Aldor Compiler Internals II:

Code Generation and Optimisation

Martin N Dunstan

Numerical Algorithms Group (NAG Ltd)

Road Map

- **Code generation:**
  - **review key data-structures**
  - **from abstract syntax to abstract machine code**
  - **what does FOAM actually look like?**
  - **the Aldor runtime system**
- **Code optimisation:**
  - **tools of the trade: flow graphs, data-flow analysis**
  - **constant propagation, common sub-expressions**
  - **inlining, environment merging, flow control**
  - **dead variable/assignment elimination**

Review Key Data-structures

- TForm: type-form
  - many begin life as tfSyntax; tfGeneral very common
  - tfSubst is also common; transformed into tfForward
  - specialised versions: tfUnion, tfMap, tfCross *etc*
  - tfTrigger for lazy-loading tforms
- Syme: symbol meaning
  - single-most used objects in the whole compiler
  - appear everywhere, hold important information
  - used from scobind through tinfer and optimiser
- Foam: intermediate code

Abstract Syntax to FOAM

- *N*-ary tree; mirrors Aldor source:
  - every node has usage info (in-value, in-type, declare)
  - every node has a unique type (tform)
  - leaf nodes have meaning (syme, defnIdx, embed)
  - many nodes have link to source position
- Whole tree processed as a domain body:
  - search for function definitions (const numbers)
  - identify deep identifier usage (lexicals)
  - create FOAM closures for function values
  - create FOAM for top-level lexicals, globals *etc*
  - generate FOAM for top-level definitions

FOAM Generation Part I

- **Some nodes are easy to compile:**
  - **recurse a lot via `genFoam()`**
- **Some nodes are harder:**
  - **functions, loops, where, add and with**
  - **generate "dumb" code and leave for optimiser**
- **Definitions, domains and categories:**
  - **order of definition/initialisation is critical**
  - **laziness is vital for performance (space and time)**

Generating `for` Loops

```
for elt in L repeat { tot := tot + elt; } show(tot);
```

```
      local source:   () -> (()->Bool, ()->(), ()->SInt, ()->SInt);
      local stepFun:  () -> ();
      local doneFun:  () -> Bool;
      local valueFun: () -> SInt;
      local boundFun: () -> SInt;

      -- Initialise iterators
      source := generator(L);
      (doneFun, stepFun, valueFun, boundFun) := source();
      goto L1;
      -- Loop body
@L0   tot := tot + elt;
      -- Iterators and loop tests
@L1   stepFun();
      if doneFun() then goto L2;
      elt := valueFun();
      goto L0;
@L2   show(tot);
```

Generated FOAM

```
(Set (Loc 0 tot)
   (Cast Word (CCall Word (Glo 13 lazyForceImport) (Lex 1 18 \0))))
(Def (Values (Loc 2) (Loc 3) (Loc 4) (Loc 5))
   (MFmt 7 (CCall NOp (CCall Clos (Lex 1 15 generator) (Par 0 L)))))
(Def (Loc 6) (Loc 2)) -- done?: () -> Boolean
(Def (Loc 7) (Loc 3)) -- step!: () -> ()
(Def (Loc 8) (Loc 4)) -- value: () -> SingleInteger
(Def (Loc 9) (Loc 5)) -- bound: () -> SingleInteger
(Goto 1)
(Label 0)
(Set (Loc 0 tot) (CCall Word (Lex 1 19 +) (Loc 0 tot) (Loc 1 elt)))
(Label 1)
(CCall NOp (Loc 7)) -- step!()
(If (Cast Bool (CCall Word (Loc 6))) 2) -- if (done?())
(Set (Loc 1 elt) (CCall Word (Loc 8)))  -- elt := value()
(Goto 0)
(Label 2)
(PCall C NOp (Glo 12 show) (Loc 0 tot))
```

Generating Generators

```
generate { while L repeat { yield first L; L := rest L; } }
```

```
local generDone?:Bool := false;
local yieldPlace:SInt := 0;
local yieldValue:T;

-- Main generator function
step!():() ==
{
        select yieldPlace in { 0 => goto L0; 1 => goto L2; }
   @L0  generDone? := false;
   @L1  if (empty? L) then goto L3;
        yieldPlace := 1; yieldValue := first L; return;
   @L2  L := rest L;
        goto L1;
   @L3  generDone? := true;
        return;
}

value():T == yieldValue;
done?():Bool == generDone?;
bound():SInt == -1;
```

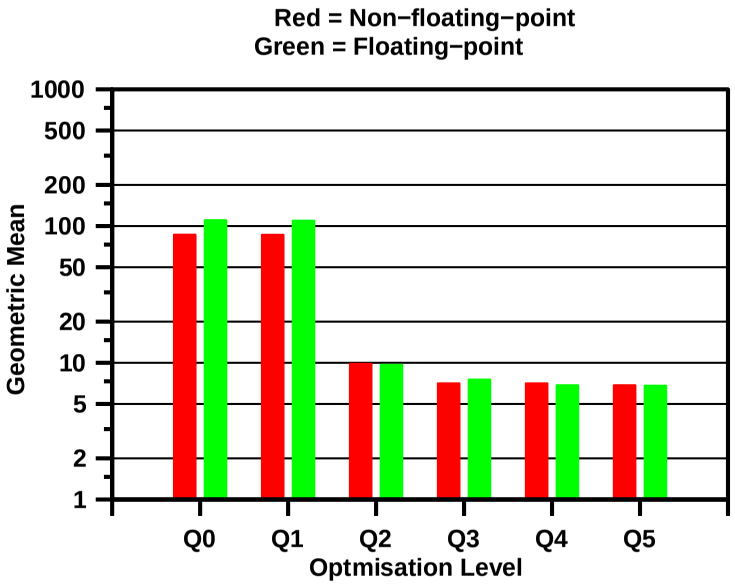
FOAM Generation Part II

- Generating code for add bodies:
  - represented by Record with several slots
  - created with one field (closure A) initialised
  - when called, A fills in hash and another closure B
  - B invoked: full initialisation
  - order of export initialisation is delicate
  - imports are also lazy; need forcing
- Why are we so lazy?
  - domain initialisation is expensive (*e.g.* SUP)
  - don't want to initialise whole domain for its hash
  - laziness unavoidable for mutually recursive domains

Loose Ends

- **Runtime system written in C and Aldor:**
  - **2400 lines of code in Aldor**
  - **5000 lines of library support code (AxlLib variant)**
  - **20000 lines of C support functions (GC, big-ints *etc*)**
- **Particular issues:**
  - **no "gets" allowed in `runtime.as` (infinite recursion)**

Why Use Optimisation?

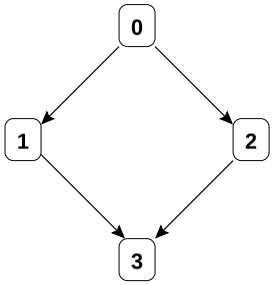


Optimisation Tools

- **Flow-graphs (flogs):**
  - **directed graphs with *basic blocks* (BBs) as nodes**
  - **holds data-flow class, entry-point BB, original FOAM**
- **Basic-blocks:**
  - **have exactly one entry point and one exit point**
  - **may be reached by zero or more other BBs**
  - **may target zero or more other BBs**
  - **may correspond to labelled FOAM sequences**
  - **slots for application-specific information**
- **Data-flow analysis**

Flow-graph/BB Example

```
if (a < b) then
   min := a;
else
   min := b;
print << min;
```




```
0: (If (BCall SIntLT
     (Loc 0) (Loc 1)) 2)
1: (Set (Loc 2) (Loc 1))
   (Goto 3)
2: (Set (Loc 2) (Loc 0))
3: (CCall Word
     (Lex 0 0) (Loc 2))
```

Data-flow Analysis

- **Compiler needs data-flow information:**
  - **examples include liveness, reachability, use-def**
  - **obtained by solving system of (data-flow) equations**
  - $out[S] = gen[S] \cup (in[S] - kill[S])$ ***i.e.*** **info-*out* is either *generated* in statement S, or is info-*in* not *killed* by S**
- **How do we define and solve these equations?**
  - **number each statement; bit-vectors for sets**
  - **problem-specific meanings for *in*, *out*, *gen*, *kill***
  - **BBs as statements: compute gen/kill + in/out**
  - **apply standard algorithm to whole flog**
  - **solve by forward or backward iteration until stable**

"Simple" Optimisations

- Copy propagation (cprop)
  - perform *use-def* data-flow analysis
  - transform L1 := L0; L2 := L1; foo(L2) into foo(L0)
- Constant and hash folding (cfold, ffold, hfold)
  - cprop first then evaluate constants/bcalls
  - transform L1 := 2*4 + 15 into L1 := 23
  - hfold eliminates `domainGetHash!()` calls
- Peep-hole: similar to cfold
  - transform L1 := complexExpr $\wedge$ false into L1 := false
  - only apply to side-effect free expressions

Optimisation: CSE

- Common sub-expression elimination:
  - walk sequence building table of equal expressions
  - data-flow analysis to identify *reaching* expressions
  - replace common expressions with locals
  - L0 := (t1*t1) + (t1*t1) becomes L1 := t1*t1; L0 := L1 + L1
- As always, must be careful:
  - lex-vars and fun-calls: (+ (+ (Lex 2 1) 1) (CCall foo ...))
  - common expressions must share execution paths
- CSE conflicts with cprop

Example of CSE

```
@L0   if someTest then goto L3;
      select (3*y rem 1) in { 0 => goto L1; 1 => goto L2; never; }
@L1   t0 := x + 3*y;
      t1 := 4*a*c;
      goto L3;
@L2   t0 := z + 3*y;
      t1 := 5*a*c;
@L3   t2 := t0 + t1 + a*c;
```

- **Common expressions: 3*y (okay) and a*c (bad)**

```
@L0   if someTest then goto L3;
      T := 3*y;
      select (T rem 1) in { 0 => goto L1; 1 => goto L2; never; }
@L1   t0 := x + T;
      t1 := 4*a*c;
      goto L3;
@L2   t0 := z + T;
      t1 := 5*a*c;
@L3   t2 := t0 + t1 + a*c;
```

Optimisation: Inliner

- Vital component of the optimiser:
  - replaces function calls with function body
  - reduces cost of function and closure calls
  - parameters replaced with locals if necessary
  - cprop may eliminate some/all extra locals
- Enables other optimisations to work better:
  - removing fun-calls means less heap escapes (emerge)
  - low-level if-tests can be eliminated (jflow)
- Must limit the growth in program size:
  - however, inlining often *decreases* program size

Inliner Details

- **Preparation:**
  - **build priority queue of all OCall/CCalls in a prog**
  - **priority based on number of calls, inlinee size *etc***
  - **love small inlinees and those not touching their env**
  - **use symes to trace inlinee origin and value**
- **Inlining:**
  - **remove call at front of queue and inline (repeatedly)**
  - **stop when prog too big (bytes, statements, bbs)**
  - **drag across remote OCalls invoked by inlinee**
  - **add inlinee's calls to priority queue**
  - **push new env if inliner/inlinee have different parents**

Inliner Example: Part I

- **Starting with the code:**

```
tot := 0;
for elt in generator(L) repeat { tot := tot + elt; }
show(tot);
```

- **. . . we inline `generator(L)` to obtain:**

```
    tot := 0;
    goto L1;
@L0 tot := tot + elt;
@L1 generStep();
    if generDone() then goto L2;
    elt := generValue();
    goto L0;
@L2 show(tot);
```

- **then we inline `generStep()`, re-order *etc* . . .**

Inliner Example: Part II

- . . . and we obtain the following:

```
@L0   tot := 0;
      yieldPlace := 0;
@L1   select yieldPlace in { 0 => L5; 1 => L10 }
@L5   generDone? := false;
@L8   if (nil? L) then goto L9;
      yieldPlace := 1;
      yieldValue := L.first;
      t9 := yieldValue;
@L11  if (generDone?) then goto L2;
      tot := tot + t9;
      goto L1;
@L9   generDone? := true;
      goto L11;
@L10  L := L.rest;
      goto L8;
@L2   show(tot);
```

Optimisation: JFlow

- **Jump removal and control-flow modification:**
  - **removes "mess" introduced by inlining generators**
  - **re-target goto statements to ultimate destination**
  - **merge consecutive, unconditional basic-blocks**
  - **normalise if-tests and eliminate repetitions**
- **Examine control-variables:**
  - **clone basic-blocks for finite sets of values**
  - **eliminate if/select nodes with known execution path: transform `if true goto L23` into `goto L23`**

Re-targetting and Merging

BB	Before	After
0:	`(Set (Loc 0) (SInt 2))` `(Goto 2)`	`(Set (Loc 0) (SInt 2))` `(Set (Loc 1) (Loc 0))`
1:	`(Goto 3)`	
2:	`(Goto 1)`	
3:	`(Set (Loc 1) (Loc 0))`	

JFlow Example: Part I

- We reach jflow after inlining with:

```
@L0   tot := 0;
      yieldPlace := 0;
@L1   select yieldPlace in { 0 => L5; 1 => L10 }
@L5   generDone? := false;
@L8   if (nil? L) then goto L9;
@L13  yieldPlace := 1;
      yieldValue := L.first;
      t9 := yieldValue;
@L11  if (generDone?) then goto L2;
@L12  tot := tot + t9;
      goto L1;
@L9   generDone? := true;
      goto L11;
@L10  L := L.rest;
      goto L8;
@L2   show(tot);
```

- Specialise L0, retarget gotos, merge L12 with L1

JFlow Example: Part II

```
@L0  tot := 0;
     yieldPlace := 0;
@L5  generDone? := false;
@L8  if (nil? L) then goto L9;
@L13 yieldPlace := 1;
     yieldValue := L.first;
     t9 := yieldValue;
@L11 if (generDone?) then goto L2;
@L12 tot := tot + t9;
     select yieldPlace in { 0 => L5; 1 => L10 }
@L9  generDone? := true;
     goto L11;
@L10 L := L.rest;
     goto L8;
@L2  show(tot);
```

- **Specialise L9 and L11, re-target and merge . . .**

JFlow Example: Part III

```
@L0   tot := 0;
      yieldPlace := 0;
@L5   generDone? := false;
@L8   if (nil? L) then goto L9;
@L12  yieldPlace := 1;
      yieldValue := L.first;
      t9 := yieldValue;
      tot := tot + t9;
      select yieldPlace in { 0 => L5; 1 => L10 }
@L9   generDone? := true;
      goto L2;
@L10  L := L.rest;
      goto L8;
@L2   show(tot);
```

- **Specialise select to L10, re-target, merge *etc* . . .**

JFlow Example: Part IV

```
@L0   tot := 0;
      yieldPlace := 0;
      generDone? := false;
@L8   if (nil? L) then goto L9;
@L12  yieldPlace := 1;
      yieldValue := L.first;
      t9 := yieldValue;
      tot := tot + t9;
      L := L.rest;
      goto L8;
@L9   generDone? := true;
      show(tot);
```

- **Dead code elimination, cprop *etc*:**

```
      tot := 0;
@L0   if (nil? L) then goto L1;
      t9 := L.first;
      tot := tot + t9;
      L := L.rest;
      goto L0;
@L1   show(tot);
```

Optimisation: Emerger

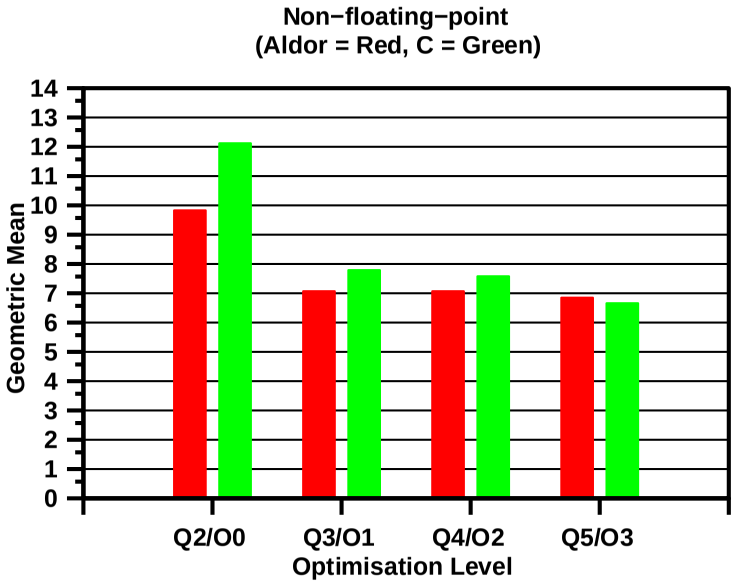
- Place heap-allocated objects on the stack:
  - eliminates environments and other heap objects
  - walk sequence marking def/sets and inferring type
  - process RNew, ANew, PushEnv instructions
  - explode non-escaping heap-store into locals

```
(Set (Loc 1) (RNew 8))
(Set (RElt 8 (Loc 1) 0) (DFlo 2.5))
(Set (Loc 0) (RNew 8))
(Set (RElt 8 (Loc 0) 0) (DFlo 3.5))
(Set (Loc 2) (RNew 8))
(Set (RElt 8 (Loc 2) 0)
  (BCall DFloPlus
    (RElt 8 (Loc 1) 0)
    (RElt 8 (Loc 0) 0)))
```

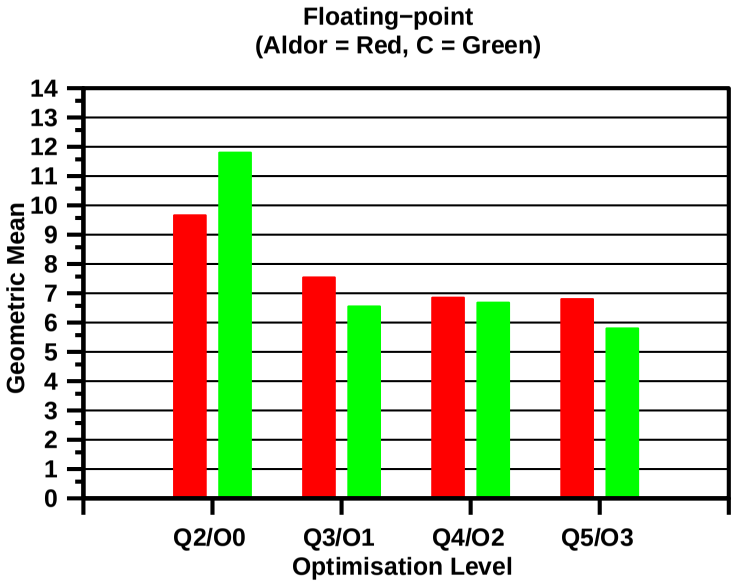


```
(Set (Loc 0)
  (BCall DFloPlus
    (DFlo 2.5)
    (DFlo 3.5)))
```

Aldor *vs* C (Part I)



Aldor *vs* C (Part II)



Summary

- **Our intermediate code:**
  - **provides uniform platform for the code generator**
  - **can be loaded from libraries at compile time**
  - **allows heavy, platform-independent optimisation**
  - **can support different object codes (including LISP)**
- **Compiler and language:**
  - **continually being improved and updated**