Optimising Compilers

Computer Science Tripos Part II

Timothy Jones

Lecture 1
Introduction

A non-optimising compiler

- character stream
  - lexing
- token stream
  - parsing
- parse tree
  - translation
- intermediate code
  - code generation
- target code

Optimisation

(real “amelioration”!)

Good humans write simple, maintainable, general code.
Compilers should then remove unused generality, and hence hopefully make the code:
- Smaller
- Faster
- Cheaper (e.g. lower power consumption)

Analysis + Transformation

- Transformation does something dangerous.
- Analysis determines whether it’s safe.

An optimising compiler

- character stream
  - lexing
- token stream
  - parsing
- parse tree
  - translation
- intermediate code
  - code generation
- target code

- optimisation
- optimisation
- optimisation
- decompilation

Optimisation

= Analysis + Transformation

- An analysis shows that your program has some property...
- ...and the transformation is designed to be safe for all programs with that property...
- ...so it’s safe to do the transformation.
int main(void) 
{ 
    return 42;  
} 

int f(int x) 
{ 
    return x * 2;  
}

while (i <= k*2) { 
    j = j * i;  
    i = i + 1; 
}

while (i <= k*2) { 
    k = k - i;  
    i = i + 1; 
}
Stack-oriented code

```c
iload 0
iload 1
iadd
iload 2
iload 3
iadd
imul
ireturn
```

3-address code

```c
ENTRY fact
MOV t32, arg1
CMPEQ t32, #0, lab1
SUB arg1, t32, #1
CALL fact
MUL res1, t32, res1
EXIT
```

C into 3-address code

```c
int fact (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```

Flowgraphs

- A graph representation of a program
- Each node stores 3-address instruction(s)
- Each edge represents (potential) control flow:

```
pred(n) = \{ n' \mid (n', n) \in edges(G) \}
succ(n) = \{ n' \mid (n, n') \in edges(G) \}
```

Basic blocks

A maximal sequence of instructions $n_1, ..., n_k$ which have

- exactly one predecessor (except possibly for $n_1$)
- exactly one successor (except possibly for $n_k$)
Basic blocks

 ENTRY fact
    MOV t32, arg1
    CMPEQ t32, #0
    SUB arg1, t32, #1
    CALL fact
    MUL res1, t32, res1
 EXIT
    MOV res1, #1
 EXIT

A basic block doesn’t contain any interesting control flow.

Types of analysis
(and hence optimisation)

- Within basic blocks (“local” / “peephole”)
- Between basic blocks (“global” / “intra-procedural”)
  - e.g. live variable analysis, available expressions
- Whole program (“inter-procedural”)
  - e.g. unreachable-procedure elimination

Reduce time and space requirements
for analysis algorithms
by calculating and storing data flow information
once per block
(and recomputing within a block if required)
instead of
once per instruction.

Peephole optimisation

ADD t32, arg1, #1
MOV r0, r1
MOV r1, r0
MUL t33, r0, t32
ADD t32, arg1, #1
MOV r0, r1
MUL t33, r0, t32

matches
MOV x, y
MOV y, x

replace
MOV x, y
with
MOV x, y
Types of analysis
(and hence optimisation)

Type of information:
• Control flow
  • Discovering control structure (basic blocks, loops, calls between procedures)
• Data flow
  • Discovering data flow structure (variable uses, expression evaluation)

Finding basic blocks

1. Find all the instructions which are leaders:
   • the first instruction is a leader;
   • the target of any branch is a leader; and
   • any instruction immediately following a branch is a leader.
2. For each leader, its basic block consists of itself and all instructions up to the next leader.

ENTRY fact
MOV t32, arg1
CMPEQ t32, #0, lab1
SUB arg1, t32, #1
CALL fact
MUL res1, t32, res1
EXIT
lab1: MOV res1, #1
EXIT

Summary
• Structure of an optimising compiler
• Why optimise?
• Optimisation = Analysis + Transformation
• 3-address code
• Flowgraphs
• Basic blocks
• Types of analysis
• Locating basic blocks

Control-flow analysis

Discovering information about how control (e.g. the program counter) may move through a program.

Intra-procedural analysis

An intra-procedural analysis collects information about the code inside a single procedure.

We may repeat it many times (i.e. once per procedure), but information is only propagated within the boundaries of each procedure, not between procedures.

One example of an intra-procedural control-flow optimisation (an analysis and an accompanying transformation) is unreachable-code elimination.

Dead vs. unreachable code

int f(int x, int y) {
  int z = x * y; // DEAD
  return x + y;
}

Dead code computes unused values. (Waste of time.)
Dead vs. unreachable code

Unreachable code cannot possibly be executed. (Waste of space.)

Dead vs. unreachable code

Unreachability is a control-flow property:
“May control ever arrive here?”

Safety of analysis

In general, this is undecidable.
(Arithmetic is undecidable; cf. halting problem.)

Safety of analysis

• Many interesting properties of programs are undecidable and cannot be computed precisely...
• ...so they must be approximated.
• A broken program is much worse than an inefficient one...
• ...so we must err on the side of safety.

Safety of analysis

• If we decide that code is unreachable then we may do something dangerous (e.g. remove it!)...
• ...so the safe strategy is to overestimate reachability.
• If we can’t easily tell whether code is reachable, we just assume that it is. (This is conservative.)
• For example, we assume
  • both branches of a conditional are reachable
  • and that loops always terminate.
Safety of analysis

Naïvely,

```java
if (false) {
    int z = x * y;
}
```

this instruction is reachable,

```java
while (true) {
    // Code without ‘break’
}
```

and so is this one.

Another source of uncertainty is encountered when constructing the original flowgraph: the presence of indirect branches (also known as “computed jumps”).

Again, this is a conservative overestimation of reachability.

In the worst-case scenario in which branch-address computations are completely unrestricted (i.e. the target of a jump could be absolutely anywhere), the presence of an indirect branch forces us to assume that all instructions are potentially reachable in order to guarantee safety.

“reachable”

program instructions

sometimes executed

never executed

imprecision
Unreachable code

This naïve reachability analysis is simplistic, but has the advantage of corresponding to a very straightforward operation on the flowgraph of a procedure:

1. mark the procedure’s entry node as reachable;
2. mark every successor of a marked node as reachable and repeat until no further marking is required.

Unreachable code

Programmers rarely write code which is completely unreachable in this naïve sense. Why bother with this analysis?

• Naïvely unreachable code may be introduced as a result of other optimising transformations.
• With a little more effort, we can do a better job.

Unreachable code

Obviously, if the conditional expression in an if statement is literally the constant “false”, it’s safe to assume that the statements within are unreachable.

```c
if (false) {
  int z = x * y;
} UNREACHABLE
```

But programmers never write code like that either.

Unreachable code

However, other optimisations might produce such code. For example, copy propagation:

```c
bool debug = false;
if (debug) {
  int z = x * y;
}
```

Unreachable code

However, other optimisations might produce such code. For example, copy propagation:

```c
if (false) {
  int z = x * y;
} UNREACHABLE
```

We can try to spot (slightly) more subtle things too.

• if (!true) {... }
• if (false && ...) {... }
• if (x != x) {... }
• while (true) {... } ...
• ...
Unreachable code

Note, however, that the reachability analysis no longer consists simply of checking whether any paths to an instruction exist in the flowgraph, but whether any of the paths to an instruction are actually executable.

With more effort we may get arbitrarily clever at spotting non-executable paths in particular cases, but in general the undecidability of arithmetic means that we cannot always spot them all.

Although unreachable-code elimination can only make a program smaller, it may enable other optimisations which make the program faster.

Unreachable code

For example, straightening is an optimisation which can eliminate jumps between basic blocks by coalescing them:

Unreachable procedures

Unreachable-procedure elimination is very similar in spirit to unreachable-code elimination, but relies on a different data structure known as a call graph.

Inter-procedural analysis

An inter-procedural analysis collects information about an entire program.

Information is collected from the instructions of each procedure and then propagated between procedures.

One example of an inter-procedural control-flow optimisation (an analysis and an accompanying transformation) is unreachable-procedure elimination.

Call graphs

Unreachable-procedure elimination is very similar in spirit to unreachable-code elimination, but relies on a different data structure known as a call graph.
Call graphs

Again, the precision of the graph is compromised in the presence of indirect calls.

Unreachable procedures

The reachability analysis is virtually identical to that used in unreachable-code elimination, but this time operates on the call graph of the entire program (vs. the flowgraph of a single procedure):

1. mark procedure main as callable;
2. mark every successor of a marked node as callable and repeat until no further marking is required.

Safety of transformations

- All instructions/procedures to which control may flow at execution time will definitely be marked by the reachability analyses...
- ...but not vice versa, since some marked nodes might never be executed.
- Both transformations will definitely not delete any instructions/procedures which are needed to execute the program...
- ...but they might leave others alone too.

If simplication

Empty then in if-then

if (f(x)) {
}

(Assuming that f has no side effects.)

Unreachable procedures

In general, we assume that a procedure containing an indirect call has all address-taken procedures as successors in the call graph — i.e., it could call any of them.

This is obviously safe; it is also obviously imprecise.

As before, it might be possible to do better by application of more careful methods (e.g. tracking data-flow of procedure variables).

Call graphs

In general, we assume that a procedure containing an indirect call has all address-taken procedures as successors in the call graph — i.e., it could call any of them.

This is obviously safe; it is also obviously imprecise.

As before, it might be possible to do better by application of more careful methods (e.g. tracking data-flow of procedure variables).
If simplification

Empty else in if-then-else

```c
if (f(x)) {
    z = x * y;
} else {
}
```

If simplification

Empty then in if-then-else

```c
if (!f(x)) {
} else {
    z = x * y;
}
```

If simplification

Empty then and else in if-then-else

```c
if (f(x)) {
} else {
}
```

If simplification

Constant condition

```c
if (true) {
    z = x * y;
}
```

If simplification

Nested if with common subexpression

```c
if (x > 3 && t) {
    if (x > 3) {
        z = x * y;
    } else {
        z = y - x;
    }
}
```

Loop simplification

```c
int x = 0;
int i = 0;
while (i < 4) {
    i = i + 1;
    x = x + i;
}
```

Loop simplification

```c
int x = 10;
int i = 4;
```

Summary

- Control-flow analysis operates on the control structure of a program (flowgraphs and call graphs)
- Unreachable-code elimination is an intra-procedural optimisation which reduces code size
- Unreachable-procedure elimination is a similar, inter-procedural optimisation making use of the program's call graph
- Analyses for both optimisations must be imprecise in order to guarantee safety
Lecture 3
Live variable analysis

Motivation
Programs may contain
• code which gets executed but which has no useful effect on the program’s overall result;
• occurrences of variables being used before they are defined; and
• many variables which need to be allocated registers and/or memory locations for compilation.

The concept of variable liveness is useful in dealing with all three of these situations.

Liveness
At each instruction, each variable in the program is either live or dead.

We therefore usually consider liveness from an instruction’s perspective: each instruction (or node of the flowgraph) has an associated set of live variables.

Semantic vs. syntactic
There are two kinds of variable liveness:
• Semantic liveness
• Syntactic liveness

Semantic vs. syntactic
A variable \( x \) is semantically live at a node \( n \) if there is some execution sequence starting at \( n \) whose (externally observable) behaviour can be affected by changing the value of \( x \).

Semantic vs. syntactic
A variable \( x \) is semantically live at a node \( n \) if there is some execution sequence starting at \( n \) whose (externally observable) behaviour can be affected by changing the value of \( x \).
Semantic vs. syntactic

Semantic liveness is concerned with the execution behaviour of the program. This is undecidable in general. (e.g. Control flow may depend upon arithmetic.)

Semantic vs. syntactic

Syntactic liveness is concerned with properties of the syntactic structure of the program. Of course, this is decidable.

So what's the difference?

Semantic vs. syntactic

int \( t = x \times y \); \( t \) DEAD
if \( ((x+1) \times (x+1) == y) \) {
  \( t = 1; \)
}
if \( (x^2 + 2x + 1 != y) \) {
  \( t = 2; \)
}
return \( t \);

Semantically: one of the conditions will be true, so on every execution path \( t \) is redefined before it is returned. The value assigned by the first instruction is never used.

Semantic vs. syntactic

On this path through the flowgraph, \( t \) is not redefined before it's used, so \( t \) is syntactically live at the first instruction.

Note that this path never actually occurs during execution.

Semantic vs. syntactic

So, as we’ve seen before, syntactic liveness is a computable approximation of semantic liveness.
Semantic vs. syntactic

\[ \text{sem-live}(n) \subseteq \text{syn-live}(n) \]

Using syntactic methods, we safely overestimate liveness.

Live variable analysis

Variable liveness flows (backwards) through the program in a continuous stream.

Each instruction has an effect on the liveness information as it flows past.

Live variable analysis

An instruction makes a variable live when it references (uses) it.

Live variable analysis

An instruction makes a variable dead when it defines (assigns to) it.

Live variable analysis

We can devise functions \( \text{ref}(n) \) and \( \text{def}(n) \) which give the sets of variables referenced and defined by the instruction at node \( n \).

\[
\begin{align*}
\text{ref}(x = 3) &= \{ \} & \text{ref}(\text{print } x) &= \{ x \} \\
\text{def}(x = 3) &= \{ x \} & \text{def}(\text{print } x) &= \{ \}
\end{align*}
\]

\[
\begin{align*}
\text{ref}(x = x + y) &= \{ x, y \} \\
\text{def}(x = x + y) &= \{ x \}
\end{align*}
\]
Live variable analysis

As liveness flows backwards past an instruction, we want to modify the liveness information by adding any variables which it references (they become live) and removing any which it defines (they become dead).

\[
\begin{align*}
\text{ref( print } x \text{) = } & \{ x \} \\
\text{def( } x = 3 \text{) = } & \{ x \} \\
\text{out-live } (n) = & \{ x, y \} \\
\text{in-live } (n) = & \{ s, t, x, y \}
\end{align*}
\]

Live variable analysis

So, if we consider \( \text{in-live}(n) \) and \( \text{out-live}(n) \), the sets of variables which are live immediately before and immediately after a node, the following equation must hold:

\[
\text{in-live}(n) = \left( \text{out-live}(n) \setminus \text{def}(n) \right) \cup \text{ref}(n)
\]

Live variable analysis

So we know how to calculate \( \text{in-live}(n) \) from the values of \( \text{def}(n) \), \( \text{ref}(n) \) and \( \text{out-live}(n) \). But how do we calculate \( \text{out-live}(n) \)?

\[
\begin{align*}
\text{in-live}(n) = \left( \text{out-live}(n) \setminus \text{def}(n) \right) \cup \text{ref}(n) \\
\text{out-live}(n) = & \ ?
\end{align*}
\]

Live variable analysis

In straight-line code each node has a unique successor, and the variables live at the exit of a node are exactly those variables live at the entry of its successor.

Live variable analysis

In general, however, each node has an arbitrary number of successors, and the variables live at the exit of a node are exactly those variables live at the entry of any of its successors.
Live variable analysis

So the following equation must also hold:
\[
\text{out-live}(n) = \bigcup_{s \in \text{succ}(n)} \text{in-live}(s)
\]

Data-flow equations

These are the data-flow equations for live variable analysis, and together they tell us everything we need to know about how to propagate liveness information through a program.

\[
in-live(n) = \left( \text{out-live}(n) \setminus \text{def}(n) \right) \cup \text{ref}(n)
\]
\[
\text{out-live}(n) = \bigcup_{s \in \text{succ}(n)} \text{in-live}(s)
\]

Algorithm

“Doing the analysis” consists of computing a value \( \text{live}(n) \) for each node \( n \) in a flowgraph such that the liveness data-flow equations are satisfied.

A simple way to solve the data-flow equations is to adopt an iterative strategy.
Algorithm

\[
\text{for } i = 1 \text{ to } n \text{ do } \text{live}[i] := \{\}
\]
\[
\text{while (live[]} changes) \text{ do }
\]
\[
\text{for } i = 1 \text{ to } n \text{ do }
\]
\[
\text{live}[i] := \left( \bigcup_{s \in \text{succ}(i)} \text{live}[s] \right) \setminus \text{def}(i) \cup \text{ref}(i)
\]

Algorithm

Implementation notes:

- If the program has \( n \) variables, we can implement each element of \( \text{live}[] \) as an \( n \)-bit value, with each bit representing the liveness of one variable.
- We can store liveness once per basic block and recompute inside a block when necessary. In this case, given a basic block \( n \) of instructions \( i_1, ..., i_k \):

\[
\text{live}(n) = \left( \bigcup_{s \in \text{succ}(n)} \text{live}(s) \right) \setminus \text{def}(i_k) \cup \text{ref}(i_k) \cdots \setminus \text{def}(i_1) \cup \text{ref}(i_1)
\]

Safety of analysis

- Syntactic liveness safely overapproximates semantic liveness.
- The usual problem occurs in the presence of address-taken variables (cf. labels, procedures): ambiguous definitions and references. For safety we must
  - overestimate ambiguous references (assume all address-taken variables are referenced) and
  - underestimate ambiguous definitions (assume no variables are defined); this increases the size of the smallest solution.

Summary

- Data-flow analysis collects information about how data moves through a program
- Variable liveness is a data-flow property
- Live variable analysis (LVA) is a backwards data-flow analysis for determining variable liveness
- LVA may be expressed as a pair of complementary data-flow equations, which can be combined
- A simple iterative algorithm can be used to find the smallest solution to the LVA data-flow equations

Motivation

Programs may contain code whose result is needed, but in which some computation is simply a redundant repetition of earlier computation within the same program.

The concept of expression availability is useful in dealing with this situation.
Expressions

Any given program contains a finite number of expressions (i.e. computations which potentially produce values), so we may talk about the set of all expressions of a program.

```plaintext
int z = x * y;
print s + t;
int w = u / v;
⋮
```

program contains expressions \{ x*y, s+t, u/v, ... \}

Availability

Availability is a data-flow property of expressions: “Has the value of this expression already been computed?”

```plaintext
int z = x * y;
print s + t;
```

Availability

At each instruction, each expression in the program is either available or unavailable.

We therefore usually consider availability from an instruction’s perspective: each instruction (or node of the flowgraph) has an associated set of available expressions.

```plaintext
int z = x * y;
print s + t;
int w = u / v;
⋮
```

\( \text{avail}(n) = \{ x*y, s+t \} \)

Semantic vs. syntactic

For example, availability differs from earlier examples in a subtle but important way: we want to know which expressions are definitely available (i.e. have already been computed) at an instruction, not which ones may be available.

As before, we should consider the distinction between semantic and syntactic (or, alternatively, dynamic and static) availability of expressions, and the details of the approximation which we hope to discover by analysis.

An expression is \textit{semantically} available at a node \( n \) if its value gets computed (and not subsequently invalidated) along every execution sequence ending at \( n \).

An expression is \textit{syntactically} available at a node \( n \) if its value gets computed (and not subsequently invalidated) along every path from the entry of the flowgraph to \( n \).

And, as expected, only the latter is decidable.
Semantic vs. syntactic

if ((x+1)*(x+1) == y) {
    s = x + y;
}
if (x*x + 2*x + 1 != y) {
    t = x + y;
}
return x + y;  \text{ \textsc{x+y AVAILABLE}}

Semantically: one of the conditions will be true, so on every execution path \textsc{x+y} is computed twice. The recomputation of \textsc{x+y} is redundant.

Semantic vs. syntactic

On this path through the flowgraph, \textsc{x+y} is only computed once, so \textsc{x+y} is syntactically unavailable at the last instruction.

Note that this path never actually occurs during execution.

Semantic vs. syntactic

\text{\Fbox{ADD t32,x,#1}}
\text{\Fbox{MUL t33,t32,t32}}
\text{\Fbox{CMPEQ t33,y,lab1}}
\text{\Fbox{ADD s,x,y}}

lab1: \text{\Fbox{MUL t34,x,x}}
\text{\Fbox{MUL t35,x,#2}}
\text{\Fbox{ADD t36,t34,t35}}
\text{\Fbox{ADD t37,t36,#1}}
\text{\Fbox{CMPEQ t37,y,lab2}}
\text{\Fbox{ADD t,x,y}}

lab2: \text{\Fbox{ADD res1,x,y}}

If an expression is deemed to be available, we may do something dangerous (e.g. remove an instruction which recomputes its value).

Whereas with live variable analysis we found safety in assuming that more variables were live, here we find safety in assuming that fewer expressions are available.

Semantic vs. syntactic

program expressions

\[ \text{sem-avail}(n) \supseteq \text{syn-avail}(n) \]

This time, we safely underestimate availability. (cf. \text{sem-live}(n) \subseteq \text{syn-live}(n))

Warning

Danger: there is a standard presentation of available expression analysis (textbooks, notes for this course) which is formally satisfying but contains an easily-overlooked subtlety.

We'll first look at an equivalent, more intuitive bottom-up presentation, then amend it slightly to match the version given in the literature.
Available expression analysis

Available expressions is a forwards data-flow analysis: information from past instructions must be propagated forwards through the program to discover which expressions are available.

```
int z = x * y;
if (x*y > 0)
    t = x * y;
print a*b;
```  

Available expression analysis

Unlike variable liveness, expression availability flows forwards through the program.
As in liveness, though, each instruction has an effect on the availability information as it flows past.

```
{ a*b, c+1, d/e, d-1 }
```

Available expression analysis

An instruction makes an expression available when it generates (computes) its current value.

```
gen( print x+1 ) = { x+1 }  
generate( x = 3 ) = { x+1 }  
gen( a*b ) = { a*b, d+1, f/g }  
generate( x = x + y ) = { x+y }  
generate( print x+1 ) = { x+1 }  
generate( x = x + y ) = E_x
```

Available expression analysis

An instruction makes an expression unavailable when it kills (invalidates) its current value.

```
{ a*b, c+1, d/e, d-1 }  
kill( x = 3 ) = { a*b, d+1, f/g }  
kill( print x+1 ) = { }  
kill( x = x + y ) = E_x
```

Available expression analysis

As in LVA, we can devise functions `gen(n)` and `kill(n)` which give the sets of expressions generated and killed by the instruction at node \( n \).

The situation is slightly more complicated this time: an assignment to a variable \( x \) kills all expressions in the program which contain occurrences of \( x \).

So, in the following, \( E_x \) is the set of expressions in the program which contain occurrences of \( x \).
Available expression analysis

As availability flows forwards past an instruction, we want to modify the availability information by adding any expressions which it generates (they become available) and removing any which it kills (they become unavailable).

\[
\begin{align*}
gen(\text{print } x+1) &= \{ x+1 \} \\
\text{kill}(x = 3) &= E_x \\
\end{align*}
\]

Available expression analysis

If an instruction both generates and kills expressions, we must remove the killed expressions after adding the generated ones (cf. removing def(n) before adding ref(n)).

\[
\begin{align*}
gen( x = x + y ) &= \{ x+y \} \\
\text{kill}(x = x + y) &= E_x \\
\end{align*}
\]

Available expression analysis

So, if we consider in-avail(n) and out-avail(n), the sets of expressions which are available immediately before and immediately after a node, the following equation must hold:

\[
\text{out-avail}(n) = (\text{in-avail}(n) \cup \text{gen}(n)) \setminus \text{kill}(n)
\]

Available expression analysis

As in LVA, we have devised one equation for calculating out-avail(n) from the values of gen(n), kill(n) and in-avail(n), and now need another for calculating in-avail(n).

\[
\begin{align*}
in-avail(n) &= ? \\
\text{out-avail}(n) &= (\text{in-avail}(n) \cup \text{gen}(n)) \setminus \text{kill}(n)
\end{align*}
\]

Available expression analysis

When a node n has a single predecessor m, the information propagates along the control-flow edge as you would expect: in-avail(n) = out-avail(m).

When a node has multiple predecessors, the expressions available at the entry of that node are exactly those expressions available at the exit of all of its predecessors (cf. “any of its successors” in LVA).

Available expression analysis

So the following equation must also hold:

\[
in-avail(n) = \bigcap_{p \in \text{pred}(n)} \text{out-avail}(p)
\]
Data-flow equations

These are the data-flow equations for available expression analysis, and together they tell us everything we need to know about how to propagate availability information through a program.

\[ \text{in-avail}(n) = \bigcap_{p \in \text{pred}(n)} \text{out-avail}(p) \]
\[ \text{out-avail}(n) = \left( \text{in-avail}(n) \cup \text{gen}(n) \right) \setminus \text{kill}(n) \]

Data-flow equations

Danger: we have overlooked one important detail.

\[ \text{avail}(n) = \bigcap_{p \in \text{pred}(n)} \left( \left( \text{avail}(p) \cup \text{gen}(p) \right) \setminus \text{kill}(p) \right) \]

Clearly there should be no expressions available here, so we must stipulate explicitly that \( \text{avail}(n) = \{\} \) if \( \text{pred}(n) = \{\} \).

Data-flow equations

The functions and equations presented so far are correct, and their definitions are fairly intuitive.

However, we may wish to have our data-flow equations in a form which more closely matches that of the LVA equations, since this emphasises the similarity between the two analyses and hence is how they are most often presented.

A few modifications are necessary to achieve this.

\[ \text{in-live}(n) = \left( \text{out-live}(n) \setminus \text{def}(n) \right) \cup \text{ref}(n) \]
\[ \text{out-live}(n) = \bigcup_{s \in \text{succ}(n)} \text{in-live}(s) \]

These differences are inherent in the analyses.

Data-flow equations

We might instead have decided to define \( \text{gen}(n) \) and \( \text{kill}(n) \) to coincide with the following (standard) definitions:

- A node generates an expression \( e \) if it must compute the value of \( e \) and does not subsequently redefine any of the variables occurring in \( e \).
- A node kills an expression \( e \) if it may redefine some of the variables occurring in \( e \) and does not subsequently recompute the value of \( e \).
Data-flow equations

By the old definition:
\[
\begin{align*}
\text{gen}(x &= x + y) = \{x+y\} \\
\text{kill}(x &= x + y) = E_x
\end{align*}
\]

By the new definition:
\[
\begin{align*}
\text{gen}(x &= x + y) = \{ \} \\
\text{kill}(x &= x + y) = E_x
\end{align*}
\]

(The new kill(n) may visibly differ when n is a basic block.)

Data-flow equations

From this new equation for out-avail(n) we may produce our final data-flow equation for expression availability:

\[
avail(n) = \left\{ \bigcap_{p \in \text{pred}(n)} ((\text{avail}(p) \setminus \text{kill}(p)) \cup \text{gen}(p)) \right\}
\]

This is the equation you will find in the course notes and standard textbooks on program analysis; remember that it depends on these more subtle definitions of gen(n) and kill(n).

Algorithm

for i = 1 to n do avail[i] := U
while (avail[] changes) do
  for i = 1 to n do
    avail[i] := \(\bigcap_{p \in \text{pred}(i)} ((\text{avail}[p] \setminus \text{kill}(p)) \cup \text{gen}(p))\)

We can do better if we assume that the flowgraph has a single entry node (the first node in avail[]).

Then avail[1] may instead be initialised to the empty set, and we need not bother recalculating availability at the first node during each iteration.

Algorithm

\[
avail[1] := \{ \}
\]

for i = 2 to n do avail[i] := U
while (avail[] changes) do
  for i = 2 to n do
    avail[i] := \(\bigcap_{p \in \text{pred}(i)} ((\text{avail}[p] \setminus \text{kill}(p)) \cup \text{gen}(p))\)

As with LVA, this algorithm is guaranteed to terminate since the effect of one iteration is monotonic (it only removes expressions from availability sets) and an empty availability set cannot get any smaller.

Any solution to the data-flow equations is safe, but this algorithm is guaranteed to give the largest (and therefore most precise) solution.

Algorithm

\[
\begin{align*}
\text{out-avail}(n) &= (\text{in-avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n) \\
\text{in-live}(n) &= (\text{out-live}(n) \setminus \text{def}(n)) \cup \text{ref}(n)
\end{align*}
\]
Algorithm

Implementation notes:

- If we arrange our programs such that each assignment assigns to a distinct temporary variable, we may number these temporaries and hence number the expressions whose values are assigned to them.

- If the program has \( n \) such expressions, we can implement each element of \( \text{avail}[] \) as an \( n \)-bit value, with the \( m \)th bit representing the availability of expression number \( m \).

Safety of analysis

- Syntactic availability safely underapproximates semantic availability.

- Address-taken variables are again a problem. For safety we must
  - underestimate ambiguous generation (assume no expressions are generated) and
  - overestimate ambiguous killing (assume all expressions containing address-taken variables are killed); this decreases the size of the largest solution.

Analysis framework

\[
\begin{align*}
\text{in-live}(n) &= \left( \text{out-live}(n) \setminus \text{def}(n) \right) \cup \text{ref}(n) \\
\text{out-live}(n) &= \bigcup_{s \in \text{succ}(n)} \text{in-live}(s) \\
\text{in-avail}(n) &= \bigcap_{p \in \text{pred}(n)} \text{out-avail}(p) \\
\text{out-avail}(n) &= \left( \text{in-avail}(n) \setminus \text{kill}(n) \right) \cup \text{gen}(n)
\end{align*}
\]

Analysis framework

The two data-flow analyses we’ve seen, LVA and AVAIL, clearly share many similarities.

In fact, they are both instances of the same simple data-flow analysis framework: some program property is computed by iteratively finding the most precise solution to data-flow equations, which express the relationships between values of that property immediately before and immediately after each node of a flowgraph.

\[
\begin{align*}
\text{avail}(n) &= \bigcap_{p \in \text{pred}(n)} (\text{avail}(p) \setminus \text{kill}(p[1]) \cup \text{gen}(p[1])) \cdots \setminus \text{kill}(p[k_p]) \cup \text{gen}(p[k_p])) \bigcup \text{in}(s) \quad (s \in \text{succ}(n)) \\
\text{in}(n) &= \bigcup_{p \in \text{pred}(n)} \text{out}(p) \\
\text{out}(n) &= \bigcup_{s \in \text{succ}(n)} \text{in}(s)
\end{align*}
\]

Analysis framework

LVA’s data-flow equations have the form

\[
\begin{align*}
\text{in}(n) &= (\text{out}(n) \setminus \text{...}) \cup \text{...} \\
\text{out}(n) &= \bigcup_{s \in \text{succ}(n)} \text{in}(s) \\
\text{union over successors}
\end{align*}
\]

AVAIL’s data-flow equations have the form

\[
\begin{align*}
\text{out}(n) &= (\text{in}(n) \setminus \text{...}) \cup \text{...} \\
\text{in}(n) &= \bigcap_{p \in \text{pred}(n)} \text{out}(p) \\
\text{intersection over predecessors}
\end{align*}
\]

So, given a single algorithm for iterative solution of data-flow equations of this form, we may compute all these analyses and any others which fit into the framework.

...and others
Summary

• Expression availability is a data-flow property
• Available expression analysis (AVAIL) is a forwards data-flow analysis for determining expression availability
• AVAIL may be expressed as two complementary data-flow equations, which may be combined
• A simple iterative algorithm can be used to find the largest solution to the data-flow equations
• AVAIL and LVA are both instances (among others) of the same data-flow analysis framework

Motivation

Both human- and computer-generated programs sometimes contain data-flow anomalies. These anomalies result in the program being worse, in some sense, than it was intended to be. Data-flow analysis is useful in locating, and sometimes correcting, these code anomalies.

Dead code

Dead code is a simple example of a data-flow anomaly, and LVA allows us to identify it. Recall that code is dead when its result goes unused; if the variable \( x \) is not live on exit from an instruction which assigns some value to \( x \), then the whole instruction is dead.

For this kind of anomaly, an automatic remedy is not only feasible but also straightforward: dead code with no live side effects is useless and may be removed.

Optimisation vs. debugging

Data-flow anomalies may manifest themselves in different ways: some may actually “break” the program (make it crash or exhibit undefined behaviour), others may just make the program “worse” (make it larger or slower than necessary).

Any compiler needs to be able to report when a program is broken (i.e. “compiler warnings”), so the identification of data-flow anomalies has applications in both optimisation and bug elimination.

Lecture 5

Data-flow anomalies and clash graphs

Dead code

For this kind of anomaly, an automatic remedy is not only feasible but also straightforward: dead code with no live side effects is useless and may be removed.

Successive iterations may yield further improvements.

\begin{align*}
\text{\{a, b, z\}} & \\
\text{\{x, y, z\}} & \\
a &= x + 11; & \\
b &= y + 13; & \\
c &= a \times b; & \text{DEAD} \\
\text{print z;} & \text{\{z\}} \\
& \text{\{\}}
\end{align*}
Dead code

The program resulting from this transformation will remain correct and will be both smaller and faster than before (cf. just smaller in unreachable code elimination), and no programmer intervention is required.

Uninitialised variables

In some languages, for example C and our 3-address intermediate code, it is syntactically legitimate for a program to read from a variable before it has definitely been initialised with a value.

If this situation occurs during execution, the effect of the read is usually undefined and depends upon unpredictable details of implementation and environment.

Uninitialised variables

This kind of behaviour is often undesirable, so we would like a compiler to be able to detect and warn of the situation.

Happily, the liveness information collected by LVA allows a compiler to see easily when a read from an undefined variable is possible.

Uninitialised variables

If any variables are still live at the beginning of a program, they represent uses which are potentially unmatched by corresponding definitions, and hence indicate a program with potentially undefined (and therefore incorrect) behaviour.

In this situation, the compiler can issue a warning: “variable z may be used before it is initialised”.

However, because LVA computes a safe (syntactic) overapproximation of variable liveness, some of these compiler warnings may be (semantically) spurious.

Uninitialised variables

x = 11; { }
\{x\}
y = 13; \{x, y\}
z = 17; \{x, y\}
print x; \{x, y\}
print y; \{y\}
\}
\}

x = 11; { z } z LIVE
\{x, z\}
y = 13; \{x, y, z\}
: \{x, y, z\}
print x; \{x, y, z\}
print y; \{y, z\}
print z; \{z\}
\}
\}
\}
\}

✓

✗
Uninitialised variables

Here the analysis is being too safe, and the warning is unnecessary, but this imprecision is the nature of our computable approximation to semantic liveness.

So the compiler must either risk giving unnecessary warnings about correct code ("false positives") or failing to give warnings about incorrect code ("false negatives"). Which is worse?

Opinions differ.

Uninitialised variables

Although dead code may easily be remedied by the compiler, it’s not generally possible to automatically fix the problem of uninitialised variables.

As just demonstrated, even the decision as to whether a warning indicates a genuine problem must often be made by the programmer, who must also fix any such problems by hand.

Uninitialised variables

Write-write anomalies

While LVA is useful in these cases, some similar data-flow anomalies can only be spotted with a different analysis.

Write-write anomalies are an example of this. They occur when a variable may be written twice with no intervening read; the first write may then be considered unnecessary in some sense.

```
x = 11;
x = 13;
```

```
print x;
```

Write-write anomalies

A simple data-flow analysis can be used to track which variables may have been written but not yet read at each node.

In a sense, this involves doing LVA in reverse (i.e. forwards!): at each node we should remove all variables which are referenced, then add all variables which are defined.

```
in-wrn(n) = \bigcup_{p \in \text{pred}(n)} \text{out-wrn}(p)
```

```
out-wrn(n) = (in-wrn(n) \setminus \text{ref}(n)) \cup \text{def}(n)
```

```
wnr(n) = \bigcup_{p \in \text{pred}(n)} ((\text{wref}(p) \setminus \text{ref}(p)) \cup \text{def}(p))
```
Write-write anomalies

\[
x = 11; \\
y = 13; \\
z = 17; \\
\vdots \\
\text{print } x; \\
y = 19; \\
\vdots \\
\{ \}
\]

But, although the second write to a variable may turn an earlier write into dead code, the presence of a write-write anomaly doesn’t necessarily mean that a variable is dead — hence the need for a different analysis.

Write-write anomalies

\[
x = 11; \\
\text{if (p) } \{ \\
  x = 13; \\
\} \text{ else } \{ \\
  x = 11; \\
\} \\
\text{print } x;
\]

This code does the same job, but avoids writing to \( x \) twice in succession on any control-flow path.

Write-write anomalies

\[
\text{if (p) } \{ \\
  x = 13; \\
\} \text{ else } \{ \\
  x = 11; \\
\} \\
\text{print } x;
\]

As with uninitialised variable anomalies, the programmer must be relied upon to investigate the compiler’s warnings and fix any genuine problems which they indicate.

Write-write anomalies

\[
\text{if (p) } \{ \\
  x = 13; \\
\} \text{ else } \{ \\
  x = 11; \\
\} \\
\text{print } x;
\]

Again, the analysis may be too approximate to notice that a particular write-write anomaly may never occur during any execution, so warnings may be inaccurate.

Clash graphs

The ability to detect data-flow anomalies is a nice compiler feature, but LVA’s main utility is in deriving a data structure known as a clash graph (aka interference graph).

Clash graphs

When generating intermediate code it is convenient to simply invent as many variables as necessary to hold the results of computations; the extreme of this is “normal form”, in which a new temporary variable is used on each occasion that one is required, with none being reused.
This makes generating 3-address code as straightforward as possible, and assumes an imaginary target machine with an unlimited supply of “virtual registers”, one to hold each variable (and temporary) in the program.

Such a naïve strategy is obviously wasteful, however, and won’t generate good code for a real target machine.

Before we can work on improving the situation, we must collect information about which variables actually need to be allocated to different registers on the target machine, as opposed to having been incidentally placed in different registers by our translation to normal form.

LVA is useful here because it can tell us which variables are simultaneously live, and hence must be kept in separate virtual registers for later retrieval.

In a program’s clash graph there is one vertex for each virtual register and an edge between vertices when their two registers are ever simultaneously live.
Summary

• Data-flow analysis is helpful in locating (and sometimes correcting) data-flow anomalies
• LVA allows us to identify dead code and possible uses of uninitialised variables
• Write-write anomalies can be identified with a similar analysis
• Imprecision may lead to overzealous warnings
• LVA allows us to construct a clash graph

Motivation

Normal form is convenient for intermediate code.

However, it's extremely wasteful.

Real machines only have a small finite number of registers, so at some stage we need to analyse and transform the intermediate representation of a program so that it only requires as many (architectural) registers as are really available.

This task is called register allocation.

Graph colouring

Register allocation depends upon the solution of a closely related problem known as graph colouring.

For general (non-planar) graphs, however, four colours are not sufficient; there is no bound on how many may be required.
Graph colouring

Algorithms

1. Choose a vertex (i.e. virtual register) which has the least number of incident edges (i.e. clashes).
2. Remove the vertex and its edges from the graph, and push the vertex onto a LIFO stack.
3. Repeat until the graph is empty.
4. Pop each vertex from the stack and colour it in the most conservative way which avoids the colours of its (already-coloured) neighbours.

Finding the minimal colouring for a graph is NP-hard, and therefore difficult to do efficiently.

However, we may use a simple heuristic algorithm which chooses a sensible order in which to colour vertices and usually yields satisfactory results on real clash graphs.

How many colours (i.e. architectural registers) are necessary to colour a clash graph such that no two connected vertices have the same colour (i.e. such that no two simultaneously live virtual registers are stored in the same arch. register)?

• What colour should each vertex be?

_allocation by colouring_

This is essentially the same problem that we wish to solve for clash graphs.

MOV r0,#11
MOV r1,#13
ADD r0,r0,r1
MUL r2,r0,#2
MOV r0,#17
MOV r1,#19
MUL r0,r0,r1
ADD r2,r2,r0

 allocation by colouring

Finding the minimal colouring for a graph is NP-hard, and therefore difficult to do efficiently.

However, we may use a simple heuristic algorithm which chooses a sensible order in which to colour vertices and usually yields satisfactory results on real clash graphs.

MOV r0,#11
MOV y,#13
ADD t1,x,y
MUL z,t1,#2
MOV a,#17
MOV b,#19
MUL t2,a,b
ADD z,z,t2
MOV x,#11
MOV y,#13...
Algorithm

Bear in mind that this is only a heuristic.

A better (more minimal) colouring may exist.

Spilling

This algorithm tries to find an approximately minimal colouring of the clash graph, but it assumes new colours are always available when required.

In reality we will usually have a finite number of colours (i.e. architectural registers) available; how should the algorithm cope when it runs out of colours?

Spilling

The quantity of architectural registers is strictly limited, but it is usually reasonable to assume that fresh memory locations will always be available.

So, when the number of simultaneously live values exceeds the number of architectural registers, we may spill the excess values into memory.

Operating on values in memory is of course much slower, but it gets the job done.

Algorithm

- Choose a vertex with the least number of edges.
- If it has fewer edges than there are colours, 
  - remove the vertex and push it onto a stack,
  - otherwise choose a register to spill — e.g. the least-accessed one — and remove its vertex.
- Repeat until the graph is empty.
- Pop each vertex from the stack and colour it.
- Any uncoloured vertices must be spilled.

Spilling

ADD a, b, c

vs.

LDR t1,#0xFFA4
LDR t2,#0xFFA8
ADD t3,t1,t2
STR t3,#0xFFA0

Algorithm

a: 3, b: 5, c: 7, d: 11, w: 13, x: 17, y: 19, z: 23

Algorithm

r0
r1

a and b spilled to memory

a: 3, b: 5, c: 7, d: 11, w: 13, x: 17, y: 19, z: 23
Algorithm

Choosing the right virtual register to spill will result in a faster, smaller program.

The static count of “how many accesses?” is a good start, but doesn’t take account of more complex issues like loops and simultaneous liveness with other spilled values.

One easy heuristic is to treat one static access inside a loop as (say) 4 accesses; this generalises to $4^n$ accesses inside a loop nested to level $n$.

Algorithm

When we are popping vertices from the stack and assigning colours to them, we sometimes have more than one colour to choose from.

If the program contains an instruction “MOV $a, b$” then storing $a$ and $b$ in the same arch. register (as long as they don’t clash) will allow us to delete that instruction.

We can construct a preference graph to show which pairs of registers appear together in MOV instructions, and use it to guide colouring decisions.

Algorithm

We have assumed that we are free to choose architectural registers however we want to, but this is simply not the case on some architectures.

- The x86 MUL instruction expects one of its arguments in the AL register and stores its result into AX.
- The VAX MOV C3 instruction zeroes r0, r2, r4 and r5, storing its results into r1 and r3.

We must be able to cope with such irregularities.

Non-orthogonal instructions

We can handle the situation tidily by pre-allocating a virtual register to each of the target machine’s arch. registers, e.g. keep v0 in r0, v1 in r1, ..., v31 in r31.

When generating intermediate code in normal form, we avoid this set of registers, and use new ones (e.g. v32, v33, ...) for temporaries and user variables.

In this way, each architectural register is explicitly represented by a unique virtual register.

Non-orthogonal instructions

We must now do extra work when generating intermediate code:

- When an instruction requires an operand in a specific arch. register (e.g. x86 MUL), we generate a preceding MOV to put the right value into the corresponding virtual register.
- When an instruction produces a result in a specific arch. register (e.g. x86 MUL), we generate a trailing MOV to transfer the result into a new virtual register.

Non-orthogonal instructions

If (hypothetically) ADD on the target architecture can only perform $r0 = r1 + r2$:

\[
\begin{align*}
x &= 19; \\
y &= 23; \\
z &= x + y;
\end{align*}
\]

\[
\begin{align*}
&\text{MOV } v32, #19 \\
&\text{MOV } v33, #23 \\
&\text{MOV } v1, v32 \\
&\text{MOV } v2, v33 \\
&\text{ADD } v0, v1, v2 \\
&\text{MOV } v34, v0
\end{align*}
\]

This may seem particularly wasteful, but many of the MOV instructions will be eliminated during register allocation if a preference graph is used.
Non-orthogonal instructions

This may seem particularly wasteful, but many of the MOV instructions will be eliminated during register allocation if a preference graph is used.

This may seem particularly wasteful, but many of the MOV instructions will be eliminated during register allocation if a preference graph is used.

Non-orthogonal instructions

And finally, we may also synthesise MOV instructions to ensure that arguments and results end up in the correct registers, and use the preference graph to guide colouring such that most of these MOVs can be deleted again.

Procedure calling standards

This final technique of synthesising edges on the clash graph in order to avoid corrupted registers is helpful for dealing with the procedure calling standard of the target architecture.

Such a standard will usually dictate that procedure calls (e.g. CALL and CALLI instructions in our 3-address code) should use certain registers for arguments and results, should preserve certain registers over a call, and may corrupt any other registers if necessary.

Procedure calling standards

On the ARM, for example:

- Arguments should be placed in r0-r3 before a procedure is called.
- Results should be returned in r0 and r1.
- r4-r8, r10 and r11 should be preserved over procedure calls, and r9 might be depending on the platform.
- r12-r15 are special registers, including the stack pointer and program counter.

Non-orthogonal instructions

MOV r1, #19
MOV r2, #23
MOV r1, r1
MOV r2, r2
ADD r0, r1, r2
MOV r0, r0

Non-orthogonal instructions

If (hypothetically) MUL on the target architecture corrupts the contents of r0:

MOV r1, #6
MOV r2, #7
MUL r0, r1, r2

Non-orthogonal instructions

MOV r1, #6
MOV r2, #7
MUL r0, r1, r2

Non-orthogonal instructions

MOV r1, #19
MOV r2, #23
MOV r0, r0

Procedure calling standards

Since a procedure call instruction may corrupt some of the registers (r0-r3 and possibly r9 on the ARM), we can synthesise edges on the clash graph between the corrupted registers and all other virtual registers live at the call instruction.

As before, we may also synthesise MOV instructions to ensure that arguments and results end up in the correct registers, and use the preference graph to guide colouring such that most of these MOVs can be deleted again.

Non-orthogonal instructions

This final technique of synthesising edges on the clash graph in order to avoid corrupted registers is helpful for dealing with the procedure calling standard of the target architecture.

Such a standard will usually dictate that procedure calls (e.g. CALL and CALLI instructions in our 3-address code) should use certain registers for arguments and results, should preserve certain registers over a call, and may corrupt any other registers if necessary.

Procedure calling standards

On the ARM, for example:

- Arguments should be placed in r0-r3 before a procedure is called.
- Results should be returned in r0 and r1.
- r4-r8, r10 and r11 should be preserved over procedure calls, and r9 might be depending on the platform.
- r12-r15 are special registers, including the stack pointer and program counter.

Non-orthogonal instructions

This may seem particularly wasteful, but many of the MOV instructions will be eliminated during register allocation if a preference graph is used.

Non-orthogonal instructions

And finally, we may also synthesise MOV instructions to ensure that arguments and results end up in the correct registers, and use the preference graph to guide colouring such that most of these MOVs can be deleted again.

Procedure calling standards

This final technique of synthesising edges on the clash graph in order to avoid corrupted registers is helpful for dealing with the procedure calling standard of the target architecture.

Such a standard will usually dictate that procedure calls (e.g. CALL and CALLI instructions in our 3-address code) should use certain registers for arguments and results, should preserve certain registers over a call, and may corrupt any other registers if necessary.

Procedure calling standards

On the ARM, for example:

- Arguments should be placed in r0-r3 before a procedure is called.
- Results should be returned in r0 and r1.
- r4-r8, r10 and r11 should be preserved over procedure calls, and r9 might be depending on the platform.
- r12-r15 are special registers, including the stack pointer and program counter.

Non-orthogonal instructions

This may seem particularly wasteful, but many of the MOV instructions will be eliminated during register allocation if a preference graph is used.

Non-orthogonal instructions

And finally, we may also synthesise MOV instructions to ensure that arguments and results end up in the correct registers, and use the preference graph to guide colouring such that most of these MOVs can be deleted again.

Procedure calling standards

This final technique of synthesising edges on the clash graph in order to avoid corrupted registers is helpful for dealing with the procedure calling standard of the target architecture.

Such a standard will usually dictate that procedure calls (e.g. CALL and CALLI instructions in our 3-address code) should use certain registers for arguments and results, should preserve certain registers over a call, and may corrupt any other registers if necessary.

Procedure calling standards

On the ARM, for example:

- Arguments should be placed in r0-r3 before a procedure is called.
- Results should be returned in r0 and r1.
- r4-r8, r10 and r11 should be preserved over procedure calls, and r9 might be depending on the platform.
- r12-r15 are special registers, including the stack pointer and program counter.

Non-orthogonal instructions

This may seem particularly wasteful, but many of the MOV instructions will be eliminated during register allocation if a preference graph is used.

Non-orthogonal instructions

And finally, we may also synthesise MOV instructions to ensure that arguments and results end up in the correct registers, and use the preference graph to guide colouring such that most of these MOVs can be deleted again.

Procedure calling standards

This final technique of synthesising edges on the clash graph in order to avoid corrupted registers is helpful for dealing with the procedure calling standard of the target architecture.

Such a standard will usually dictate that procedure calls (e.g. CALL and CALLI instructions in our 3-address code) should use certain registers for arguments and results, should preserve certain registers over a call, and may corrupt any other registers if necessary.

Procedure calling standards

On the ARM, for example:

- Arguments should be placed in r0-r3 before a procedure is called.
- Results should be returned in r0 and r1.
- r4-r8, r10 and r11 should be preserved over procedure calls, and r9 might be depending on the platform.
- r12-r15 are special registers, including the stack pointer and program counter.
Summary

- A register allocation phase is required to assign each virtual register to an architectural one during compilation.
- Registers may be allocated by colouring the vertices of a clash graph.
- When the number of arch. registers is limited, some virtual registers may be spilled to memory.
- Non-orthogonal instructions may be handled with additional MOVs and new edges on the clash graph.
- Procedure calling standards also handled this way.

Motivation

Some expressions in a program may cause redundant recomputation of values. If such recomputation is safely eliminated, the program will usually become faster. There exist several redundancy elimination optimisations which attempt to perform this task in different ways (and for different specific meanings of “redundancy”).

Common subexpressions

Recall that an expression is available at an instruction if its value has definitely already been computed and not been subsequently invalidated by assignments to any of the variables occurring in the expression. If the expression \( e \) is available on entry to an instruction which computes \( e \), the instruction is performing a redundant computation and can be modified or removed.

Common subexpressions

We consider this redundantly-computed expression to be a common subexpression: it is common to more than one instruction in the program, and in each of its occurrences it may appear as a subcomponent of some larger expression.
Common subexpressions

We can eliminate a common subexpression by storing its value into a new temporary variable when it is first computed, and reusing that variable later when the same value is required.

Algorithm

- Find a node \( n \) which computes an already-available expression \( e \)
- Replace the occurrence of \( e \) with a new temporary variable \( t \)
- On each control path backwards from \( n \), find the first instruction calculating \( e \) and add a new instruction to store its value into \( t \)
- Repeat until no more redundancy is found

Common subexpressions

Our transformed program performs (statically) fewer arithmetic operations: \( y*z \) is now computed in three places rather than four.

However, three register copy instructions have also been generated; the program is now larger, and whether it is faster depends upon characteristics of the target architecture.

Copy propagation

This simple formulation of CSE is fairly careless, and assumes that other compiler phases are going to tidy up afterwards.

In addition to register allocation, a transformation called copy propagation is often helpful here.

In copy propagation, we scan forwards from an \( x=y \) instruction and replace \( x \) with \( y \) wherever it appears (as long as neither \( x \) nor \( y \) have been modified).
Copy propagation
\[
\begin{align*}
  t_3 &= y \times z \\
  a &= t_3 \\
  t_2 &= y \times z \\
  b &= t_2 \\
  t_1 &= t_2 \\
  c &= t_1 \\
  d &= t_1 \\
\end{align*}
\]

Code motion

Transformations such as CSE are known collectively as code motion transformations: they operate by moving instructions and computations around programs to take advantage of opportunities identified by control- and data-flow analysis.

Code motion is particularly useful in eliminating different kinds of redundancy.

It's worth looking at other kinds of code motion.

Code hoisting

Code hoisting reduces the size of a program by moving duplicated expression computations to the same place, where they can be combined into a single instruction.

Hoisting relies on a data-flow analysis called very busy expressions (a backwards version of AVAIL) which finds expressions that are definitely going to be evaluated later in the program; these can be moved earlier and possibly combined with each other.

Code hoisting

Hoisting may have a different effect on execution time depending on the exact nature of the code. The resulting program may be slower, faster, or just the same speed as before.

Loop-invariant code motion

Some expressions inside loops are redundant in the sense that they get recomputed on every iteration even though their value never changes within the loop.

Loop-invariant code motion recognises these redundant computations and moves such expressions outside of loop bodies so that they are only evaluated once.
Loop-invariant code motion

```
a = ...; 
b = ...; 
x = a + b; 
while (...) { 
  ... 
} 
print x;
```

This transformation depends upon a data-flow analysis to discover which assignments may affect the value of a variable ("reaching definitions").

If none of the variables in the expression are redefined inside the loop body (or are only redefined by computations involving other invariant values), the expression is invariant between loop iterations and may safely be relocated before the beginning of the loop.

Partial redundancy

*Partial redundancy elimination* combines common-subexpression elimination and loop-invariant code motion into one optimisation which improves the performance of code.

An expression is *partially redundant* when it is computed more than once on *some* (vs. all) paths through a flowgraph; this is often the case for code inside loops, for example.

This example gives a faster program of the same size.

Partial redundancy elimination can be achieved in its own right using a complex combination of several forwards and backwards data-flow analyses in order to locate partially redundant computations and discover the best places to add and delete instructions.

Putting it all together

```
a = x + y; 
b = x + y; 
r = z; 
if (a == 42) { 
  r = a + b; 
  s = x + y; 
} else { 
  s = a + b; 
} 
t = b + r;  
u = x + y; 
⋮
return r+s+t+u;
```

```
ADD a,x,y 
ADD b,x,y 
MOV r,z 
ADD r,a,b 
ADD s,x,y 
ADD s,a,b 
ADD t,b,r 
ADD u,x,y
```

Putting it all together

```
x+y COMMON
ADD a,x,y 
ADD b,x,y 
MOV r,z 
ADD r,a,b 
ADD s,x,y 
ADD s,a,b 
ADD t,b,r 
ADD u,x,y 
x+y COMMON
```

```
x+y COMMON
ADD t1,x,y 
MOV a,t1 
MOV b,t1 
MOV r,z 
ADD a,x,y 
ADD b,x,y 
ADD s,x,y 
ADD s,a,b 
ADD t,b,r 
ADD u,x,y 
x+y COMMON
```

```
x+y COMMON
ADD t,b,r 
ADD u,x,y
```

```
x+y COMMON
ADD t1,x,y 
MOV a,t1 
MOV b,t1 
MOV r,z 
ADD a,x,y 
ADD b,x,y 
ADD s,x,y 
ADD s,a,b 
ADD t,b,r 
ADD u,x,y 
x+y COMMON
```

```
x+y COMMON
ADD t,b,r 
ADD u,x,y
```

```
x+y COMMON
ADD t1,x,y 
MOV a,t1 
MOV b,t1 
MOV r,z 
ADD a,x,y 
ADD b,x,y 
ADD s,x,y 
ADD s,a,b 
ADD t,b,r 
ADD u,x,y 
x+y COMMON
```

```
x+y COMMON
ADD t,b,r 
ADD u,x,y
```

```
x+y COMMON
ADD t1,x,y 
MOV a,t1 
MOV b,t1 
MOV r,z 
ADD a,x,y 
ADD b,x,y 
ADD s,x,y 
ADD s,a,b 
ADD t,b,r 
ADD u,x,y 
x+y COMMON
```

```
x+y COMMON
ADD t,b,r 
ADD u,x,y
```
Summary

- Some optimisations exist to reduce or remove redundancy in programs
- One such optimisation, common-subexpression elimination, is enabled by AVAIL
- Copy propagation makes CSE practical
- Other code motion optimisations can also help to reduce redundancy
- These optimisations work together to improve code
Intermediate code in normal form permits maximum flexibility in allocating temporary variables to architectural registers.

This flexibility is not extended to user variables, and sometimes more registers than necessary will be used. Register allocation can do a better job with user variables if we first translate code into SSA form.

User variables are often reassigned and reused many times over the course of a program, so that they become live in many different places.

Our intermediate code generation scheme assumes that each user variable is kept in a single virtual register throughout the entire program.

This results in each virtual register having a large live range, which is likely to cause clashes.

We may remedy this situation by performing a transformation called live range splitting, in which live ranges are made smaller by using a different virtual register to store a variable’s value at different times, thus reducing the potential for clashes.

```c
extern int f(int);
extern void h(int,int);
void g()
{
  int a,b,c;
  a = f(1); b = f(2); h(a,b);
  b = f(3); c = f(4); h(b,c);
  c = f(5); a = f(6); h(c,a);
}
```

```c
extern int f(int);
extern void h(int,int);
void g()
{
  int a1,a2, b1,b2, c1,c2;
  a1 = f(1); b2 = f(2); h(a1,b2);
  b1 = f(3); c2 = f(4); h(b1,c2);
  c1 = f(5); a2 = f(6); h(c1,a2);
}
```
Static single-assignment

The ϕ-functions in SSA keep track of which variables are merged at control-flow join points. They are not executable since they do not record which variable to choose (cf. gated SSA form).

It is straightforward to transform straight-line code into SSA form: each variable is renamed by being given a subscript, which is incremented every time that variable is assigned to.

v₁ = 3;
v₂ = v₁ + 1;
v₃ = v₂ + w;
w₂ = v₃ + 2;

Static single-assignment

When the program’s control flow is more complex, extra effort is required to retain the original data-flow behaviour.

Where control-flow edges meet, two (or more) differently-named variables must now be merged together.

v₁ = 3;
v₂ = v₁ + 1;
v₃ = v₂ + w;
w₂ = v₃ + 2;

Static single-assignment

“Slight lie”: SSA is useful for much more than register allocation!

In fact, the main advantage of SSA form is that, by representing data dependencies as precisely as possible, it makes many optimising transformations simpler and more effective, e.g., constant propagation, loop-invariant code motion, partial-redundancy elimination, and strength reduction.

The ϕ-functions in SSA keep track of which variables are merged at control-flow join points.

They are not executable since they do not record which variable to choose (cf. gated SSA form).

Live ranges

Live range splitting is a useful transformation: it gives the same benefits for user variables as normal form gives for temporary variables.

However, if each virtual register is only ever assigned to once (statically), we needn’t perform live range splitting, since the live ranges are already as small as possible.

Code in static single-assignment (SSA) form has this important property.
Phase ordering

We now have many optimisations which we can perform on intermediate code.

It is generally a difficult problem to decide in which order to perform these optimisations; different orders may be more appropriate for different programs.

Certain optimisations are antagonistic: for example, CSE may superficially improve a program at the expense of making the register allocation phase more difficult (resulting in spills to memory).

Part B

Higher-level optimisations

- More modern optimisations than those in Part A
- Part A was mostly imperative
- Part B is mostly functional
- Now operating on syntax of source language vs. an intermediate representation
- Functional languages make the presentation clearer, but many optimisations will also be applicable to imperative programs

Higher-level optimisations

- More modern optimisations than those in Part A
- Part A was mostly imperative
- Part B is mostly functional
- Now operating on syntax of source language vs. an intermediate representation
- Functional languages make the presentation clearer, but many optimisations will also be applicable to imperative programs

Algebraic identities

The idea behind peephole optimisation of intermediate code can also be applied to abstract syntax trees.

There are many trivial examples where one piece of syntax is always (algebraically) equivalent to another piece of syntax which may be smaller or otherwise “better”; simple rewriting of syntax trees with these rules may yield a smaller or faster program.

Algebraic identities

These optimisations are boring, however, since they are always applicable to any syntax tree.

We're interested in more powerful transformations which may only be applied when some analysis has confirmed that they are safe.

In a lazy functional language,

\[
\text{let } x = e \text{ in if } e' \text{ then } \ldots x \ldots \text{ else } e''
\]

\[
\text{if } e' \text{ then let } x = e \text{ in } \ldots x \ldots \text{ else } e''
\]

provided \(e'\) and \(e''\) do not contain \(x\).

This is still quite boring.
More interesting analyses (i.e. ones that aren’t purely syntactic) enable more interesting transformations.

Strength reduction is an optimisation which replaces expensive operations (e.g. multiplication and division) with less expensive ones (e.g. addition and subtraction).

It is most interesting and useful when done inside loops.

For example, it may be advantageous to replace multiplication (2*e) with addition (let x = e in x + x) as before.

Multiplication may happen a lot inside loops (e.g. using the loop variable as an index into an array), so if we can spot a recurring multiplication and replace it with an addition we should get a faster program.

```c
int i;
for (i = 0; i < 100; i++)
{
    v[i] = 0;
}
```

```c
int i; char *p;
for (i = 0; i < 100; i++)
{
    p = (char *)v + 4*i;
    p[0] = 0; p[1] = 0;
}
```
Strength reduction

```
int i; int *p;
p = v;
for (i = 0; p < v + 100; i++)
{  
  *p = 0;
  p++;
}
```

Muliplication has been replaced with addition.

Strength reduction

Note that, while this code is now almost optimal, it has obfuscated the intent of the original program.

Don’t be tempted to write code like this!

For example, when targeting a 64-bit architecture, the compiler may be able to transform the original loop into fifty 64-bit stores, but will have trouble with our more efficient version.

Strength reduction

It might be easier to perform strength reduction on the intermediate code, but only if annotations have been placed on the flowchart to indicate loop structure.

At the syntax tree level, all loop structure is apparent.

Strength reduction

We are not restricted to replacing multiplication with addition, as long as we have

- induction variable: $i = i \oplus c$
- another variable: $j = c_2 \otimes (c_1 \otimes i)$

for some operations $\oplus$ and $\otimes$ such that $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$

Summary

- Live range splitting reduces register pressure
- In SSA form, each variable is assigned to only once
- SSA uses $\phi$-functions to handle control-flow merges
- SSA aids register allocation and many optimisations
- Optimal ordering of compiler phases is difficult
- Algebraic identities enable code improvements
- Strength reduction uses them to improve loops

Motivation

We reason about programs statically, but we are really trying to make predictions about their dynamic behaviour.

Why not examine this behaviour directly?

It isn’t generally feasible (e.g. termination, inputs) to run an entire computation at compile-time, but we can find things out about it by running a simplified version.

This is the basic idea of abstract interpretation.
Abstract interpretation

The key idea is to use an abstraction: a model of (otherwise unmanageable) reality, which
• discards enough detail that the model becomes manageable (e.g. small enough, computable enough), but
• retains enough detail to provide useful insight into the real world.

Try to plan a journey by exploring the concrete world instead of the abstraction (i.e. driving around aimlessly) is either very expensive or virtually impossible.

A trustworthy map makes it possible — even easy.

This is a real application of abstract interpretation, but in this course we’re more interested in computer programs.

### Multiplying integers

A canonical example is the multiplication of integers.

If we want to know whether \(-1515 \times 37\) is positive or negative, there are two ways to find out:

• Compute in the concrete world (arithmetic), using the standard interpretation of multiplication. \(-1515 \times 37 = -56055\), which is negative.
• Compute in an abstract world, using an abstract interpretation of multiplication: call it \(\otimes\).

\[
\begin{align*}
\otimes & \quad (-) \quad (0) \quad (+) \\
(-) & \quad (+) \quad (0) \quad (-) \\
(0) & \quad (0) \quad (0) \quad (0) \\
(+)& \quad (-) \quad (0) \quad (+)
\end{align*}
\]

We need to define the abstract operator \(\otimes\). Luckily, we have been to primary school.

\[
\begin{align*}
(-) &= \{ z \in \mathbb{Z} \mid z < 0 \} \\
(0) &= \{ 0 \} \\
(+) &= \{ z \in \mathbb{Z} \mid z > 0 \}
\end{align*}
\]

In the concrete world we have all the integers; in the abstract world we have only the values \((-), (0)\) and \((+).
Multiplying integers

Armed with our abstraction, we can now tackle the original problem.

\[ \text{abs}(-1515) = (-) \]
\[ \text{abs}(37) = (+) \]
\[ (-) \otimes (+) = (-) \]

So, without doing any concrete computation, we have discovered that \(-1515 \times 37\) has a negative result.

Safety

As always, there are important safety issues.

Because an abstraction discards detail, a computation in the abstract world will necessarily produce less precise results than its concrete counterpart.

It is important to ensure that this imprecision is safe.

Adding integers

A good example is the addition of integers. How do we define the abstract operator \(\oplus\)?

<table>
<thead>
<tr>
<th></th>
<th>(-)</th>
<th>(0)</th>
<th>(+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-)</td>
<td>(-)</td>
<td>(-)</td>
<td>(?)</td>
</tr>
<tr>
<td>(0)</td>
<td>(-)</td>
<td>(0)</td>
<td>(+)</td>
</tr>
<tr>
<td>(+)</td>
<td>(?)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
</tbody>
</table>

Adding integers

(?) is less precise than (-), (0) and (+); it means “I don’t know”, or “it could be anything”.

Because we want the abstraction to be safe, we must put up with this weakness.

Adding integers

\[ \text{abs}(19 + 23) = \text{abs}(42) = (+) \]
\[ \text{abs}(19) \oplus \text{abs}(23) = (+) \otimes (+) = (+) \]

Adding integers

This is just a toy example, but it demonstrates the methodology: state a problem, devise an abstraction that retains the characteristics of that problem, solve the problem in the abstract world, and then interpret the solution back in the concrete world.

This abstraction has avoided doing arithmetic; in compilers, we will mostly be interested in avoiding expensive computation, nontermination or undecidability.

Adding integers

When adding integers, their (relative) magnitudes are important in determining the sign of the result, but our abstraction has discarded this information.

As a result, we need a new abstract value: (?)

\[ (-) = \{ z \in \mathbb{Z} \mid z < 0 \} \]
\[ (0) = \{ 0 \} \]
\[ (+) = \{ z \in \mathbb{Z} \mid z > 0 \} \]
\[ (?) = \mathbb{Z} \]
Adding integers

\[ \text{abs}(-1515 + 37) = \text{abs}(-1478) = (-) \]
\[ \text{abs}(-1515) \odot \text{abs}(37) = (-) \odot (+) = (?) \]

\[ \begin{array}{ccc}
-1515, 37 & \rightarrow & -1478 \\
\text{abs} & \downarrow & \text{abs} \\
(-), (+) & \odot & (?)
\end{array} \]

Safety

Here, safety is represented by the fact that \((-) \subseteq (?)\):

\[ \{ z \in \mathbb{Z} \mid z < 0 \} \subseteq \mathbb{Z} \]

The result of doing the abstract computation is less precise, but crucially includes the result of doing the concrete computation (and then abstracting), so the abstraction is safe and hasn't missed anything.

Abstraction

Formally, an abstraction of some concrete domain \( D \) (e.g. \( \wp(\mathbb{Z}) \)) consists of

- an abstract domain \( D^\# \) (e.g. \{ \((-), (0), (+), (?) \} ),
- an abstraction function \( \alpha : D \rightarrow D^\# \) (e.g. \text{abs}), and
- a concretisation function \( \gamma : D^\# \rightarrow D \), e.g.:
  - \((-) \rightarrow \{ z \in \mathbb{Z} \mid z < 0 \},
  - (0) \rightarrow \{ 0 \}, \text{etc.} \)

Abstraction

Given a function \( f \) from one concrete domain to another (e.g. \( \hat{\ast} : \wp(\mathbb{Z}) \times \wp(\mathbb{Z}) \rightarrow \wp(\mathbb{Z}) \)), we require an abstract function \( f^\# \) (e.g. \( \odot \)) between the corresponding abstract domains.

\[ \hat{\ast}(\{ 2, 5 \}, \{ -3, -7 \}) = \{ -1, -5, 2, -2 \} \]
\[ \odot((+), (-)) = (?) \]

Abstraction

These mathematical details are formally important, but are not examinable on this course.

Recognise that this is what we were doing in data-flow analysis: interpreting 3-address instructions as operations on abstract values — e.g. live variable sets — and then “executing” this abstract program.

Summary

- Abstractions are manageably simple models of unmanageably complex reality
- Abstract interpretation is a general technique for executing simplified versions of computations
- For example, the sign of an arithmetic result can be sometimes determined without doing any arithmetic
- Abstractions are approximate, but must be safe
- Data-flow analysis is a form of abstract interpretation
Lecture 10
Strictness analysis

Motivation
The operations and control structures of imperative languages are strongly influenced by the way most real computer hardware works.

This makes imperative languages relatively easy to compile, but (arguably) less expressive; many people use functional languages, but these are harder to compile into efficient imperative machine code.

Strictness optimisation can help to improve the efficiency of compiled functional code.

Call-by-value evaluation

Strict (“eager”) functional languages (e.g. ML) use a call-by-value evaluation strategy:

\[
\frac{e_2 \Downarrow v_2 \quad e_1[v_2/x] \Downarrow v_1}{(\lambda x. e_1) \ e_2 \Downarrow v_1}
\]

- Efficient in space and time, but
- might evaluate more arguments than necessary.

Call-by-name evaluation

Non-strict (“lazy”) functional languages (e.g. Haskell) use a call-by-name evaluation strategy:

\[
\frac{e_1[e_2/x] \Downarrow v}{(\lambda x. e_1) \ e_2 \Downarrow v}
\]

- Only evaluates arguments when necessary, but
- copies (and redundantly re-evaluates) arguments.

Call-by-need evaluation

One simple optimisation is to use call-by-need evaluation instead of call-by-name.

If the language has no side-effects, duplicated instances of an argument can be shared, evaluated once if required, and the resulting value reused.

This avoids recomputation and is better than call-by-name, but is still more expensive than call-by-value.

Call-by-need evaluation

plus(x,y) = if x=0 then y else plus(x-1,y+1)

Using call-by-value:

plus(3,4) \Rightarrow if 3=0 then 4 else plus(3-1,4+1)

\Rightarrow plus(2,5)
\Rightarrow plus(1,6)
\Rightarrow plus(0,7)
\Rightarrow 7

Using call-by-need:

plus(3,4) \Rightarrow if 3=0 then 4 else plus(3-1,4+1)

\Rightarrow plus(2-1,4+1+1)
\Rightarrow plus(1-1,4+1+1+1)
\Rightarrow 4+1+1+1
\Rightarrow 5+1+1
\Rightarrow 6+1
\Rightarrow 7

Replacing CBN with CBV

So why not just replace call-by-name with call-by-value?

Because, while replacing call-by-name with call-by-need never changes the semantics of the original program (in the absence of side-effects), replacing CBN with CBV does.

In particular, the program’s termination behaviour changes.
Replacing CBN with CBV

Assume we have some nonterminating expression, Ω.

• Using CBN, the expression \((λx. 42) \ Ω\) will evaluate to 42.
• But using CBV, evaluation of \((λx. 42) \ Ω\) will not terminate: \(Ω\) gets evaluated first, even though its value is not needed here.

We should therefore try to use call-by-value wherever possible, but not when it will affect the termination behaviour of a program.

Neededness

Intuitively, it will be safe to use CBV in place of CBN whenever an argument is definitely going to be evaluated.

We say that an argument is needed by a function if the function will always evaluate it.

• \(λx,y. x+y\) needs both its arguments.
• \(λx,y. x+1\) needs only its first argument.
• \(λx,y. 42\) needs neither of its arguments.

These needed arguments can safely be passed by value: if their evaluation causes nontermination, this will just happen sooner rather than later.

In fact, neededness is too conservative:

\(λx,y,z. \text{if } x \text{ then } y \text{ else } Ω\)

This function might not evaluate \(y\), so only \(x\) is needed.

But actually it’s still safe to pass \(y\) by value: if \(y\) doesn’t get evaluated then the function doesn’t terminate anyway, so it doesn’t matter if eagerly evaluating \(y\) causes nontermination.

Strictness

What we really want is a more refined notion:

It is safe to pass an argument by value when the function fails to terminate whenever the argument fails to terminate.

When this more general statement holds, we say the function is strict in that argument.

\(λx,y,z. \text{if } x \text{ then } y \text{ else } Ω\)

is strict in \(x\) and strict in \(y\).

If we can develop an analysis that discovers which functions are strict in which arguments, we can use that information to selectively replace CBN with CBV and obtain a more efficient program.

Strictness analysis

We can perform strictness analysis by abstract interpretation.

First, we must define a concrete world of programs and values.

We will use the simple language of recursion equations, and only consider integer values.

Recursion equations

\[
F_1(x_1, \ldots, x_{k_1}) = e_1 \\
\cdots = \cdots \\
F_n(x_1, \ldots, x_{k_n}) = e_n
\]

\[e ::= x_i | A_i(e_1, \ldots, e_{r_i}) | F_i(e_1, \ldots e_{k_i})\]

where each \(A_i\) is a symbol representing a built-in (predefined) function of arity \(n\).
Recursion equations

For our earlier example,

\[ \text{plus}(x, y) = \begin{cases} y & \text{if } x = 0 \\ \text{plus}(x-1, y+1) & \text{else} \end{cases} \]

we can write the recursion equation

\[ \text{plus}(x, y) = \text{cond}(\text{eq}(x, 0), y, \text{plus}(\text{sub1}(x), \text{add1}(y))) \]

where \text{cond}, \text{eq}, \text{sub1} and \text{add1} are built-in functions.

Standard interpretation

Each built-in function needs a standard interpretation.

We will interpret each \( A_i \) as a function \( a_i \) on values in \( D \):

\[
\begin{align*}
\text{cond}(\perp, x, y) & = \perp \\
\text{cond}(0, x, y) & = y \\
\text{cond}(p, x, y) & = x \text{ otherwise}
\end{align*}
\]

\[
\begin{align*}
\text{eq}(\perp, y) & = \perp \\
\text{eq}(x, \perp) & = \perp \\
\text{eq}(x, y) & = x =_Z y \text{ otherwise}
\end{align*}
\]

Abstract interpretation

Our abstraction must capture the properties we’re interested in, while discarding enough detail to make analysis computationally feasible.

Strictness is all about termination behaviour; and in fact this is all we care about: does evaluation of an expression definitely not terminate (as with \( \Omega \)), or may it eventually terminate and return a result?

Our abstract domain \( D^\# \) is therefore \{ 0, 1 \}.

Abstract interpretation

A formal relationship exists between the standard and abstract interpretations of each built-in function; the mathematical details are in the lecture notes.

Informally, we use the same technique as for multiplication and addition of integers in the last lecture: we define the abstract operations using what we know about the behaviour of the concrete operations.

Standard interpretation

We must have some representation of nontermination in our concrete domain.

As values we will consider the integer results of terminating computations, \( \mathbb{Z} \), and a single extra value to represent nonterminating computations: \( \perp \).

Our concrete domain \( D \) is therefore \( \mathbb{Z}_\perp = \mathbb{Z} \cup \{ \perp \} \).

Abstract interpretation

For each built-in function \( A_i \) we need a corresponding strictness function \( a_i^\# \) — this provides the strictness interpretation for \( A_i \).

Whereas the standard interpretation of each built-in is a function on concrete values from \( D \), the strictness interpretation of each will be a function on abstract values from \( D^\# \) (i.e. 0 and 1).

Abstract interpretation

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( \text{eq}^#(x,y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Abstract interpretation

<table>
<thead>
<tr>
<th>$p$</th>
<th>$x$</th>
<th>$y$</th>
<th>$\text{cond}^#(p,x,y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Abstract interpretation

\[
\text{cond}^\#(p, x, y) = (p \land y) \lor (p \land x)
\]

Abstract interpretation

\[
\text{eq}^\#(x, y) = x \land y
\]

Abstract interpretation

\[
\text{plus}^\#(x, y) = \text{cond}^\#(\text{eq}^\#(x, 0\#), y, \text{plus}^\#(\text{sub}1^\#(x), \text{add}1^\#(y)))
\]

Strictness analysis

So far, we have set up

- a concrete domain, $D$, equipped with
  - a standard interpretation $a_i$ of each built-in $A_i$, and
  - a standard interpretation $f_i$ of each user-defined $F_i$;
- and an abstract domain, $D^\#$, equipped with
  - an abstract interpretation $a_i^\#$ of each built-in $A_i$.

But recall that the recursion equations show us how to build up each user-defined function, by composition and recursion, from all the built-in functions:

\[
\text{plus}^\#(x, y) = \text{cond}^\#(\text{eq}^\#(x, 0\#), y, \text{plus}^\#(\text{sub}1^\#(x), \text{add}1^\#(y)))
\]

So we can build up the $f_i^\#$ from the $a_i^\#$ in the same way:

\[
\text{plus}^\#(x, y) = \text{cond}^\#(\text{eq}^\#(x, 0\#), y, \text{plus}^\#(\text{sub}1^\#(x), \text{add}1^\#(y)))
\]

Strictness analysis

The point of this analysis is to discover the missing piece: what is the strictness function $f_i^\#$ corresponding to each user-defined $F_i$?

These strictness functions will show us exactly how each $F_i$ is strict with respect to each of its arguments — and that’s the information that tells us where we can replace lazy, CBN-style parameter passing with eager CBV.

We already know all the other strictness functions:

\[
\begin{align*}
\text{cond}^\#(p, x, y) &= p \land (x \lor y) \\
\text{eq}^\#(x, y) &= x \land y \\
0^\# &= 1 \\
\text{sub}1^\#(x) &= x \\
\text{add}1^\#(x) &= x
\end{align*}
\]

So we can use these to simplify the expression for $\text{plus}^\#$.
Algorithm

\[
\text{for } i = 1 \text{ to } n \text{ do } f^\#[i] := \lambda x,y. 0
\]
while (f\#[] changes) do
  \[
  \text{for } i = 1 \text{ to } n \text{ do }
  \]
  \[
  f^\#[i] := \lambda x. e^\#
  \]
\]
e^\# means “e (from the recursion equations) with each } A_j \text{ replaced with } a^\# \text{ and each } F_j \text{ replaced with } f^\#[j].”

Algorithm

On the first iteration, we calculate \( e_1^\# \):

• The recursion equations say
  \[
  e_1 = \text{cond}(\text{eq}(x, 0), y, \text{plus}(\text{sub}l(x), \text{add}l(y)))
  \]
• The current contents of \( f^\#[] \) say \( f_1^\# \) is \( \lambda x,y. 0 \).
• So:
  \[
  e_1^\# = \text{cond}(\text{eq}^\#(x, 0^\#), y, (\lambda x,y. 0^\#) (\text{sub}l^\#(x), \text{add}l^\#(y)))
  \]

Algorithm

On the second iteration, we recalculate \( e_1^\# \):

• The recursion equations still say
  \[
  e_1 = \text{cond}(\text{eq}(x, 0), y, \text{plus}(\text{sub}l(x), \text{add}l(y)))
  \]
• The current contents of \( f^\#[] \) say \( f_1^\# \) is \( \lambda x,y. x \land y \)
• So:
  \[
  e_1^\# = \text{cond}(\text{eq}^\#(x, 0^\#), y, (\lambda x,y. x \land y) (\text{sub}l^\#(x), \text{add}l^\#(y)))
  \]

\[
\text{plus}^\#(x, y) = x \land (y \lor \text{plus}^\#(x, y))
\]

This is a recursive definition, and so unfortunately doesn’t provide us with the strictness function directly.

We want a definition of \( \text{plus}^\# \) which satisfies this equation — actually we want the least fixed point of this equation, which (as ever!) we can compute iteratively.

\[
\text{for } i = 1 \text{ to } n \text{ do } f^\#[i] := \lambda x.0
\]
while (f\#[ ] changes) do
  \[
  \text{for } i = 1 \text{ to } n \text{ do }
  \]
  \[
  f^\#[i] := \lambda x. e^\#
  \]
\]
e^\# means “e (from the recursion equations) with each } A_j \text{ replaced with } a^\# \text{ and each } F_j \text{ replaced with } f^\#[j].”

Algorithm

We have only one user-defined function, \( \text{plus} \), and so only one recursion equation:
\[
\text{plus}(x, y) = \text{cond}(\text{eq}(x, 0), y, \text{plus}(\text{sub}l(x), \text{add}l(y)))
\]

We initialise the corresponding element of our \( f^\#[] \) array to contain the always-0 strictness function of the appropriate arity:
\[
\begin{align*}
  f^\#[1] & := \lambda x,y. 0
\end{align*}
\]

Algorithm

\[
\begin{align*}
  e_1^\# & =\text{cond}(\text{eq}^\#(x, 0^\#), y, (\lambda x,y. 0^\#) (\text{sub}l^\#(x), \text{add}l^\#(y)))
  \end{align*}
\]

Simplifying:
\[
\begin{align*}
  e_1^\# & = \text{cond}^\#(\text{eq}^\#(x, 0^\#), y, 0^\#)
  \end{align*}
\]

Using definitions of \( \text{cond}^\#, \text{eq}^\# \) and \( 0^\# \):
\[
\begin{align*}
  e_1^\# & = (x \land 1^\#) \land (y \lor 0^\#)
  \end{align*}
\]

Simplifying again:
\[
\begin{align*}
  e_1^\# & = x \land y
  \end{align*}
\]
Algorithm
\[ e_1^# = \text{cond}(\text{eq}(x, 0^#), \lambda x. x \land y) \ (\text{sub}_1^#(x), \text{add}_1^#(y))) \]

Simplifying:
\[ e_1^# = \text{cond}(\text{eq}(x, 0^#), \text{sub}_1^#(x) \land \text{add}_1^#(y)) \]

Using definitions of \text{cond}^#, \text{eq}^#, 0^#, \text{sub}_1^# and \text{add}_1^#:
\[ e_1^# = (x \land 1) \land (y \lor (x \land y)) \]

Simplifying again:
\[ e_1^# = x \land y \]

Algorithm
\[ \text{plus}^#(x, y) = x \land y \]

Optimisation
So, at the end of the second iteration,
\[ f^#[1] := \lambda x, y. x \land y \]

This is the same result as last time, so we stop.

Summary
- Functional languages can use CBV or CBN evaluation
- CBV is more efficient but can only be used in place of CBN if termination behaviour is unaffected
- Strictness shows dependencies of termination
- Abstract interpretation may be used to perform strictness analysis of user-defined functions
- The resulting strictness functions tell us when it is safe to use CBV in place of CBN

Motivation
Intra-procedural analysis depends upon accurate control-flow information.

In the presence of certain language features (e.g. indirect calls) it is nontrivial to predict accurately how control may flow at execution time — the naive strategy is very imprecise.

A constraint-based analysis called 0CFA can compute a more precise estimate of this information.
Constraint-based analysis

Many of the analyses in this course can be thought of in terms of solving systems of constraints.

For example, in LVA, we generate equality constraints from each instruction in the program:

\[
\text{in-live}(m) = (\text{out-live}(m) \setminus \text{def}(m)) \cup \text{ref}(m)
\]
\[
\text{out-live}(m) = \text{in-live}(n) \cup \text{in-live}(o)
\]
\[
\text{in-live}(n) = (\text{out-live}(n) \setminus \text{def}(n)) \cup \text{ref}(n)
\]
⋮
and then iteratively compute their minimal solution.

0CFA

0CFA — “zeroth-order control-flow analysis” — is a constraint-based analysis for discovering which values may reach different places in a program.

When functions (or pointers to functions) are present, this provides information about which functions may potentially be called at each call site.

We can then build a more precise call graph.

Specimen language

Functional languages are a good candidate for this kind of analysis; they have functions as first-class values, so control flow may be complex.

We will use a minimal syntax for expressions:

\[
e ::= x \mid c \mid \lambda x. e \mid e_1 e_2 \mid \text{let } x = e_1 \text{ in } e_2
\]

A program in this language is a closed expression.

Specimen program

let id = \lambda x. x in id id 7

let id2 = (\lambda x. x^5)^3 in ((id^8 id^9)^7 7^{10})^6

Program points

(let id^2 = (\lambda x. x^5)^3 in ((id^8 id^9)^7 7^{10})^6)

Each program point \( i \) has an associated flow variable \( \alpha_i \).

Each \( \alpha_i \) represents the set of flow values which may be yielded at program point \( i \) during execution.

For this language the flow values are integers and function closures; in this particular program, the only values available are \( 7^{10} \) and \( (\lambda x. x^5)^3 \).

The precise value of each \( \alpha_i \) is undecidable in general, so our analysis will compute a safe overapproximation.

From the structure of the program we can generate a set of constraints on the flow variables, which we can then treat as data-flow inequations and iteratively compute their least solution.
Using solutions

\[ \alpha_{10} = \{ 7^{10} \} \]

\[ \alpha_2, \alpha_3, \alpha_8, \alpha_9 = \{ (\lambda x^4. x^5)^3 \} \]

\[ \alpha_1, \alpha_4, \alpha_5, \alpha_6, \alpha_7 = \{ (\lambda x^4. x^5)^3, 7^{10} \} \]

(let \( id^2 = (\lambda x^4. x^5)^3 \) in \((id^8 \ id^9) \ 7 \ 7^{10} \ 6\) )

Limitations

0CFA is still imprecise because it is monovariant: each expression has only one flow variable associated with it, so multiple calls to the same function allow multiple values into the single flow variable for the function body, and these values "leak out" at all potential call sites.
Limitations

\[
\begin{align*}
\alpha_7 &= \{ (\lambda x^4. x^5)^3 \} \\
\alpha_9 &= \{ (\lambda x^4. x^5)^3 \} \\
\alpha_{10} &= \{ 7^{10} \}
\end{align*}
\]

\[
\begin{align*}
(\alpha_10 \mapsto \alpha_6) &\supseteq \alpha_7 \\
(\alpha_9 \mapsto \alpha_7) &\supseteq \alpha_8 \\
\alpha_1 &\supseteq \alpha_6 \\
\alpha_2 &\supseteq \alpha_3 \\
\alpha_5 &\supseteq \alpha_4 \\
\alpha_8 &\supseteq \alpha_2 \\
\alpha_9 &\supseteq \alpha_2 \\
\alpha_{10} &\supseteq \{ \ldots \alpha_5 \alpha_4 \supseteq \alpha_{10} \alpha_6 \supseteq \alpha_5 \}
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{id}^2 &= (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{id}^9)7^{10})^6^1
\end{align*}
\]

Summary

- Many analyses can be formulated using constraints
- 0CFA is a constraint-based analysis
- Inequality constraints are generated from the syntax of a program
- A minimal solution to the constraints provides a safe approximation to dynamic control-flow behaviour
- Polyvariant (as in 1CFA) and polymorphic approaches may improve precision

Motivation

In this part of the course we’re examining several methods of higher-level program analysis.

We have so far seen abstract interpretation and constraint-based analysis, two general frameworks for formally specifying (and performing) analyses of programs.

Another alternative framework is inference-based analysis.

Inference-based analysis

Inference systems consist of sets of rules for determining program properties.

Typically such a property of an entire program depends recursively upon the properties of the program’s subexpressions; inference systems can directly express this relationship, and show how to recursively compute the property.

Inference-based analysis

An inference system specifies judgements:

\[
\Gamma \vdash e : \phi
\]

- \(e\) is an expression (e.g. a complete program)
- \(\Gamma\) is a set of assumptions about free variables of \(e\)
- \(\phi\) is a program property
Optimisation

In the absence of a compile-time type checker, all values must be tagged with their types and run-time checks must be performed to ensure types match appropriately.

If a type system has shown that the program is well-typed, execution can proceed safely without these tags and checks; if necessary, the final result of evaluation can be tagged with its inferred type.

Hence the final result of evaluation is identical, but less run-time computation is required to produce it.

Type systems

Consider the ML type system, for example.

This particular inference system specifies judgements about a well-typedness property:

\[ \Gamma \vdash e : t \]

means “under the assumptions in \( \Gamma \), the expression \( e \) has type \( t \)”.

Type systems

\( \Gamma \) is a set of type **assumptions** of the form

\[ \{ x_1 : t_1, ..., x_n : t_n \} \]

where each identifier \( x_i \) is assumed to have type \( t_i \).

We write

\[ \Gamma[x : t] \]

to mean \( \Gamma \) with the additional assumption that \( x \) has type \( t \) (overriding any other assumption about \( x \)).

Type systems

\[ \Gamma[x : t] \vdash x : t \] \hspace{1cm} (VAR)

\[ \Gamma[x : t] \vdash e : t' \] \hspace{1cm} (LAM)

\[ \Gamma \vdash \lambda x.e : t \rightarrow t' \]

\[ \Gamma \vdash e_1 : t \rightarrow t' \hspace{0.5cm} \Gamma \vdash e_2 : t \]

\[ \Gamma \vdash e_1 e_2 : t' \] \hspace{1cm} (APP)

Safety

The safety condition for this inference system is

\[ (\{\} \vdash e : t) \Rightarrow ([e] \in [t]) \]

where \([e]\) and \([t]\) are the **denotations** of \( e \) and \( t \) respectively; \([e]\) is the value obtained by evaluating \( e \), and \([t]\) is the set of all values of type \( t \).

This condition asserts that the run-time behaviour of the program will agree with the type system's prediction.
Type-checking is just one application of inference-based program analysis.

The properties do not have to be types; in particular, they can carry more (or completely different!) information than traditional types do.

We'll consider a more program-analysis–related example: detecting odd and even numbers.

Odds and evens

Type systems are the best-known example where properties give more detailed information than traditional types do.

Type-checking is just one application of inference-based program analysis.

This time, the program property \( \phi \) has the form

\[
\phi ::= \text{odd} \mid \text{even} \mid \phi_1 \rightarrow \phi_2
\]

The safety condition for this inference system is

\[
\text{odd} \in \{ z \in \mathbb{Z} \mid z \text{ is even} \}, \quad \text{even} \in \{ z \in \mathbb{Z} \mid z \text{ is odd} \},
\]

We need more than one assumption about \( \text{multiply} \):

\[
\Gamma = \{ ..., \text{multiply} : \text{even} \rightarrow \text{even} \rightarrow \text{even}, \quad \text{multiply} : \text{odd} \rightarrow \text{even} \rightarrow \text{even}, ... \}
\]

Summary

- Inference-based analysis is another useful framework
- Inference rules are used to produce judgements about programs and their properties
- Type systems are the best-known example
- Richer properties give more detailed information
- An inference system used for analysis has an associated safety condition
Lecture 13
Effect systems

Motivation
We have so far seen many analyses which deal with control- and data-flow properties of pure languages. However, many languages contain operations with side-effects, so we must also be able to analyse and safely transform these impure programs. Effect systems, a form of inference-based analysis, are often used for this purpose.

Side-effects
A side-effect is some event — typically a change of state — which occurs as a result of evaluating an expression.

• "x++" changes the value of variable x.
• "malloc(42)" allocates some memory.
• "print 42" outputs a value to a stream.

As an example language, we will use the lambda calculus extended with read and write operations on “channels”.

\[ e ::= x | \lambda x . e | e_1 e_2 | \xi ? x . e | \xi ! e_1 . e_2 \]

• \( \xi \) represents some channel name.
• \( \xi ? x . e \) reads an integer from the channel named \( \xi \), binds it to \( x \), and returns the result of evaluating \( e \).
• \( \xi ! e_1 . e_2 \) evaluates \( e_1 \), writes the resulting integer to channel \( \xi \), and returns the result of evaluating \( e_2 \).

Some example expressions:

\[ \xi ? x . x \]  
read an integer from channel \( \xi \) and return it

\[ \xi ! x . y \]  
write the (integer) value of \( x \) to channel \( \xi \) and return the value of \( y \)

\[ \xi ? x . \xi ! x . x \]  
read an integer from channel \( \xi \), write it to channel \( \xi \), and return it

Side-effects

\[ \Gamma [x : \text{int}] \vdash e : t \]
\[ \Gamma \vdash \xi ? x . e : t \]  (READ)

\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : t \]
\[ \Gamma \vdash \xi ! e_1 . e_2 : t \]  (WRITE)

Effect systems
However, in order to perform any transformations on a program in this language it would be necessary to pay attention to its potential side-effects.

For example, we might need to devise an analysis to tell us which channels may be read or written during evaluation of an expression.

We can do this by modifying our existing type system to create an effect system (or “type and effect system”).
Effect systems

First we must formally define our effects:

An expression has effects $F$. $F$ is a set containing elements of the form:

- $\text{R}_\xi$ read from channel $\xi$
- $\text{W}_\xi$ write to channel $\xi$

Effect systems

But we also need to be able to handle expressions like

$$\lambda x. \xi!x. x$$

whose evaluation doesn't have any immediate effects.

In this case, the effect $\text{W}_\xi$ may occur later, whenever this newly-created function is applied.

Effect systems

So, although it has no immediate effects, the type of

$$\lambda x. \xi!x. x$$

is

$$\text{int} \rightarrow \text{int} \left\{ \text{W}_\xi \right\}$$

Effect systems

To handle these latent effects we extend the syntax of types so that function types are annotated with the effects that may occur when a function is applied:

$$t ::= \text{int} | t_1 \rightarrow t_2$$

Effect systems

We can now modify the existing type system to make an effect system — an inference system which produces judgements about the type and effects of an expression:

$$\Gamma \vdash e : t, F$$

Effect systems

The rules for type inference are:

- (READ)
  $$\frac{ \Gamma[x : \text{int}] \vdash e : t }{ \Gamma \vdash \xi?x.x : t }$$

- (WRITE)
  $$\frac{ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : t }{ \Gamma \vdash \xi!e_1.e_2 : t }$$

- (READ)
  $$\frac{ \Gamma \vdash e_1 : \text{int}, F \quad \Gamma \vdash e_2 : t, F' }{ \Gamma \vdash \xi!e_1.e_2 : t, F \cup \{ \text{W}_\xi \} \cup F' }$$

Effect systems

For example:

- $\xi?x. x \quad F = \{ \text{R}_\xi \}$
- $\xi!x. y \quad F = \{ \text{W}_\xi \}$
- $\xi?x. \xi!x. x \quad F = \{ \text{R}_\xi, \text{W}_\xi \}$
Effect systems

\[ \Gamma[x : t] \vdash x : t, \{ \} \quad \text{(VAR)} \]

\[ \Gamma[x : t] \vdash e : t', F \]

\[ \Gamma \vdash \lambda x.e : t \xrightarrow{F} t', \{ \} \quad \text{(LAM)} \]

\[ \Gamma \vdash e_1 : t \xrightarrow{F''} t', F \quad \Gamma \vdash e_2 : t, F' \]

\[ \Gamma \vdash e_1 e_2 : t', F \cup F' \cup F'' \quad \text{(APP)} \]

Effect subtyping

We would probably want more expressive control structure in a real programming language.

For example, we could add if-then-else:

\[ e ::= x \mid \lambda x.e \mid e_1 e_2 \mid \xi!x.e \mid \xi?e_1.e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

However, there are some valid uses of if-then-else which this rule cannot handle by itself.

\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : t, F' \quad \Gamma \vdash e_3 : t, F'' \quad \text{(COND)} \]

\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t, F \cup F' \cup F'' \]

Effect subtyping

We can solve this problem by adding a new rule to handle subtyping.

We would probably want more expressive control structure in a real programming language.

For example, we could add if-then-else.
If we use a different representation of effects, and use different operations on them, we can keep track of more information.

One option is to use sequences of effects and use an append operation when combining them.

The information discovered by the effect system is useful when deciding whether particular transformations are safe.

An expression with no immediate side-effects is referentially transparent: it can safely be replaced with another expression (with the same value and type) with no change to the semantics of the program.

For example, referentially transparent expressions may safely be removed if LVA says they are dead.

In this analysis we are using sets of effects.

As a result, we aren’t collecting any information about how many times each effect may occur, or the order in which they may happen.

If we use a different representation of effects, and use different operations on them, we can keep track of more information.

One option is to use sequences of effects and use an append operation when combining them.
In the new system, these expressions all have different effects:

\[ \xi?x. \, \zeta!x. \, x \quad F = \langle R_{\xi}; W_{\zeta} \rangle \]
\[ \zeta!y. \, \xi?x. \, x \quad F = \langle W_{\zeta}; R_{\xi} \rangle \]
\[ \zeta!y. \, \xi?x. \, \zeta!x. \, x \quad F = \langle W_{\zeta}; R_{\xi}; W_{\zeta} \rangle \]

Whether we use sequences instead of sets depends upon whether we care about the order and number of effects. In the channel example, we probably don’t. But if we were tracking file accesses, it would be important to ensure that no further read or write effects occurred after a file had been closed. And if we were tracking memory allocation, we would want to ensure that no block of memory got deallocated twice.

Summary

• Effect systems are a form of inference-based analysis
• Side-effects occur when expressions are evaluated
• Function types must be annotated to account for latent effects
• A type system can be modified to produce judgements about both types and effects
• Subtyping may be required to handle annotated types
• Different effect structures may give more information

Motivation

We have seen optimisation techniques which involve removing and reordering code at both the source- and intermediate-language levels in an attempt to achieve the smallest and fastest correct program.

These techniques are platform-independent, and pay little attention to the details of the target architecture.

We can improve target code if we consider the architectural characteristics of the target processor.

Part C

Instruction scheduling

In single-cycle processor designs, an entire instruction is executed in a single clock cycle.

Each instruction will use some of the processor’s processing stages:

<table>
<thead>
<tr>
<th>Instruction fetch (IF)</th>
<th>Register fetch (RF)</th>
<th>Execute (EX)</th>
<th>Memory access (MEM)</th>
<th>Register write-back (WB)</th>
</tr>
</thead>
</table>

For example, a load instruction uses all five.
Single-cycle implementation

On these processors, the order of instructions doesn’t make any difference to execution time: each instruction takes one clock cycle, so \( n \) instructions will take \( n \) cycles and can be executed in any (correct) order.

In this case we can naively translate our optimised 3-address code by expanding each intermediate instruction into the appropriate sequence of target instructions; clever reordering is unlikely to yield any benefits.

Pipelined implementation

In pipelined processor designs (e.g. MIPS R2000), each processing stage works independently and does its job in a single clock cycle, so different stages can be handling different instructions simultaneously.

These stages are arranged in a pipeline, and the result from each unit is passed to the next one via a pipeline register before the next clock cycle.

Pipelined implementation

In this multicycle design the clock cycle is much shorter (one pipeline stage vs. one complete instruction) and ideally we can still execute one instruction per cycle when the pipeline is full.

Programs will therefore execute more quickly.

Pipeline hazards

However, it is not always possible to run the pipeline at full capacity.

Some situations prevent the next instruction from executing in the next clock cycle: this is a pipeline hazard.

On interlocked hardware (e.g. SPARC) a hazard will cause a pipeline stall; on non-interlocked hardware (e.g. MIPS) the compiler must generate explicit NOPs to avoid errors.

Pipeline hazards

Consider data hazards: these occur when an instruction depends upon the result of an earlier one.

\[
\begin{align*}
\text{add } &\$3,\$1,\$2 \\
\text{add } &\$5,\$3,\$4
\end{align*}
\]

The pipeline must stall until the result of the first add has been written back into register \$3.
Pipeline hazards

The severity of this effect can be reduced by using *bypassing*: extra paths are added between functional units, allowing data to be used before it has been written back into registers.

```
add $3, $1, $2
add $5, $3, $4
```

Pipeline hazards

But even when bypassing is used, some combinations of instructions will always result in a stall.

```
lw $1, 0($0)
add $2, $2, $1
lw $3, 4($0)
add $4, $4, $3
```

Instruction order

Since particular combinations of instructions cause this problem on pipelined architectures, we can achieve better performance by reordering instructions where possible.

```
lw $1, 0($0)
add $2, $2, $1
lw $3, 4($0)
add $4, $4, $3
lw $1, 0($0)
add $2, $2, $1
lw $3, 4($0)
add $4, $4, $3
```

Instruction order

```
lw $1, 0($0)
add $2, $2, $1
lw $3, 4($0)
add $4, $4, $3
lw $1, 0($0)
lw $3, 4($0)
add $2, $2, $1
add $4, $4, $3
```

Instruction order

```
lw $1, 0($0)
lw $3, 4($0)
add $2, $2, $1
add $4, $4, $3
```

Instruction order

```
lw $1, 0($0)
lw $3, 4($0)
add $2, $2, $1
add $4, $4, $3
```

Instruction order

```
lw $1, 0($0)
lw $3, 4($0)
add $2, $2, $1
add $4, $4, $3
```

Instruction order

```
lw $1, 0($0)
lw $3, 4($0)
add $2, $2, $1
add $4, $4, $3
```
Instruction dependencies

We can only reorder target-code instructions if the meaning of the program is preserved.

We must therefore identify and respect the data dependencies which exist between instructions.

In particular, whenever an instruction is dependent upon an earlier one, the order of these two instructions must not be reversed.

There are three kinds of data dependency:

• Read after write
• Write after read
• Write after write

Whenever one of these dependencies exists between two instructions, we cannot safely permute them.

Instruction scheduling

We would like to reorder the instructions within each basic block in a way which

• preserves the dependencies between those instructions (and hence the correctness of the program), and
• achieves the minimum possible number of pipeline stalls.

We can address these two goals separately.

Preserving dependencies

Firstly, we can construct a directed acyclic graph (DAG) to represent the dependencies between instructions:

• For each instruction in the basic block, create a corresponding vertex in the graph.
• For each dependency between two instructions, create a corresponding edge in the graph.
  • This edge is directed: it goes from the earlier instruction to the later one.

```
1 lw $1,0($0)
2 lw $2,4($0)
3 add $3,$1,$2
4 sw $3,12($0)
5 lw $4,8($0)
6 add $3,$1,$4
7 sw $3,16($0)
```

Diagram:
Preserving dependencies

Any topological sort of this DAG (i.e. any linear ordering of the vertices which keeps all the edges “pointing forwards”) will maintain the dependencies and hence preserve the correctness of the program.

Minimising stalls

Secondly, we want to choose an instruction order which causes the fewest possible pipeline stalls.

Unfortunately, this problem is (as usual) NP-complete and hence difficult to solve in a reasonable amount of time for realistic quantities of instructions.

However, we can devise some static scheduling heuristics to help guide us; we will hence choose a sensible and reasonably optimal instruction order, if not necessarily the absolute best one possible.

Algorithm

Armed with the scheduling DAG and the static scheduling heuristics, we can now devise an algorithm to perform instruction scheduling.

• While the candidate list is non-empty:
  • If possible, emit a candidate instruction satisfying all three of the static scheduling heuristics;
  • if no instruction satisfies all the heuristics, either emit NOP (on MIPS) or an instruction satisfying only the last two heuristics (on SPARC).
  • Remove the instruction from the DAG and insert the newly minimal elements into the candidate list.

Algorithm

• Construct the scheduling DAG.
  • We can do this in $O(n^2)$ by scanning backwards through the basic block and adding edges as dependencies arise.
  • Initialise the candidate list to contain the minimal elements of the DAG.

Candidates: 

\{ 1, 2, 5 \}

1 lw $1,0($0)
Instruction scheduling is important for getting the best performance out of a processor; if the compiler does a bad job (or doesn't even try), performance will suffer.

As a result, modern processors have dedicated hardware for performing instruction scheduling dynamically as the code is executing.

This may appear to render compile-time scheduling rather redundant.
Dynamic scheduling

But:

- This is still compiler technology, just increasingly being implemented in hardware.
- Somebody — now hardware designers — must still understand the principles.
- Embedded processors may not do dynamic scheduling, or may have the option to turn the feature off completely to save power, so it’s still worth doing at compile-time.

Summary

- Instruction pipelines allow a processor to work on executing several instructions at once
- Pipeline hazards cause stalls and impede optimal throughput, even when bypassing is used
- Instructions may be reordered to avoid stalls
- Dependencies between instructions limit reordering
- Static scheduling heuristics may be used to achieve near-optimal scheduling with an $O(n^2)$ algorithm

Allocation vs. scheduling

We have seen why register allocation is a useful compilation phase: when done well, it can make the best use of available registers and hence reduce the number of spills to memory.

Unfortunately, by maximising the utilisation of architectural registers, register allocation makes instruction scheduling significantly more difficult.

Allocation vs. scheduling

Can we reorder them to avoid stalls?

1 lw $5,0($1)
2 sw $5,0($2)
3 lw $5,0($3)
4 sw $5,0($4)

No: this is the only correct schedule for these instructions.

Allocation vs. scheduling

We might have done better if register $5$ wasn’t so heavily used.

If only our register allocation had been less aggressive!
Allocation vs. scheduling

So, if we are less zealous about reusing registers, this should hopefully result in a better instruction schedule while not incurring any extra spills.

In general, however, it is rather difficult to predict exactly how our allocation and scheduling phases will interact, and this particular solution is quite ad hoc.

Some (fairly old) research (e.g. CRAIG system in 1995, Touati’s PhD thesis in 2002) has improved the situation.

Allocation vs. scheduling

One option is to try to allocate architectural registers cyclically rather than re-using them at the earliest opportunity.

It is this eager re-use of registers that causes stalls, so if we can avoid it — and still not spill any virtual registers to memory — we will have a better chance of producing an efficient program.

Allocation vs. scheduling

There is clearly antagonism between register allocation and instruction scheduling: one reduces spills by using fewer registers, but the other can better reduce stalls when more registers are used.

This is related to the phase-order problem discussed earlier in the course, in which we would like to defer optimisation decisions until we know how they will affect later phases in the compiler.

It’s not clear how best to resolve the problem.

Allocation vs. scheduling

In practise this means that, when doing register allocation by colouring for a basic block, we should

• satisfy all of the important constraints as usual (i.e. clash graph, preference graph),

• see how many spare architectural registers we still have left over, and then

• for each unallocated virtual register, try to choose an architectural register distinct from all others allocated in the same basic block.

Allocation vs. scheduling

The same problem also shows up in dynamic scheduling done by hardware.

Executable x86 code, for example, has lots of registers available.

Modern machines cope by actually having more registers than advertised; it does dynamic recolouring using this larger register set, which then enables more effective scheduling.
Part D
Decomposition and reverse engineering

Motivation

The job of an optimising compiler is to turn human-readable source code into efficient, executable target code.

Although executable code is useful, software is most valuable in source code form, where it can be easily read and modified.

The source code corresponding to an executable is not always available — it may be lost, missing or secret — so we might want to use decompilation to recover it.

Reverse engineering

In general terms, engineering is a process which decreases the level of abstraction of some system.

In contrast, reverse engineering is the process of increasing the level of abstraction of some system, making it less suitable for implementation but more suitable for comprehension and modification.

Legality and ethics

It is quite feasible to decompile and otherwise reverse-engineer most software.

So if reverse-engineering software is technologically possible, is there any ethical barrier to doing it?

In particular, when is it legal to do so?
Clean room design

Despite the complexity of legislation, it is possible to do useful reverse-engineering without breaking the law.

In 1982, Compaq produced the first fully IBM-compatible personal computer by using clean room design (aka “Chinese wall technique”) to reverse-engineer the proprietary IBM BIOS.

This technique is effective in legally circumventing copyrights and trade secrets, although not patents.

Summary

- Register allocation makes scheduling harder by creating extra dependencies between instructions
- Less aggressive register allocation may be desirable
- Some processors allocate and schedule dynamically
- Reverse engineering is used to extract source code and specifications from executable code
- Existing copyright legislation may permit limited reverse engineering for interoperability purposes
Lecture 16
Decompilation

Why decompilation?
This course is ostensibly about Optimising Compilers.
It is really about program analysis and transformation.
Decompilation is achieved through analysis and transformation of target code; the transformations just work in the opposite direction.

The decompilation problem
Even simple compilation discards a lot of information:
- Comments
- Function and variable names
- Structured control flow
- Type information

The decompilation problem
Optimising compilation is even worse:
- Dead code and common subexpressions are eliminated
- Algebraic expressions are rewritten
- Code and data are inlined; loops are unrolled
- Unrelated local variables are allocated to the same architectural register
- Instructions are reordered by code motion optimisations and instruction scheduling

The decompilation problem
Some of this information is never going to be automatically recoverable (e.g. comments, variable names); some of it we may be able to partially recover if our techniques are sophisticated enough.
Compilation is not injective. Many different source programs may result in the same compiled code, so the best we can do is to pick a reasonable representative source program.

Intermediate code
It is relatively straightforward to extract a flowgraph from an assembler program.
Basic blocks are located in the same way as during forward compilation; we must simply deal with the semantics of the target instructions rather than our intermediate 3-address code.

Intermediate code
For many purposes (e.g. simplicity, retargetability) it might be beneficial to convert the target instructions back into 3-address code when storing it into the flowgraph.

Control reconstruction
A compiler apparently destroys the high-level control structure which is evident in a program’s source code.

This presents its own problems: for example, many architectures include instructions which test or set condition flags in a status register, so it may be necessary to laboriously reconstruct this behaviour with extra virtual registers and then use dead-code elimination to remove all unnecessary instructions thus generated.

After building a flowgraph during decompilation, we can recover some of this structure by attempting to match intervals of the flowgraph against some fixed set of familiar syntactic forms from our high-level language.
Finding loops

Any structured loops from the original program will have been compiled into tests and branches; they will look like arbitrary (“spaghetti”) control flow.

In order to recover the high-level structure of these loops, we must use dominance.

Dominance

A node \( m \) dominates another node \( n \) if control must go through \( m \) before it can reach \( n \).

A node \( m \) strictly dominates another node \( n \) if \( m \) dominates \( n \) and \( m \neq n \).

The immediate dominator of a node \( n \) is the unique node that strictly dominates \( n \) but doesn’t dominate any other strict dominator of \( n \).

Dominance

A node \( n \) is in the dominance frontier of a node \( m \) if \( m \) does not strictly dominate \( n \) but does dominate an immediate predecessor of \( n \).

Intuitively this is the set of nodes where \( m \)’s dominance stops.

We can represent this dominance relation with a dominance tree in which each edge connects a node with its immediate dominator.

Back edges

We can now define the concept of a back edge.

In a flowgraph, a back edge is one whose head dominates its tail.

Finding loops

Each back edge has an associated loop.

The head of a back edge points to the loop header, and the loop body consists of all the nodes from which the tail of the back edge can be reached without passing through the loop header.
Finding loops

Once each loop has been identified, we can examine its structure to determine what kind of loop it is, and hence how best to represent it in source code.

Finding loops

Here, the loop header contains a conditional which determines whether the loop body is executed, and the last node of the body unconditionally transfers control back to the header.

This structure corresponds to source-level
while (...) {...}
syntax.

Finding loops

Here, the loop header unconditionally allows the body to execute, and the last node of the body tests whether the loop should execute again.

This structure corresponds to source-level
do {...} while (...)
syntax.

Finding conditionals

A similar principle applies when trying to reconstruct conditionals: we look for structures in the flowgraph which may be represented by particular forms of high-level language syntax.

Finding conditionals

The first node in this interval transfers control to one node if some condition is true, and another node if the condition is false; control always reaches some later node.

This structure corresponds to source-level
if (...) then {...} else {...}
syntax.

Finding conditionals

The first node in this interval transfers control to one node if some condition is true, otherwise it transfers control to another node (which control also eventually reaches along the first branch).

This structure corresponds to source-level
if (...) then {...} else {...}
syntax.

Control reconstruction

We can keep doing this for whatever other control-flow constructs are available in our source language.

Once an interval of the flowgraph has been matched against a higher-level control structure in this way, its entire subgraph can be replaced with a single node which represents that structure and contains all of the information necessary to generate the appropriate source code.
Type reconstruction

Many source languages also contain rich information about the types of variables: integers, booleans, arrays, pointers, and more elaborate data-structure types such as unions and structs.

At the target code level there are no variables, only registers and memory locations.

Types barely exist here: memory contains arbitrary bytes, and registers contain integers of various bit-widths (possibly floating-point values too).

Reconstruction of the types of source-level variables is made more difficult by the combination of SSA and register allocation performed by an optimising compiler.

SSA splits one user variable into many variables — one for each static assignment — and any of these variables with disjoint live ranges may be allocated to the same architectural register.

So each user variable may be spread between several registers — and each register may hold the value of different variables at different times.

It’s therefore a bit hopeless to try to give a type to each architectural register; the notional type of the value held by any given register will change during execution.

```
int x = 42;

char *y = "42";
```

Happily, we can undo the damage by once again converting to SSA form: this will split a single register into many registers, each of which can be assigned a different type if necessary.

```
MOV r3,#42
MOV r3,#0xFF34
```

```
int foo (int *x) {
  return x[1] + 2;
}
```

```
int f (int r0) {
  r0 = *(int *)(r0 + 4);
  return r0;
}
```

```
int f (int *r0a) {
  int r0b = *(int *)(r0a + 4);
  int r0c = r0b + 2;
  return r0c;
}
```

```
int f (int r0) {
  r0 = r0 + 2;
  return r0;
}
```

```
int f (int r0a) {
  int r0b = *(int *)(r0a + 4);
  int r0c = r0b + 2;
  return r0c;
}
```

```
int f (int r0) {
  r0 = r0 + 2;
  return r0;
}
```

```
int f (int *r0a) {
  int r0b = *(int *)(r0a + 4);
  int r0c = r0b + 2;
  return r0c;
}
```

```
int f (int r0a) {
  int r0b = *(int *)(r0a + 4);
  int r0c = r0b + 2;
  return r0c;
}
```

```
int f (int r0a) {
  int r0b = *(int *)(r0a + 4);
  int r0c = r0b + 2;
  return r0c;
}
```

```
int f (int r0a) {
  int r0b = *(int *)(r0a + 4);
  int r0c = r0b + 2;
  return r0c;
}
```

```
int f (int r0a) {
  int r0b = *(int *)(r0a + 4);
  int r0c = r0b + 2;
  return r0c;
}
```
Type reconstruction

int f (int *r0a) {
    int r0b = *(r0a + 1);
    int r0c = r0b + 2;
    return r0c;
}

Type reconstruction

int f (int *r0a) {
    int r0b = *(r0a + 1);
    int r0c = r0b + 2;
    return r0c;
}

Type reconstruction

In fact, the return type could be anything, so more generally:

T f (T *r0a) {
    return r0a[1] + 2;
}

This is all achieved using constraint-based analysis: each target instruction generates constraints on the types of the registers, and we then solve these constraints in order to assign types at the source level.

Typing information is often incomplete intraprocedurally (as in the example); constraints generated at call sites help to fill in the gaps.

We can also infer unions, structs, etc.

Summary

• Decompilation is another application of program analysis and transformation
• Compilation discards lots of information about programs, some of which can be recovered
• Loops can be identified by using dominator trees
• Other control structure can also be recovered
• Types can be partially reconstructed with constraint-based analysis