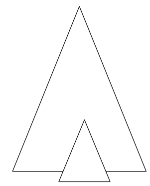
Part I: Specifying Transformations

Oege de Moor
Ganesh Sittampalam
Programming Tools Group, Oxford

AST rewriting



To apply rule:
rewrite(pat_0, pat_1)

- Match:
  focus = ϕ(pat_0)
- Replace:
  focus := ϕ(pat_1)

Excellent for transforming declarative programs

Imperative object programs

Constant propagation:

```
{
   n = 10;
   if (n > 0) {
      c = 0;
      n = n-1;
   } else {
      c = 0;
      n = n+1;
   }
   x = c;
}
```

⇨

```
{
   n = 10;
   if (n > 0) {
      c = 0;
      n = n-1;
   } else {
      c = 0;
      n = n+1;
   }
   x = 0;
}
```

The need for path queries

rewrite(assign(localvar(X),localvar(V)),
  assign(localvar(X),const(C))) :-
    ``*all paths from entry to focus guarantee V=C*".

To apply this rule:
- match:
  focus = ϕ(assign(localvar(X),localvar(V))
- solve path query:
  $\psi \in$ all(focus.fromEntry, "guarantee V=C")
- replace:
  focus := ψ(ϕ(assign(localvar(X),const(C)))

fromentry

```
{
   n = 10;
   if (n > 0) {
      c = 0;
      n = n-1;
   } else {
      c = 0;
      n = n+1;
   }
   x = c;
}
```

focus.fromentry
=
{
  [n =10; ?(n>0); c=0; n=n-1; x=c],
  [n =10; ?(n>0); c=0; n=n+1; x=c]
}

paths are sequences of "atomic" statements

fromentry, *paths*, *toexit*

fromentry : all paths from program entry up to including the focus

paths : all paths that start and end at focus (e.g. all paths through a loop body)

toexit : all paths from (and including) the focus to the program exit

these are *attributes* of each AST node

Defining paths

paths : defined bottom-up
upto : (like *fromentry*, but excluding focus) defined top-down

stmt$_0$ ::= **while** expr stmt$_1$

stmt$_0$.paths = expr.paths ; (stmt$_1$.paths ; expr.paths)*

expr.upto = stmt$_0$.upto; (expr.paths ; stmt$_1$.paths)*

stmt$_1$.upto = expr.upto; expr.paths

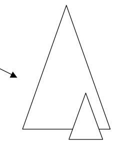
stmt$_0$.fromentry = stmt$_0$.upto ; stmt$_0$.paths

Path logic programming

Prolog query with new primitives → interpret with respect to both

fromentry,
toexit,
paths

to check properties of *all elements* of the corresponding attributes of the focus



logic program:
P(X,Y) :- Q(X), R(X,Y).
A(u,v).
B(s,t).
....

Constant propagation (0)

```
const_prop(E,F)
  :-
    baseinstr(E),
    fromentry(

        {}*;

        {'assign_const(V,C,T)};
        {cnot('def_var(V))}*;

        {'use_var_type(V,T)}
       ),
    subst(C,expr_type(localvar(V),T),E,F) .
```

all paths to focus satisfy this pattern:
- *zero or more statements that don't matter*
- *an assignment V := C*
- *zero or more statements that do not redefine V*
- *a use of V*

Constant Propagation (1)

```
const_prop(E,F)
  :-
    baseinstr(E),
    fromentry(
        {}*;
        {'assign_const(V,C,T)};
        {cnot('def_var(V))}*;
        {'use_var_type(V,T)}
       ),
    subst(C,expr_type(localvar(V),T),E,F) .
```

normal Prolog predicate:

baseinstr(*instr_label*(_,I,_))
  :- **not**(nonterminal(I)).
baseinstr(*condexpr*(_)).

subst(V,X,X,V).

subst(V,X,*assign*(L0,R0),*assign*(L1,R1))
  :- **not**(X=L0),
  subst(V,X,L0,L1),
  subst(V,X,R0,R1).

… *etc* …

Constant Propagation (2)

```
const_prop(E,F)
  :-
    baseinstr(E),
    fromentry(
        {}*;
        {'assign_const(V,C,T)};
        {cnot('def_var(V))}*;
        {'use_var_type(V,T)}
       ),
    subst(C,expr_type(localvar(V),T),E,F) .
```

ticked predicate 'P(X)
P(X,I) : a property of an instruction I

localvar(*expr_type*(*localvar*(V),T),V,T).
const(*expr_type*(*const*(N),T),N,T).
assign(L,R,T,*expr_type*(*assign*(L,R),T)).

assign_const(V,C,T,I)
  :- assign(L,C,T,I),
  localvar(L,V,T),
  const(C,_,T).

use_var_type(V,T,I) :- use(*expr_type*(*localvar*(V),T),I).

use(E,I) :- **not**(isassign(I)),occurs(I,E).
use(E,*exp*(F)) :- assign(_,R,_,F),occurs(R,E).
use(E,*exp*(F)) :- assign(L,_,_,F),**not**(L=E),occurs(L,E).

Constant propagation (3)

```
const_prop(E,F)
  :-
    baseinstr(E),
    fromentry(
        {}*;
        {'assign_const(V,C,T)};
        {cnot('def_var(V))}*;
        {'use_var_type(V,T)}
       ),
    subst(C,expr_type(localvar(V),T),E,F) .
```

def_var(V,*exp*(E)) :-
  assign(L,_,_,E),localvar(L,V,_).

Implementation notes:
Predicates inside {…} are solved independently: cannot assume any variables to be bound.

Solutions of {…} (substitution sets) are turned into *constraints* (propositional formulae combining equations).

cnot denotes *constraint negation*

cnot(def_var(V,"x=3"))	≡	V ≠ x
not(def_var(V,"x=3"))	≡	false

Common Subexp Elimination

```
x = ack(10,20);

… no changes to x or to ack(10,20) …

y = 20 * ack(10,20);
```

⇩

```
x = ack(10,20);

… no changes to x or to ack(10,20) …

y = 20 * x;
```

Common Subexp Elimination

```
cse(I,J)
 :-
   baseinstr(I),

   fromentry(
               {}*;

               { 'assign_var(V,Exp,T),pure(Exp),nontriv(Exp),
                 not(occurs(Exp,localvar(V)))};

               { cnot('def_var(V)), cnot(delay('somedef(Exp))) }*;

               { 'use(Exp) }        ),

   subst(expr_type(localvar(V),T),Exp,I,J).
```

"delay" needed: *somedef*(*Exp*,*I*) may give infinite number of *Exp* for fixed *I*

Dead assignment elimination

```
while (x > 0) {
        x = x-1;
        v = v+1;
}
… no further use of v …
```

⇨

```
while (x > 0) {
        x = x-1;
}
…
```

Dead assignment elimination

```
dead_code(instr_label(Labs,E,Annot),
        instr_label(Labs,exp(expr_type(applyatom(nop,nil),void)),Annot))
 :-
   not(nonterminal(E)),

   toexit(
        { 'assign_var(V,R,_), pure(R),  atnode(N) };

        ( {cnot('use_var(V))} | {atnode(N)} )*;

        ( ε  |  ( { 'def_var(V),not('use_var(V)) } ; {}* )  )  ).
```

atomic statements have identity

Unique use propagation

```
x = ack(10,20);
…. no defs of x or
        uses of x or
        defs of ack(10,20) …
y = x+1;
…. no uses of x anywhere else …
```

⇘

```
x = ack(10,20);
…. no defs of x or
        uses of x or
        defs of ack(10,20) …
y = ack(10,20) +1;
…. no uses of x anywhere else …
```

Unique use propagation

```
unique_prop(I,J)
 :-
   baseinstr(I),

   fromentry({ }*;
           { 'assign_var(V,Exp,T), pure(Exp),
             not(occurs(Exp,localvar(V))) };
           { cnot('def_var(V)), cnot(delay('somedef(Exp))) }*;
           { atnode(N),'uniq_use_var_type(V,T) }),

   fromentry( ({cnot('use_var(V))} | {atnode(N)})* ; { } ),

   toexit( { } ; ({cnot('use_var(V))} |{atnode(N)})* ),

   subst(Exp,expr_type(localvar(V),T),I,J).
```

Strength reduction

```
while (i > 0) {
    n = i * ack(10,20);
    t = t + n;
    i = i + 1;
}
```

⇨

```
{
  long c,x;
  c = ack(10,20);
  x = i*c;
  while (i > 0) {
      n = x;
      t = t + n;
      i = i + 1;
      x = x + c;
  }
}
```

Strength reduction

strengthred(*while*(Cond,Body),*seq*(Init,*while*(Cond,Body')))
:-
paths(Body, { **cnot**('def_var(I)) **cor** 'incr(I) }*;
{ 'incr(I) };
{ **cnot**('def_var(I)) **cor** 'incr(I) }*),
paths(Cond, **cnot**('def_var(I))),

times(E,I,C), occurs(E,Body), pure(C),

paths(Body, {**cnot**(**delay**('some_def(C)))}*),
paths(Cond, {**cnot**(**delay**('some_def(C)))}*),

… *construct Init and Body'* … .

Combining rewrite rules

strategy: a way of applying rewrite rules
apply : rule → strategy

s_0 <+ s_1	: choice
s_0 ; s_1	: sequencing
id	: skip

try(s) = s <+ id

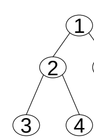
exhaustively(s) = try(s;exhaustively(s))

[Visser *et al*: Stratego]

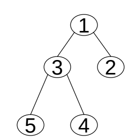
Moving the focus

below(s) :	apply to immediate descendants, left-to-right
woleb(s) :	same, right-to-left

leftmost(s) = s;below(leftmost(s))



rightmost(s) = s;woleb(rightmost(s))



A complete strategy

incremental : (logprog × strategy) → strategy

incrlil =
incremental("optimise.lp" , ← *path logic program defining trafos*

forward analyses: fromentry + leftmost
leftmost(exhaustively(apply(*unique_prop*));

exhaustively(apply(*const_prop*)));

backward analyses: toexit + rightmost
rightmost(try(apply(*dead_code*)));

leftmost(exhaustively(apply(*cse*))))

Summary

- Transformation by rewriting AST
- Individual rules are specified as "*path logic programs*":
  - fromentry, toexit
  - interpreted relative to logic fact base + focus in AST
- Rules are combined with *strategies*

Part II: Chip-chop Theory

Conway's "language factors"

Checking conditions

program paths $\subseteq$ pattern

context-free *regular*

pattern may contain free variables

(ignore variables for now)

Obvious algorithm

Let P be push-down automaton for program traces
Let S be regular automaton for negation of pattern

Compute product automaton $P \times S$

If language of product is empty, we have

prog $\subseteq$ pattern

Incremental checking

- re-checking after each rewrite is expensive
- hence aim to re-use intermediary results
- need to check “parts” of a regular expression

Defining paths (reminder)

paths, fromentry : defined bottom-up
upto : (like *fromentry*, but excluding focus) defined top-down

$stmt_0$ ::= **while** expr $stmt_1$

$stmt_0$.paths = expr.paths ; ($stmt_1$.paths ; expr.paths)*

expr.upto = $stmt_0$.upto; (expr.paths ; $stmt_1$.paths)*

$stmt_1$.upto = expr.upto; expr.paths

$stmt_0$.fromentry = $stmt_0$.upto ; $stmt_0$.paths

Wishful thinking (1)

test prog = prog $\subseteq$ pat

would like a compositional form (for some $\oplus$, $\otimes$):

test (p_1 + p_2) = test p_1 $\oplus$ test p_2
test (p_1 ; p_2) = test p_1 $\otimes$ test p_2

this would allow computation of inclusions while we build up attributes

after each rewrite, recompute (synthesised) *paths* attributes on path to root; recompute (inherited) *fromentry* and *toexit* while walking back down to focus

Wishful thinking (2)

test prog = prog ⊆ pat

would like a compositional form (for some ⊕, ⊗):

test (p_1 + p_2) = test p_1 ⊕ test p_2
test (p_1 ; p_2) = test p_1 ⊗ test p_2

⊕, ⊗ do not exist, need to generalise:

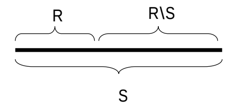
tests prog = { (pat', prog ⊆ pat') | pat' is a "part" of pat }

but what is the definition of "part"?

Chip and chop

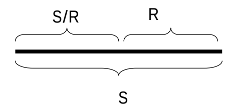
chip R from front of S

$T \subseteq R \setminus S \quad \equiv \quad R ; T \subseteq S$



chop R from back of S

$T \subseteq S / R \quad \equiv \quad T ; R \subseteq S$



Properties of chip and chop

$T \subseteq R \setminus S \quad \equiv \quad R ; T \subseteq S$

$T \subseteq S / R \quad \equiv \quad T ; R \subseteq S$

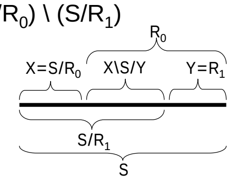
prove:

$(R \setminus S) / T \quad = \quad R \setminus (S / T)$
$R / (S ; T) \quad = \quad R \setminus (T \setminus S)$

Chip-chop theory

- S is regular iff
  there are finitely many chops S/R
- for any S, X and Y:
  $X \setminus S / Y = (S/R_0) \setminus (S/R_1)$
  for some R_0, R_1.

X\S/Y is a "part" of S



Theorem

$U;V \subseteq (S / R_0) \setminus (S / R_1)$

iff

$\exists R: \quad U \subseteq (S / R_0) \setminus (S / R)$
$\wedge$
$V \subseteq (S / R) \setminus (S / R_1)$

Chip-chop matrix of regular S

indexed by finite set of chops of S

$M(S)_{X,Y} = X \setminus Y$

S occurs in M(S)

Example 1

pattern: S = a* ; b ; c* ; d

chops: 0, a*, a*;b;c*, S, Σ*

chip-chop matrix:

$$\begin{pmatrix} \Sigma^* & \Sigma^* & \Sigma^* & \Sigma^* & \Sigma^* \\ 0 & a^* & a^*;b;c^* & S & \Sigma^* \\ 0 & 0 & c^* & c^*;d & \Sigma^* \\ 0 & 0 & 0 & \varepsilon & \Sigma^* \\ 0 & 0 & 0 & 0 & \Sigma^* \end{pmatrix}$$

Example 2

pattern: S = (a*;b + b;b*;a)*

chops: 0, S;b;b*, S, (a+b)*

chip-chop matrix:

$$\begin{pmatrix} (a+b)^* & (a+b)^* & (a+b)^* & (a+b)^* \\ 0 & (\varepsilon + b^*;a);S;b;b^* + b^* & (\varepsilon+b^*;a);S & (a+b)^* \\ 0 & S;b;b^* & S & (a+b)^* \\ 0 & a^*;b;S;b;b^* & a^*;b;S & (a+b)^* \end{pmatrix}$$

Matching matrix of pattern S

$B(R)_{X,Y} = R \subseteq M(S)_{X,Y}$

$$\begin{aligned} B(R_0 + R_1) &= B(R_0) \wedge B(R_1) \\ B(R_0 \,;\, R_1) &= B(R_0) \times B(R_1) \\ B(R^*) &= B(\varepsilon) \;\wedge\; (B(R) \times B(R^*)) \end{aligned}$$

Complexity

C	:	number of chops
N	:	size of program
D	:	depth of AST
P	:	number of rewrite steps
Q	:	maximum number of new nodes per rewrite

time:
$\mathcal{O}(C^4 \cdot (N + P \cdot (Q + D)))$
space:
$\mathcal{O}(C^2 \cdot N)$

Summary

- Chips and chops are the "parts" of a formal language
- For regular language, finite number of chips and chops
- Arranging these in a matrix reduces the language inclusion problem to simple matrix operations

Part III: Implementation

- generalise from inclusion problem to free variables
- how to represent constraints

Free variables

prop	= "finite set of atomic propositions"
constraints	= "equalities, inequalities, and, or"
statement	= prop $\rightarrow$ constraints

p_0= { assign(X,C), constant (C) }
p_1= { not (def(X)) }

prop = { p_0, p_1 }

(a := 3) $\equiv$ { $p_0 \rightarrow$ (X=a $\wedge$ C=3), $p_1 \rightarrow$ (X$\neq$a) }

Free variables, continued

prop	= "finite set of atomic propositions"
constraints	= "equalities, inequalities, and, or"
statement	= prop $\rightarrow$ constraints

program : statement list set

pattern : prop list set

wish to compute constraint:
C(program,pattern) =
$\wedge$ (x $\in$ program :
$\vee$ (y $\in$ pattern :
$\wedge$ (x $\odot$ y))) **pointwise application**

Example

p_0= { assign(X,C), constant (C) }
p_1= { not (def(X)) }

prop = { p_0, p_1 }

(a := 3) $\equiv$ { $p_0 \rightarrow$ (X=a $\wedge$ C=3), $p_1 \rightarrow$ (X$\neq$a) }
(? c==0) $\equiv$ { $p_0 \rightarrow$ false, $p_1 \rightarrow$ true }

prog : a := 3; if (d==0) {b:=0;} {b:=0;a :=2;}
pat : (true*; p_0; p_1*) | p_1*

C(prog,pat) $\equiv$ (X=b $\wedge$ C=0) $\vee$ (X $\neq$ a $\wedge$ X $\neq$ b)

Generalised inclusion

prop	= "finite set of atomic propositions"
constraints	= "equalities, inequalities, and, or"
statement	= prop $\rightarrow$ constraints

program : statement list set

pattern : prop list set

wish to compute constraint:
C(program,pattern) =
$\wedge$ (x $\in$ program :
$\vee$ (y $\in$ pattern :
$\wedge$ (x $\odot$ y)))

$pat_1 \subseteq pat_2 \equiv C(f(pat_1),pat_2)$
where
f(p)(q) $\equiv$ if (p=q) then true else false

A *false* conjecture

$B(prog)_{X,Y}$ = C($M(pat)_{X,Y}$, prog)

$B(R_0 + R_1)$	=	$B(R_0) \wedge B(R_1)$
$B(R_0 ; R_1)$	?=	$B(R_0) \times B(R_1)$
$B(R^*)$	=	$B(\varepsilon) \wedge (B(R) \times B(R^*))$

no!

Intuitively, it fails because
a statement may match multiple propositions

Lifting the pattern

Given
  pat : prop list set
Define
  pat' : prop set list set
by
  $[xs_0, xs_1, \ldots, xs_{n-1}] \in pat'$
  $\equiv$
  $\exists\, x_i \in xs_i : [x_0, x_1, \ldots, x_{n-1}] \in pat$

Generalised application of statement s : prop $\rightarrow$ constraint
  $s(\{p_0, p_1, \ldots, p_{k-1}\}) = s(p_0) \wedge \ldots \wedge s(p_{k-1})$

Theorem: C(pat,prog) $\equiv$ C(pat',prog)

Fixing the conjecture

$B(prog)_{X,Y} = C(\ M(pat')_{X,Y}, prog)$

$B(R_0 + R_1)$	$=$	$B(R_0) \wedge B(R_1)$	
$B(R_0\ ;\ R_1)$	$?=$	$B(R_0) \times B(R_1)$	yes!
$B(R^*)$	$=$	$B(\varepsilon)\ \wedge\ (B(R) \times B(R^*))$	

Proofs rather technical... see draft paper

Representing constraints

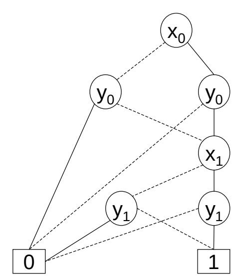
Example:
$(X{=}b \wedge C{=}0) \vee (X \neq a \wedge X \neq b)$

Characteristics:
- Boolean combinations of equations (var = val)
- finite number of variables
- finite number of values
- many, many "similar" constraints

Encode constraints via Binary Decision Diagrams

Binary Decision Diagrams

A representation of propositional formulae



$(x_0 \equiv y_0) \wedge (x_1 \equiv y_1)$

true
false

easy to encode
constraints over arbitrary
finite domain
via bit representation

Finite domains?

cse(I,J)
 :-
  baseinstr(I),

  fromentry(
      {}*;

      { 'assign_var(V,Exp,T),pure(Exp),nontriv(Exp),
       not(occurs(Exp,*localvar*(V)))};

      { **cnot**('def_var(V)), **cnot**(**delay**('somedef(Exp))) }*;

      { 'use(Exp) }),

  subst(*expr_type*(*localvar*(V),T),Exp,I,J).

"delay" needed:
somedef(*Exp*,*I*)
may give
infinite number of
Exp for fixed *I*

Finite domains!

somedef(Exp,a =3) = "Exp contains a"

introduce special boolean variable
for each meta-variable E and program variable a:

  $E \unrhd a$

while solving path query:

somedef(Exp,a := 3)
$\equiv$
$(Exp \unrhd a)$

when a transformation is to be applied:
- let C be the constraint for the path query
- for each a, let $X(a) = \{ e_0, e_1, \ldots e_{n-1}\}$ be the expressions that contain e
- let C'
  $= C \wedge$
  $\bigwedge E, a : (E \unrhd a \equiv (\vee\, e_i : e_i \in X(a) : E{=}e_i))$
- transformation is applicable for all solutions of C'

Local variables: example (0)

p_0= { assign(X,C), constant (C) }
p_1= { not (def(X)) }

prop = { p_0, p_1 }

(a := 3) $\equiv$ { $p_0 \to$ (X=a $\wedge$ C=3), $p_1 \to$ (X≠a) }
(? c==0) $\equiv$ { $p_0 \to$ false, $p_1 \to$ true }

prog : a := 3; if (d==0) {b:=0;} {b:=0;a:=2;}
pat : (true*; p_0; p_1*) | p_1*

C(prog,pat) $\equiv$ (X=b $\wedge$ C=0) $\vee$ (X $\neq$ a $\wedge$ X $\neq$ b)

what to do when b is no longer in scope?

Local variables: example (1)

prog : **{int b;** a := 3; if (d==0) {b:=0;} {b:=0;a:=2;} **}**

C(prog,pat) $\equiv$ $\cancel{(X=b \wedge C=0) \vee (X \neq a \wedge X \neq b)}$

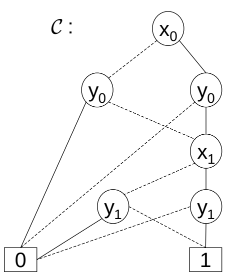
remove constraints that "refer to b" from BDD

How do we implement these BDD transformations?

(X=b)	:=	false
(X $\neq$ b)	:=	true

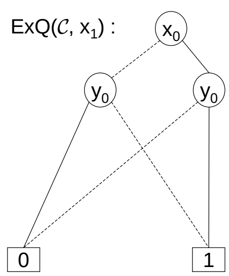
Hiding BDD variables (0)

$\mathcal{C}$:



($x_0 \equiv y_0$) $\wedge$ ($x_1 \equiv y_1$)

ExQ($\mathcal{C}$, x_1) :



($x_0 \equiv y_0$)

Hiding BDD variables (1)

ExQ(X,$\mathcal{C}$)
$\equiv$
$\mathcal{C}$[0\X] $\vee$ $\mathcal{C}$[1\X]

For finite domains:

For any substitution ϕ
such that $\phi(\mathcal{C})$
we have for any v
($\phi \oplus$ (X $\to$ v)) (ExQ(X,$\mathcal{C}$))

Scoping and BDDs

Introduce special value $*$
that occurs nowhere in program

$*$ signifies "*any variable not explicitly mentioned in the code fragment*"

Let b be a local variable.
To remove bindings of the form (X=b) from $\mathcal{C}$:

$\mathcal{C}$ ' $\equiv$ **if** X=b **then** ExQ(X, $\mathcal{C} \wedge$ X = $*$) **else** $\mathcal{C}$

(similar for E $\trianglerighteq$ b)

Interprocedural analysis

Compute matching matrix B(Pb) for each procedure body Pb

value parameter f:
call P(a)
is analysed as
{ **var** f; f := a; Pb }

result parameter f :
{ **var** f; Pb; a := f }

matching matrix B(Pb) is all we need to summarise a procedure

Summary

- Chip-chop algorithm with constraints in lieu of Booleans
- Represent constraints with BDDs
- Use ExQ operation to remove local variables from constraints
- An interprocedural analysis directly generated from the specifications

Research problems

- fast algorithms for computing chips and chops
- compare our interprocedural analyses with traditional methods
- generate fast specialised algorithms from path queries