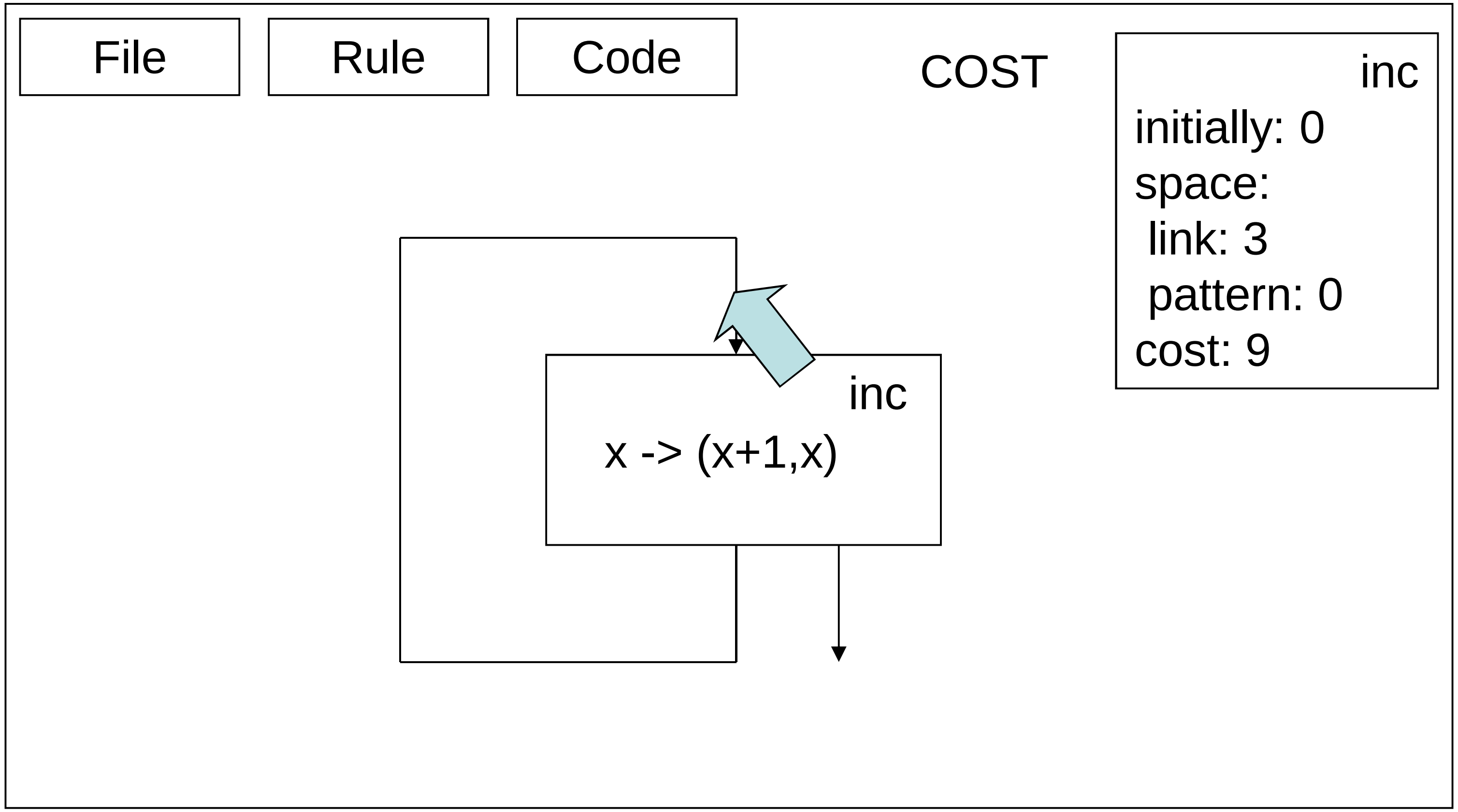
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 3
pattern: 0
cost: 6
inc
x -> (x,y)

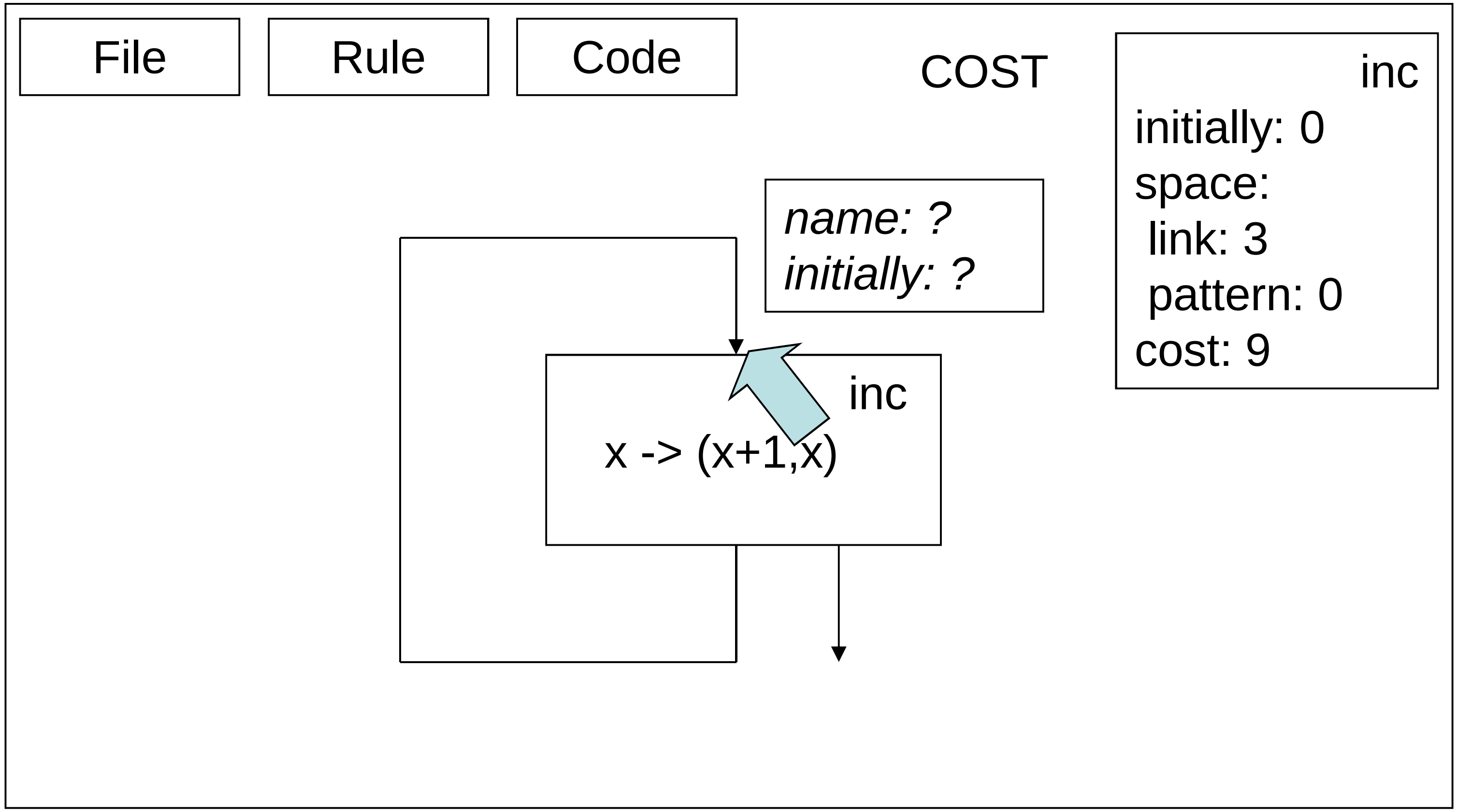
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 3
pattern: 0
cost: 6
inc
x -> (x,x)

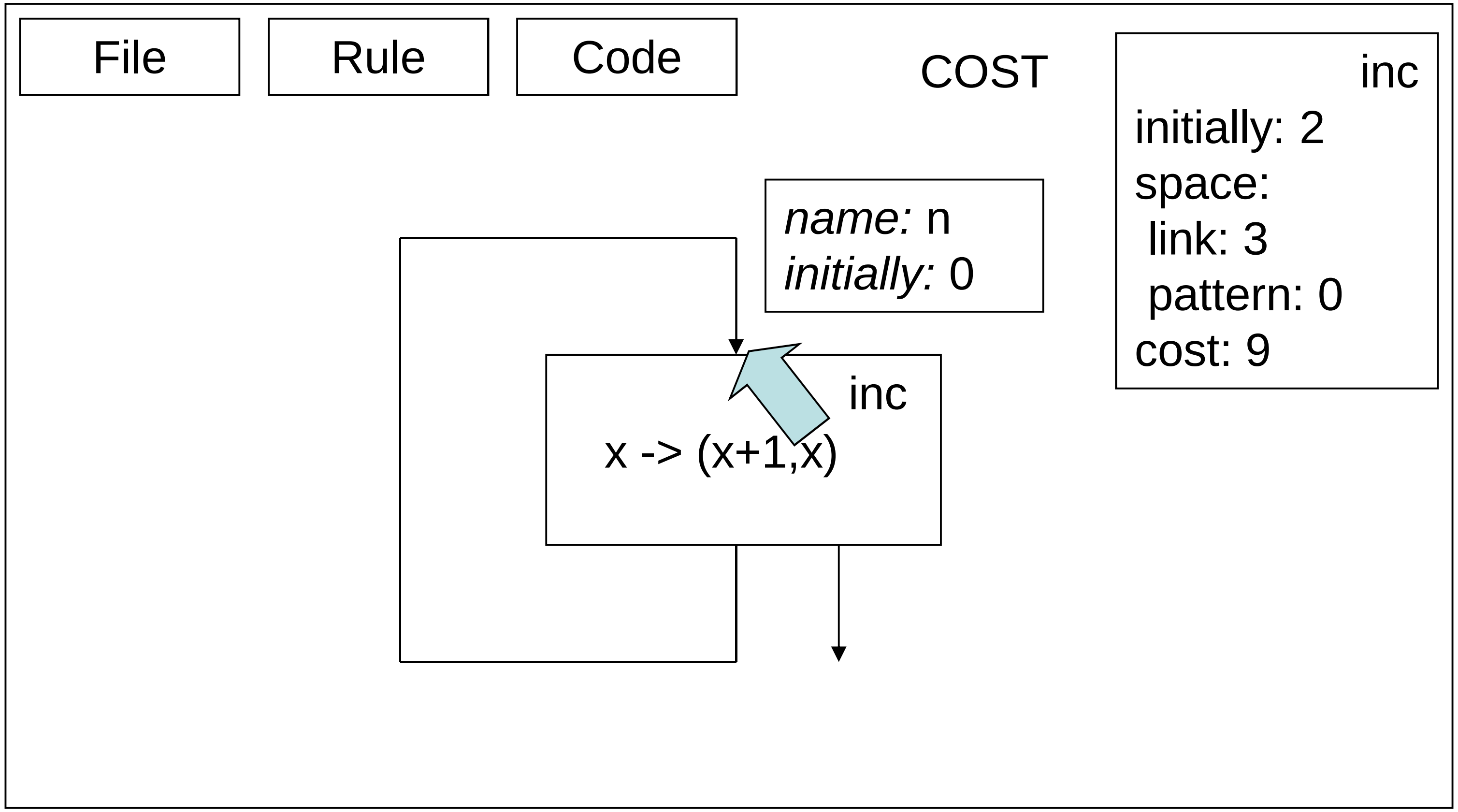
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 3
pattern: 0
cost: 9
inc
x -> (x+1,x)

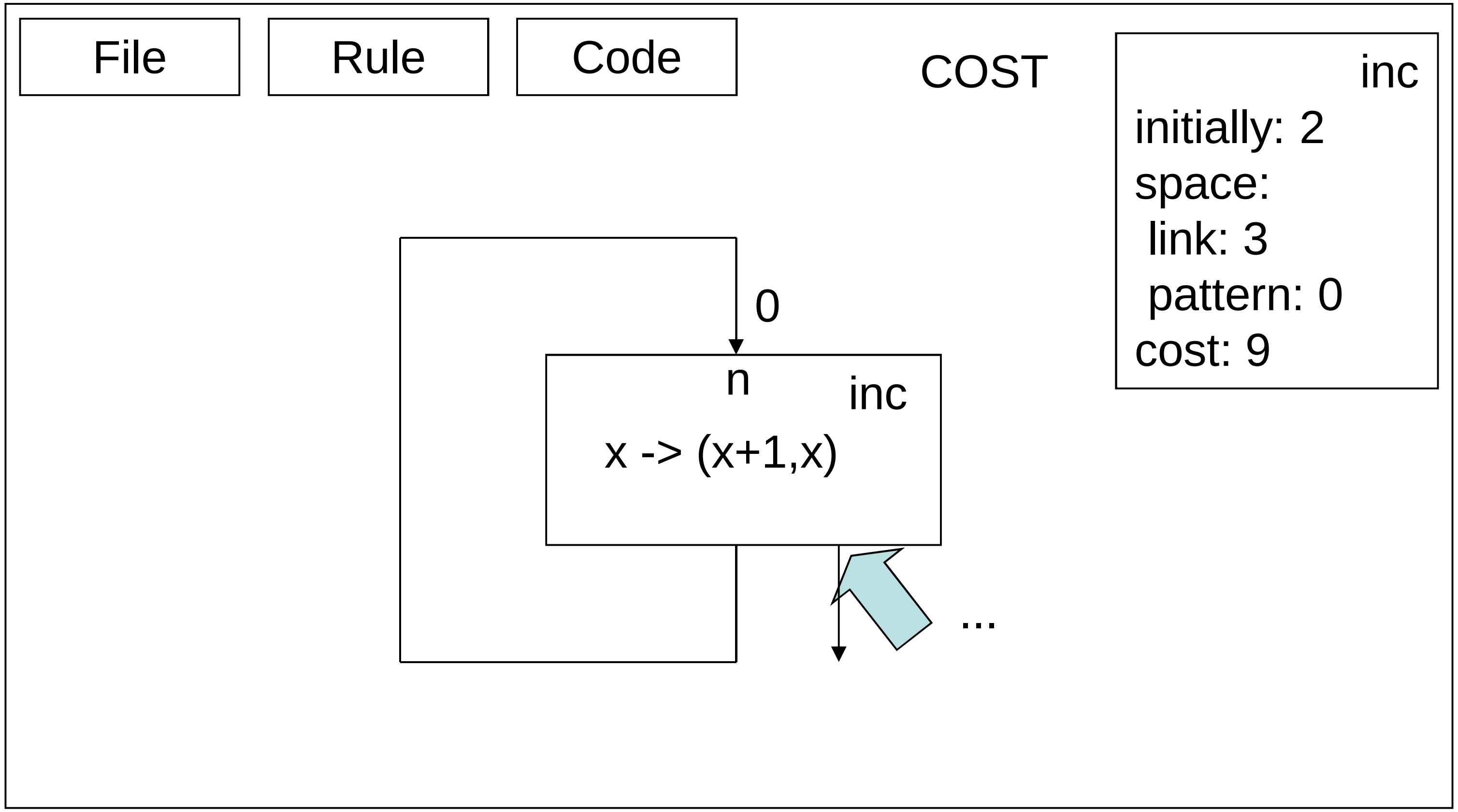
Costulator


File
Rule
Code
COST
inc
initially: 0
space:
link: 3
pattern: 0
cost: 9
name: ?
initially: ?
inc
x -> (x+1,x)

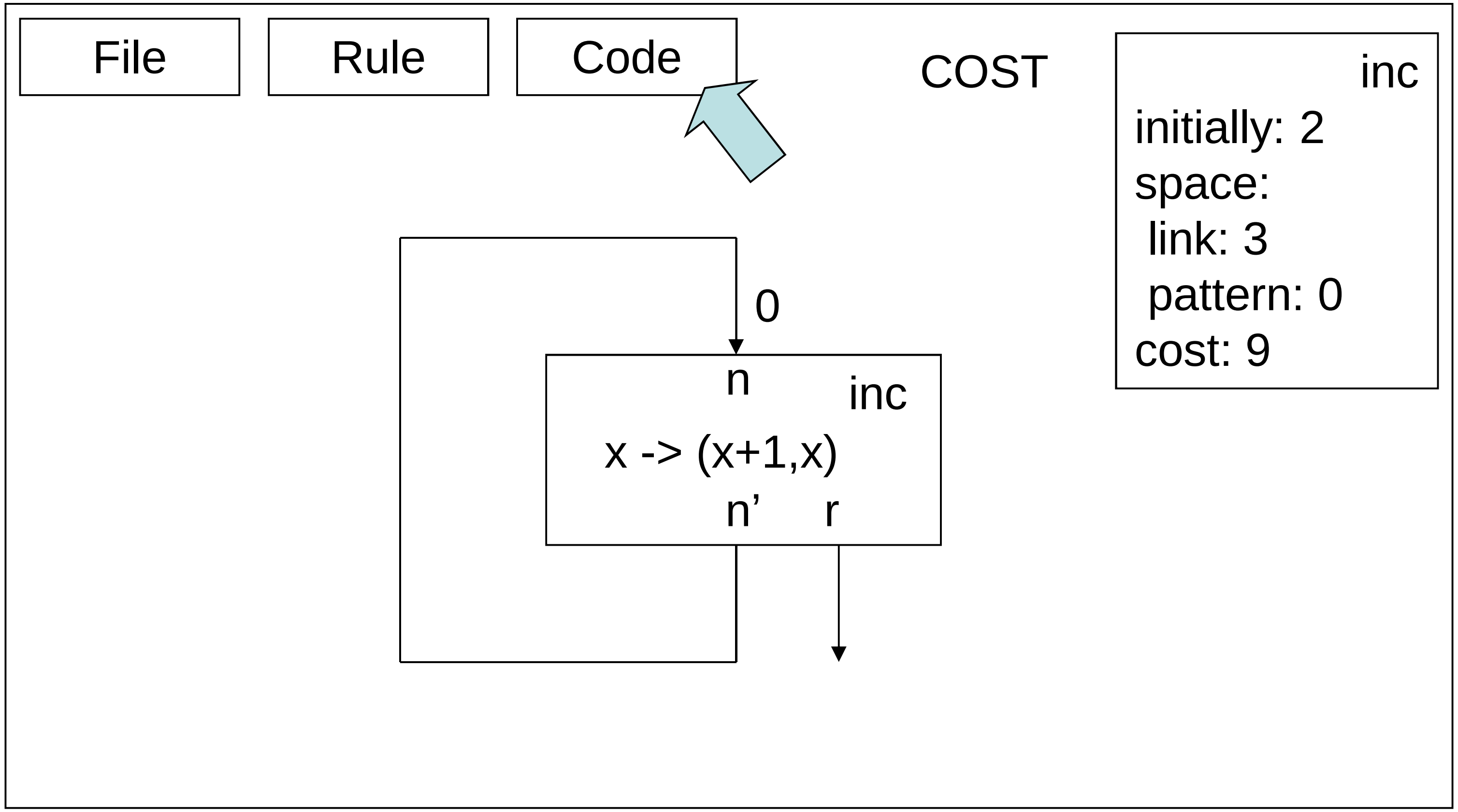
Costulator


File
Rule
Code
COST
inc
initially: 2
space:
link: 3
pattern: 0
cost: 9
name: n
initially: 0
inc
x -> (x+1,x)

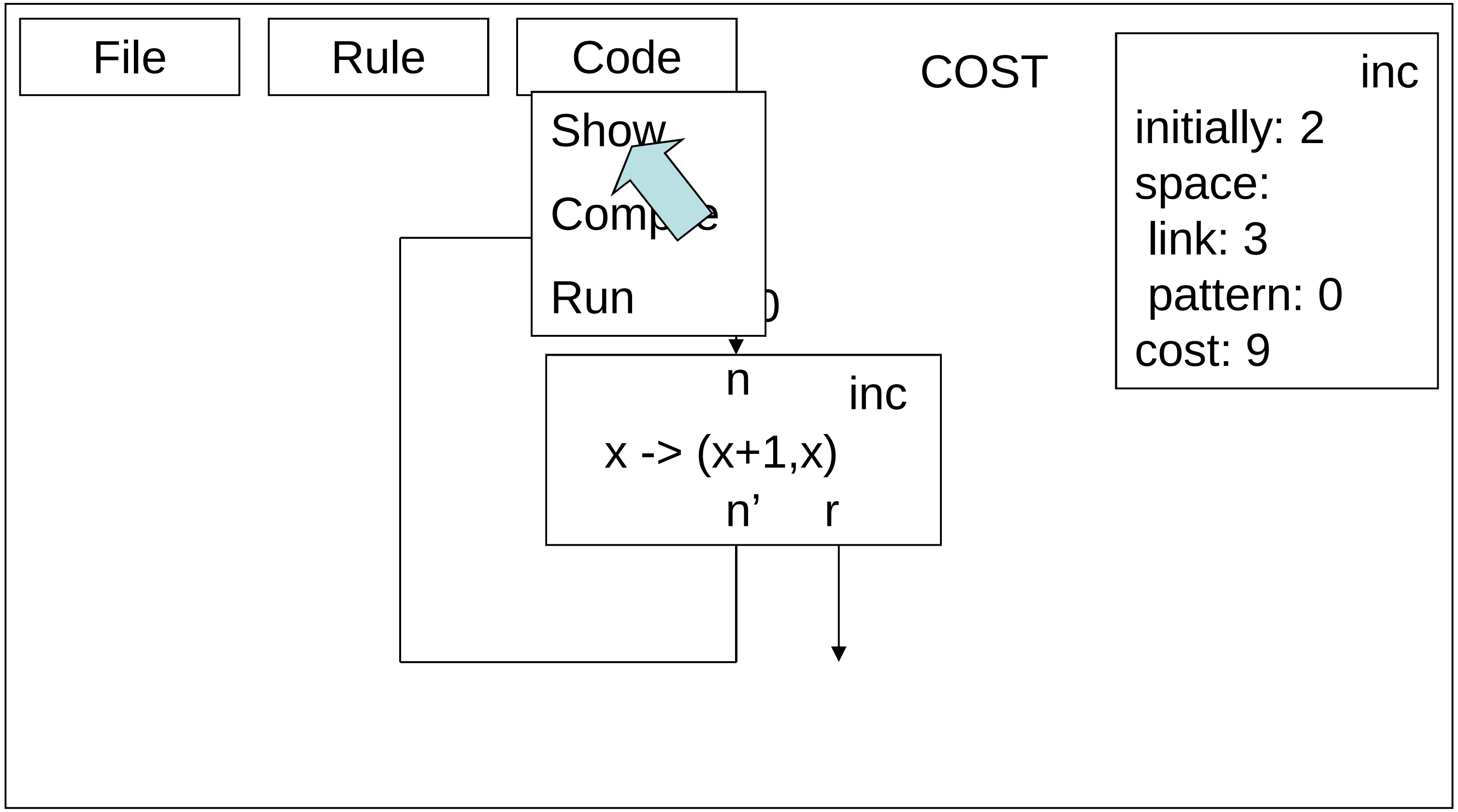
Costulator


File
Rule
Code
COST
inc
initially: 2
space:
link: 3
pattern: 0
cost: 9
0
n
inc
x -> (x+1,x)
...

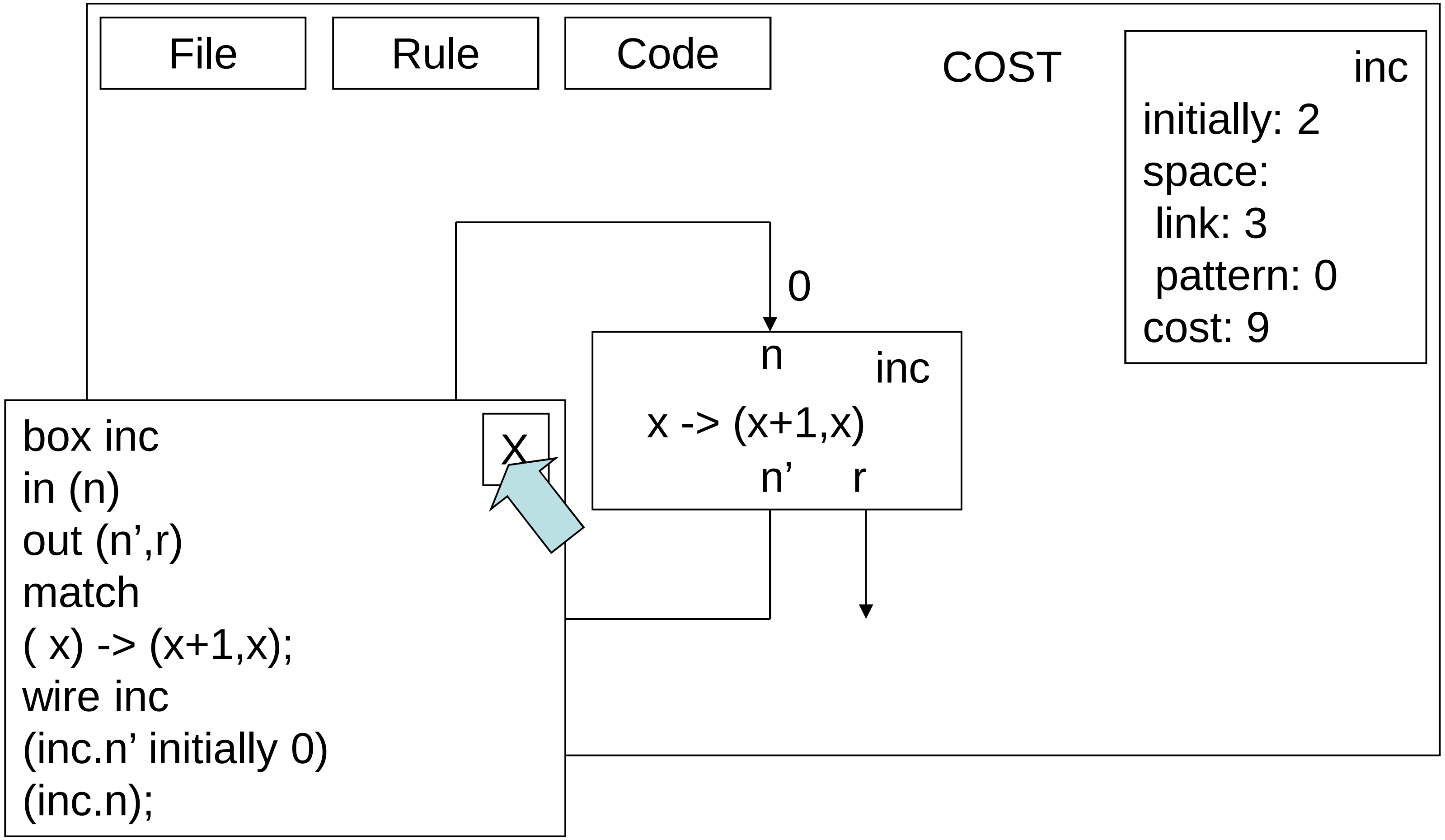
Costulator


File
Rule
Code
COST
inc
initially: 2
space:
link: 3
pattern: 0
cost: 9
0
n
inc
x -> (x+1,x)
n’
r

Costulator


File
Rule
Code
Show
Comp
e
Run
COST
inc
initially: 2
space:
link: 3
pattern: 0
cost: 9
n
inc
x -> (x+1,x)
n'
r

Costulator


File
Rule
Code
COST
inc
initially: 2
space:
link: 3
pattern: 0
cost: 9
0
n
inc
x -> (x+1,x)
n’
r
box inc
in (n)
out (n’,r)
match
(x) -> (x+1,x);
wire inc
(inc.n’ initially 0)
(inc.n);
X

X by construction

- syntax directed editing
  - also visual editing
  - editor obliges user to only follow grammar rules in constructing/changing program
- correctness by construction
  - theorem prover obliges user to only apply correctness preserving rules to composing correct constructs
  - e.g. Martin-Löf type theory

X by construction

- too restrictive
- typically can’t make bad moves to get to a good state
- bad constructs are:
  – ungrammatical (syntax)
  – incorrect (correctness)

X by construction

- costing isn’t like that!
- everything has a cost, even “wrong” bits of program
  - cost is 0
  - cost is a variable
- maybe not the cost you want though so need cost monitoring

Conclusion

- Costing by construction:
  - lets you watch how your programming affects costs as the program develops
  - does not oblige you to form grammatical/correct/well costed constructs as you go along
  - might cleanly augment an IDE

Conclusion

- mini-Hume compiler + stack machine + cost analysis all written in Haskell
- *Costulator* longer term project
  - Kos Devyatov
- thanks to:
  - Kevin Hammond: Hume + costs
  - Gudmund Grov: box calculus
- http://www.macs.hw.ac.uk/~greg/hume