Representation of programs

Primary goals:
- analysis is easy & effective
- just a few cases to handle
- provide support for linking things of interest
- transformations are easy
- general, across input languages & target machines

Additional goals:
- compact in memory
- easy to translate to and from
- tracks info for source-level debugging, profiling, etc.
- extensible (new optimizations, targets, language features)
- displayable

Example IRs:
- C?
- Java bytecode?
- ...

High-level syntax-based representation

Represent source-level control structures & expressions directly

Examples
- (Attributed) AST
- Lisp S-expressions
- lambda calculus? Java bytecode?

Source:
for i := 1 to 10 do
  a[i] := b[i] * 5;
end

AST:

Low-level representation

Translate input programs into low-level primitive chunks, often close to the target machine

Examples
- assembly code, virtual machine code (e.g. stack machine)
- three address code, register transfer language (RTLs)
- lambda calculus? Java bytecode?

Standard RTL operators:

<table>
<thead>
<tr>
<th>Operation</th>
<th>RTL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>x := y;</td>
</tr>
<tr>
<td>unary op</td>
<td>x := op y;</td>
</tr>
<tr>
<td>binary op</td>
<td>x := y op z;</td>
</tr>
<tr>
<td>address-of</td>
<td>p := &amp;y;</td>
</tr>
<tr>
<td>load</td>
<td>x := *(p + o);</td>
</tr>
<tr>
<td>store</td>
<td>*(p + o) := x;</td>
</tr>
<tr>
<td>call</td>
<td>x := f(...);</td>
</tr>
<tr>
<td>unary compare</td>
<td>op x ?</td>
</tr>
<tr>
<td>binary compare</td>
<td>x op y ?</td>
</tr>
</tbody>
</table>

Control flow graph containing RTL instructions:

Source:
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
Comparison

Advantages of high-level rep:
• analysis can exploit high-level knowledge of constructs
  • probably faster to analyze
• supports semantics-based reasoning about correctness etc. of analysis
• easy to map to source code terms for debugging, profiling
• may be more compact

Advantages of low-level rep:
• can do low-level, machine-specific optimizations
  (if target-based representation)
• high-level rep may not be able to express some transformations
• can have relatively few kinds of instructions to analyze
• can be language-independent

High-level rep suitable for a source-to-source or special-purpose optimizer, e.g. inliner, parallelizer

Can mix multiple representations in single compiler
Can sequence compilers using different reps

Components of representation

Operations

Dependences between operations
• control dependences: sequencing of operations
  • evaluation of then & else arms depends on result of test
  • side-effects of statements occur in right order
• data dependences: flow of values from definitions to uses
  • operands computed before operation
  • values read from variable before being overwritten

Ideal: represent just those dependences that matter
• dependences constrain transformations
• fewest dependences ⇒ most flexibility in implementation

Representing control dependences

Option 1: high-level representation
• control flow implicit in semantics of AST nodes

Option 2: control flow graph
• nodes are basic blocks
  • instructions in basic block sequence side-effects
  • edges represent branches
    (control flow between basic blocks)

Some fancier options:
• control dependence graph,
  part of program dependence graph (PDG) [Ferrante et al. 87]
• convert into data dependences on a memory state,
  in value dependence graph (VDG) [Weise et al. 94]

Kinds of data dependences

read-after-write (RAW): true/flow dependence
• reflects real data flow, operands to operation

write-after-read (WAR): anti-dependence
write-after-write (WAW): output dependence
• reflects overwriting of memory, not real data flow
  ⇒ can sometimes be eliminated by optimization

read-after-read (RAR): no dependence
• can occur in any order
Example

```plaintext
x := 3
if q != NULL then
    y := x + 2
    w := *q
    x := z * 10
else
    x := 4
endif
```

### Representing data dependences

#### Within basic block:

**Option 1:** sequence of instructions
- simple, source-like
- fixed ordering supports easy analysis
- may overconstrain order of operations

**Option 2:** expression tree/DAG
- natural, abstract
- directly captures data dependences within basic block
- DAG supports local CSE
- can be compact
- conceptually harder to analyze, transform
- must linearize eventually

#### Representing data dependences, cont.

**Across basic blocks:**

**Option 1:** implicitly through variable defs/uses
- simple
- analysis wants important things explicit ⇒ analysis can be slow

**Option 2:** def/use chains, linking each def with each use
- explicit ⇒ analysis can be fast
- must be computed, maintained after transformations
- may be space-consuming

**Fancier options:**
- **static single assignment (SSA) form** [Alpern et al. 88]
- value dependence graphs (VDGs)
- dependence flow graphs (DFGs)
- ...

Example

**Source:**

```plaintext
x := (z/y) + (y*4)
y := x + (y*4)
z := z + (y*4)
```

**Linear RTL:**

```plaintext
t1 := z/y
t2 := y*4
x := t1 + t2

t3 := y*4
y := x + t3

t4 := y*4
z := z + t4
```
Data flow analysis

Want to compute some info about program
• at program points
• to identify opportunities for improving transformations

Can model data flow analysis as solving a system of constraints
• each node in CFG imposes a constraint relating info at predecessor and successor points
• solution to constraints is result of analysis

Solution must be safe/sound
Solution can be conservative

Key issues:
• how to know if constraint system defines the analysis correctly?
• how to represent info efficiently?
• how to represent & solve constraints efficiently?
• how long does constraint solving take? finite time?
• what if multiple solutions are possible?
• how to synchronize transformations with analysis?

Example: reaching definitions

For each program point, want to compute set of definitions (statements) that may reach that point
• reach: are the last definition of some variable

Info = set of var→rtl bindings
E.g.:
\{x→s_1, y→s_5, y→s_9\}

Can use reaching definition info to:
• build def-use chains
• do constant & copy propagation
• ...

Safety rule (for these intended uses of this info): can have more bindings than the “true” answer, but can’t miss any

Constraints for reaching definitions

Main constraints:

A simple assignment removes any old reaching defs for the lhs and replaces them with this stmt:
• strong update
  \[ s: x := \ldots : \quad \text{info}_{\text{succ}} = \text{info}_{\text{pred}} - \{x→s' | \forall s' \} \cup \{x→s\} \]

A pointer assignment may modify anything, but doesn’t definitely replace anything
• weak update
  \[ s: \ast p := \ldots : \quad \text{info}_{\text{succ}} = \text{info}_{\text{pred}} \cup \{x→s | \forall x ∈ \text{may-point-to}(p)\} \]

Other statements: do nothing
\[ \text{info}_{\text{succ}} = \text{info}_{\text{pred}} \]
Constraints for reaching definitions, continued

Branches pass through reaching defs to both successors

\[ \text{info}_{\text{succ}}[i] = \text{info}_{\text{pred}} \]

Merges take the union of all incoming reaching defs
- we don’t know which path is being taken at run-time
  \( \Rightarrow \) be conservative

\[ \text{info}_{\text{succ}} = \bigcup_i \text{info}_{\text{pred}}[i] \]

Conditions at entry to CFG: definitions of formals

\[ \text{info}_{\text{entry}} = \{ x \rightarrow \text{entry} \mid \forall x \in \text{formals} \} \]

Solving constraints

A given program yields a system of constraints
Need to solve constraints

For reaching definitions,
  can traverse instructions in forward topological order,
  computing successor info from predecessor info

Example

1. \( x := \ldots \)
2. \( y := \ldots \)
3. \( y := \ldots \)
4. \( p := \ldots \)

5. \( x \ldots \)
6. \( x := \ldots \)
7. \( *p := \ldots \)

8. \( x \ldots \)
9. \( y \ldots \)
10. \( y := \ldots \)

Another example

1. \( x := \ldots \)
2. \( y := \ldots \)
3. \( y := \ldots \)
4. \( p := \ldots \)

5. \( x := \ldots \)
6. \( x := \ldots \)
7. \( *p := \ldots \)

8. \( x \ldots \)
9. \( y \ldots \)
10. \( y := \ldots \)

Topological order not defined!
Loop terminology

**loop**: strongly-connected component in CFG with single entry

**loop entry edge**: source not in loop, target in loop

**loop exit edge**: the reverse

**back edge**: target is loop head node

**loop head node**: target of loop entry edge

**loop tail node**: source of back edge

**loop preheader node**: single node that’s source of loop entry edge

**nested loop**: loop whose head is inside another loop

**reducible flow graph**: all SCC’s have single entry

Example

```
preheader
entry edge

head
back edge
exit edge
```

Analysis of loops

If CFG has a loop, data flow constraints are recursively defined:

\[ \text{info loop-head} = \text{info loop-entry} \cup \text{info back-edge} \]
\[ \text{info back-edge} = \ldots \text{info loop-head} \ldots \]

Substituting definition of info back-edge:

\[ \text{info loop-head} = \text{info loop-entry} \cup (\ldots \text{info loop-head} \ldots) \]

Summarizing r.h.s. as \( F \):

\[ \text{info loop-head} = F(\text{info loop-head}) \]

Legal solution to constraints is a **fixed-point** of \( F \)

Recursive constraints can have many solutions

- want **least** or **greatest** fixed-point, whichever corresponds to the most precise answer

How to find least/greatest fixed-point of \( F \)?

- for restricted CFGs can use specialized methods
  - e.g. interval analysis for reducible CFGs
  - for arbitrary CFGs, can use **iterative** approximation

Iterative data flow analysis

1. Start with initial guess of info at loop head:
   \[ \text{info loop-head} = \text{guess} \]

2. Solve equations for loop body:
   \[ \text{info back-edge} = F(\text{info loop-head}) \]
   \[ \text{info loop-head} = \text{info loop-entry} \cup \text{info back-edge} \]

3. Test if found fixed-point:
   \[ \text{info loop-head} = \text{info loop-head} ? \]

A. if same, then done

B. if not, then adopt result as (better) guess and repeat:
   \[ \text{info back-edge} = F(\text{info loop-head}) \]
   \[ \text{info loop-head} = \text{info loop-entry} \cup \text{info back-edge} \]
   \[ \text{info loop-head} = \text{info loop-head} ? \]
   
   ...
When does iterating work?

1. need to be able to make an initial guess
2. info^{n+1} must be closer to the fixed-point than info^n (constraints must be monotonic)
3. must eventually reach the fixed-point in a finite number of iterations (info must be drawn from a finite-height domain)

To reach best fixed-point, initial guess for loop head should be optimistic
• easy choice: info_{loop-head} = info_{loop-entry}
  (Even if guess is overly optimistic, iteration will ensure we won’t stop analysis until the answer is safe.)

To speed iterative analysis, want to test guess ASAP
• ideal: solve constraints along shortest path from loop head to loop tail
• practical: avoid solving constraints outside of loop until fixed-point is reached within loop

Another example, again

Direction of dataflow analysis

In what order are constraints solved, in general?

Constraints are declarative, not directional/procedural, so may require mixing forward & backward solving, or other more global solution methods

But often constraints can be solved by (directional) propagation & iteration
• may be forward or backward propagation of info
• topological traversals of acyclic subgraphs minimize analysis time

Directional constraints often called flow functions
• often written as functions on input info to compute output
  \[
  \begin{align*}
  RD_{x := \ldots} :&= \ldots (in) = in - \{(x \rightarrow s') \land \forall x \in \text{may-point-to}(p)\} \\
  RD_{*p := \ldots} :&= \ldots (in) = in - \{(x \rightarrow s') \land \forall x \in \text{may-point-to}(p)\} \\
  \end{align*}
  \]

GEN and KILL sets

For even more structure, can often think of flow functions in terms of each’s GEN set and KILL set
• GEN = new information added
• KILL = old information removed

Then
\[
F_{\text{instr}}(in) = in - \text{KILL}_{\text{instr}} \cup \text{GEN}_{\text{instr}}
\]

E.g., for reaching defs:
\[
\begin{align*}
RD_{x := \ldots} :&= \ldots (in) = in - \{(x \rightarrow s') \land \forall x \in \text{may-point-to}(p)\} \\
RD_{*p := \ldots} :&= \ldots (in) = in - \{(x \rightarrow s') \land \forall x \in \text{may-point-to}(p)\}
\end{align*}
\]
Bit vectors

Can sometimes represent info/KILL/GEN sets as bit vectors
• if can express abstractly as set of things (e.g. statements, vars),
drawn from a statically known set of things, each thing getting a statically determined bit position
• bitvector encodes characteristic function of set

E.g., for reaching defs:
info = bitvector over statements,
each stmt getting a distinct bit position
• statement implies which variable is defined

Bit vectors compactly represent sets
Bit-vector operations efficiently perform set difference & union

Flow function may be able to be represented simply by a pair of bit vectors, if they don’t depend on input bit vector
• can merge the KILL and GEN bit vectors of a whole block of instructions into a single overall KILL and GEN set, for faster iterating

Another example: constant propagation

Goal: data flow analysis that implements constant propagation

What info computed for each program point?
I is a conservative approximation to true info \( I_{true} \) iff:

\[
\begin{align*}
CP_{x} & := N \times \\
CP_{x} & := y + z \times \\
CP_{p} & := q + r \times 
\end{align*}
\]

Merge function?

Initial info at program points?
Direction of analysis?
Can use bit vectors?

Example

```
\begin{align*}
x & := 5 \\
x & := x + 1 \\
w & := 3 \\
y & := x \times 2 \\
z & := y + 5 \\
w & := w + 2
\end{align*}
```

May vs. must info

Some kinds of info imply guarantees: must info
Some kinds of info imply possibilities: may info
• the complement of may info is must not info

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>desired info</td>
<td>small set</td>
<td>big set</td>
</tr>
<tr>
<td>safe</td>
<td>overly big set</td>
<td>overly small set</td>
</tr>
<tr>
<td>GEN</td>
<td>add everything that might be true</td>
<td>add only if guaranteed true</td>
</tr>
<tr>
<td>KILL</td>
<td>remove only if guaranteed wrong</td>
<td>remove everything possibly wrong</td>
</tr>
<tr>
<td>MERGE</td>
<td>( \cup )</td>
<td>( \cap )</td>
</tr>
</tbody>
</table>
Another example: live variables

Want the set of variables that are live at each pt. in program
- live: might be used later in the program
Supports dead assignment elimination, register allocation

What info computed for each program point?
May or must info?
I is a conservative approximation to true info \( I_{\text{true}} \) iff:

\[
\text{LV}_x := y + z
\]

\[
\text{LV}_{*p} := *q + *z
\]

Merge function?
Initial info at program points?
Direction of analysis?
Can use bit vectors?

Example

\[
\begin{align*}
x &:= 5 \\
y &:= x + 2 \\
\text{... } y \ldots
\end{align*}
\]

Lattice-Theoretic Data Flow Analysis Framework

Goals:
- provide a single, formal model that describes all DFAs
- formalize notions of “safe”, “conservative”, “optimistic”
- place precise bounds on time complexity of DF analysis
- enable connecting analysis to underlying semantics for correctness proofs

Plan:
- define domain of program properties computed by DFA
  - domain has a set of elements
  - each element represents one possible value of the property
  - order elements to reflect their relative precision
  - domain = set of elements + order over elements = lattice
- define flow functions & merge function over this domain, using standard lattice operators
- benefit from lattice theory in attacking above issues

History: Kildall [POPL 73], Kam & Ullman [JACM 76]

Lattices

Define lattice \( D = (S, \leq) \):
- \( S \) is a set of elements of the lattice
- \( \leq \) is a binary relation over elements of \( S \)

Required properties of \( \leq \):
- \( \leq \) induces a partial order over \( S \)
  - reflexive, transitive, & anti-symmetric
- every pair of elements of \( S \) has
  - a unique greatest lower bound (a.k.a. meet) and
  - a unique least upper bound (a.k.a. join)

Height of \( D \) = longest path through partial order from greatest to least
- infinite lattice can have finite height (but infinite width)

Top (T) = unique element of \( S \) that’s greatest, if exists
Bottom (\( \bot \)) = unique element of \( S \) that’s least, if exists
Lattice models in data flow analysis

Model data flow information by elements of a lattice domain
• top = best case info
• bottom = worst case info
• if \( a \leq b \), then \( a \) is a conservative approximation to \( b \)
• merge function = g.l.b. (meet) on lattice elements
  (the most precise element that’s a conservative approximation to both input elements)
• initial info for optimistic analysis (at least back edges): top

(Opposite up/down conventions used in PL semantics!)

Examples

Reaching definitions:
• elements:
  • \( \leq \):
  • top:
  • bottom:
  • meet:

Reaching constants:
• elements:
  • \( \leq \):
  • top:
  • bottom:
  • meet:

Tuples of lattices

Often helpful to break down a complex lattice into a tuple of lattices, one per variable being analyzed
Formally: \( D_T = \langle S_T, \leq_T \rangle = (D = \langle S, \leq \rangle)^N \)
• \( S_T = S_1 \times S_2 \times \ldots \times S_N \)
• element of tuple domain is a tuple of elements from each variable’s domain
• \( i \)th component of tuple is info about \( i \)th variable
• \( \langle \ldots, d_{1i}, \ldots \rangle \leq_T \langle \ldots, d_{2i}, \ldots \rangle \equiv d_{1i} \leq d_{2i}, \forall i \)
• i.e. pointwise ordering
• meet: pointwise meet
• top: tuple of tops
• bottom: tuple of bottoms
• height(D_T) = N \times height(D)

E.g. reaching constants
• lattice for single variable is 3-level lattice:

```
... x=2 x=1 x=0 x=1 x=2 ...
```
• whole problem is tuple of individual lattices

Some typical lattice domains

Single-point lattice: just bottom
• trivial do-nothing analysis

Two-point lattice: top and bottom
• computes a boolean property
• a tuple of two-point lattices \( \Leftrightarrow \) a bit-vector

A lifted set: a set of incomparable values, plus top & bottom
• e.g. reaching constants domain

Powerset lattice: set of all subsets of a set \( S \), ordered somehow
• top & bottom = \( \emptyset \) & \( S \), or vice versa
• “a collecting analysis”
• isomorphic to tuple of booleans indicating membership in subset of elements of \( S \)
Analysis of loops in lattice model

Consider:

Assume \( B(d) \) computes info at back-edge given \( d \), info at head

Want solution to constraints:

\[
\begin{align*}
\text{d}_{\text{head}} &= \text{d}_{\text{entry}} \cap \text{d}_{\text{backedge}} \\
\text{d}_{\text{backedge}} &= B(\text{d}_{\text{head}})
\end{align*}
\]

Let \( F(d) = \text{d}_{\text{entry}} \cap B(d) \)

Then want fixed-point of \( F \):

\[
\begin{align*}
\text{d}_{\text{head}} &= F(\text{d}_{\text{head}})
\end{align*}
\]

Iterative analysis in lattice model

Iterative analysis computes fixed-point
by iterative approximation:

\[
\begin{align*}
F^0 &= \text{d}_{\text{entry}} \cap T \\
F^1 &= \text{d}_{\text{entry}} \cap B(F^0) = F(F^0) = F(d_{\text{entry}}) \\
F^2 &= \text{d}_{\text{entry}} \cap B(F^1) = F(F^1) = F(F(F^0)) = F(F(d_{\text{entry}})) \\
&\quad \vdots \\
F^k &= \text{d}_{\text{entry}} \cap B(F^{k-1}) = F(F^{k-1}) = F(F(\cdots(F(d_{\text{entry}}))\cdots)) \\
\text{until} \\
F^{k+1} &= \text{d}_{\text{entry}} \cap B(F^k) = F(F^k) = F^k
\end{align*}
\]

Is \( k \) finite?
If so, how big can it be?

Termination of iterative analysis

In general, \( k \) need not be finite

Sufficient conditions for finiteness:

- flow functions (e.g. \( F \)) are \textbf{monotonic}
- lattice is of finite height

A function \( F \) is \textbf{monotonic} iff:

\[
\begin{align*}
d_1 \leq d_2 &\quad \Rightarrow \quad F(d_1) \leq F(d_2)
\end{align*}
\]

For monotonic \( F \) over domain \( D \), the maximum number of times that \( F \) can be applied to itself, starting w' any element of \( D \), w/o reaching fixed-point, is height(\( D \))-1

- start at top of \( D \\
- go down one level in lattice each application of \( F \\
- eventually must hit fixed-point or bottom \\
  (which is guaranteed to be a fixed-point), \\
  if \( D \) of finite height

Complexity of iterative analysis

How long does iterative analysis take?

\[
\begin{align*}
l &: \text{depth of loop nesting} \\
\rho &: \# \text{ of stmts in loop} \\
t &: \text{time to execute one flow function} \\
k &: \text{height of lattice}
\end{align*}
\]
Another example: integer range analysis

For each program point,
for each integer-typed variable,
calculate (an approximation to) the set of integer values
that can be taken on by the variable
• use info for constant folding comparisons,
  for eliminating array bounds checks,
  for (in)dependence testing of array accesses,
  for eliminating overflow checks

What domain to use?
• what is its height?

What flow functions to use?
• are they monotonic?

Example

```
for i := 0 to N-1
  ...
  a[i]
  ...
end
```

Widening operators

If domain is tall, then can introduce artificial generalizations
(called **widenings**) when merging at loop heads
• ensure that only a finite number of widenings are possible

Sharlit

A data flow analyzer generator [Tjiang & Hennessy 92]
• analogous to YACC

User writes basic primitives:
• control flow graph representation
  • nodes are instructions, not basic blocks
• domain (“flow value”) representation and key operations
  • init
  • copy
  • is_equal
  • meet
• flow functions for each kind of instruction
• action routines to optimize after analysis

Sharlit constructs iterative dataflow analyzer from these pieces
+ easy to build, extend
  – not highly efficient, in this first mode of use
**Path compression**

Can improve analysis efficiency by summarizing effect of sequences of nodes

User can define path compression operations to collapse nodes together
- linear joining of sequential nodes ⇒ summarizes effect of whole BB
- presumes a fixed GEN/KILL bit-vector structure to be effective
- merge trees into extended BB’s
- merge merges, loops as in interval analysis
  - simplifies reducible parts, applies iteration to nonreducible parts

+ gets efficiency, preserves modularity & generality
– doesn’t support data-dependent flow functions, cannot simulate optimizations during analysis

Performance results for code quality of generated optimizer, but not for compilation speed of optimizer

**Vortex IDFA framework**

Like Sharlit, except a compiler library rather than a compiler-compiler

User defines a subclass of `AnalysisInfo` to represent elements of domain
- `copy`
- `merge` (lattice g.l.b. operator)
- generalizing_merge (g.l.b. with optional widening)
  - `as_general_as` (lattice ≤ operator)

User invokes `traverse` to perform analysis:

```plaintext
cfg.traverse(direction, is_iterative?, initial_analysis_info, λ(rtl, info){ rtl.flow_fn(info) })
```

Flow function returns an `AnalysisResult`: one of
- keep instruction and continue analysis w/ updated info(s)
- delete instruction/constant-fold branch
- replace instruction with instruction or subgraph

`ComposedAnalysis` supports running multiple analyses interleaved at each instruction

**Features of Vortex IDFA**

Big idea: separate analyses and transformations, make framework compose them appropriately
- don’t have to simulate the effect of transformations during analysis
- can run analyses in parallel if each provides opportunities for the other
  - sometimes can achieve strictly better results this way than if run separately in a loop
- more general transformations supported (e.g. inlining) than Sharlit

Exploit inheritance & closures

Analysis speed is not stressed
- no path compression
- no “compilation” of analysis with framework

[Vortex’s interprocedural analysis support discussed later]