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 idI1
VAR idI2
...
VAR idO1
VAR idO2
...

stream -> stream id [from/to] "path" ==> 
VAR id
Compilation - box

box -> box id in (vars) out (vars)
    match (patt) -> (exp) | ... ==> 

LABEL id1 :
<<patt1>>
<<exp1>>
JMP idEND

... 

LABEL idN+1 :
LABEL idEND:
Compilation - pattern

• for box id

\[ \text{patt}_i \rightarrow \text{int} \Rightarrow \]
\[ \text{PUSHM id} i \]
\[ \text{PUSHI int} \]
\[ \text{SUB} \]
\[ \text{JNZERO id} + 1 \]

\[ \text{patt}_i \rightarrow \text{patt}_1, \ \text{patt}_2 \Rightarrow \]
\[ \lll \text{patt}_1 \rrr \]
\[ \lll \text{patt}_2 \rrr \]
Compilation - expression

• for box id

\[ \text{exp}_i \rightarrow \text{int} \Rightarrow \text{PUSH int} \]

\[ \text{exp}_i \rightarrow \text{var}_i \Rightarrow \text{PUSHM id}_i \]

\[ \text{exp}_i \rightarrow \text{exp}_1 \text{ op } \text{exp}_2 \Rightarrow \]
\[ \text{<<exp1>>} \]
\[ \text{<<exp2>>} \]
\[ \text{<<op>>} \]
Compilation - expression

<<+>> ==> ADD
<<->> ==> SUB
<<*>> ==> MULT
<</>> ==> DIV

- for box id

\[ \text{exp}_i \rightarrow \text{exp}_1, \text{exp}_2 \rightarrow \text{exp}_1 \]
\[ \text{POPM id}_i \]
\[ \text{exp}_2 \]
Compilation - wire super step

wire -> \texttt{wire id (ins) (outs)} \implies

- for out\textsubscript{i},

\texttt{var\textsubscript{1}.var\textsubscript{2}} \implies

\texttt{PUSHM id} \texttt{i}

\texttt{POPM id} \texttt{i}

\texttt{PUSHM input wired to var\textsubscript{1}.var\textsubscript{2}}

\texttt{POPM var}
Compilation - initially

- for wire id, out$_i$

\[ \text{var}_1.\text{var}_2 \text{ initially int} \Rightarrow \]
\[ \text{PUSHI int} \]
\[ \text{POPM id}_i \]
Compilation - program

memory
inititally
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
PUSHI 1
ADD
POPM incdI2
PUSHM incdI0
PUSHI 1
ADD
POPM incdI1
PUSHM incdI0
PUSHI 1
ADD
POPM incdO2
JMP incdEND

- pattern 1
LABEL incd0
LABEL incd1
... 
LABEL incd2

- super step
LABEL incdEND
PUSHM incdO0
PUSHM incdI0
PUSHM incdO1
PUSHM 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 ((Σcost(patt_i)) + cost(exp_i)) + 1 - JMP

patt -> int
   var
   patt_1, patt_2
   - cost(patt_1) + cost(patt_2)

exp -> int
   var
   exp_1 op exp_2
   exp_1, exp_2
   - cost(exp_1) + cost(exp_2) + 1 - op
   - cost(exp_1) + 1 + cost(exp_2) - POPM
```
Cost model

- wire costs in super step

wire -> \text{wire id (ins) (outs)}

ins -> \text{var}

\text{var.var [initially int]}

[+\text{PUSHI,POPM}]

ins_1, ins_2

outs -> \text{var}

\text{var.var}

outs_1, outs_2

stream -> \text{stream id to “path”}

- cost(ins)+cost(outs)

- 2 \text{ – PUSHM,POPM}

- 2[+2] \text{ – PUSHM,POPM}

- cost(ins_1)+cost(ins_2)

- cost(outs_1)+cost(outs_2)

- 1 \text{ – 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

PUSHM genI0
PUSHI 0
PUSHM genI0
POPM doubleI0
PUSHM doubleO0
POPM output
SHOW
JMP _MAIN

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

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: $x \rightarrow x$
- Vertical split/join: $x \rightarrow f(g \ x)$, $x \rightarrow g \ x$, $y \rightarrow f \ y$, $y \rightarrow g \ y$
- Horizontal split/join: $(x,y) \rightarrow (f \ x, g \ y)$, $x \rightarrow f \ x$, $y \rightarrow g \ y$
Box calculus

input introduction/elimination

output introduction/elimination
Example: divide & conquer

\[(x) \leftarrow (conquer \leftarrow (process f \leftarrow (divide x))))\]

process \(f \ x \ y = (f \ x,f \ y)\)
Example: divide & conquer

\[(x) \leftarrow \text{(divide } x)\]

\[(x,y) \leftarrow \text{(conquer } \text{(process } f \text{ } (x,y))\]

- vertical split
Example: divide & conquer

\[
(x) \leftarrow \text{(divide } x) \\
(x,y) \leftarrow \text{(process } f (x,y)) \\
(x',y') \leftarrow \text{(conquer } (x',y'))
\]

• vertical split
Example: divide & conquer

- unfold

\[(x) \leftarrow \text{divide } x\]

\[(x,y) \leftarrow f\ x, f\ y\]

\[(x',y') \leftarrow \text{conquer}\]

\[(x',y')\]
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

Identity:

- $x \rightarrow x$
- $+/- \text{ cost patt } x$
- $+/- \text{ cost exp } x$
- $+/- 2$ - super step from $x$
- $+/- 2$ - super step to $x$

Vertical split/join:

- $x \rightarrow f(g(x))$
- $+/- \text{ cost patt } y$
- $+/- 2$ - super step from $g(x)$
- $+/- 2$ - super step to $y$

Box calculus + costs

**(x, y)** → **(f x, g y)**

**horizontal split/join**

no additional cost

**x** → **f x**

**y** → **g y**
Box calculus + costs

input introduction/elimination

output introduction/elimination
Example: divide & conquer

(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

- vertical split

(\(x\) <- (divide \(x\))

(\(x,y\) <- (conquer
  (process \(f\)
    (\(x,y\)))

box \text{sumsq2} \text{ in (x) out (x1,x2)}
match \(x\) -> (x \text{ div 2}, x \text{ div 3});

box \text{conq in (x1,x2) out (s)}
match \(x1,x2\) -> (x1*x1+x2*x2);

\text{sumsq2}: \text{space 3 pattern 0 exp 9}
\text{total cost 9}
\text{sumsq2: initially 0 superstep 4}

\text{conq: space 3 pattern 0 exp 9}
\text{total cost 9}
\text{conq: initially 0 superstep 2}
Example: divide & conquer

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

*(x) <- (divide x)*
*(x,y) <- (f x,f y)*
*(x',y') <- (conquer (x',y'))*

• vertical split/unfold
Example: divide & conquer

\[(x) \leftarrow (\text{divide } x)\]

\[(x') \leftarrow (f \ x)\]

\[(y') \leftarrow (f \ y)\]

\[(x'',y'') \leftarrow (\text{conquer } (x'',y''))\]

- horizontal split

box sumsq4 in \((x)\) out \((x_1,x_2)\)
match \((x)\) -> \((x \div 2, x \div 3)\);

box process1 in \((x_1)\) out \((x_1')\)
match \((x_1)\) -> \((x_1 \times x_1)\);

box process2 in \((x_2)\) out \((x_2')\)
match \((x_2)\) -> \((x_2 \times x_2)\);

box conq in \((x_1,x_2)\) out \((s)\)
match \((x_1,x_2)\) -> \((x_1 + x_2)\);

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 superstep 2
Costulator

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

File  Rule  Code  COST
Costulator

File  Rule  Code  COST
Costulator

File  Rule  Code  COST

Identity Split b Join box Add in Add out ...

name?

x -> x

name?

initially: 0
space:
link: 2
pattern: 0
cost: 4
Costulator

File  Rule  Code  COST

x -> x  inc

initially: 0  
space: 
link: 2  
pattern: 0  
cost: 4
Costulator

File
Rule
Identity
Split box >
Join box >
Add in
Add out...

Code

x -> (x,y)

COST

inc

initially: 0
space:
link: 3
pattern: 0
cost: 6
Costulator

- File
- Rule
- Code

COST

- inc
  - initially: 0
  - space: 0
  - link: 3
  - pattern: 0
  - cost: 6

\[ x \rightarrow (x, y) \]
Costulator

\[
x \rightarrow (x, y)
\]

**COST**

- inc
- initially: 0
- space: 
- link: 3
- pattern: 0
- cost: 6
Costulator

File  Rule  Code  COST

\[ x \rightarrow (x,y) \]

inc

initially: 0
space: link: 3
pattern: 0
cost: 6
Costulator

\[ x \rightarrow (x, y) \]

- **File**: inc
- **Rule**: pattern: 0
- **Code**: link: 3
- **COST**: initially: 0
  - space: 
  - link: 3
  - pattern: 0
  - cost: 6

26/8/13 CPA 2013 Napier
Costulator

\[ x \rightarrow (x, x) \]
Costulator

\[ x \rightarrow (x+1, x) \]

initially: 0
space: link: 3
pattern: 0
cost: 9
Costulator

File  Rule  Code

COST

name: ?
initially: ?

x -> (x+1,x)

inc

initially: 0
space:
link: 3
pattern: 0
cost: 9
Costulator

File | Rule | Code | COST

name: n
initially: 0

x -> (x+1, x)

cost: 9

initially: 2
space: 3
link: 3
pattern: 0

Costulator

\[ x \rightarrow (x + 1, x) \]
Costulator

File  Rule  Code  COST

initially: 2
space: 0
link: 3
pattern: 0
cost: 9

x -> (x+1,x)

0

n  inc

n'  r
Costulator

```
Show

Compile

Run

n
x -> (x+1,x)

n'
r

COST

inc

initially: 2
space:
link: 3
pattern: 0
cost: 9
```
Costulator

box inc
in (n)
out (n',r)
match
(x) -> (x+1,x);
wire inc
(inc.n' initially 0)
(inc.n);

0
n
x -> (x+1,x)
n'
r

COST

inc

initially: 2
space: 2
link: 3
pattern: 0
cost: 9
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](http://www.macs.hw.ac.uk/~greg/hume)