Representing types [33]

The lineage of a type can be represented without reference to specific type names. A bit-vector representation is convenient for construction and for comparison:

Each of the types extenders is assigned a bit pattern from $k$ bits:

<table>
<thead>
<tr>
<th>Type</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>01</td>
</tr>
<tr>
<td>array</td>
<td>10</td>
</tr>
<tr>
<td>func</td>
<td>11</td>
</tr>
</tbody>
</table>

So that a pointer to type $X$ is represented as

$$01X$$

Note that this scheme does not track array index types or function parameter types.

Wisely leaving one pattern free, we now assign the base types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>0000</td>
</tr>
<tr>
<td>char</td>
<td>0001</td>
</tr>
<tr>
<td>int</td>
<td>0010</td>
</tr>
<tr>
<td>float</td>
<td>0011</td>
</tr>
</tbody>
</table>

Runtime storage management

Most block-structured languages require manipulation of runtime structures to maintain efficient access to appropriate data and machine resources. For our purposes, a procedure is either a named function or an inline (parameterless) block.

Let's examine the activity normally associated with invoking a procedure $P$:

1. Some machine state might be saved: general registers, vector registers, condition codes, interrupt masks, etc.
2. Access must be established to $P$'s local variables and compiler-generated temporaries.
3. Access must be established to outer scope variables (but not for C).
4. The caller of $P$ must be recorded so that $P$ can return when done.
5. Parameters might be received prior to executing $P$.
6. A return value might be prepared prior to returning from $P$.

Each procedure invocation causes creation of an activation record or frame to hold such runtime information.

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local x</td>
</tr>
<tr>
<td>Local y</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Dynamic Link</td>
</tr>
<tr>
<td>Static Link</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>

It's convenient to have each local occupy a fixed amount of storage in the frame. Therefore, arrays and other large objects are often indirectly accessed from a procedure's frame, with the actual storage allocated on stack after the frame.
At procedure entry, a link is inserted in the frame to a procedure’s next outer scope, whose frame is linked to its outer scope, and so on. The static link is deallocated along with the frame.

Establishing the link is fast, but accessing the $k$th enclosing scope requires $k$ indirections using static links. However, a good register allocator would cache these in the procedure’s registers.

Most programs make almost exclusive use of local and outermost scopes, with scant use of intermediate scopes. This is especially true in C, where the language offers no access to intermediate scopes except by explicit pointers.
Example of a high-level intermediate language

The language **FRIL** [10] was developed to ease code generation, primarily by resembling **LISP** and by offering a declarative mechanism for storage association. Each symbol **FRIL** is declared at most once as any procedure’s local or parameter. Each “expression” declares the static depth of its frame, and provides a pointer to its outer scope.

```c
int a1;
extern int a2;
int one;
void main() {
  int i;
  int factorial(X)
  int X;
  {
    int Y;
    Y = X;
    if (Y > 0) Y*factorial(X-1);
    else one;
  }
  one = 1;
  a1 = factorial(i=5);
  a2 = factorial(3);
}
```

In a compiler course, the choice of code generation strategy is key to a successful experience. Many courses stop just before code generation, in which case the students do not experience the elation of watching their compilers actually work.

If the target of translation is reasonably high-level (e.g., a **LISP**-like intermediate language), then **ad hoc** methods are feasible. In this case, an interpreter should be provided to execute the translated programs.

Otherwise, experience with an automatic code generator is more beneficial. Watch for developments in the **lcc** system [18], which can be obtained by contacting Dave Hanson (drh@princeton.edu). If the **MIPS** instruction set were targeted, then Lazarus’s **SPIM** simulator [30](Appendix A) can greatly facilitate debugging the generated code.
Ad hoc. methods

For example, for a binary + node, the code generator would be called recursively to place the result of the left and right subtrees in two known locations (say, registers $R_1$ and $R_2$). Code would then be emitted to form the sum, placing the result in yet another known location.

![Diagram of binary tree with nodes A, B, and +]

I usually provide procedures for generating FRIL's symbol table, for generating a PushLevel, and for indenting and formatting the output. The students must decide what constitutes an expression. For example, FRIL has only one control transfer operator: the procedure call. Thus, the body of an iterative loop must be invoked recursively to achieve iteration.

Students write some 200 lines of code to complete the ad hoc. code generator for FRIL.

A systematic method — Tree Rewriting

While the ad hoc. method services the AST a node at a time, tree rewriting systems can examine larger subtrees and searching for more optimal instruction sequences.

The AST shown to the right is representative of the code fragment

\[
*(x+4) = a[k];
\]

Note that left and right value analysis has already taken place.

Let's assume that from the perspective of code generation, the nodes $x$, $a$, and $k$ represent constants. This would be the case had the compiler assigned storage to these variables. If not, then the AST should reflect a level of indirection (probably off a popular register) to reach those variables.

Given the richness of most instruction sets, trying all combinations of instructions to cover the tree would be prohibitively expensive. Most tree matching algorithms use dynamic programming, so that results previously holding for some subtree can be reused without additional cost.
### Tree rewriting

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rewrite</th>
<th>Instruction</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="tree1.png" alt="" /></td>
<td>$R_i \leftarrow R_i + \text{const}$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td><img src="tree2.png" alt="" /></td>
<td>$R_i \leftarrow M[R_i + \text{const}]$</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td><img src="tree3.png" alt="" /></td>
<td>$R_i \leftarrow M[\text{const}]$</td>
<td>3</td>
</tr>
</tbody>
</table>

Not shown are the rules that account for the symmetry of addition.

### Tree rewriting

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rewrite</th>
<th>Instruction</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td><img src="tree4.png" alt="" /></td>
<td>$M[R_i + \text{const}] \leftarrow R_j$</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td><img src="tree5.png" alt="" /></td>
<td>$M[R_i] \leftarrow M[R_j]$</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td><img src="tree6.png" alt="" /></td>
<td>$R_i \leftarrow \text{const}$</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td><img src="tree7.png" alt="" /></td>
<td>$R_i \leftarrow R_i + R_j$</td>
<td>1</td>
</tr>
</tbody>
</table>
Example – one way to cover the nodes

\[
\begin{align*}
\text{Rule} & \quad \text{Instr} & \quad \text{Cost} \\
2 & R_i \rightarrow M[R_i + \text{const}] & 5 \\
3 & R_i \rightarrow M[\text{const}] & 3 \\
4 & M[R_i + \text{const}] \rightarrow R_j & 5
\end{align*}
\]

Example – another way to cover the nodes

\[
\begin{align*}
\text{Rule} & \quad \text{Instr} & \quad \text{Cost} \\
1 & R_i \rightarrow R_i + \text{const} & 1 \\
3 & R_i \rightarrow M[\text{const}] & 3 \\
5 & M[R_i] \rightarrow M[R_j] & 6
\end{align*}
\]
Some thoughts

Misconceptions

Optimization optimizes your program.
There’s probably a better algorithm or sequence of program transformations. While optimization hopefully improves your program, the result is usually not optimal.

Optimization requires (much) more compilation time. For example, dead code elimination can reduce the size of program text such that overall compile time is also reduced.

A clever programmer is a good substitute for an optimizing compiler.
While efficient coding of an algorithm is essential, programs should not be obfuscated by “tricks” that are architecture- (and sometimes compiler-) specific.

All too often . . .

Optimization is disabled by default. Debugging optimized code can be treacherous [45, 23]. Optimization is often the primary suspect of program misbehavior—sometimes deservedly so. “No, not the third switch!”

Optimization is slow. Transformations are often applied to too much of a program. Optimizations are often textbook recipes, applied without proper thought.

Optimization produces incorrect code. Although recent work is encouraging [42], optimizations are usually developed ad hoc.

Programmers are trained by their compilers. A style is inevitably developed that is conducive to optimization.

Optimization is like sex:

- Everybody claims to get good results using exotic techniques;
- Nobody is willing to provide the details.
It's very easy to devote much time and effort toward choosing the "right" IL. Below are some guidelines for choosing or developing a useful intermediate language:

- The IL should be a bona fide language, and not just an aggregation of data structures.
- The semantics of the IL should be cleanly defined and readily apparent.
- The IL's representation should not be overly verbose:
  - Although some expansion is inevitable, the IL-to-source token ratio should be as low as possible.
  - It's desirable for the IL to have a verbose, human-readable form.
- The IL should be easily and cleanly extensible.
- The IL should be sufficiently general to represent the important aspects of multiple front-end languages.
- The IL should be sufficiently general to support efficient code generation for multiple back-end targets.

A sampling of difficult issues:

- How should a string operation be represented (intact or as a "loop")?
- How much detail of a procedure's behavior is relevant?

Ideally, an IL has fractal characteristics: optimization can proceed at a given level; the IL can be "lowered"; optimization is then applied to the freshly exposed description.

Architecting an intermediate language reduces the incremental cost of accommodating new source languages or target architectures [5]. Moreover, many optimizations can be performed directly on the intermediate language text, so that source- and machine-independent optimizations can be performed by a common middle-end.
Unoptimized matrix multiply

\[
\text{for } i = 1 \text{ to } N \text{ do } \\
\quad \text{for } j = 1 \text{ to } N \text{ do } \\
\quad \quad A[i, j] <- 0 \quad \\
\quad \quad \text{for } k = 1 \text{ to } N \text{ do } \\
\quad \quad \quad A[i, j] <- A[i, j] + B[i, k] \times C[k, j] \quad \\
\quad \text{od} \\
\text{od} \\
\text{od}
\]

Note that \( A[i, j] \) is really

\[
\text{Addr}(A) + ((i - 1) \times K_1 + (j - 1)) \times K_2
\]

which takes 6 integer operations.

The innermost loop of this “textbook” program takes

\[
\begin{align*}
24 & \text{ integer ops} \\
3 & \text{ loads} \\
1 & \text{ floating add} \\
1 & \text{ floating mpy} \\
1 & \text{ store} \\
30 & \text{ instructions}
\end{align*}
\]
Optimizing matrix multiply

for i = 1 to N do
  for j = 1 to N do
    a ← &A[i, j]
    for k = 1 to N do
      a ← a + B[i, k] × C[k, j]
    od
  od
od

for i = 1 to N do
  b ← &B[i, 1]
  for j = 1 to N do
    a ← &A[i, j]
    for k = 1 to N do
      a ← a + b × C[k, j]
    od
    b ← b + K_B
  od
od

Similar analysis for C yields only 2 integer operations in the innermost loop, for a speedup of nearly 5. We can do better, especially for large arrays.

If optimization is...

so great because:

A good compiler can sell (even a slow) machine. Optimizing compilers easily provide a factor of two in performance. Moreover, the analysis performed during program optimization can be incorporated into the "programming environment" [29, 7, 43].

New languages and architectures motivate new program optimizations. Although some optimizations are almost universally beneficial, the advent of functional and parallel programming languages has increased the intensity of research into program analysis and transformation.

Programs can be written with attention to clarity, rather than performance.

There is no substitute for a good algorithm. However, the expression of an algorithm should be as independent as possible of any specific architecture.

then:

Why does it take so long? Compilation time is usually 2-5 times slower, and programs with large procedures often take longer. Often this is the result of poor engineering: better data structures or algorithms can help in the optimizer.

Why does the resulting program sometimes exhibit unexpected behavior? Sometimes the source program is at fault, and a bug is uncovered when the optimized code is executed; sometimes the optimizing compiler is itself to blame.

Why is "no-opt" the default? Most compilations occur during the software development cycle. Unfortunately, most debuggers cannot provide useful information when the program has been optimized [45, 23]. Even more unfortunately, optimizing compilers sometimes produce incorrect code. Often, insufficient time is spent testing the optimizer, and with no-opt the default, bugs in the optimizer may remain hidden.
Ingredients in a data flow framework (cont’d)

Meet lattice which determines the outcome when disparate solutions combine. The lattice is specified with distinguished elements:

\[ \top \] which represents the best possible solution, and

\[ \bot \] which represents the worst possible solution.

Transfer functions which transform one solution into another.

We’ll use the meet lattice to summarize the effects of convergent paths in the data flow graph, and transfer functions to model the effects of a data flow graph path on the data flow solution.

We’ll begin with some simple bit-vectoring data flow problems, classically solved as operations on bit-vectors. For ease of exposition, we’ll associate data flow solutions with the edges, rather than the nodes, of the data flow graph.
Available expressions

An expression $expr$ is available ($\text{Avail}$) at flow graph edge $e$ if any past behavior of the program includes a computation of the value of $expr$ at $e$.

Consider the expression $(v + w)$ in the flow graph shown to the right. If the expression is available at the assignment to $z$, then it need not be recomputed.

- This is a forward problem, so the data flow graph will have the same edges and $\text{Start}$ and $\text{Stop}$ nodes as the flow graph.
- The solution for any given $expr$ is either $\text{Avail}$ or $\neg \text{Avail}$.
- The “best” solution for an expression is $\text{Avail}$. We thus obtain the two-level lattice:
  - $\top$ is $\text{Avail}$.
  - $\bot$ is $\neg \text{Avail}$.

Available expressions (cont’d)

Nodes that compute an expression make that expression available. We also assume that every expression is available from $\text{Start}$.

The transfer function for each highlighted node makes the expression $(v + w)$ $\text{Avail}$, regardless of the solution present at the node’s input.
Nodes that assign to any variable in an expression make that expression not available, even if the variable’s value is unchanged.

The transfer function for each highlighted node makes the expression \((v + w)\) *Available*, regardless of the solution present at the node’s input.

Here we see the global solution for availability of the expression \((v + w)\).

Each of the highlighted nodes shown previously asserts a solution on its output edge(s). It’s the job of global data flow analysis to assign the best possible solution to every edge in the data flow graph, consistent with the asserted solutions.

The expression \((v + w)\) need not be computed in the assignment to \(z\). The relevant value is held either in \(x\), or \(y\), depending on program flow.

To solve this problem using bit-vectors, assign each expression a position in the bit-vector. When an expression is available, its associated bit is 1.
Live variables

A variable \( v \) is **live** at edge \( e \) if the future behavior of the program may reference the value of \( v \) at \( e \).

If a variable \( v \) is not live, then any resources associated with \( v \) (registers, storage, etc.) may be reclaimed.

- This is a backward problem.
- In the bit-vector representation, each variable is associated with a bit.
- The “best” solution is \( \text{Live} \), so we obtain the two-level lattice:
  - \( \top \) is \( \text{Live} \).
  - \( \bot \) is \( \text{Live} \).

---

Live variables (cont’d)

Each of the highlighted nodes affects the data flow solution:

- If a node uses \( v \), then the node’s output asserts that \( v \) is \( \text{Live} \).
- If a node kills \( v \), then the node’s output asserts that \( v \) is \( \text{Live} \).
If a node $Y$ preserves $v$ (as might a procedure call), then the node does not affect the solution.

- If $v$ is \textit{Live} on “input” to $Y$, then $Y$ cannot make $v$ \textit{Live}.
- If $v$ is \textit{Not Live} on “input” to $Y$, then $Y$ does not make $v$ \textit{Live}.

Node $Y$’s transfer function is therefore the \textit{identity} function:

$$f_Y(IN) = IN$$

assuming node $Y$ does not use $v$.

\[ \text{Global solution: Live variables} \]

\[ \text{Live variables (cont’d)} \]

\( \text{The data flow graph} \)

\[ G_{df} = (N_{df}, E_{df}) \]

has been described previously:

- its edges are oriented in the direction of the data flow problem;
- $G_{df}$ is augmented with nodes \textit{Start} and \textit{Stop} and an edge (\textit{Start}, \textit{Stop}), suitably inserted with respect to the direction of the data flow problem.

Successors and predecessors are also defined with respect to the direction of the data flow problem:

\[ \text{Successors} (Y) = \{ Z \mid (Y, Z) \in E_{df} \} \]

\[ \text{Predecessors} (Y) = \{ X \mid (X, Y) \in E_{df} \} \]

\( \text{The meet semilattice is} \)

\[ L = (A, T, \perp, \leq, \wedge) \]

$A$ is a set (usually a powerset), whose elements form the domain of the data flow problem,

$T$ and $\perp$ are distinguished elements of $A$, usually called “top” and “bottom”, respectively,

$\leq$ is a reflexive partial order, and

$\wedge$ is the associative and commutative \textit{meet} operator, such that for any $a, b \in A$,

\[ a \leq b \iff a \wedge b = a \]

\[ a \wedge a = a \]

\[ a \wedge b \leq a \]

\[ a \wedge b \leq b \]

\[ a \wedge T = a \]

\[ a \wedge \perp = \perp \]

These rules allow formal reasoning about $T$ and $\perp$ in a framework.
We'll now examine some special algorithms for optimization, based on a single assignment representation.
Static Single Assignment (SSA) form

Below are shown a program and its reaching definitions.

Notice how the use of $v$ at $G$ is reached by two definitions, and the use at $H$ is reached by four definitions. If each use were reached by just a single definition, data flow analysis based on definitions could consult one definition per use.

SSA form (cont’d)

Here we see the SSA form of the program.

- Each definition of $v$ is with respect to a distinct symbol: $v_1$ is as different from $v_2$ as $x$ would be from $y$.
- Where multiple definitions reach a node, a $\phi$-function is inserted, with arguments sufficient to receive a different “name” for $v$ on each in-edge.
- Each use is appropriately renamed to the distinct definition that reaches it.
- Although $\phi$-functions could have been placed at every node, the program shown has exactly the right number and placement of $\phi$-functions to combine multiple defs from the original program.
- Our example assumes that procedure $f$ does not modify $v$. 

Copyright © 1994 Ron K. Cytron. All rights reserved
SSA form (cont'd)

Each def is now regarded as a “killing” def, even those usually regarded as preserving
defs. For example, if \( v \) is potentially modified by the call site, then the old value for
\( v \) must be passed into the called procedure, so that its value can be assigned to the
name for \( v \) that always emerges from the procedure.

Procedure `foo(v)`

```plaintext
if (c) then
  v ← 7
else
  /* Do nothing */
fi
```

Procedure `foo(v_out, v_in)`

```plaintext
v_0 ← v_in
if (c) then
  v_1 ← 7
else
  /* Do nothing */
fi

v_2 ← \( \phi(v_0, v_1) \)
v_out ← v_2
```

SSA form can be computed by a data flow framework, in which the transfer function
for a node with multiple reaching defs of \( v \) generates its own def of \( v \). Uses are then
named by the solution in effect at the associated node.

SSA form construction [9]

1. Every preserving def is turned into a killing def, by copying potentially unmodified
   values (at subscripted defs, call sites, aliased defs, etc.).
2. Each ordinary definition of \( v \) defines a new name.
3. At each node in the flow graph where multiple definitions of \( v \) meet, a \( \phi \)-function
   is introduced to represent yet another new name for \( v \).
4. Uses are renamed by their dominating definition (where uses at a \( \phi \)-function are
   regarded as belonging to the appropriate predecessor node of the \( \phi \)-function).

```