Optimising Compilers

Computer Science Tripos Part II

Timothy Jones

Lent 2021

timothy.jones@cl.cam.ac.uk
http://www.cl.cam.ac.uk/Teaching/2021/OptComp/

Learning Guide

The course as lectured proceeds fairly evenly through these notes, with 7 lectures devoted to part A, 5 to part B and 3 or 4 devoted to parts C and D. Part A mainly consists of analysis/transformation pairs on flowgraphs whereas part B consists of more sophisticated analyses (typically on representations nearer to source languages) where typically a general framework and an instantiation are given. Part C consists of an introduction to instruction scheduling and part D an introduction to decompilation and reverse engineering.

One can see part A as intermediate-code to intermediate-code optimisation, part B as (already typed if necessary) parse-tree to parse-tree optimisation and part C as target-code to target-code optimisation. Part D is concerned with the reverse process.

Rough contents of each lecture are:

Lecture 1: Introduction, flowgraphs, call graphs, basic blocks, types of analysis

Lecture 2: (Transformation) Unreachable-code elimination

Lecture 3: (Analysis) Live variable analysis

Lecture 4: (Analysis) Available expressions

Lecture 5: (Transformation) Uses of LVA

Lecture 6: (Continuation) Register allocation by colouring

Lecture 7: (Transformation) Uses of Avail; Code motion

Lecture 8: Static Single Assignment; Strength reduction

Lecture 9: (Framework) Abstract interpretation

Lecture 10: (Instance) Strictness analysis

Lecture 11: (Framework) Constraint-based analysis;
(Instance) Control-flow analysis (for λ-terms)

Lecture 12: (Framework) Inference-based program analysis

Lecture 13: (Instance) Effect systems

Lecture 13a: Points-to and alias analysis

Lecture 14: Instruction scheduling

Lecture 15: Same continued, slop

Lecture 16: Decompilation.

Books

- Aho, A.V., Sethi, R. & Ullman, J.D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986. Now a bit long in the tooth and only covers part A of the course.
  See `http://www.aw-bc.com/catalog/academic/product/0,1144,0321428900,00.html`

- Appel A. *Modern Compiler Implementation in C/ML/Java* (2nd edition). CUP 1997.
  See `http://www.cs.princeton.edu/~appel/modern/`

- Hankin, C.L., Nielson, F., Nielson, H.R. *Principles of Program Analysis*. Springer 1999. Good on part A and part B.
  See `http://www.springer.de/cgi-bin/search_book.pl?isbn=3-540-65410-0`

- Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
  See `http://books.elsevier.com/uk/mk/uk/subindex.asp?isbn=1558603204`

- Wilhelm, R. *Compiler Design*. Addison-Wesley, 1995.
  See `http://www.awprofessional.com/bookstore/product.asp?isbn=0201422905`

Questions, Errors and Feedback

Please let me know if you spot an error in the lecture notes or slides—I will maintain an errata on the course web page, which will hopefully remain empty. Also on the web I will post a list of frequently-asked questions and answers; please feel free to email me if you have anything to ask. In this most different of years, I will provide some office hours to supplement the video lectures—please see the course web page for dates and times. I would also be very happy to receive any comments you may have (positive and negative) on the notes, lectures, or course in general.

Acknowledgement

I first lectured this course in Lent 2017, taking over from Professor Alan Mycroft and adapting his notes from the Michaelmas 2015 Optimising Compilers course, for which I thank him sincerely. Had I started from scratch, this would have been a far less thorough course on the subject. In the intervening years I have made only minor changes to these and the accompanying slides, which were written by Tom Stuart and to whom I am also extremely grateful.
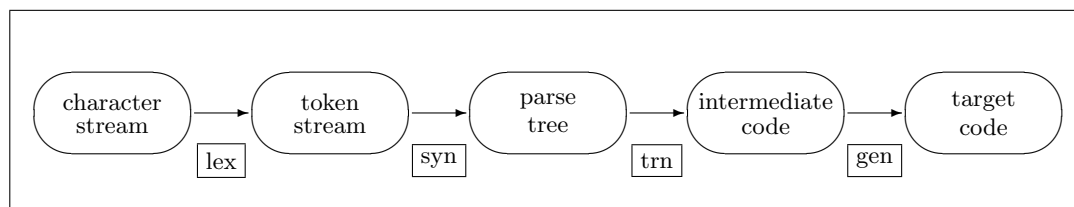
I hope you enjoy this course.

<div align="right">

Timothy Jones
`timothy.jones@cl.cam.ac.uk`

</div>

Part A: Classical 'Dataflow' Optimisations

1 Introduction

Recall the structure of a simple non-optimising compiler (e.g. from CST Part IB).

In such a compiler "intermediate code" is typically a stack-oriented abstract machine code (e.g. OCODE in the BCPL compiler or JVM for Java). Note that stages 'lex', 'syn' and 'trn' are in principle source language-dependent, but not target architecture-dependent whereas stage 'gen' is target dependent but not language dependent.

To ease optimisation (really 'amelioration'!) we need an intermediate code which makes inter-instruction dependencies explicit to ease moving computations around. Typically we use 3-address code (sometimes called 'quadruples'). This is also near to modern RISC architectures and so facilitates target-dependent stage 'gen'. This intermediate code is stored in a flowgraph G—a graph whose nodes are labelled with 3-address instructions (or later 'basic blocks'). We write

$$
\begin{aligned}
pred(n) &= \{n' \mid (n', n) \in edges(G)\} \\
succ(n) &= \{n' \mid (n, n') \in edges(G)\}
\end{aligned}
$$

for the sets of predecessor and successor nodes of a given node; we assume common graph theory notions like path and cycle.

Forms of 3-address instructions (a, b, c are operands, f is a procedure name, and lab is a label):

- ENTRY f: no predecessors;

- EXIT: no successors;

- ALU a, b, c: one successor (ADD, MUL, ...);

- CMP$\langle cond \rangle$ a, b, lab: two successors (CMPNE, CMPEQ, ...) — in straight-line code these instructions take a label argument (and fall through to the next instruction if the branch doesn't occur), whereas in a flowgraph they have two successor edges.

Multi-way branches (e.g. case) can be considered for this course as a cascade of CMP instructions. Procedure calls (CALL f) and indirect calls (CALLI a) are treated as atomic instructions like ALU a, b, c. Similarly one distinguishes MOV a, b instructions (a special case of ALU ignoring one operand) from indirect memory reference instructions (LDI a, b and STI a, b) used to represent pointer dereference including accessing array elements. Indirect branches (used for local goto $\langle exp \rangle$) terminate a basic block (see later); their successors must include all the

possible branch targets (see the description of Fortran ASSIGNED GOTO). A safe way to over-estimate this is to treat as successors all labels which occur other than in a direct goto *l* form. Arguments to and results from procedures are presumed to be stored in standard places, e.g. global variables arg1, arg2, res1, res2, etc. These would typically be machine registers in a modern procedure-calling standard.

As a brief example, consider the following high-level language implementation of the factorial function:

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

This might eventually be translated into the following 3-address code:

```
      ENTRY fact           ; begins a procedure called "fact"
      MOV t32,arg1         ; saves a copy of arg1 in t32
      CMPEQ t32,#0,lab1    ; branches to lab1 if arg1 == 0
      SUB arg1,t32,#1      ; decrements arg1 in preparation for CALL
      CALL fact            ; leaves fact(arg1) in res1 (t32 is preserved)
      MUL res1,t32,res1
      EXIT                 ; exits from the procedure
lab1: MOV res1,#1
      EXIT                 ; exits from the procedure
```

Slogan: Optimisation = Analysis + Transformation

Transformations are often simple (e.g. delete this instruction) but may need complicated analysis to show that they are valid. Note also the use of Analyses without corresponding Transformations for the purposes of compile-time debugging (e.g. see the later use of LVA to warn about the dataflow anomaly of possibly uninitialised variables).

Hence a new structure of the compiler:
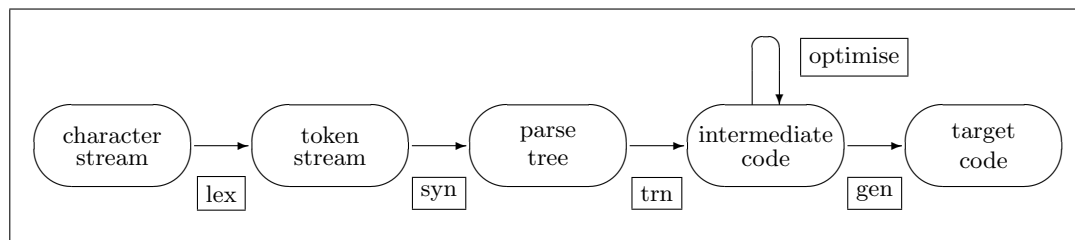
This course only considers the optimiser, which in principle is both source-language and target-architecture independent, but certain gross target features may be exploited (e.g. number of user allocatable registers for a register allocation phase).

Often we group instructions into *basic blocks*: a basic block is a maximal sequence of instructions $n_1, \ldots, n_k$ which have

- exactly one predecessor (except possibly for n_1)

- exactly one successor (except possibly for n_k)

The basic blocks in our example 3-address code factorial procedure are therefore:

```
       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
```

Basic blocks reduce space and time requirements for analysis algorithms by calculating and storing data-flow information once-per-block (and recomputing within a block if required) over storing data-flow information once-per-instruction.

It is common to arrange that stage 'trn', which translates a tree into a flowgraph, uses a new temporary variable on each occasion that one is required. Such a basic block (or flowgraph) is referred to as being *in normal form*. For example, we would translate

```
       x = a*b+c;
       y = a*b+d;
```

into

```
       MUL t1,a,b
       ADD x,t1,c
       MUL t2,a,b
       ADD y,t2,d.
```

Later we will see how general optimisations can map these code sequences into more efficient ones.

1.1 Forms of analysis

Form of analysis (and hence optimisation) are often classified:

- 'local' or 'peephole': within a basic block;

- 'global' or 'intra-procedural': outwith a basic block, but within a procedure;

- 'inter-procedural': over the whole program.

This course mainly considers intra-procedural analyses in part A (an exception being 'unreachable-procedure elimination' in section 1.3) whereas the techniques in part B often are applicable intra-or inter-procedurally (since the latter are not flowgraph-based, further classification by basic block is not relevant).

1.2 Simple example: unreachable-code elimination

(Reachability) Analysis = 'find reachable blocks'; Transformation = 'delete code which reachability does not mark as reachable'. Analysis:

- mark entry node of each procedure as reachable;

- mark every successor of a marked node as reachable and repeat until no further marks are required.

Analysis is *safe*: every node to which execution may flow at execution will be marked by the algorithm. The converse is in general false:

$$\texttt{if } tautology\texttt{(x) then } C_1 \texttt{ else } C_2.$$

The undecidability of arithmetic (cf. the halting problem) means that we can never spot all such cases. Note that safety requires the successor nodes to $\texttt{goto } \langle exp \rangle$ (see earlier) not to be under-estimated. Note also that *constant propagation* (not covered in this course) could be used to propagate known values to tests and hence sometimes to reduce (safely) the number of successors of a comparison node.

1.3 Simple example: unreachable-procedure elimination

(A simple interprocedural analysis.) Analysis = 'find callable procedures'; Transformation = 'delete procedures which analysis does not mark as callable'. Data-structure: call-graph, a graph with one node for each procedure and an edge (f, g) whenever f has a CALL g statement *or f has a CALLI a statement and we suspect that the value of a may be g. A safe (i.e. over-estimate in general) interpretation is to treat CALLI a as calling any procedure in the program which occurs other than in a direct call context—in C this means (implicitly or explicitly) address taken. Analysis:

- mark procedure `main` as callable;

- mark every successor of a marked node as callable and repeat until no further marks are required.

Analysis is *safe*: every procedure which may be invoked during execution will be marked by the algorithm. The converse is again false in general. Note that label variable and procedure variables may reduce optimisation compared with direct code—do not use these features of a programming language unless you are sure they are of overall benefit.

2   Live Variable Analysis (LVA)

A variable x is *semantically live*[1] at node n if there is some execution sequence starting at n whose I/O behaviour can be affected by changing the value of x.

A variable x is *syntactically live* at node n if there is a path in the flowgraph to a node n' at which the current value of x may be used (i.e. a path from n to n' which contains no definition of x and with n' containing a reference to x). Note that such a path may not actually occur during any execution, e.g.

```
l1:  ; /* is 't' live here?  */
     if ((x+1)*(x+1) == y) t = 1;
     if (x*x+2*x+1 != y) t = 2;
l2:  print t;
```

Because of the optimisations we will later base on the results of LVA, safety consists of over-estimating liveness, i.e.

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

where $live(n)$ is the set of variable live at n. Logicians might note the connection of semantic liveness and $\models$ and also syntactic liveness and $\vdash$.

From the non-algorithmic definition of syntactic liveness we can obtain *dataflow equations*:

$$live(n) = \left(\bigcup_{s \in succ(n)} live(s) \right) \setminus def(n) \cup ref(n)$$

You might prefer to derive these in two stages, writing *in-live*(n) for variables live on entry to node n and *out-live*(n) for those live on exit. This gives

$$\begin{aligned} in\text{-}live(n) &= out\text{-}live(n) \setminus def(n) \cup ref(n) \\ out\text{-}live(n) &= \bigcup_{s \in succ(n)} in\text{-}live(s) \end{aligned}$$

Here $def(n)$ is the set of variables defined at node n, i.e. $\{x\}$ in the instruction `x = x+y` and $ref(n)$ the set of variables referenced at node n, i.e. $\{x, y\}$.

Notes:

- These are 'backwards' flow equations: liveness depends on the future whereas normal execution flow depends on the past;

- Any solution of these dataflow equations is safe (w.r.t. semantic liveness).

Problems with address-taken variables—consider:

```
int x,y,z,t,*p;
x = 1, y = 2, z = 3;
p = &x;
if (...) p = &y;
*p = 7;
if (...) p = &x;
t = *p;
print z+t;
```

---

[1] Mention the words 'extensional' for this notion and 'intentional' for the 'syntactic' property below.

Here we are unsure whether the assignment `*p = 7;` assigns to x or y. Similarly we are uncertain whether the reference `t = *p;` references x or y (but we are certain that both *reference* p). These are *ambiguous* definitions and references. For safety we treat (for LVA) an ambiguous reference as referencing *any* address-taken variable (cf. label variable and procedure variables—an indirect reference is just a 'variable' variable). Similarly an ambiguous definition is just ignored. Hence in the above, for `*p = 7;` we have $ref = \{\texttt{p}\}$ and $def = \{\}$ whereas `t = *p;` has $ref = \{\texttt{p}, \texttt{x}, \texttt{y}\}$ and $def = \{\texttt{t}\}$.

Algorithm (implement *live* as an array `live[]`):

```
for i=1 to N do live[i] := {}
while (live[] changes) do
   for i=1 to N do
```
$$\texttt{live[i]} := \left(\bigcup_{s \in succ(\texttt{i})} \texttt{live}[s] \right) \setminus def(\texttt{i}) \cup ref(\texttt{i}).$$

Clearly if the algorithm terminates then it results in a solution of the dataflow equation. Actually the theory of complete partial orders (cpo's) means that it always terminates with the *least* solution, the one with as few variables as possible live consistent with safety. (The powerset of the set of variables used in the program is a finite lattice and the map from old-liveness to new-liveness in the loop is continuous.)

Notes:

- we can implement the `live[]` array as a bit vector with bit k being set to represent that variable $\texttt{x}_k$ (according to a given numbering scheme) is live.

- we can speed execution and reduce store consumption by storing liveness information only once per basic block and re-computing within a basic block if needed (typically only during the use of LVA to validate a transformation). In this case the dataflow equations become:

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

where $(i_1, \ldots, i_k)$ are the instructions in basic block n.

3 Available Expressions (AVAIL)

Available expressions analysis has many similarities to LVA. An expression e (typically the RHS of a 3-address instruction) is *available* at node n if on every path leading to n the expression e has been evaluated and not invalidated by an intervening assignment to a variable occurring in e. Note that the e on each path does not have to come from the same instruction.

This leads to dataflow equations:

$$
\begin{aligned}
avail(n) &= \bigcap_{p \in pred(n)} \left(avail(p) \setminus kill(p) \cup gen(p) \right) && \text{if } pred(n) \neq \{\} \\
avail(n) &= \{\} && \text{if } pred(n) = \{\}.
\end{aligned}
$$

Here $gen(n)$ gives the expressions freshly computed at n: $gen(\texttt{x = y+z}) = \{\texttt{y} + \texttt{z}\}$, for example; but $gen(\texttt{x = x+z}) = \{\}$ because, although this instruction does compute $\texttt{x} + \texttt{z}$, it then changes

the value of x, so if the expression x + z is needed in the future it must be recomputed in light of this.[2] Similarly $kill(n)$ gives the expressions killed at n, i.e. all expressions containing a variable updated at n. These are 'forwards' equations since $avail(n)$ depends on the past rather than the future. Note also the change from $\cup$ in LVA to $\cap$ in AVAIL. You should also consider the effect of ambiguous $kill$ and gen (cf. ambiguous ref and def in LVA) caused by pointer-based access to address-taken variables.

Again any solution of these equations is safe but, given our intended use, we wish the greatest solution (in that it enables most optimisations). This leads to an algorithm (assuming flowgraph node 1 is the only entry node):

```
avail[1] := {}
for i=2 to N do avail[i] := U
while (avail[] changes) do
   for i=2 to N do
      avail[i] := ⋂      (avail[p] \ kill(p) ∪ gen(p)).
              p∈pred(i)
```

Here U is the set of all expressions; it suffices here to consider all RHS's of 3-address instructions. Indeed if one arranges that every assignment assigns to a distinct temporary (a little strengthening of normal form for temporaries) then a numbering of the temporary variables allows a particularly simple bit-vector representation of `avail[]`.

4 Uses of LVA

There are two main uses of LVA:

- to report on dataflow anomalies, particularly a warning to the effect that "variable 'x' may be used before being set";

- to perform 'register allocation by colouring'.

For the first of these it suffices to note that the above warning can be issued if 'x' is live at entry to the procedure (or scope) containing it. (Note here 'safety' concerns are different—it is debatable whether a spurious warning about code which avoids executing a seeming error for rather deep reasons is better or worse than omitting to give a possible warning for suspicious code; decidability means we cannot have both.) For the second, we note that if there is no 3-address instruction where two variables are both live then the variables can share the same memory location (or, more usefully, the same register). The justification is that when a variable is not live its value can be corrupted arbitrarily without affecting execution.

4.1 Register allocation by colouring

Generate naïve 3-address code assuming all variables (and temporaries) are allocated a different (virtual) register (recall 'normal form'). Gives good code, but real machines have a finite number of registers, e.g. 8 in x86 or 31 in MIPS. Derive a graph (the 'clash graph') whose nodes are

---

[2]This definition of $gen(n)$ is rather awkward. It would be tidier to say that $gen(\text{x = x+z}) = \{\text{x} + \text{z}\}$, because x + z is certainly computed by the instruction regardless of the subsequent assignment. However, the given definition is chosen so that $avail(n)$ can be defined in the way that it is; I may say more in lectures.

virtual registers and there is an edge between two virtual registers which are ever simultaneously live (this needs a little care when liveness is calculated merely for basic block starts—we need to check for simultaneous liveness within blocks as well as at block start!). Now try to colour (= give a different value for adjacent nodes) the clash graph using the real (target architecture) registers as colours. (Clearly this is easier if the target has a large-ish number of interchangeable registers—not an early 8086.) Although planar graphs (corresponding to terrestrial maps) can always be coloured with four colours this is not generally the case for clash graphs (exercise).

Graph colouring is NP-complete but here is a simple heuristic for choosing an order to colour virtual registers (and to decide which need to be *spilt* to memory where access can be achieved via LD/ST to a dedicated temporary instead of directly by ALU register-register instructions):

- choose a virtual register with the least number of clashes;

- if this is less than the number of colours then push it on a LIFO stack since we can guarantee to colour it after we know the colour of its remaining neighbours. Remove the register from the clash graph and reduce the number of clashes of each of its neighbours.

- if all virtual registers have more clashes than colours then one will have to be spilt. Choose one (e.g. the one with least number of accesses[3]) to spill and reduce the clashes of all its neighbours by one.

- when the clash graph is empty, pop in turn the virtual registers from the stack and colour them in any way to avoid the colours of their (already-coloured) neighbours. By construction this is always possible.

Note that when we have a free choice between several colours (permitted by the clash graph) for a register, it makes sense to choose a colour which converts a `MOV r1,r2` instruction into a no-op by allocating `r1` and `r2` to the same register (provided they do not clash). This can be achieved by keeping a separate 'preference' graph.

4.2   Non-orthogonal instructions and procedure calling standards

A central principle which justifies the idea of register allocation by colouring at all is that of having a reasonably large interchangeable register set from which we can select at a later time. It is assumed that if we generate a (say) *multiply* instruction then registers for it can be chosen later. This assumption is a little violated on the 80x86 architecture where the multiply instruction always uses a standard register, unlike other instructions which have a reasonably free choice of operands. Similarly, it is violated on a VAX where some instructions corrupt registers `r0`–`r5`.

However, we can design a uniform framework in which such small deviations from uniformity can be gracefully handled. We start by arranging that architectural registers are a subset of virtual registers by arranging that (say) virtual registers `v0`–`v31` are pre-allocated to architectural registers `r0`–`r31` and virtual registers allocated for temporaries and user variables start from 32. Now

---

[3]Of course this is a static count, but can be made more realistic by counting an access within a loop nesting of n as worth 4^n non-loop accesses. Similarly a user `register` declaration can be here viewed as an extra (say) 1000 accesses.

- when an instruction requires an operand in a given architectural register, we use a MOV to move it to the virtual encoding of the given architectural register—the preference graph will try to ensure calculations are targeted to the given source register;

- similarly when an instruction produces a result in a given architectural register, we move the result to an allocatable destination register;

- finally, when an instruction corrupts (say) rx during its calculation, we arrange that its virtual correspondent vx has a clash with every virtual register live at the occurrence of the instruction.

Note that this process has also solved the problem of handling register allocation over procedure calls. A typical procedure calling standard specified n registers for temporaries, say r0–r[n-1] (of which the first m are used for arguments and results—these are the standard places arg1, arg2, res1, res2, etc. mentioned at the start of the course) and k registers to be preserved over procedure call. A CALL or CALLI instruction then causes each variable live over a procedure call to clash with each non-preserved architectural register which results in them being allocated a preserved register. For example,

```
int f(int x) { return g(x)+h(x)+1;}
```

might generate intermediate code of the form

```
        ENTRY f
        MOV v32,r0      ; save arg1 in x
        MOV r0,v32      ; omitted (by "other lecturer did it" technique)
        CALL g
        MOV v33,r0      ; save result as v33
        MOV r0,v32      ; get x back for arg1
        CALL h
        ADD v34,v33,r0 ; v34 = g(x)+h(x)
        ADD r0,v34,#1  ; result = v34+1
        EXIT
```

which, noting that v32 and v33 clash with all non-preserved registers (being live over a procedure call), might generate code (on a machine where r4 upwards are specified to be preserved over procedure call)

```
f:      push  {r4,r5}  ; on ARM we do:  push {r4,r5,lr}
        mov   r4,r0
        call  g
        mov   r5,r0
        mov   r0,r4
        call  h
        add   r0,r5,r0
        add   r0,r0,#1
        pop   {r4,r5}  ; on ARM we do:  pop {r4,r5,pc} which returns ...
        ret            ; ... so don't need this on ARM.
```

Note that `r4` and `r5` need to be push'd and pop'd at entry and exit from the procedure to preserve the invariant that these registers are preserved over a procedure call (which is exploited by using these registers over the calls to `g` and `h`. In general a sensible procedure calling standard specifies that some (but not all) registers are preserved over procedure call. The effect is that store-multiple (or push-multiple) instructions can be used more effectively than sporadic `ld`/`st` to stack.

4.3   Global variables and register allocation

The techniques presented have implicitly dealt with register allocation of *local* variables. These are live for (at most) their containing procedure, and can be saved and restored by called procedures. Global variables (e.g. C static or extern) are in general live on entry to, and exit from, a procedure and in general cannot be allocated to a register except for a whole program "reserve register $r\langle n\rangle$ for variable $\langle x\rangle$" declaration. The allocator then avoids such registers for local variables (because without whole program analysis it is hard to know whether a call may indirectly affect $r\langle n\rangle$ and hence $\langle x\rangle$).

An amusing exception might be a C *local static* variable which is not live on entry to a procedure—this does not have to be preserved from call-to-call and can thus be treated as an ordinary local variable (and indeed perhaps the programmer should be warned about sloppy code). The Green Hills C compiler used to do this optimisation.

5   Uses of AVAIL

The main use of AVAIL is common sub-expression elimination, CSE, (AVAIL provides a technique for doing CSE outwith a single basic block whereas simple-minded tree-oriented CSE algorithms are generally restricted to one expression without side-effects). If an expression e is available at a node n which computes e then we can ensure that the calculations of e on each path to n are saved in a new variable which can be re-used at n instead of re-computing e at n.

In more detail (for any ALU operation $\oplus$):

- for each node n containing $x := a \oplus b$ with $a \oplus b$ available at n:

- create a new temporary t;

- replace $n : x := a \oplus b$ with $n : x := t$;

- on each path scanning backwards from n, for the first occurrence of $a \oplus b$ (say $n' : y := a \oplus b$) in the RHS of a 3-address instruction (which we know exists by AVAIL) replace n' with two instructions $n' : t := a \oplus b$; $n'' : y := t$.

Note that the additional temporary t above can be allocated by register allocation (and also that it encourages the register allocator to choose the same register for t and as many as possible of the various y). If it becomes spilt, we should ask whether the common sub-expression is big enough to justify the LD/ST cost of spilling of whether the common sub-expression is small enough that ignoring it by re-computing is cheaper. (See Section 8).

One subtlety which I have rather side-stepped in this course is the following issue. Suppose we have source code

```
        x := a*b+c;
        y := a*b+c;
```

then this would become 3-address instructions:

```
        MUL t1,a,b
        ADD x,t1,c
        MUL t2,a,b
        ADD y,t2,c
```

CSE as presented converts this to

```
        MUL t3,a,b
        MOV t1,t3
        ADD x,t1,c
        MOV t2,t3
        ADD y,t2,c
```

which is not obviously an improvement! There are two solutions to this problem. One is to consider bigger CSE's than a single 3-address instruction RHS (so that effectively `a*b+c` is a CSE even though it is computed via two different temporaries). The other is to use *copy propagation*—we remove `MOV t1,t3` and `MOV t2,t3` by the expedient of renaming `t1` and `t2` as `t3`. This is only applicable because we know that `t1`, `t2` and `t3` are not otherwise updated. The result is that `t3+c` becomes another CSE so we get

```
        MUL t3,a,b
        ADD t4,t3,c
        MOV x,t4
        MOV y,t4
```

which is just about optimal for input to register allocation (remember that `x` or `y` may be spilt to memory whereas `t3` and `t4` are highly unlikely to be; moreover `t4` (and even `t3`) are likely to be allocated the same register as either `x` or `y` if they are not spilt).

6 Code Motion

Transformations such as CSE are known collectively as *code motion* transformations. Another famous one (more general than CSE[4]) is Partial Redundancy Elimination. Consider

```
    a = ...;
    b = ...;
    do
    {   ... = a+b;        /* here */
        a = ...;
        ... = a+b;
    } while (...)
```

---

[4]One can see CSE as a method to remove *totally* redundant expression computations.

the marked expression `a+b` is redundantly calculated (in addition to the non-redundant calculation) every time round the loop except the first. Therefore it can be time-optimised (even if the program stays the same size) by first transforming it into:

```
a = ...;
b = ...;
... = a+b;
do
{   ... = a+b;        /* here */
    a = ...;
    ... = a+b;
} while (...)
```

and then the expression marked '`here`' can be optimised away by CSE.

7   Static Single Assignment (SSA) Form

Register allocation re-visited: sometimes the algorithm presented for register allocation is not optimal in that it assumes a single user-variable will live in a single place (store location or register) for the whole of its scope. Consider the following illustrative program:

```
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);
}
```

Here `a`, `b` and `c` all mutually clash and so all get separate registers. However, note that the first variable on each line could use (say) `r4`, a register preserved over function calls, and the second variable a distinct variable (say) `r1`. This would reduce the need for registers from three to two, by having distinct registers used for a given variable at different points in its scope. (Note this may be hard to represent in debugger tables.)

The transformation is often called *live range splitting* and can be seen as resulting from source-to-source transformation:

```
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);
}
```

This problem does not arise with temporaries because we have arranged that every need for a temporary gets a new temporary variable (and hence virtual register) allocated (at least before register colouring). The critical property of temporaries which we wish to extend to

15

user-variables is that each temporary is assigned a value only once (statically at least—going round a loop can clearly assign lots of values dynamically).

This leads to the notion of Static Single Assignment (SSA) form and the transformation to it.

SSA form (see e.g. Cytron et al. [2]) is a compilation technique to enable repeated assignments to the same variable (in flowgraph-style code) to be replaced by code in which each variable occurs (statically) as a destination exactly once.

In straight-line code the transformation to SSA is straightforward, each variable v is replaced by a numbered instance v_i of v. When an update to v occurs this index is incremented. This results in code like

```
v = 3; v = v+1; v = v+w; w = v*2;
```

(with next available index 4 for `w` and 7 for `v`) being mapped to

$$v_7 = 3; \; v_8 = v_7+1; \; v_9 = v_8+w_3; \; w_4 = v_9*2;$$

On path-merge in the flowgraph we have to ensure instances of such variables continue to cause the same data-flow as previously. This is achieved by placing a logical (static single) assignment to a new common variable on the path-merge arcs. Because flowgraph nodes (rather than edges) contain code this is conventionally represented by a invoking a so-called ϕ-function at entry to the path-merge node. The intent is that $\phi(x, y)$ takes value x if control arrived from the left arc and y if it arrived from the right arc; the value of the ϕ-function is used to define a new singly-assigned variable. Thus consider

```
{ if (p) { v = v+1; v = v+w; } else v=v-1; } w = v*2;
```

which would map to (only annotating `v` and starting at 4)

```
{ if (p) { v₄ = v₃+1; v₅ = v₄+w; } else v₆=v₃-1; } v₇ = ϕ(v₅,v₆); w = v₇*2;
```

8 The Phase-Order Problem

The 'phase-order problem' refers to the issue in compilation that whenever we have multiple optimisations to be done on a single data structure (e.g. register allocation and CSE on the flowgraph) we find situations where doing any given optimisation yields better results for some programs if done after another optimisation, but better results if done before for others. A slightly more subtle version is that we might want to bias *choices* within one phase to make more optimisations possible in a later phase. These notes just assume that CSE is done before register allocation and if SSA is done then it is done between them.

We just saw the edge of the phase order problem: what happens if doing CSE causes a cheap-to-recompute expression to be stored in a variable which is spilt into expensive-to-access memory. In general other code motion operations (including Instruction Scheduling in Part C) have harder-to-resolve phase order issues.

9 Compiling for Multi-Core

Multi-core processors are now the norm with the inability of additional transistors due to Moore's Law to translate into higher processor clock frequencies (cf. failure of Dennard scaling and Part II Comparative Architectures).

Effectively compiling for them is, however, a challenging task and current industrial offerings are far from satisfactory. One key issue is whether we wish to write in a sequential language and then hope that the compiler can parallelise it (this is liable to be rather optimistic for languages which contain aliasing, especially on NUMA architectures, but also on x86-style multi-core) since "alias analysis" (determining whether two pointers may point to the same location) is undecidable in theory and tends to be ineffective in practice (see Section 18 for an $O(n^3)$ approach). Otherwise a compiler for a sequential language needs hints about where parallelism is possible and/or safe. Open/MP and `Cilk++` are two general-purpose offerings with very different flavours.

The alternative is writing explicitly parallel code, but this easily becomes target-specific and hence non-portable. Languages with explicit message passing (MPI) are possibilities, and for graphics cards Nvidia's CUDA or OpenCL (which targets heterogeneous systems in general) are standard.

A promising direction is that of languages which explicitly express the isolation of two processes (disjointness of memory accesses).

For time reasons this course will not say more on this topic, but it is worth noting that the change from uni-processing to multi-core is bigger than almost any other change in computing, and the sequential languages which we learned how to compile efficiently for sequential machines seem no longer appropriate.

Part B: Higher-Level Optimisations

This second part of the course concerns itself with more modern optimisation techniques than the first part. A simplistic view is that the first part concerned classical optimisations for imperative languages and this part concerns mainly optimisations for functional languages but this somewhat misrepresents the situation. For example even if we perform some of the optimisations (like strictness optimisations) detailed here on a functional language, we may still wish to perform flowgraph-based optimisations like register allocation afterwards. The view I would like to get across is that the optimisations in this part tend to be interprocedural ones and these can often be seen with least clutter in a functional language. So a more correct view is that this part deals with analyses and optimisations at a higher level than that which is easily represented in a flowgraph. Indeed they tend to be phrased in terms of the original (or possibly canonicalised) syntax of the programming language, so that flowgraph-like concepts are not easily available (whether we want them to be or not!).

As a final remark aimed at discouraging the view that the techniques detailed here 'are only suited to functional languages', one should note that for example 'abstract interpretation' is a very general framework for analysis of programs written in any paradigm and it is only the instantiation of it to strictness analysis given here which causes it to be specialised to programs written in a functional paradigm. Similarly 'rule-based program property inference' can be seen as a framework which can be specialised into type checking and inference systems (the subject of another CST Part II course) in addition to the techniques given here.

One must remark however, that the research communities for dataflow analyses and higher-level program analyses have not always communicated sufficiently for unified theory and notation to have developed.

We start by looking at classical intra-procedural optimisations which are typically done at the syntax tree level. Note that these can be seen as code motion transformations (see Section 6).

10   Algebraic Identities

One form of transformation which is is not really covered here is the (rather boring) purely algebraic tree-to-tree transformation such as $e + 0 \longrightarrow e$ or $(e + n) + m \longrightarrow e + (n + m)$ which usually hold universally (without the need to do analysis to ensure their validity, although neither need hold in floating point arithmetic!). A more programming-oriented rule with a trivial analysis might be transforming

```
let x = e in if e' then ... x ... else e''
```

in a lazy language to

```
if e' then let x = e in ... x ... else e''
```

when `e'` and `e''` do not contain `x`. The flavour of transformations which concern us are those for which a non-trivial (i.e. not purely syntactic) property is required to be shown by analysis to validate the transformation.

10.1   Strength reduction

A slightly more exciting example is that of strength reduction. Strength reduction generally refers to replacing some expensive operator with some cheaper one. A trivial example given by

an simple algebraic identity such as $2 * e \longrightarrow$ `let` $x = e$ `in` $x + x$. It is more interesting/useful to do this in a loop.

First find loop *induction variables*, those whose only assignment in the loop is $i := i \oplus c$ for some operator $\oplus$ and some constant[5] c. Now find other variables j, whose only assignment in the loop is $j := c_2 \oplus c_1 \otimes i$, where $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ and c_1, c_2 are constants (we assume this assignment to j is before the update to i to make the following explanation simpler).

The optimisation is to move the assignment $j := c_2 \oplus c_1 \otimes i$ to the entry to the loop[6], and add an end-of-loop-body assignment $j := j \oplus (c_1 \otimes c)$. Now that we know the relation of i to j we can, for example, change any loop-termination test using i to one using j and therefore sometimes eliminate i entirely. For example, assume `int v[100];` and `int`s to be 4 bytes wide on a byte addressed machine. Let us write `&&v` for the *byte* address of the first element of array `v`, noting it is a constant, and consider

```
for (i=0; i<100; i++) v[i] = 0;
```

Although this code is sometimes optimal, many machines need to calculate the physical byte address $\&\&v + 4 * i$ separately from the store instruction, so the code is really

```
for (i=0; i<100; i++) { p = &&v + 4*i; Store4ZerobytesAt(p); }
```

Strength reduction gives:

```
for ((i=0, p=&&v); i<100; (i++, p+=4)) Store4ZerobytesAt(p);
```

and rewriting the loop termination test gives

```
for ((i=0, p=&&v); p<&&v+400; (i++, p+=4)) Store4ZerobytesAt(p);
```

Dropping the i (now no longer used) gives, and re-expressing in proper C gives

```
int *p;
for (p=&v[0]; p<&v[100]; p++) *p = 0;
```

which is often (depending on exact hardware) the optimal code, and is perhaps the code that the C-hackers of you might have been tempted to write. Let me discourage you—this latter code may save a few bytes on your current hardware/compiler, but because of pointer-use, is *much* harder to analyse—suppose your shiny new machine has 64-bit operations, then the loop as originally written can (pretty simply, but beyond these notes) be transformed to be a loop of 50 64-bit stores, but most compilers will give up on the 'clever C programmer' solution.

I have listed strength reduction in this tree-oriented-optimisation section. In many ways it is easier to perform on the flowgraph, but only if loop structure has been preserved as annotations to the flowgraph (recovering this is non-trivial—see the Decompilation section).

---

[5]Although I have written 'constant' here I really only need "expression not affected by execution of (invariant in) the loop".

[6]If i is seen to be assigned a constant on entry to the loop then the RHS simplifies to constant.

11 Abstract Interpretation

In this course there is only time to give the briefest of introductions to abstract interpretation.

We observe that to justify why $(-1515) \times 37$ is negative there are two explanations. One is that $(-1515) \times 37 = -56055$ which is negative. Another is that -1515 is negative, 37 is positive and 'negative $\times$ positive is negative' from school algebra. We formalise this as a table

$\otimes$	$(-)$	(0)	$(+)$
$(-)$	$(+)$	(0)	$(-)$
(0)	(0)	(0)	(0)
$(+)$	$(-)$	(0)	$(+)$

Here there are two calculation routes: one is to calculate in the real world (according to the *standard interpretation* of operators (e.g. $\times$ means multiply) on the *standard space of values*) and then to determine the whether the property we desire holds; the alternative is to *abstract* to an *abstract* space of values and to compute using *abstract interpretations* of operators (e.g. $\times$ means $\otimes$) and to determine whether the property holds there. Note that the abstract interpretation can be seen as a 'toy-town' world which models certain aspects, but in general not all, of reality (the standard interpretation).

When applying this idea to programs undecidability will in general mean that answers cannot be precise, but we wish them to be *safe* in that "if a property is exhibited in the abstract interpretation then the corresponding real property holds". (Note that this means we cannot use logical negation on such properties.) We can illustrate this on the above rule-of-signs example by considering $(-1515) + 37$: real-world calculation yields -1478 which is clearly negative, but the abstract operator $\oplus$ on signs can only safely be written

$\oplus$	$(-)$	(0)	$(+)$
$(-)$	$(-)$	$(-)$	$(?)$
(0)	$(-)$	(0)	$(+)$
$(+)$	$(?)$	$(+)$	$(+)$

where $(?)$ represents an additional abstract value conveying no knowledge (the always-true property), since the sign of the sum of a positive and a negative integer depends on their relative magnitudes, and our abstraction has discarded that information. Abstract addition $\oplus$ operates on $(?)$ by $(?) \oplus x = (?) = x \oplus (?)$ — an unknown quantity may be either positive or negative, so the sign of its sum with any other value is also unknown. Thus we find that, writing *abs* for the abstraction from concrete (real-world) to abstract values we have

$$abs((-1515) + 37) \quad = abs(-1478) = (-), \quad \text{but}$$
$$abs(-1515) \oplus abs(37) = (-) \oplus (+) \quad = (?).$$

Safety is represented by the fact that $(-) \subseteq (?)$, i.e. the values predicted by the abstract interpretation (here everything) include the property corresponding to concrete computation (here $\{z \in \mathbb{Z} \mid z < 0\}$).

Note that we may extend the above operators to accept $(?)$ as an input, yielding the definitions

$\otimes$	$(-)$	(0)	$(+)$	$(?)$
$(-)$	$(+)$	(0)	$(-)$	$(?)$
(0)	(0)	(0)	(0)	(0)
$(+)$	$(-)$	(0)	$(+)$	$(?)$
$(?)$	$(?)$	(0)	$(?)$	$(?)$

$\oplus$	$(-)$	(0)	$(+)$	$(?)$
$(-)$	$(-)$	$(-)$	$(?)$	$(?)$
(0)	$(-)$	(0)	$(+)$	$(?)$
$(+)$	$(?)$	$(+)$	$(+)$	$(?)$
$(?)$	$(?)$	$(?)$	$(?)$	$(?)$

and hence allowing us to compose these operations arbitrarily; for example,

$$(abs(-1515) \otimes abs(37)) \oplus abs(42) = ((-) \otimes (+)) \oplus (+) = (?), \quad \text{or}$$
$$(abs(-1515) \oplus abs(37)) \otimes abs(0) \ = ((-) \oplus (+)) \otimes (0) \ = (0).$$

Similar tricks abound elsewhere e.g. 'casting out nines' (e.g. 123456789 divides by 9 because $1+2+3+4+5+6+7+8+9 = 45$ does, 45 because 4+5 does).

One point worth noting, because it turns up in programming equivalents, is that two different syntactic forms which have the same standard meaning may have differing abstract meanings. An example for the rule-of-signs is $(x+1) \times (x-3) + 4$ which gives (?) when $x = (-)$ whereas $(x \times x) + (-2 \times x) + 1$ gives (+).

Abstract interpretation has been used to exhibit properties such as live variable sets, available expression sets, types etc. as abstract values whose computation can be seen as pre-evaluating the user's program but using non-standard (i.e. abstract) operators during the computation. For this purpose it is useful to ensure the abstract computation is finite, e.g. by choosing finite sets for abstract value domains.

12   Strictness Analysis

This is an example of abstract interpretation which specialises the general framework to determining when a function in a lazy functional language is *strict* in a given formal parameter (i.e. the actual parameter will necessarily have been evaluated whenever the function returns). The associated optimisation is to use call-by-value (eager evaluation) to implement the parameter passing mechanism for the parameter. This is faster (because call-by-value is closer to current hardware than the suspend-resume of lazy evaluation) and it can also reduce asymptotic space consumption (essentially because of tail-recursion effects). Note also that strict parameters can be evaluated in parallel with each other (and with the body of the function about to be called!) whereas lazy evaluation is highly sequential.

In these notes we will not consider full lazy evaluation, but a simple language of recursion equations; eager evaluation is here call-by-value (CBV—evaluate argument once before calling the function); lazy evaluation corresponds to call-by-need (CBN—pass the argument unevaluated and evaluate on its first use (if there is one) and re-use this value on subsequent uses—argument is evaluated 0 or 1 times). In a language free of side-effects CBN is semantically indistinguishable (but possibly distinguishable by time complexity of execution) from call-by-name (evaluate a parameter each time it is required by the function body—evaluates the argument 0,1,2,...times).

The running example we take is

```
plus(x,y) = cond(x=0,y,plus(x-1,y+1)).
```

To illustrate the extra space use of CBN over CBV we can see that

```
plus(3,4) ↦ cond(3=0,4,plus(3-1,4+1))
          ↦ plus(3-1,4+1)
          ↦ plus(2-1,4+1+1)
          ↦ plus(1-1,4+1+1+1)
          ↦ 4+1+1+1
          ↦ 5+1+1
```

$$\mapsto \texttt{6+1}$$
$$\mapsto \texttt{7.}$$

The language we consider here is that of recursion equations:

$$
\begin{aligned}
F_1(x_1, \ldots, x_{k_1}) &= e_1 \\
\cdots &= \cdots \\
F_n(x_1, \ldots, x_{k_n}) &= e_n
\end{aligned}
$$

where e is given by the syntax

$$e ::= x_i \mid A_i(e_1, \ldots, e_{r_i}) \mid F_i(e_1, \ldots e_{k_i})$$

where the A_i are a set of symbols representing built-in (predefined) function (of arity r_i). The technique is also applicable to the full λ-calculus but the current formulation incorporates recursion naturally and also avoids difficulties with the choice of associated strictness optimisations for higher-order situations.

We now interpret the A_i with standard and abstract interpretations (a_i and $a_i^\sharp$ respectively) and deduce standard and abstract interpretations for the F_i (f_i and $f_i^\sharp$ respectively).

Let $D = \mathbb{Z}_\perp (= \mathbb{Z} \cup \{\perp\})$ be the space of integer values (for terminating computations of expressions e) augmented with a value $\perp$ (to represent non-termination). The standard interpretation of a function A_i (of arity r_i) is a value $a_i \in D^{r_i} \to D$. For example

$$
\begin{aligned}
+(\perp, y) &= \perp \\
+(x, \perp) &= \perp \\
+(x, y) &= x +_{\mathbb{Z}} y \quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
cond(\perp, x, y) &= \perp \\
cond(0, x, y) &= y \\
cond(p, x, y) &= x \quad \text{otherwise}
\end{aligned}
$$

(Here, and elsewhere, we treat 0 as the *false* value for *cond* and any non-0 value as *true*, as in C.)

We can now formally define the notion that a function A (of arity r) with semantics $a \in D^r \to D$ is strict in its ith parameter (recall earlier we said that this was if the parameter had necessarily been evaluated whenever the function returns). This happens precisely when

$$(\forall d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_r \in D) \; a(d_1, \ldots, d_{i-1}, \perp, d_{i+1}, \ldots, d_r) = \perp.$$

We now let $D^\sharp = 2 \stackrel{\text{def}}{=} \{0, 1\}$ be the space of abstract values and proceed to define an $a_i^\sharp$ for each a_i. The value '0' represents the property 'guaranteed looping' whereas the value '1' represents 'possible termination'.

Given such an $a \in D^r \to D$ we define $a^\sharp : 2^r \to 2$ by

$$
\begin{aligned}
a^\sharp(x_1, \ldots, x_r) &= 0 \;\; \text{if } (\forall d_1, \ldots, d_r \in D \text{ s.t. } (x_i = 0 \Rightarrow d_i = \perp)) \; a(d_1, \ldots, d_r) = \perp \\
&= 1 \;\; \text{otherwise.}
\end{aligned}
$$

This gives the *strictness function* $a_i^\sharp$ which provides the *strictness interpretation* for each A_i. Note the equivalent characterisation (to which we shall return when we consider the relationship of $f^\sharp$ to f)

$$a^\sharp(x_1, \ldots, x_r) = 0 \Leftrightarrow (\forall d_1, \ldots, d_r \in D \text{ s.t. } (x_i = 0 \Rightarrow d_i = \bot))\ a(d_1, \ldots, d_r) = \bot$$

For example we find

$$
\begin{aligned}
+^\sharp(x, y) &= x \wedge y \\
cond^\sharp(p, x, y) &= p \wedge (x \vee y)
\end{aligned}
$$

We build a table into our analyser giving the strictness function for each built-in function.

Strictness functions generalise the above notion of "being strict in an argument". For a given built-in function a, we have that a is strict in its ith argument iff

$$a^\sharp(1, \ldots, 1, 0, 1, \ldots, 1) = 0$$

(where the '0' is in the ith argument position). However strictness functions carry more information which is useful for determining the strictness property of one (user) function in terms of the functions which it uses. For example consider

```
let f1(x,y,z) = if x then y else z
let f2(x,y,z) = if x then y else 42
let g1(x,y) = f1(x,y,y+1)
let g2(x,y) = f2(x,y,y+1)
```

Both `f1` and `f2` are strict in `x` and nothing else—which would mean that the strictness of `g1` and `g2` would be similarly deduced identical—whereas their strictness functions differ

$$
\begin{aligned}
f1^\sharp(x, y, z) &= x \wedge (y \vee z) \\
f2^\sharp(x, y, z) &= x
\end{aligned}
$$

and this fact enables us (see below) to deduce that `g1` is strict in `x` and `y` while `g2` is merely strict in `x`. This difference between the strictness behaviour of `f1` and `f2` can also be expressed as the fact that `f1` (unlike `f2`) is *jointly strict* in `y` and `z` (i.e. $(\forall x \in D)f(x, \bot, \bot) = \bot$) in addition to being strict in `x`.

Now we need to define strictness functions for user-defined functions. The most exact way to calculate these would be to calculate them as we did for base functions: thus

```
f(x,y) = if tautology(x) then y else 42
```

would yield

$$f^\natural(x, y) = x \wedge y$$

assuming that `tautology` was strict. (Note use of $f^\natural$ in the above—we reserve the name $f^\sharp$ for the following alternative.) Unfortunately this is undecidable in general and we seek a decidable alternative (see the corresponding discussion on semantic and syntactic liveness).

To this end we define the $f_i^\sharp$ not directly but instead in terms of the same composition and recursion from the $a_i^\sharp$ as that which defines the F_i in terms of the A_i. Formally this can be seen as: the f_i are the solution of the equations

$$
\begin{aligned}
F_1(x_1, \ldots, x_{k_1}) &= e_1 \\
\cdots &= \cdots \\
F_n(x_1, \ldots, x_{k_n}) &= e_n
\end{aligned}
$$

when the A_i are interpreted as the a_i whereas the $f_i^\sharp$ are the solutions when the A_i are interpreted as the $a_i^\sharp$.

Safety of strictness can be characterised by the following: given user defined function F (of arity k) with standard semantics $f : D^k \to D$ and strictness function $f^\sharp : 2^k \to 2$ by

$$
f^\sharp(x_1, \ldots, x_k) = 0 \Rightarrow (\forall d_1, \ldots, d_k \in D \text{ s.t. } (x_i = 0 \Rightarrow d_i = \bot)) \, f(d_1, \ldots, d_k) = \bot
$$

Note the equivalent condition for the A_i had $\Rightarrow$ strengthened to $\Leftrightarrow$—this corresponds to the information lost by composing the abstract functions instead of abstracting the standard composition. An alternative characterisation of safety is that $f^\natural(\vec{x}) \leq f^\sharp(\vec{x})$.

Returning to our running example

```
plus(x,y) = cond(x=0,y,plus(x-1,y+1)).
```

we derive equation

$$
plus^\sharp(x, y) = cond^\sharp(eq^\sharp(x, 0^\sharp), y, plus^\sharp(sub1^\sharp(x), add1^\sharp(y)). \tag{1}
$$

Simplifying with built-ins

$$
\begin{aligned}
eq^\sharp(x, y) &= x \wedge y \\
0^\sharp &= 1 \\
add1^\sharp(x) &= x \\
sub1^\sharp(x) &= x
\end{aligned}
$$

gives

$$
plus^\sharp(x, y) = x \wedge (y \vee plus^\sharp(x, y)).
$$

Of the six possible solutions (functions in $2 \times 2 \to 2$ which do not include negation—negation corresponds to 'halt iff argument does not halt')

$$
\{\lambda(x, y).0, \quad \lambda(x, y).x \wedge y, \quad \lambda(x, y).x, \quad \lambda(f, y).y, \quad \lambda(x, y).x \vee y, \quad \lambda(x, y).1\}
$$

we find that only $\lambda(x, y).x$ and $\lambda(x, y).x \wedge y$ satisfy equation (1) and we choose the latter for the usual reasons—all solutions are safe and this one permits most strictness optimisations.

Mathematically we seek the least fixpoint of the equations for $plus^\sharp$ and algorithmically we can solve any such set of equations (using $f\#[i]$ to represent $f_i^\sharp$, and writing $e_i^\sharp$ to mean e_i with the F_j and A_j replaced with $f_j^\sharp$ and $a_j^\sharp$) by:

```
for i=1 to n do f#[i] := λx⃗.0
while (f#[] changes) do
    for i=1 to n do
        f#[i] := λx⃗.e#.
```

Note the similarity to solving dataflow equations—the only difference is the use of functional dataflow values. Implementation is well served by an efficient representation of such boolean functions. ROBDDs[7] are a rational choice in that they are a fairly compact representation with function equality (for the convergence test) being represented by simple pointer equality.

For $plus^\sharp$ we get the iteration sequence $\lambda(x.y).0$ (initial), $\lambda(x,y).x \wedge y$ (first iteration), $\lambda(x,y).x \wedge y$ (second iteration, halt as converged).

Since we can now see that $plus^\sharp(0,1) = plus^\sharp(1,0) = 0$ we can deduce that `plus` is strict in `x` and in `y`.

We now turn to *strictness optimisation*. Recall we suppose our language requires each parameter to be passed *as if* using CBN. As indicated earlier any parameter shown to be strict can be implemented using CBV. For a thunk-based implementation of CBN this means that we continue to pass a closure $\lambda().e$ for any actual parameter e not shown to be strict and evaluate this on first use inside the body; whereas for a parameter shown to be strict, we evaluate e before the call by passing it using CBV and then merely use the value in the body.

13   Constraint-Based Analysis

In constraint-based analysis, the approach taken is that of walking the program emitting constraints (typically, but not exclusively) on the sets of values which variables or expressions may take. These sets are related together by constraints. For example if x is constrained to be an even integer then it follows that $x + 1$ is constrained to be an odd integer.

Rather than look at numeric problems, we choose as an example analysis the idea of control-flow analysis (CFA, technically 0-CFA for those looking further in the literature); this attempts to calculate the set of functions callable at every call site.

13.1   Constraint systems and their solution

This is a non-examinable section, here to provide a bit of background.

Many program analyses can be seen as solving a system of constraints. For example in LVA, the constraints were that a "set of live variables at one program point is *equal* to some (monotonic) function applied to the sets of live variables at other program points". Boundary conditions were supplied by entry and/or exit nodes. I used the "other lecturer did it" technique (here 'semantics') to claim that such sets of such constraints have a minimal solution. Another example is Hindley-Milner type checking—we annotate every expression with a type t_i, e.g. $(e_1^{t_1} e_2^{t_2})^{t_3}$ and then walk the program graph emitting constraints representing the need for consistency between neighbouring expressions. The term above would emit the constraint $t_1 = (t_2 \to t_3)$ and then recursively emit constraints for e_1 and e_2. We can then solve these constraints (now using unification) and the least solution (substituting types to as few t_i as possible) corresponds to ascribing all expressions their most-general type.

In the CFA below, the constraints are inequations, but they again have the property that a minimal solution can be reached by initially assuming that all sets α_i are empty, then for each constraint $\alpha \supseteq \phi$ (note we exploit that the LHS is always a flow variable) which fails to hold, we update α to be ϕ and loop until all equations hold.

---

[7]ROBBD means Reduced Ordered Binary Decision Diagram, but often OBDD or BDD is used to refer to the same concept.

One exercise to think of solving inequation systems is to consider how, given a relation R, its transitive closure T may be obtained. This can be expressed as constraints:

$$R \subseteq T$$
$$\{(x,y)\} \subseteq T \wedge \{(y,z)\} \subseteq R \implies \{(x,z)\} \subseteq T$$

14 Control-Flow Analysis (For λ-Terms)

This is not to be confused with the simpler intraprocedural reachability analysis on flow graphs, but rather generalises call graphs. Given a program P the aim is to calculate, for each expression e, the set of primitive values (here integer constants and λ-abstractions) which can result from e during the evaluation of P. (This can be seen as a higher-level technique to improve the resolution of the approximation "assume an indirect call may invoke any procedure whose address is taken" which we used in calculating the call graph.)

We take the following language for concrete study (where we consider c to range over a set of (integer) constants and x to range over a set of variables):

$$e ::= x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \texttt{let } x = e_1 \texttt{ in } e_2.$$

Programs P are just terms in e with no free variables. For this lecture we will consider the program, P, given by

$$\texttt{let id} = \lambda\texttt{x.x in id id 7}$$

We now need a notion of program point (generalisation of label) which we can use to reference uniquely a given expression *in context*. This is important because the same expression may occur twice in a program but we wish it to be treated separately. Thus we label the nodes of the syntax tree of the above program uniquely with their *occurrences* in the tree (formally sequences of integers representing the route from the root to the given node, but here convenient integers). This gives

$$(\texttt{let id}^{10} = (\lambda\texttt{x}^{20}.\texttt{x}^{21})^{22} \texttt{ in } ((\texttt{id}^{30} \texttt{ id}^{31})^{32} \texttt{ } 7^{33})^{34})^1.$$

The space of *flow values* F for this program is

$$\{(\lambda\texttt{x}^{20}.\texttt{x}^{21})^{22}, 7^{33}\}$$

which again in principle require the labelling to ensure uniqueness. Now associate a *flow variable* with each program point, i.e.

$$\alpha_1, \alpha_{10}, \alpha_{20}, \alpha_{21}, \alpha_{22}, \alpha_{30}, \alpha_{31}, \alpha_{32}, \alpha_{33}, \alpha_{34}.$$

In principle we wish to associate, with each flow variable α_i associated with expression e^i, the subset of the flow values which it yields during evaluation of P. Unfortunately again this is undecidable in general and moreover can depend on the evaluation strategy (CBV/CBN). We have seen this problem before and, as before, we give a formulation to get safe approximations (here possibly over-estimates) for the α_i.[8] Moreover these solutions are safe with respect to any evaluation strategy for P (this itself is a source of some imprecision!).

We get constraints on the α_i determined by the program structure (the following constraints are in addition to the ones recursively generated by the subterms e, e_1, e_2 and e_3):

---

[8]The above is the normal formulation, but you might prefer to think in dataflow terms. α_i represents *possible-values*(i) and the equations below are dataflow equations.

- for a term x^i we get the constraint $\alpha_i \supseteq \alpha_j$ where x^j is the associated binding (via `let` $x^j = \cdots$ or $\lambda x^j . \cdots$);

- for a term c^i we get the constraint $\alpha_i \supseteq \{c^i\}$;

- for a term $(\lambda x^j.e^k)^i$ we get the constraint $\alpha_i \supseteq \{(\lambda x^j.e^k)^i\}$;

- for a term $(e_1^j e_2^k)^i$ we get the compound constraint $(\alpha_k \mapsto \alpha_i) \supseteq \alpha_j$;

- for a term $(\texttt{let } x^l = e_1^j \texttt{ in } e_2^k)^i$ we get the constraints $\alpha_i \supseteq \alpha_k$ and $\alpha_l \supseteq \alpha_j$;

- for a term $(\texttt{if } e_1^j \texttt{ then } e_2^k \texttt{ else } e_3^l)^i$ we get the constraints $\alpha_i \supseteq \alpha_k$ and $\alpha_i \supseteq \alpha_l$.

Here $(\gamma \mapsto \delta) \supseteq \beta$ represents the fact that the flow variable β (corresponding to the information stored for the function to be applied) must include the information that, when provided an argument contained within the argument specification γ, it yields results contained within the result specification δ. (Of course δ may actually be larger because of other calls.) Formally $(\gamma \mapsto \delta) \supseteq \beta$ is shorthand for the compound constraint that (i.e. is satisfied when)

$$\text{whenever } \beta \supseteq \{(\lambda x^q.e^r)^p\} \text{ we have } \alpha_q \supseteq \gamma \wedge \delta \supseteq \alpha_r.$$

You may prefer instead to to see this directly as "applications generate an implication":

- for a term $(e_1^j e_2^k)^i$ we get the constraint implication

$$\alpha_j \supseteq \{(\lambda x^q.e^r)^p\} \implies \alpha_q \supseteq \alpha_k \wedge \alpha_i \supseteq \alpha_r.$$

Now note this implication can also be written as two implications

$$\alpha_j \supseteq \{(\lambda x^q.e^r)^p\} \implies \alpha_q \supseteq \alpha_k$$
$$\alpha_j \supseteq \{(\lambda x^q.e^r)^p\} \implies \alpha_i \supseteq \alpha_r$$

Now, if you know about Prolog/logic programming then you can see these expression forms as generating clauses defining the predicate symbol $\supseteq$. Most expressions generate simple 'always true' clauses such as $\alpha_i \supseteq \{c^i\}$, whereas the application form generates two implicational clauses:

$$\alpha_q \supseteq \alpha_k \impliedby \alpha_j \supseteq \{(\lambda x^q.e^r)^p\}$$
$$\alpha_i \supseteq \alpha_r \impliedby \alpha_j \supseteq \{(\lambda x^q.e^r)^p\}$$

Compare the two forms respectively with the two clauses

```
app([],X,X).
app([A|L],M,[A|N]) :- app(L,M,N).
```

which constitutes the Prolog definition of *append*.

As noted in Section 13.1 the constraint set generated by walking a program has a unique least solution.

The above program P gives the following constraints, which we should see as dataflow inequations:

$$
\begin{array}{rcll}
\alpha_1 & \supseteq & \alpha_{34} & \texttt{let} \text{ result} \\
\alpha_{10} & \supseteq & \alpha_{22} & \texttt{let} \text{ binding} \\
\alpha_{22} & \supseteq & \{(\lambda\texttt{x}^{20}.\texttt{x}^{21})^{22}\} & \lambda\text{-abstraction} \\
\alpha_{21} & \supseteq & \alpha_{20} & \texttt{x} \text{ use} \\
\alpha_{33} & \supseteq & \{7^{33}\} & \text{constant } 7 \\
\alpha_{30} & \supseteq & \alpha_{10} & \texttt{id} \text{ use} \\
\alpha_{31} \mapsto \alpha_{32} & \supseteq & \alpha_{30} & \text{application-32} \\
\alpha_{31} & \supseteq & \alpha_{10} & \texttt{id} \text{ use} \\
\alpha_{33} \mapsto \alpha_{34} & \supseteq & \alpha_{32} & \text{application-34}
\end{array}
$$

Again all solutions are safe, but the least solution is

$$
\begin{array}{rcl}
\alpha_1 = \alpha_{34} = \alpha_{32} = \alpha_{21} = \alpha_{20} & = & \{(\lambda\texttt{x}^{20}.\texttt{x}^{21})^{22}, 7^{33}\} \\
\alpha_{30} = \alpha_{31} = \alpha_{10} = \alpha_{22} & = & \{(\lambda\texttt{x}^{20}.\texttt{x}^{21})^{22}\} \\
\alpha_{33} & = & \{7^{33}\}
\end{array}
$$

You may verify that this solution is safe, but note that is imprecise because $(\lambda\texttt{x}^{20}.\texttt{x}^{21})^{22} \in \alpha_1$ whereas the program always evaluates to 7^{33}. The reason for this imprecision is that we have only a single flow variable available for the expression which forms the body of each λ-abstraction. This has the effect that possible results from one call are conflated with possible results from another. There are various enhancements to reduce this which we sketch in the next paragraph (but which are rather out of the scope of this course).

The analysis given above is a *monovariant* analysis in which one property (here a single set-valued flow variable) is associated with a given term. As we saw above, it led to some imprecision in that P above was seen as possibly returning $\{7, \lambda\texttt{x}.\texttt{x}\}$ whereas the evaluation of P results in 7. There are two ways to improve the precision. One is to consider a *polyvariant* approaching in which multiple calls to a single procedure are seen as calling separate procedures with identical bodies. An alternative is a *polymorphic* approach in which the values which flow variables may take are enriched so that a (differently) specialised version can be used at each use. One can view the former as somewhat akin to the ML treatment of overloading where we see (letting $\wedge$ represent the choice between the two types possessed by the + function)

```
op + : int*int->int ∧ real*real->real
```

and the latter can be similarly be seen as comparable to the ML typing of

```
fn x=>x : ∀α.α->α.
```

This is an active research area and the ultimately 'best' treatment is unclear.

15 Class Hierarchy Analysis

This section is just a pointer for those of you who want to know more about optimising object-oriented programs. Dean et al. [3] " Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis" is the original source. Ryder [4] "Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages" gives a retrospective.

16 Inference-Based Program Analysis

This is a general technique in which an inference system specifies judgements of the form

$$\Gamma \vdash e : \phi$$

where ϕ is a program property and Γ is a set of assumptions about free variables of e. One standard example (covered in more detail in the CST Part II 'Types' course) is the ML type system. Although the properties are here types and thus are not directly typical of program optimisation (the associated optimisation consists of removing types of values, evaluating in a typeless manner, and attaching the inferred type to the computed typeless result; non-typable programs are rejected) it is worth considering this as an archetype. For current purposes ML expressions e can here be seen as the λ-calculus:

$$e ::= x \mid \lambda x.e \mid e_1 e_2$$

and (assuming α to range over type variables) types t of the syntax

$$t ::= \alpha \mid int \mid t \rightarrow t'.$$

Now let Γ be a set of assumptions of the form $\{x_1 : t_1, \ldots, x_n : t_n\}$ which assume types t_i for free variables x_i; and write $\Gamma[x : t]$ for Γ with any assumption about x removed and with $x : t$ additionally assumed. We then have inference rules:

$$(\text{VAR}) \frac{}{\Gamma[x : t] \vdash x : t}$$

$$(\text{LAM}) \frac{\Gamma[x : t] \vdash e : t'}{\Gamma \vdash \lambda x.e : t \rightarrow t'}$$

$$(\text{APP}) \frac{\Gamma \vdash e_1 : t \rightarrow t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : t'}.$$

Safety: the type-safety of the ML inference system is clearly not part of this course, but its formulation clearly relates to that for other analyses. It is usually specified by the *soundness* condition:

$$(\{\} \vdash e : t) \Rightarrow (\llbracket e \rrbracket \in \llbracket t \rrbracket)$$

where $\llbracket e \rrbracket$ represents the result of evaluating e (its denotation) and $\llbracket t \rrbracket$ represents the set of values which have type t. Note that (because of $\{\}$) the safety statement only applies to closed programs (those with no free variables) but its inductive proof in general requires one to consider programs with free variables.

The following gives a more program-analysis–related example; here properties have the form

$$\phi ::= odd \mid even \mid \phi \rightarrow \phi'.$$

We would then have rules:

$$(\text{VAR}) \frac{}{\Gamma[x : \phi] \vdash x : \phi}$$

$$(\text{LAM}) \frac{\Gamma[x : \phi] \vdash e : \phi'}{\Gamma \vdash \lambda x.e : \phi \rightarrow \phi'}$$

$$(\text{APP}) \frac{\Gamma \vdash e_1 : \phi \rightarrow \phi' \qquad \Gamma \vdash e_2 : \phi}{\Gamma \vdash e_1 e_2 : \phi'}.$$

Under the assumptions

$$\Gamma = \{2 : even, \quad + : even \to even \to even, \quad \times : even \to odd \to even\}$$

we could then show

$$\Gamma \vdash \lambda x.\lambda y.2 \times x + y : odd \to even \to even.$$

but note that showing

$$\Gamma' \vdash \lambda x.\lambda y.2 \times x + 3 \times y : even \to even \to even.$$

would require Γ' to have *two* assumptions for $\times$ or a single assumption of a more elaborate property, involving conjunction, such as:

$$
\begin{aligned}
\times : \quad &even \to even \to even \;\wedge \\
&even \to odd \to even \;\;\wedge \\
&odd \to even \to even \;\;\wedge \\
&odd \to odd \to odd.
\end{aligned}
$$

Exercise: Construct a system for *odd* and *even* which can show that

$$\Gamma \vdash (\lambda f.f(1) + f(2))(\lambda x.x) : odd$$

for some Γ.

17 Effect Systems

This is an example of inference-based program analysis. The particular example we give concerns an *effect system* for analysis of communication possibilities of systems.

The idea is that we have a language such as the following

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \mid \xi?x.e \mid \xi!e_1.e_2 \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3.$$

which is the λ-calculus augmented with expressions $\xi?x.e$ which reads an *int* from a channel ξ and binds the result to x before resulting in the value of e (which may contain x) and $\xi!e_1.e_2$ which evaluates e_1 (which must be an *int*) and writes its value to channel ξ before resulting in the value of e_2. Under the ML type-checking regime, side effects of reads and writes would be ignored by having rules such as:

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

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

For the purpose of this example, we suppose the problem is to determine which channels may be read or written during evaluation of a closed term P. These are the *effects* of P. Here we take the effects, ranged over by F, to be subsets of

$$\{W_\xi, R_\xi \mid \xi \text{ a channel}\}.$$

The problem with the natural formulation is that a program like

$$\xi!1.\lambda x.\zeta!2.x$$

has an *immediate effect* of writing to ξ but also a *latent effect* of writing to ζ via the resulting λ-abstraction.

We can incorporate this notion of effect into an inference system by using judgements of the form

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

whose meaning is that when e is evaluated then its result has type t and whose *immediate* effects are a subset (this represents *safety*) of F. To account for *latent* effects of a λ-abstraction we need to augment the type system to

$$t ::= int \mid t \xrightarrow{F} t'.$$

Letting $one(f) = \{f\}$ represent the singleton effect, the inference rules are then

$$(\text{VAR})\frac{}{\Gamma[x : t] \vdash x : t, \emptyset}$$

$$(\text{READ})\frac{\Gamma[x : int] \vdash e : t, F}{\Gamma \vdash \xi?x.e : t, one(R_\xi) \cup F}$$

$$(\text{WRITE})\frac{\Gamma \vdash e_1 : int, F \qquad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash \xi!e_1.e_2 : t, F \cup one(W_\xi) \cup F'}$$

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

$$(\text{APP})\frac{\Gamma \vdash e_1 : t \xrightarrow{F''} t', F \qquad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash e_1 e_2 : t', F \cup F' \cup F''}.$$

Note that by changing the space of effects into a more structured set of values (and by changing the understanding of the $\emptyset$, *one* and $\cup$ constants and operators on effects e.g. using sequences with $\cup$ being *append*) we could have captured more information such as temporal ordering since

$$\xi?x.\zeta!(x+1).42 : int, \{R_\xi\} \cup \{W_\zeta\}$$

and

$$\zeta!7.\xi?x, 42 : int, \{W_\zeta\} \cup \{R_\xi\}.$$

Similarly one can extend the system to allow transmitting and receiving more complex types than *int* over channels.

One additional point is that care needs to be taken about allowing an expression with fewer effects to be used in a context which requires more. This is an example of subtyping although the example below only shows the subtype relation acting on the effect parts. The obvious rule for if-then-else is:

$$(\text{COND})\frac{\Gamma \vdash e_1 : int, F \qquad \Gamma \vdash e_2 : t, F' \qquad \Gamma \vdash e_3 : t, F''}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : t, F \cup F' \cup F''}.$$

However, this means that

$$\text{if } x \text{ then } \lambda x.\xi!3.x + 1 \text{ else } \lambda x.x + 2$$

is ill-typed (the types of e_2 and e_3 mismatch because their latent effects differ). Thus we tend to need an additional rule which, for the purposes of this course can be given by

$$(\text{SUB})\frac{\Gamma \vdash e : t \xrightarrow{F'} t', F}{\Gamma \vdash e : t \xrightarrow{F''} t', F} \text{ (provided } F' \subseteq F'')$$

Safety can then similarly approached to that of the ML type system where semantic function $[\![e]\!]$ is adjusted to yield a pair (v, f) where v is a resulting value and f the actual (immediate) effects obtained during evaluation. The safety criterion is then stated:

$$(\{\} \vdash e : t, F) \Rightarrow (v \in [\![t]\!] \wedge f \subseteq F \text{ where } (v, f) = [\![e]\!])$$

Incidentally, various "purity analyses" for Java (which capture that a *pure method* has no effect on existing data structures) are closely related to effect systems.

18 Points-To and Alias Analysis

Consider an MP3 player containing code:

```
for (channel = 0; channel < 2; channel++)
  process_audio(channel);
```

or even

```
process_audio_left();
process_audio_right();
```

These calls can only be parallelised (useful for multi-core CPUs) if neither call writes to a memory location read or written by the other.

So, we want to know (at compile time) what locations a procedure might write to or read from at run time.

For simple variables, even including address-taken variables, this is moderately easy (we have done similar things in "ambiguous *ref*" in LVA and "ambiguous *kill*" in Avail), but note that multi-level pointers `int a, *b=&a, **c=&b;` make the problem more complicated here.

So, given a pointer value, we are interested in finding a (finite) description of what locations it *might* point to—or, given a procedure, a description of what locations it might read from or write to. If two such descriptions have empty intersection then we can parallelise.

To deal with `new()` we will adopt the crude idea that all allocations done at a single program point may alias, but allocations done at two different points cannot:

```
for (i=1; i<2; i++)
{ t = new();
  if (i==1) a=t; else b=t;
}
c = new();
d = new();
```

We see `a` and `b` as possibly aliasing (as they both point to the `new` on line 2, while `c` and `d` cannot alias with `a`, `b` or each other. A similar effect would occur in

```
for (...)
{ p = cons(a,p);
  p = cons(b,p);
}
```

Where we know that `p` points to a `new` from line 2, which points to a `new` from line 3, which points to a `new` from line 2 ....

Another approximation which we will make is to have a single points-to summary that says (e.g.) p may point to c or d, but definitely nothing else. We *could* record this information on a per-statement level which would be more accurate, but instead choose to hold this information once (per-procedure). Hence in

```
p = &c;
*p = 3;
p = &d;
q = &e;
```

we will assume that the indirect write may update `c` or `d` but not `e`.

Strategy:

- do a "points-to" analysis which associates each variable with (a description of) a set of locations.

- can now just say "x and y may alias if their results from points-to analysis is not provably disjoint".

Alias analysis techniques can become very expensive for large programs "alias analysis is undecidable in theory and intractable in practice". Simpler techniques tend to say "I don't know" too often.

We will present Andersen's $O(n^3)$ algorithm, at least in part because the constraint-solving is identical to 0-CFA! Note that we only consider the intra-procedural situation.

First assume programs have been written in 3-address code and with all *pointer-typed* operations being of the form

$$
\begin{array}{ll}
x := \mathtt{new}_\ell & \ell \text{ is a program point (label)} \\
x := \mathtt{null} & \text{optional, can see as variant of } \mathtt{new} \\
x := \&y & \text{only in C-like languages, also like } \mathtt{new} \text{ variant} \\
x := y & \text{copy} \\
x := *y & \text{field access of object} \\
*x := y & \text{field access of object}
\end{array}
$$

Note that pointer arithmetic is not considered. Also, note that while `new` can be seen as allocating a record, we only provide operations to read and write all fields at once. This means that fields are conflated, i.e. we analyse $x.f = e$ and $x.g = e$ as identical—and equivalent to $*x = e$. It is possible to consider so-called 'field-sensitive' analyses (not in this course though, so use google if you want to know more).

18.1 Andersen's analysis in detail

Define a set of abstract values

$$V = Var \cup \{\texttt{new}_\ell \mid \ell \in Prog\} \cup \{\texttt{null}\}$$

As said before, we treat all allocations at a given program point as indistinguishable.

Now consider the *points-to* relation. Here we see this a function $pt(x) : V \to \mathcal{P}(V)$. As said before, we keep one pt per procedure (intra-procedural analysis).

Each line in the program generates zero of more constraints on pt:

$$\overline{\vdash x := \&y : y \in pt(x)} \qquad \overline{\vdash x := \texttt{null} : \texttt{null} \in pt(x)}$$

$$\overline{\vdash x := \texttt{new}_\ell : \texttt{new}_\ell \in pt(x)} \qquad \overline{\vdash x := y : pt(y) \subseteq pt(x)}$$

$$\frac{z \in pt(y)}{\vdash x := *y : pt(z) \subseteq pt(x)} \qquad \frac{z \in pt(x)}{\vdash *x := y : pt(y) \subseteq pt(z)}$$

Note that the first three rules are essentially identical.

The above rules all deal with atomic assignments. The next question to consider is control-flow. Our previous analyses (e.g. LVA) have all been *flow-sensitive*, e.g. we treat

```
x = 1; print x; y = 2; print y;
```

and

```
x = 1; y =2 ; print x; print y
```

differently (as required when allocating registers to `x` and `y`). However, Andersen's algorithm is *flow-insensitive*, we simply look at the *set of* statements in the program and not at their order or their position in the syntax tree. This is faster, but loses precision. Flow-insensitive means property inference rules are essentially of the form (here C is a command, and S is a set of constraints):

$$(\text{ASS})\frac{}{\vdash e := e' : \langle \text{as above} \rangle} \qquad (\text{SEQ})\frac{\vdash C : S \quad \vdash C' : S'}{\vdash C; C' : S \cup S'}$$

$$(\text{COND})\frac{\vdash C : S \quad \vdash C' : S'}{\vdash \texttt{if } e \texttt{ then } C \texttt{ else } C' : S \cup S'}$$

$$(\text{WHILE})\frac{\vdash C : S}{\vdash \texttt{while } e \texttt{ do } C : S}$$

The safety property   A program analysis on its own never useful—we want to be able to use it for transformations, and hence need to know what the analysis guarantees about run-time execution.

Given pt solving the constraints generated by Andersen's algorithm then we have that

- at all program points during execution, the value of pointer variable x is always in the description $pt(x)$. For `null` and `&z` this is clear, for $\texttt{new}_\ell$ this means that x points to a memory cell allocated there.

Hence (alias analysis, and its uses):

- If $pt(x) \cap pt(y)$ is empty, then x and y cannot point to the same location, hence *it is safe* to (e.g.) swap the order of `n=*x; *y=m`, or even to run them in parallel.

Epilogue for Part B

You might care to reflect that program analyses and type systems have much in common. Both attempt to determine whether a given property of a program holds (in the case of type systems, this is typically that the application of an operator is type-safe). The main difference is the use to which analysis results are put—for type systems failure to guarantee type correctness causes the program to be rejected whereas for program analysis failure to show a result causes less efficient code to be generated.

Part C: Instruction Scheduling

19 Introduction

In this part we introduce instruction scheduling for a processor architecture of complexity typical of the mid-1980's. Good examples would be the MIPS R-2000 or SPARC implementations of this period. Both have simple 5-stage pipelines (IF,RF,EX,MEM,WB) with register bypassing and both have delayed branches and delayed loads. One difference is that the MIPS had no interlocks on delayed loads (therefore requiring the compiler writer, in general, to insert NOP's to ensure correct operation) whereas the SPARC had interlocks which cause pipeline stalls when a later instruction refers to an operand which is not yet available. In both cases faster execution (in one case by removing NOP's and in the other by avoiding stalls) is often possible by re-ordering the (target) instructions essentially within each basic block.

Of course there are now more sophisticated architectures: many processors have multiple dispatch into multiple pipelines. Functional units (e.g. floating point multipliers) may be scheduled separately by the pipeline to allow the pipeline to continue while they complete. They may be also duplicated. High-performance architectures go as far as re-scheduling instruction sequences dynamically, to some extent making instruction scheduling at compile time rather redundant. However, the ideas presented here are an intellectually satisfactory basis for compile-time scheduling for all architectures.

The data structure we operate upon is a graph of basic blocks, each consisting of a sequence of *target* instructions obtained from blow-by-blow expansion of the abstract 3-address intermediate code we saw in Part A of this course. Scheduling algorithms usually operate within a basic block and adjust if necessary at basic block boundaries—see later.

The objective of scheduling is to minimise the number of pipeline stalls (or the number of inserted NOP's on the MIPS). Sadly the problem of such optimal scheduling is often NP-complete and so we have to fall back on heuristics for life-size code. These notes present the $O(n^2)$ algorithm due to Gibbons and Muchnick [5].

Observe that two instructions may be permuted if neither writes to a register read or written by the other. We define a graph (actually a DAG), whose nodes are instructions within a basic block. Place an edge from instruction a to instruction b if a occurs before b in the original instruction sequence and if a and b cannot be permuted. Now observe that any of the minimal elements of this DAG (normally drawn at the top in diagrammatic form) can be validly scheduled to execute first and after removing such a scheduled instruction from the graph any of the new minimal elements can be scheduled second and so on. In general any topological sort of this DAG gives a valid scheduling sequence. Some are better than others and to achieve non-NP-complete complexity we cannot in general search freely, so the current $O(n^2)$ algorithm makes the choice of the next-to-schedule instruction *locally*, by choosing among the minimal elements with the *static scheduling heuristics*

- choose an instruction which does not conflict with the previous emitted instruction

- choose an instruction which is most likely to conflict if first of a pair (e.g. `ld.w` over `add`)

- choose an instruction which is as far as possible (over the longest path) from a graph-maximal instruction—the ones which can be validly be scheduled as the last of the basic block.

On the MIPS or SPARC the first heuristic can never harm. The second tries to get instructions which can provoke stalls out of the way in the hope that another instruction can be scheduled between a pair which cause a stall when juxtaposed. The third has similar aims—given two independent streams of instructions we should save some of each stream for inserting between stall-pairs of the other.

So, given a basic block

- construct the scheduling DAG as above; doing this by scanning backwards through the block and adding edges when dependencies arise, which works in $O(n^2)$

- initialise the *candidate list* to the minimal elements of the DAG

- while the candidate list is non-empty

  - emit an instruction satisfying the static scheduling heuristics (for the first iteration the 'previous instruction' with which we must avoid dependencies is any of the final instructions of predecessor basic blocks which have been generated so far.
  - if no instruction satisfies the heuristics then either emit NOP (MIPS) or emit an instruction satisfying merely the final two static scheduling heuristics (SPARC).
  - remove the instruction from the DAG and insert the newly minimal elements into the candidate list.

On completion the basic block has been scheduled.

One little point which must be taken into account on non-interlocked hardware (e.g. MIPS) is that if any of the successor blocks of the just-scheduled block has already been generated then the first instruction of one of them might fail to satisfy timing constraints with respect to the final instruction of the newly generated block. In this case a NOP must be appended.

20 Antagonism Between Register Allocation and Instruction Scheduling

Register allocation by colouring attempts to minimise the number of store locations or registers used by a program. As such we would not be surprised to find that the generated code for

$$x := a; \quad y := b;$$

were to be

```
ld.w    a,r0
st.w    r0,x
ld.w    b,r0
st.w    r0,y
```

This code takes 6 cycles[9] to complete (on the SPARC there is an interlock delay between each load and store, on the MIPS a NOP must be inserted). According to the scheduling theory developed above, each instruction depends on its predecessor (def-def or def-use conflicts inhibit all permutations) this is the only valid execution sequence. However if the register allocator had allocated `r1` for the temporary copying `y` to `b`, the code could have been scheduled as

---

[9]Here I am counting time in pipeline step cycles, from start of the first `ld.w` instruction to the start of the instruction following the final `st.w` instruction.

```
ld.w    a,r0
ld.w    b,r1
st.w    r0,x
st.w    r1,y
```

which then executes in only 4 cycles.

For some time there was no very satisfactory theory as to how to resolve this (it is related to the 'phase-order problem' in which we would like to defer optimisation decisions until we know how later phases will behave on the results passed to them). The CRAIG system [1] is one exception, and 2002 saw Touati's thesis [8] "Register Pressure in Instruction Level Parallelism" which addresses a related issue.

One rather *ad hoc* solution is to allocate temporary registers cyclically instead of re-using them at the earliest possible opportunity. In the context of register allocation by colouring this can be seen as attempting to select a register distinct from all others allocated in the same basic block when all other constraints and desires (recall the MOV preference graph) have been taken into account.

This problem also poses dynamic scheduling problems in pipelines for corresponding 80x86 instruction sequences which need to reuse registers as much as possible because of their limited number. High-performance processors achieve effective dynamic rescheduling by having a larger register set in the computational engine than the potentially small instruction set registers and dynamically 'recolouring' live-ranges of such registers with the larger register set. This then achieves a similar effect to the above example in which the r0-r1 pair replaces the single r0, but without the need to tie up another user register.

Part D: Decompilation and Reverse Engineering

This final lecture considers the topic of *decompilation*, the inverse process to compilation whereby assembler (or binary object) files are mapped into one of the source files which could compile to the given assembler or binary object source.

Note in particular that compilation is a many-to-one process—a compiler may well ignore variable names and even compile `x<=9` and `x<10` into the same code. Therefore we are picking a *representative* program.

There are three issues which I want to address:

- The ethics of decompilation;

- Control structure reconstruction; and

- Variable and type reconstruction.

You will often see the phrase *reverse engineering* to cover the wider topic of attempting to extract higher-level data (even documentation) from lower-level representations (such as programs). Our view is that decompilation is a special case of reverse engineering. A site dedicated to reverse engineering is:

> `http://www.reengineer.org/`

Legality/Ethics

Reverse engineering of a software product is normally forbidden by the licence terms which a purchaser agrees to, for example on shrink-wrap or at installation. However, legislation (varying from jurisdiction to jurisdiction) often permits decompilation for very specific purposes. For example the EU 1991 Software Directive (a world-leader at the time) allows the *reproduction* and *translation* of the form of program code, without the consent of the owner, only for the purpose of achieving the interoperability of the program with some other program, and only if this reverse engineering is *indispensable* for this purpose. Newer legislation is being enacted, for example the US Digital Millennium Copyright Act which came into force in October 2000 has a "Reverse Engineering" provision which

> "... permits circumvention, and the development of technological means for such circumvention, by a person who has lawfully obtained a right to use a copy of a computer program for the sole purpose of identifying and analyzing elements of the program necessary to achieve interoperability with other programs, to the extent that such acts are permitted under copyright law."

Note that the law changes with time and jurisdiction, so do it where/when it is legal! Note also that copyright legislation covers "translations" of copyrighted text, which will certainly include decompilations even if permitted by contract or by overriding law such as the above.

A good source of information is the *Decompilation Page* [9] on the web

> `http://www.program-transformation.org/Transform/DeCompilation`

in particular the "Legality Of Decompilation" link in the introduction.

Control Structure Reconstruction

Extracting the flowgraph from an assembler program is easy. The trick is then to match *intervals of the flowgraph* with higher-level control structures, e.g. loops, if-the-else. Note that non-trivial compilation techniques like loop unrolling will need more aggressive techniques to undo. Cifuentes and her group have worked on many issues around this topic. See Cifuentes' PhD [10] for much more detail. In particular pages 123–130 are mirrored on the course web site

```
http://www.cl.cam.ac.uk/Teaching/current/OptComp/
```

Variable and Type Reconstruction

This is trickier than one might first think, because of register allocation (and even CSE). A given machine register might contain, at various times, multiple user-variables and temporaries. Worse still these may have different types. Consider

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

where a single register is used to hold x, a pointer, and the result from the function, an integer. Decompilation to

```
f(int r0) { r0 = r0+4; r0 = *(int *)r0; r0 = r0 + 2; return r0; }
```

is hardly clear. Mycroft uses transformation to SSA form to undo register colouring and then type inference to identify possible types for each SSA variable. See [11] via the course web site

```
http://www.cl.cam.ac.uk/Teaching/current/OptComp/
```

References

[1] T. Brasier, P. Sweany, S. Beaty and S. Carr. "CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment". *Proceedings of the 1995 International Conference on Parallel Architectures and Compiler Techniques (PACT 95)*, Limassol, Cyprus, June 1995. URL `ftp://cs.mtu.edu/pub/carr/craig.ps.gz`

[2] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.W. "Efficiently computing static single assignment form and the control dependence graph". *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, October 1991.

[3] Dean, J., Grove, D. and Chambers, C.", "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis". Proc. ECOOP'95, Springer-Verlag LNCS vol. 952, 1995. URL `http://citeseer.ist.psu.edu/89815.html`

[4] Ryder, B.G. "Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages" Proc. CC'03, Springer-Verlag LNCS vol. 2622, 2003. URL `http://www.prolangs.rutgers.edu/refs/docs/cc03.pdf`

[5] P. B. Gibbons and S. S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture". *ACM SIGPLAN 86 Symposium on Compiler Construction*, June 1986, pp. 11-16.

[6] J. Hennessy and T. Gross, "Postpass Code Optimisation of Pipeline Constraints". *ACM Transactions on Programming Languages and Systems*, July 1983, pp. 422-448.

[7] Johnson, N.E. and Mycroft, A. "Combined Code Motion and Register Allocation using the Value State Dependence Graph". Proc. CC'03, Springer-Verlag LNCS vol. 2622, 2003. URL `http://www.cl.cam.ac.uk/users/am/papers/cc03.ps.gz`

[8] Sid-Ahmed-Ali Touati, "Register Pressure in Instruction Level Parallelism". PhD thesis, University of Versailles, 2002. URL `http://www.prism.uvsq.fr/~touati/thesis.html`

[9] Cifuentes, C. et al. "The decompilation page". URL `http://www.program-transformation.org/Transform/DeCompilation`

[10] Cifuentes, C. "Reverse compilation techniques". PhD thesis, University of Queensland, 1994. URL `http://www.itee.uq.edu.au/~cristina/dcc.html` URL `http://www.itee.uq.edu.au/~cristina/dcc/decompilation_thesis.ps.gz`

[11] Mycroft, A. Type-Based Decompilation. Lecture Notes in Computer Science: Proc. ESOP'99, Springer-Verlag LNCS vol. 1576: 208–223, 1999. URL `http://www.cl.cam.ac.uk/users/am/papers/esop99.ps.gz`

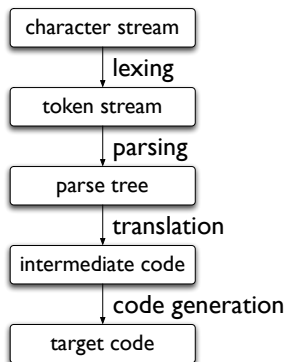[More sample papers for parts A and B need to be inserted to make this a proper bibliography.]

Optimising Compilers
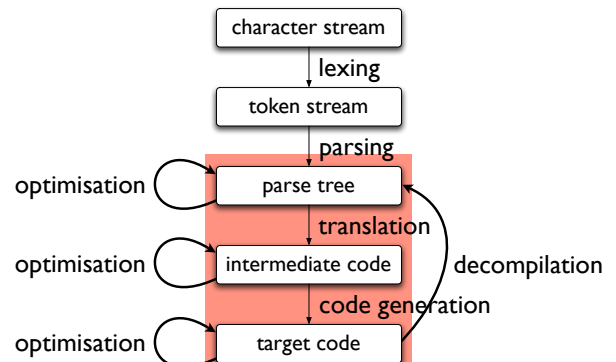
Computer Science Tripos Part II

Timothy Jones

Lecture 1
Introduction

A non-optimising compiler

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

An optimising compiler

```
            character stream
                  | lexing
            token stream
                  | parsing
optimisation → parse tree
                  | translation      ↘ decompilation
optimisation → intermediate code
                  | code generation
optimisation → target code
```

Optimisation
(really "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)

Optimisation
=
Analysis
+
Transformation

Analysis + Transformation

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

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.

Analysis + Transformation

```
int main(void)
{
  return 42;
}

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

Analysis + Transformation

```
int main(void)
{
  return 42;
}
```

✔

Analysis + Transformation

```
int main(void)
{
  return f(21);
}

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

Analysis + Transformation

```
int main(void)
{
  return f(21);
}
```

✘

Analysis + Transformation

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

Analysis + Transformation

```
int t = k * 2;
while (i <= t) {
  j = j * i;
  i = i + 1;
}
```
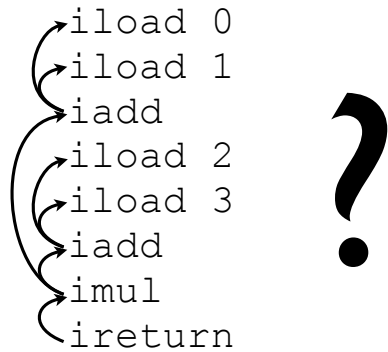
✔

Analysis + Transformation

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

Analysis + Transformation

```
int t = k * 2;
while (i <= t) {
  k = k - i;
  i = i + 1;
}
```

✘

43

Stack-oriented code

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

?

3-address code

```
MOV t32,arg1
MOV t33,arg2
ADD t34,t32,t33
MOV t35,arg3
MOV t36,arg4
ADD t37,t35,t36
MUL res1,t34,t37
EXIT
```

C into 3-address code

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

C into 3-address code

```
      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
```

Flowgraphs

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

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

Flowgraphs
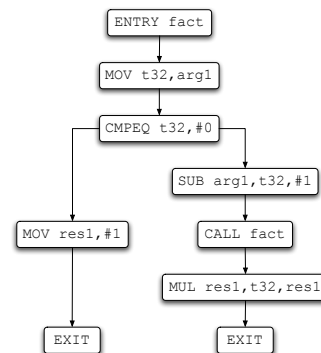
Basic blocks

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

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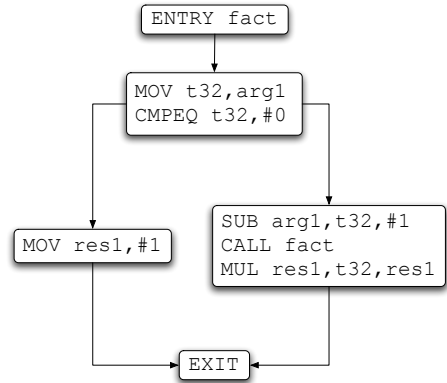- exactly one successor (except possibly for n_k)

Basic blocks

Basic blocks

```
ENTRY fact
MOV t32,arg1
CMPEQ t32,#0
```

```
MOV res1,#1
EXIT
```

```
SUB arg1,t32,#1
CALL fact
MUL res1,t32,res1
EXIT
```

Basic blocks

```
ENTRY fact
```

```
MOV t32,arg1
CMPEQ t32,#0
```

```
MOV res1,#1
```

```
SUB arg1,t32,#1
CALL fact
MUL res1,t32,res1
```

```
EXIT
```
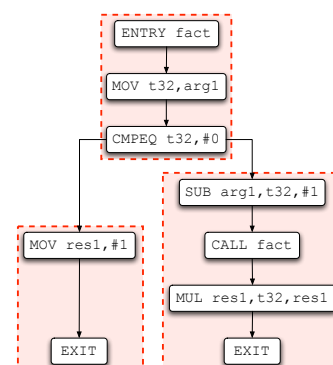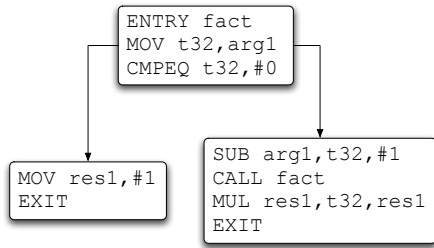
Basic blocks

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

Basic blocks

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.

Basic blocks
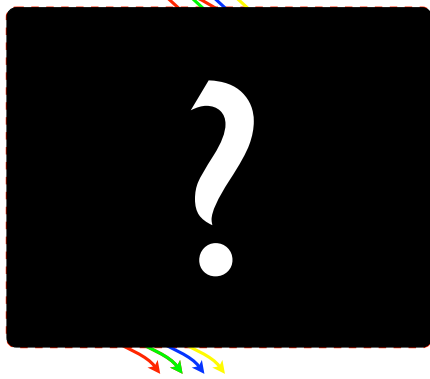
Basic blocks
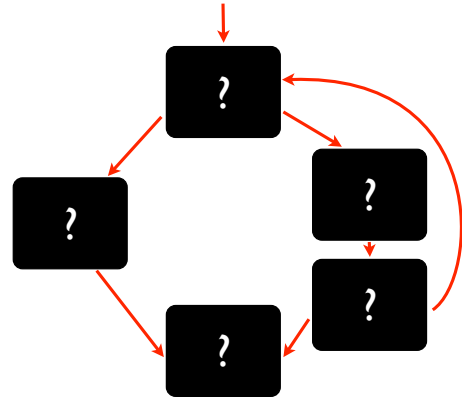
Types of analysis

(and hence optimisation)

Scope:

- 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

Peephole optimisation

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

matches

replace
```
MOV x,y
MOV y,x
```
with
```
MOV x,y
```

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

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.

Finding basic blocks

```
        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
```
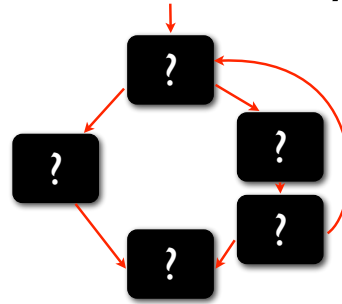
Summary

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

Lecture 2
Unreachable-code &
-procedure elimination

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

```
int f(int x, int y) {
    return x + y;
    int z = x * y;  UNREACHABLE
}
```

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

Dead vs. unreachable code

Deadness is a *data-flow* property:
"May this data ever arrive anywhere?"

```
int f(int x, int y) {
    int z = x * y;
    ⋮
```


Dead vs. unreachable code

Unreachability is a *control-flow* property:
"May control ever arrive here?"


```
    ⋮
    int z = x * y;
}
```

Safety of analysis

```
int f(int x, int y) {
    if (g(x)) {
        int z = x * y;  UNREACHABLE?
    }
    return x + y;
}

bool g(int x) {
    return false;  ✔
}
```

Safety of analysis

```
int f(int x, int y) {
  if (g(x)) {
    int z = x * y;   UNREACHABLE?
  }
  return x + y;
}

bool g(int x) {
  return ...x...;
}
```
?

Safety of analysis

```
int f(int x, int y) {
  if (g(x)) {
    int z = x * y;   UNREACHABLE?
  }
  return x + y;
}
```

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,

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

this instruction is reachable,

```
while (true) {
  // Code without 'break'
}
int z = x * y;
```

and so is this one.

Safety of analysis

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

Safety of analysis

```
     ⋮
   MOV t32,r1
   JMP lab1
     ⋮
lab1: ADD r0,r1,r2
     ⋮
```

```
⋮
MOV t32,r1
```
↓
```
ADD r0,r1,r2
⋮
```

Safety of analysis

```
⋮
MOV t33,#&lab1
MOV t34,#&lab2
MOV t35,#&lab3
⋮
JMPI t32
```

```
lab1: ADD r0,r1,r2
     ⋮
```
?
```
lab2: MUL r3,r4,r5
     ⋮
```
?
```
lab3: MOV r0,r1
     ⋮
```
?

Safety of analysis

```
MOV t33,#&lab1
MOV t34,#&lab2
MOV t35,#&lab3
⋮
```

```
ADD r0,r1,r2
⋮
```
```
MUL r3,r4,r5
⋮
```
```
MOV r0,r1
⋮
```
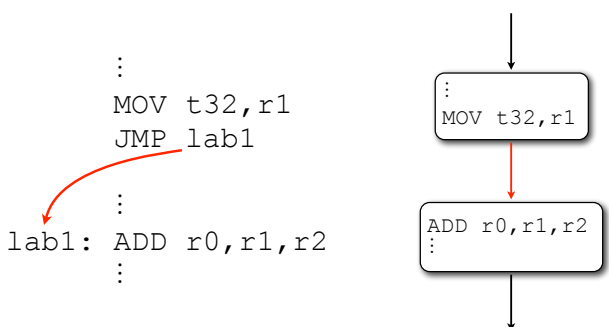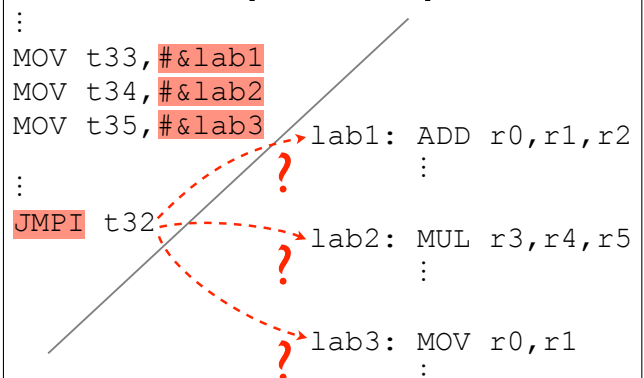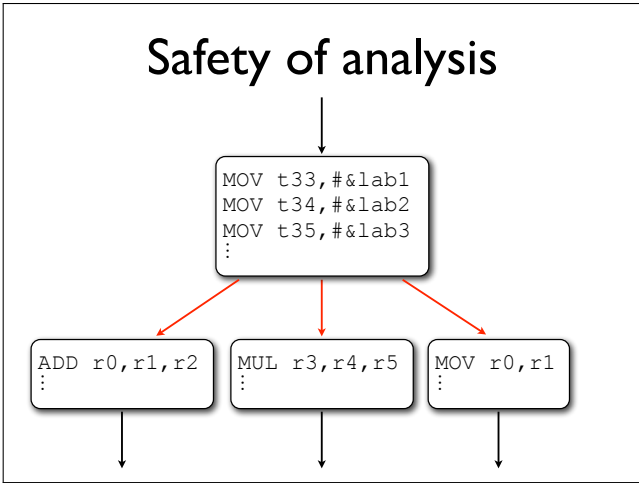
Safety of analysis

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.

Safety of analysis

program instructions

sometimes executed

never executed

Safety of analysis

"reachable"

imprecision
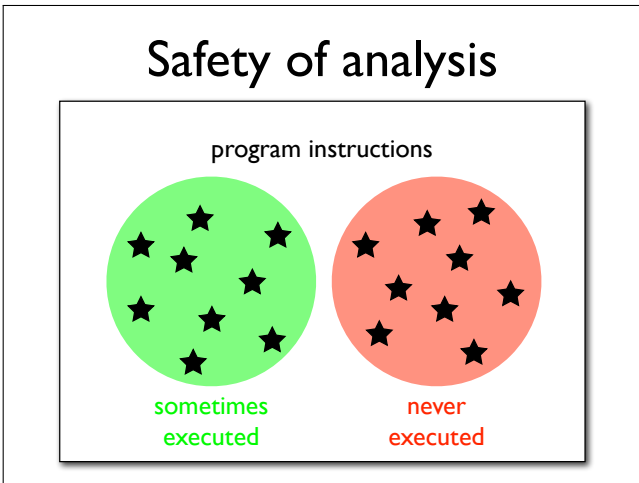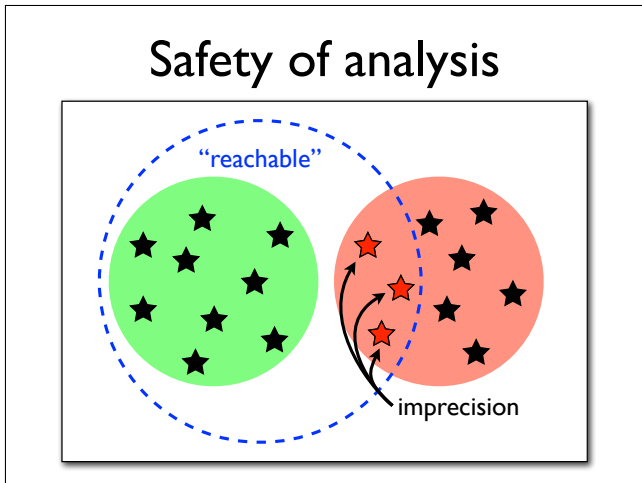
Unreachable code

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

ENTRY f

?

?

?

?

?

EXIT

Unreachable code

ENTRY f

?

?

EXIT
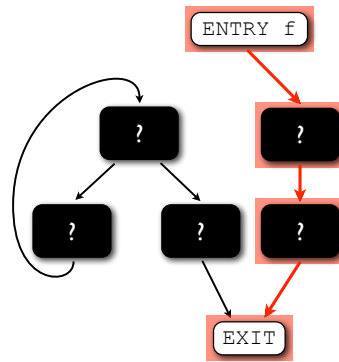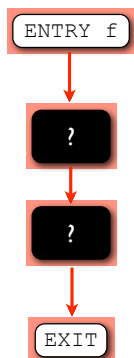
Unreachable code

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.

```
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*:

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

Unreachable code

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

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

Unreachable code

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.

Unreachable code

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:
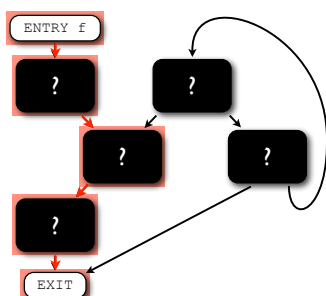
Unreachable code

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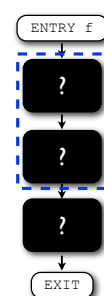
Unreachable code

For example, *straightening* is an optimisation which can eliminate jumps between basic blocks by coalescing them:
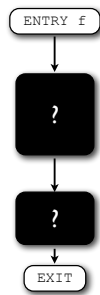
Straightening has removed a branch instruction, so the new program will execute faster.

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*.

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*.

Call graphs
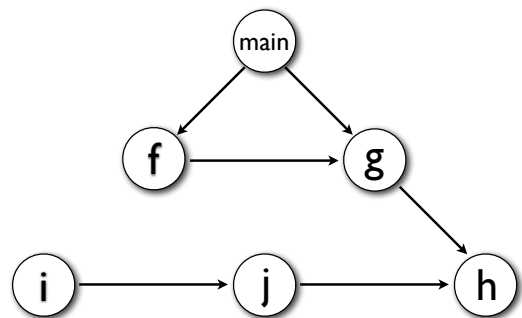
Call graphs

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

```
ENTRY main
⋮
MOV t33, #&f
MOV t34, #&g
MOV t35, #&h
⋮
CALLI t32
⋮
EXIT
```

```
ENTRY f
⋮
EXIT
```

```
ENTRY g
⋮
EXIT
```

```
ENTRY h
⋮
EXIT
```
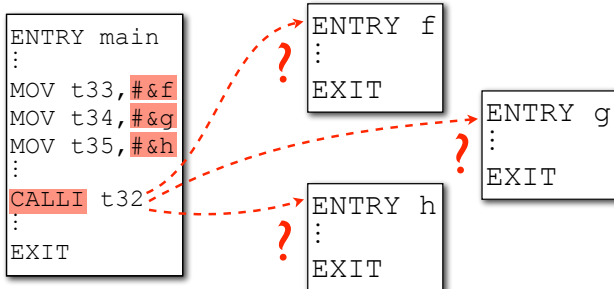
Call graphs

Again, the precision of the graph is compromised in the presence of *indirect calls*.
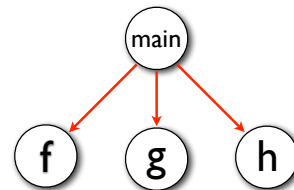
And as before, this is a *safe* overestimation of reachability.

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).

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.

Unreachable procedures
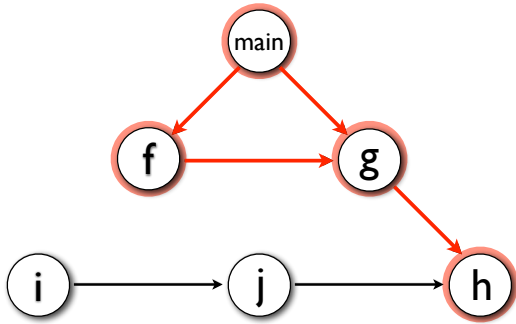
Unreachable procedures
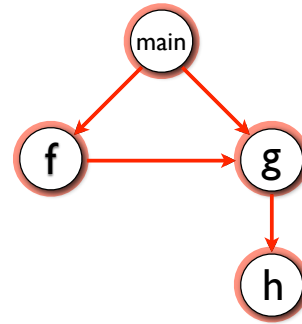
Safety of transformations

- 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 simplification

- Let's look at another set of basic control-flow transformations that can be carried out with only small amounts of analysis

- In this case, *if simplification*, which alters the structure of if statements (or removes them altogether) when possible

If simplication

Empty `then` in `if-then`

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

(Assuming that `f` has no side effects.)

If simplication

Empty `else` in `if-then-else`

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

If simplication

Empty `then` in `if-then-else`

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

If simplication

Empty `then` and `else` in `if-then-else`

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

If simplication

Constant condition

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

If simplication

Nested `if` with common subexpression

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

Loop simplification

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

Loop simplification

```
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

Data-flow analysis

```
MOV t32,arg1
MOV t33,arg2
ADD t34,t32,t33
MOV t35,arg3
MOV t36,arg4
ADD t37,t35,t36
MUL res1,t34,t37
```

Discovering information about how *data* (i.e. variables and their values) may move through a program.

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

Liveness is a data-flow property of variables:
"Is the value of this variable needed?" (cf. dead code)

```
int f(int x, int y) {
    int z = x * y;
    ⋮
```

? **?**
? **?**

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.

```
     ⋮
n: int z = x * y;      live(n) = { s, t, x, y }
   return s + t;
```

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*.

```
int x = y * z;    x LIVE
⋮
return 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*.

```
int x = y * z;    x DEAD
x = a + b;
⋮
return 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

A variable is *syntactically* live at a node if there is a path to the exit of the flowgraph along which its value may be used before it is redefined.

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 * y;    t DEAD
if ((x+1)*(x+1) == y) {
   t = 1;
}
if (x*x + 2*x + 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

```
      MUL t,x,y
      ADD t32,x,#1
      MUL t33,t32,t32
      CMPNE t33,y,lab1
      MOV t,#1
lab1: MUL t34,x,x
      MUL t35,x,#2
      ADD t36,t34,t35
      ADD t37,t36,#1
      CMPEQ t37,y,lab2
      MOV t,#2
lab2: MOV res1,t
```

Semantic vs. syntactic

t LIVE
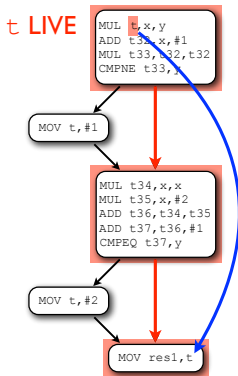
On *this* path through the flowgraph, `t` is not redefined before it's used, so `t` is *syntactically* live at the first instruction.

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

program variables

semantically live at *n*

semantically dead at *n*

Semantic vs. syntactic

syntactically live at *n*

imprecision
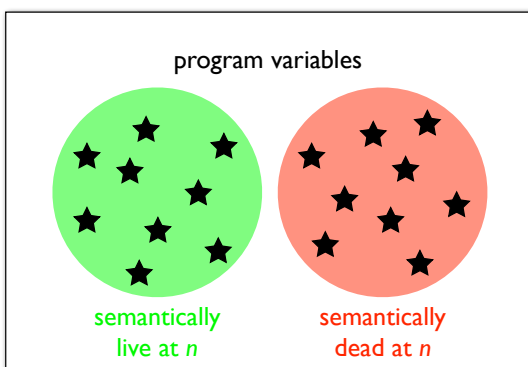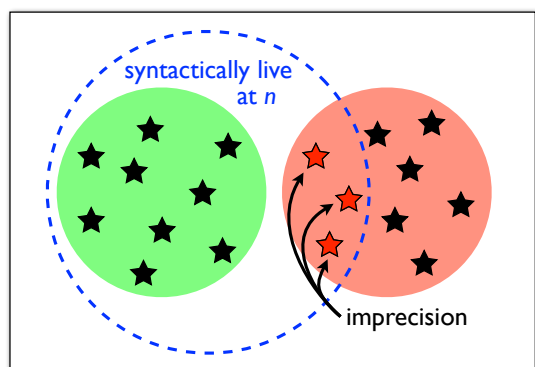
Semantic vs. syntactic

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

Using syntactic methods, we
safely overestimate liveness.

Live variable analysis

LVA is a *backwards* data-flow analysis: usage information
from *future* instructions must be propagated backwards
through the program to discover which variables are live.

```
int f(int x, int y) {
    int z = x * y;
    ⋮

                print z;
    int a = z*2;
            if (z > 5) {
```

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

```
      { b, c, e, f }
a = b * c;   REFERENCE b, c
      { e, f }
d = e + 1;   REFERENCE e
      { f }
print f;     REFERENCE f
      { }
```

Live variable analysis

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

Live variable analysis

```
      { }
a = 7;    DEFINE a
      { a }
b = 11;   DEFINE b
      { a, b }
c = 13;   DEFINE c
      { a, b, c }
```

Live variable analysis

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

ref(x = 3) = { }   *ref*(print x) = { x }
def(x = 3) = { x }   *def*(print x) = { }

ref(x = x + y) = { x, y }
def(x = x + y) = { x }

56

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).

$\{$ x, y $\}$

ref(print x) = $\{$ x $\}$

$\{$ y $\}$

$\{$ y $\}$

def(x = 3) = $\{$ x $\}$

$\{$ x, y $\}$
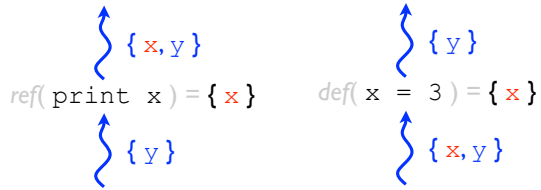
Live variable analysis

If an instruction both references and defines variables, we must remove the defined variables *before* adding the referenced ones.

$\{$ x, y, z $\}$

x = x + y

$\{$ x, z $\}$

def(x = x + y) = $\{$ x $\}$

ref(x = x + y) = $\{$ x, y $\}$
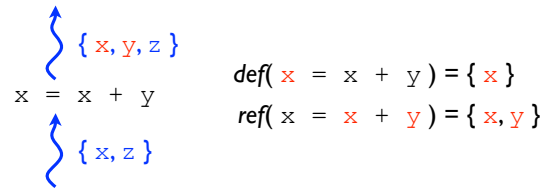
Live variable analysis

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

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n) \Big) \cup ref(n)$$

Live variable analysis

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n) \Big) \cup ref(n)$$

n: x = x + y

in-live(n) = $\big($*out-live(n)* $\setminus$ *def(n)*$\big)$ ∪ *ref(n)*
   = $(\{$ x, z $\} \setminus \{$ x $\})$ ∪ $\{$ x, y $\}$
   = $\{$ z $\}$ ∪ $\{$ x, y $\}$ = $\{$ x, y, z $\}$

out-live(n) = $\{$ x, z $\}$

def(n) = $\{$ x $\}$ *ref(n)* = $\{$ x, y $\}$
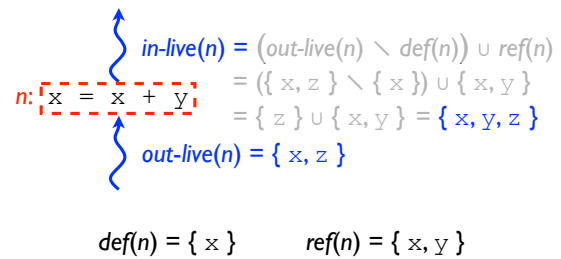
Live variable analysis

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

in-live(n) = $\big($*out-live(n)* $\setminus$ *def(n)*$\big)$ ∪ *ref(n)*

n: x = x + y

out-live(n) = ?
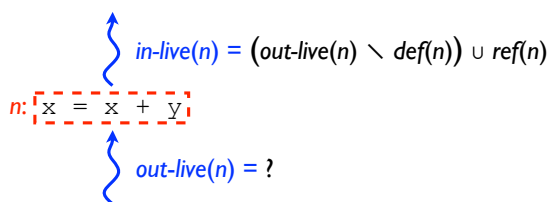
Live variable analysis

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

l:

out-live(l) = $\{$ s, t, x, y $\}$

in-live(m) = $\big($*out-live(m)* $\setminus$ *def(m)*$\big)$ ∪ *ref(m)*

m: z = x * y;

out-live(m) = $\{$ s, t, z $\}$

in-live(n) = $\big($*out-live(n)* $\setminus$ *def(n)*$\big)$ ∪ *ref(n)*

n: print s + t;

out-live(n) = $\{$ z $\}$

in-live(o) = $\big($*out-live(o)* $\setminus$ *def(o)*$\big)$ ∪ *ref(o)*

o:
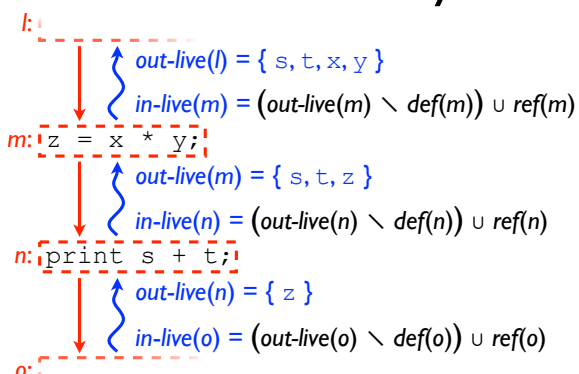
Live variable analysis

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.

57

Live variable analysis

m: `x = 17;`
{ x, z }
{ x, z }
n: `y = 19;`
$$\{\,x, z\,\} \cup \{\,y, z\,\}$$
$$= \{\,x, y, z\,\}$$
{ x, y, z }
{ x, z }   { y, z }
o: `s = x * 2;`   p: `t = y + 1;`
{ s, z }   { t, z }
{ s, z }   { t, z }
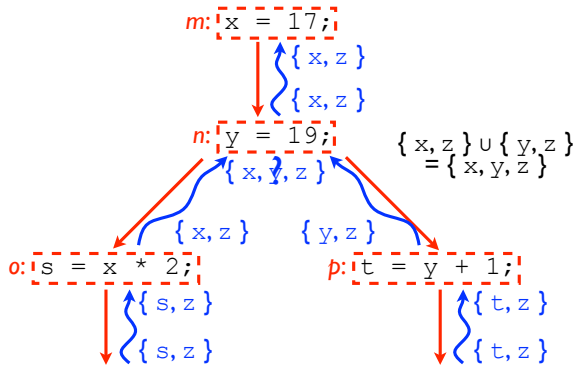
Live variable analysis

So the following equation must also hold:

$$out\text{-}live(n) = \bigcup_{s \in succ(n)} in\text{-}live(s)$$

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.

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n)\Big) \cup ref(n)$$

$$out\text{-}live(n) = \bigcup_{s \in succ(n)} in\text{-}live(s)$$

Data-flow equations

Each is expressed in terms of the other, so we can combine them to create one overall liveness equation.

$$live(n) = \left(\left(\bigcup_{s \in succ(n)} live(s)\right) \setminus def(n)\right) \cup ref(n)$$

Algorithm

We now have a formal description of liveness, but we need an actual algorithm in order to do the analysis.

Algorithm

"Doing the analysis" consists of computing a value *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

def x, y { }
def z { x, y }
ref x { x, y, z }
ref y { y, z }
ref z { z }

✗

Algorithm

def x, y { }
def z { x, y }
ref x { x, y, z }
ref y { x, y, z }
ref z { z }

✓
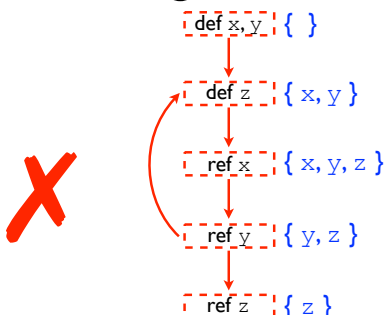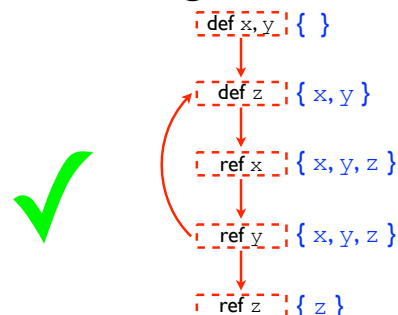
Algorithm

```
for i = 1 to n do live[i] := {}
while (live[] changes) do
  for i = 1 to n do
```

$$live[i] := \left(\left(\bigcup_{s \in succ(i)} live[s]\right) \setminus def(i)\right) \cup ref(i)$$

Algorithm

This algorithm is guaranteed to terminate since there are a *finite* number of variables in each program and the effect of one iteration is *monotonic*.

Furthermore, although any solution to the data-flow equations is safe, this algorithm is guaranteed to give the *smallest* (and therefore most precise) solution.

(See the *Knaster-Tarski theorem* if you're interested.)

Algorithm

Implementation notes:

- If the program has *n* variables, we can implement each element of `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 $i_1, ..., i_k$:

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

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.

Safety of analysis

```
    MOV x,#1
    MOV y,#2
    MOV z,#3
    MOV t32,#&x
    MOV t33,#&y
    MOV t34,#&z
     ⋮
m:  STI t35,#7    def(m) = { }
                  ref(m) = { t35 }
     ⋮
n:  LDI t36,t37   def(n) = { t36 }
                  ref(n) = { t37, x, y, z }
```

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

Lecture 4
Available expression analysis

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.

```
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?"

```
  ⋮
int z = x * y;
}
```
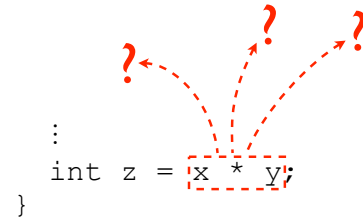
Availability

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*.

```
    int z = x * y;
    print s + t;
n:  int w = u / v;      avail(n) = { x*y, s+t }
      ⋮
```

Availability

So far, this is all familiar from live variable analysis.

Note that, while expression availability and variable liveness share many similarities (both are simple data-flow properties), they do differ in important ways.

By working through the low-level details of the availability property and its associated analysis we can see where the differences lie and get a feel for the capabilities of the general data-flow analysis framework.

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.

Semantic vs. syntactic

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

```
int x = y * z;
  ⋮
return y * z;   y*z AVAILABLE
```

Semantic vs. syntactic

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

```
int x = y * z;
⋮
y = a + b;
⋮
return y * z;   y*z UNAVAILABLE
```

Semantic vs. syntactic

An expression is *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*.

As before, semantic availability is concerned with the *execution behaviour* of the program, whereas syntactic availability is concerned with the program's *syntactic structure*.

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;   x+y AVAILABLE
```

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

Semantic vs. syntactic

```
        ADD   t32,x,#1
        MUL   t33,t32,t32
        CMPNE t33,y,lab1
        ADD   s,x,y
lab1:   MUL   t34,x,x
        MUL   t35,x,#2
        ADD   t36,t34,t35
        ADD   t37,t36,#1
        CMPEQ t37,y,lab2
        ADD   t,x,y
lab2:   ADD   res1,x,y
```

Semantic vs. syntactic
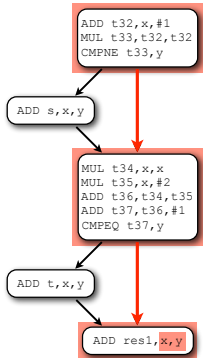
On *this* path through the flowgraph, x+y is only computed once, so x+y is *syntactically* unavailable at the last instruction.

Note that this path never actually occurs during execution.

x+y UNAVAILABLE

Semantic vs. syntactic

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

Semantic vs. syntactic

61
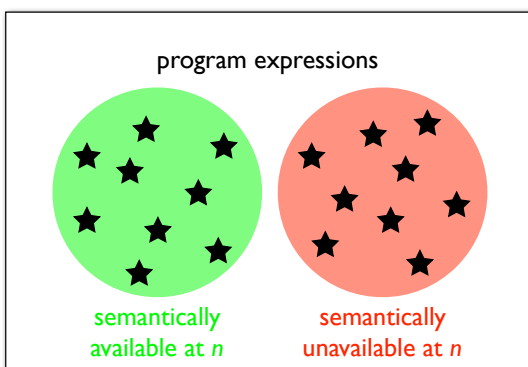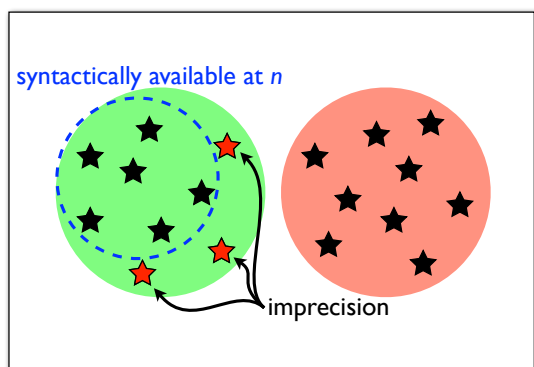
Semantic vs. syntactic

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

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

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.

```
              t = x * y;
  print x * y;            if (x*y > 0)

      :
      int z = x * y;
  }
```

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.

Available expression analysis

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

Available expression analysis

```
         { }
print a*b;   GENERATE a*b
         { a*b }
c = d + 1;   GENERATE d+1
         { a*b, d+1 }
e = f / g;   GENERATE f/g
         { a*b, d+1, f/g }
```

Available expression analysis

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

Available expression analysis

```
         { a*b, c+1, d/e, d-1 }
a = 7;      KILL a*b
         { c+1, d/e, d-1 }
c = 11;     KILL c+1
         { d/e, d-1 }
d = 13;  KILL d/e, d-1
         { }
```

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*.

Available expression analysis

So, in the following, E_x is the set of expressions in the program which contain occurrences of *x*.

gen(x = 3) = { }   *gen*(print x+1) = { x+1 }

kill(x = 3) = E_x *kill*(print x+1) = { }

gen(x = x + y) = { x+y }

kill(x = x + y) = E_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).

{ y+1 }   { x+1, y+1 }

gen(print x+1) = { x+1 }   *kill*(x = 3) = E_x

{ x+1, y+1 }   { y+1 }

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)*).

{ x+1, y+1 }

x = x + y

{ y+1 }

gen(x = x + y) = { x+y }

kill(x = x + y) = E_x

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:

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

Available expression analysis

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

in-avail(n) = { x+1, y+1 }

n: x = x + y

out-avail(n) = $\big(in\text{-}avail(n) \cup gen(n)\big) \setminus kill(n)$
= ({ x+1, y+1 } ∪ { x+y }) ∖ { x+1, x+y }
= { x+1, x+y, y+1 } ∖ { x+1, x+y } = { y+1 }

gen(n) = { x+y }   *kill(n)* = { x+1, x+y }

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)*.

in-avail(n) = ?

n: x = x + y

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

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

$$\{ \texttt{x+5} \} \qquad \{ \texttt{y-7} \}$$
$$\{ \texttt{x+5} \} \qquad \{ \texttt{y-7} \}$$
$$m: \texttt{z = x * y;} \qquad n: \texttt{print x*y;}$$
$$\{ \texttt{x+5,x*y} \} \qquad \{ \texttt{x*y,y-7} \}$$

$$\{ \texttt{x*y} \} \qquad \{ \texttt{x+5,x*y} \} \cap \{ \texttt{x*y,y-7} \}$$
$$o: \texttt{x = 11;} \qquad = \{ \texttt{x*y} \}$$
$$\{ \} $$
$$\{ \} $$
$$p: \texttt{y = 13;}$$
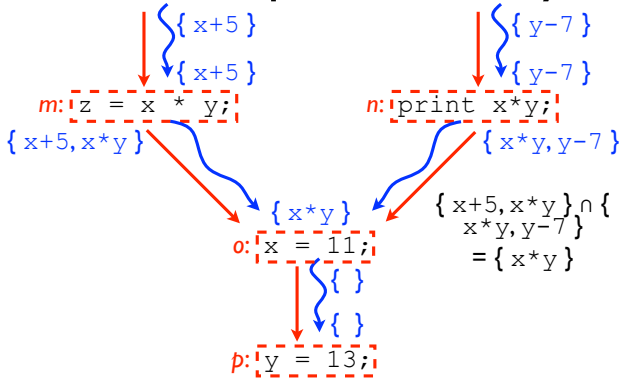
Available expression analysis

So the following equation must also hold:

$$in\text{-}avail(n) = \bigcap_{p \in pred(n)} out\text{-}avail(p)$$

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.

$$in\text{-}avail(n) = \bigcap_{p \in pred(n)} out\text{-}avail(p)$$

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

Data-flow equations

Each is expressed in terms of the other, so we can combine them to create one overall availability equation.

$$avail(n) = \bigcap_{p \in pred(n)} \Big((avail(p) \cup gen(p)) \setminus kill(p) \Big)$$

Data-flow equations

Danger: we have overlooked one important detail.

$$avail(n) = \bigcap_{p \in pred(n)} \big((avail(p) \cup gen(p)) \setminus kill(p) \big)$$

pred(n) = { }
n: x = 42;

$$= \bigcap \{ \}$$

$$= U \quad (\text{i.e. } \textit{all} \text{ expressions in the program})$$
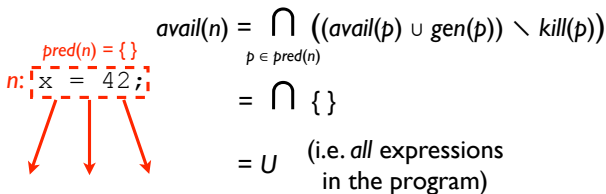
Clearly there should be *no* expressions available here, so we must stipulate explicitly that
$$avail(n) = \{ \} \text{ if } pred(n) = \{ \}.$$

Data-flow equations

With this correction, our data-flow equation for expression availability is

$$avail(n) = \begin{cases} \bigcap_{p \in pred(n)} \left((avail(p) \cup gen(p)) \setminus kill(p) \right) & \text{if } pred(n) \neq \{ \} \\ \{ \} & \text{if } pred(n) = \{ \} \end{cases}$$

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.

Data-flow equations

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n) \Big) \cup ref(n)$$

$$out\text{-}live(n) = \bigcup_{s \in succ(n)} in\text{-}live(s)$$

These differences are inherent in the analyses.

$$in\text{-}avail(n) = \bigcap_{p \in pred(n)} out\text{-}avail(p)$$

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

Data-flow equations

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n) \Big) \cup ref(n)$$

$$out\text{-}live(n) = \bigcup_{s \in succ(n)} in\text{-}live(s)$$

These differences are an arbitrary result of our definitions.

$$in\text{-}avail(n) = \bigcap_{p \in pred(n)} out\text{-}avail(p)$$

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

Data-flow equations

We might instead have decided to define *gen(n)* and *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 occuring 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:

gen(x = x + y) = { x+y }

kill(x = x + y) = E_x

By the new definition:

gen(x = x + y) = { }

kill(x = x + y) = E_x

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

Data-flow equations

Since these new definitions take account of which expressions are generated *overall* by a node (and exclude those which are generated only to be immediately killed), we may propagate availability information through a node by removing the killed expressions *before* adding the generated ones, *exactly as in LVA.*

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

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n) \Big) \cup ref(n)$$

Data-flow equations

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

$$avail(n) = \begin{cases} \bigcap_{p \in pred(n)} ((avail(p) \setminus kill(p)) \cup gen(p)) & \text{if } pred(n) \neq \{\} \\ \{\} & \text{if } pred(n) = \{\} \end{cases}$$

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

- We again use an array, `avail[]`, to store the available expressions for each node.

- We initialise `avail[]` such that each node has *all* expressions available (cf. LVA: *no* variables live).

- We again iterate application of the data-flow equation at each node until `avail[]` no longer changes.

Algorithm

```
for i = 1 to n do avail[i] := U
while (avail[] changes) do
  for i = 1 to n do
    avail[i] := ⋂    ((avail[p] \ kill(p)) ∪ gen(p))
              p∈pred(i)
```

Algorithm

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] := ⋂   ((avail[p] \ kill(p)) ∪ gen(p))
             p∈pred(i)
```

Algorithm

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

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 `avail[]` as an *n*-bit value, with the mth bit representing the availability of expression number *m*.

Algorithm

Implementation notes:

- Again, we can store availability once per basic block and recompute inside a block when necessary. Given each basic block *n* has k_n instructions $n[1], ..., n[k_n]$:

$$avail(n) = \bigcap_{p \in pred(n)} (avail(p) \setminus kill(p[1]) \cup gen(p[1]) \cdots \setminus kill(p[k_p]) \cup gen(p[k_p]))$$

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

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.

Analysis framework

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n) \Big) \cup ref(n)$$

$$out\text{-}live(n) = \bigcup_{s \in succ(n)} in\text{-}live(s)$$

$$in\text{-}avail(n) = \bigcap_{p \in pred(n)} out\text{-}avail(p)$$

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

Analysis framework

LVA's data-flow equations have the form

$$in(n) = (out(n) \setminus ...) \cup ... \qquad out(n) = \bigcup_{s \in succ(n)} in(s)$$

union over *successors*

AVAIL's data-flow equations have the form

$$out(n) = (in(n) \setminus ...) \cup ... \qquad in(n) = \bigcap_{p \in pred(n)} out(p)$$

intersection over *predecessors*

66

Analysis framework

	$\cap$	$\cup$
pred	AVAIL	RD
succ	VBE	LVA

...and others

Analysis framework

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.

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

Lecture 5
Data-flow anomalies and clash graphs

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.

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.

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.

Dead code

$$\vdots$$
```
a = x + 11;      { x, y, z }
                 ⬍
b = y + 13;      { a, y, z }
                 ⬍
c DEAD  c = a * b;   { a, b, z }
                 ⬍
⋮                { z }
print z;         { z }
                 ⬍
                 { }
```

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.

Dead code

$$\vdots$$
```
a = x + 11;      { x, y, z }
b = y + 13;      { y, z }
c = a * b;       { a, b, z }
                 { z }
⋮                { z }
print z;         { z }
                 { }
```
Successive iterations may yield further improvements.

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

In a "healthy" program, variable liveness produced by later instructions is consumed by earlier ones; if an instruction demands the value of a variable (hence making it live), it is expected that an earlier instruction will define that variable (hence making it dead again).

Uninitialised variables

```
                    { }
      x = 11;
                    { x }
      y = 13;
                    { x, y }
      z = 17;
                    { x, y }
        ⋮
                    { x, y }
      print x;
                    { y }
      print y;
                    { }
```
✔
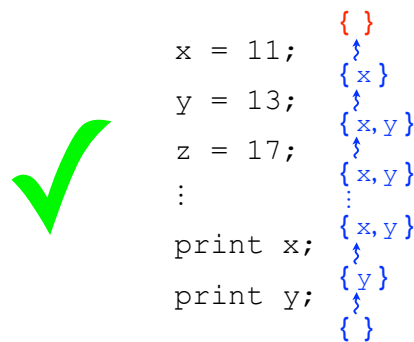
Uninitialised variables

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.

Uninitialised variables

```
                    { z }    z LIVE
      x = 11;
                    { x, z }
      y = 13;
                    { x, y, z }
        ⋮
                    { x, y, z }
      print x;
                    { y, z }
      print y;
                    { z }
      print z;
                    { }
```
✘
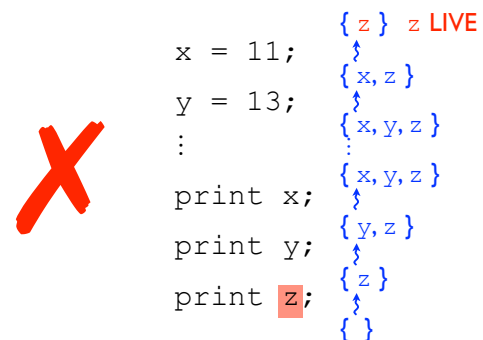
Uninitialised variables

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 }    x LIVE
      if (p) {
                    { }
        x = 42;
                    { x }
      }
                    { x }
        ⋮
                    { x }
      if (p) {
                    { x }
        print x;
                    { }
      }
                    { }
```
✘

Note: intentionally ignoring p!
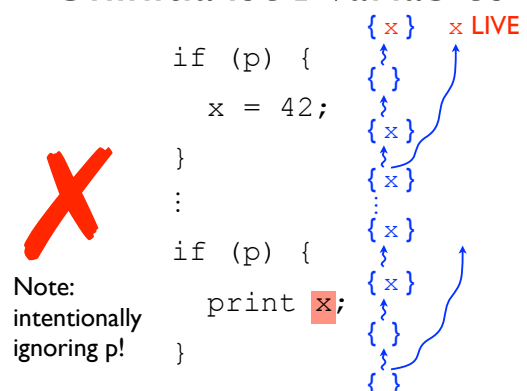
Uninitialised variables

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

Note that higher-level languages have the concept of (possibly nested) *scope*, and our expectations for variable initialisation in "healthy" programs can be extended to these.

In general we expect the set of live variables at the beginning of any scope to *not contain* any of the variables local to that scope.

Uninitialised variables

```
int x = 5;
int y = 7;
if (p) {
    int z;
    ⋮
    print z;
}
print x+y;
```

$\{ x, y, z \}$ z LIVE

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.

Write-write anomalies

$$in\text{-}wnr(n) = \bigcup_{p \in pred(n)} out\text{-}wnr(p)$$

$$out\text{-}wnr(n) = \Big(in\text{-}wnr(n) \setminus ref(n) \Big) \cup def(n)$$

$$wnr(n) = \bigcup_{p \in pred(n)} \Big((wnr(p) \setminus ref(p)) \cup def(p) \Big)$$

Write-write anomalies

y is also dead here.

y is rewritten here without ever having been read.

```
x = 11;
y = 13;
z = 17;
⋮
print x;
y = 19;
⋮
```

$\{\,\}$
$\{ x \}$
$\{ x, y \}$
$\{ x, y, z \}$
$\{ x, y, z \}$
$\{ y, z \}$
$\{ y, z \}$

Write-write anomalies

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;
if (p) {
    x = 13;
}
print x;
```

x is live throughout this code, but if p is true during execution, x will be written twice before it is read. In most cases, the programmer can remedy this.

Write-write anomalies

```
if (p) {
    x = 13;
} else {
    x = 11;
}
print x;
```

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

Write-write anomalies

```
if (p) {
    x = 13;
}
if (!p) {
    x = 11;
}
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.

Write-write anomalies

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.

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.

Clash graphs

```
x = (a*b) + c;
y = (a*b) + d;
```

lex, parse, translate

```
MUL t1,a,b
ADD x,t1,c
MUL t2,a,b
ADD y,t2,d
```

Clash graphs

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.

Clash graphs

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.

Clash graphs

```
x = 11;
y = 13;
z = (x+y) * 2;
a = 17;
b = 19;
z = z + (a*b);
```

Clash graphs

```
MOV x,#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
```

Clash graphs

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.

{ }
{ x }
{ x, y }
{ t1 }
{ z }
{ a, z }
{ a, b, z }
{ t2, z }

Clash graphs

{ }
{ x }
{ x, y }
{ t1 }
{ z }
{ a, z }
{ a, b, z }
{ t2, z }

Clash graphs


```
MOV a,#11
MOV b,#13
ADD a ,a,b
MUL z,a ,#2
MOV a,#17
MOV b,#19
MUL a ,a,b
ADD z,z,a
```
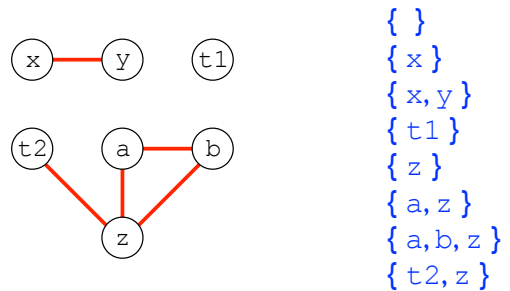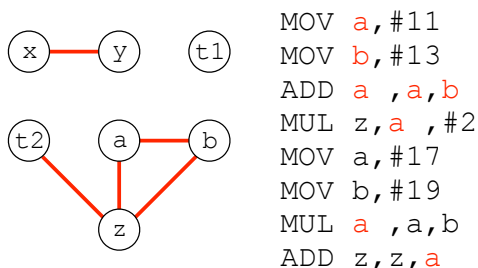
Summary

- 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

Lecture 6
Register allocation

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*.
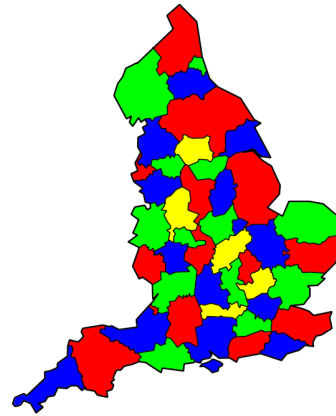
Graph colouring
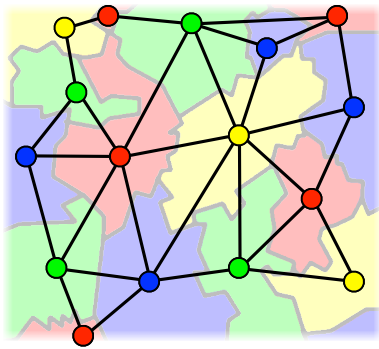
Graph colouring

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

Graph colouring

red
green
blue
yellow

Graph colouring

red
green
blue
yellow
purple
brown
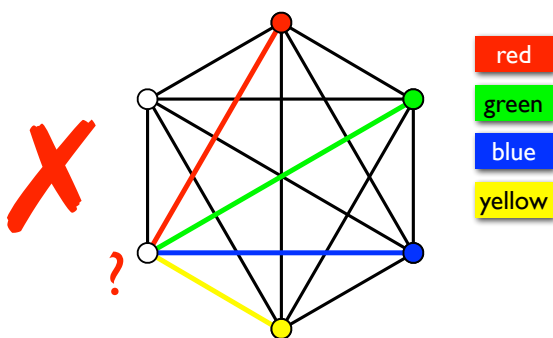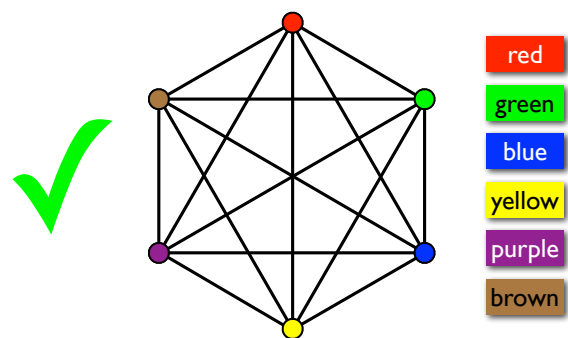
Allocation by colouring

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

```
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
```
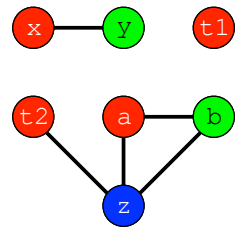

Algorithm

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.

Algorithm

- Choose a vertex (i.e. virtual register) which has the least number of incident edges (i.e. clashes).

- Remove the vertex and its edges from the graph, and push the vertex onto a LIFO stack.

- Repeat until the graph is empty.

- Pop each vertex from the stack and colour it in the most conservative way which avoids the colours of its (already-coloured) neighbours.

Algorithm

Algorithm

r0
r1
r2
r3
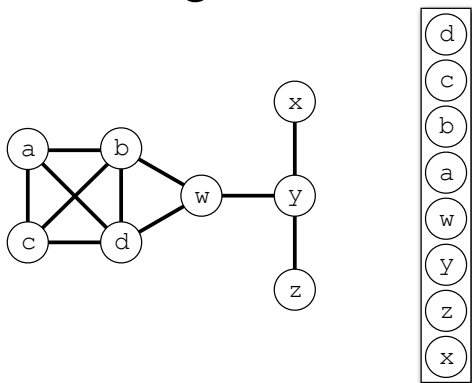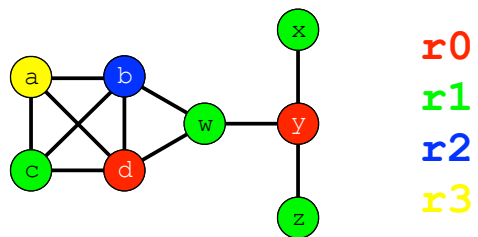
Algorithm

Bear in mind that this is only a heuristic.
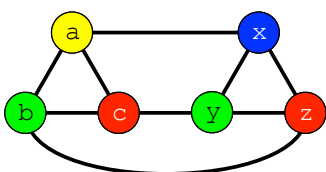
Algorithm

Bear in mind that this is only a heuristic.
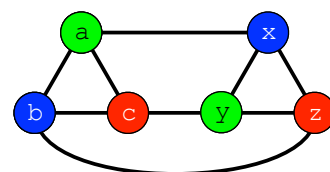
A better (more minimal) colouring may exist.

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.

Spilling

```
ADD a,b,c
```

vs.

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

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.

Algorithm

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

Algorithm

Algorithm

r0
r1

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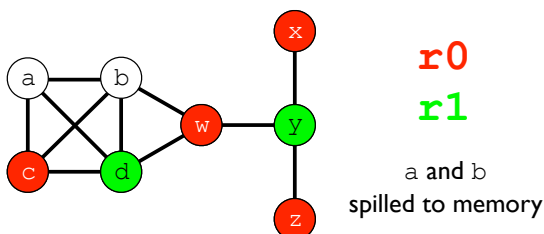
a: 3, b: 5, c: 7, d: 11, w: 13, x: 17, y: 19, z: 23

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.

75

Algorithm

"Slight lie": when spilling to memory, we (normally) need one free register to use as temporary storage for values loaded from and stored back into memory.

If any instructions operate on two spilled values simultaneously, we may need *two* such temporary registers to store both values.

So, in practise, when a spill is detected we may need to restart register allocation with one (or two) fewer architectural registers available so that these can be kept free for temporary storage of spilled values.

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.

Non-orthogonal instructions

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 MOVC3 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:

```
x = 19;
y = 23;
z = x + y;
```
→
```
MOV v32,#19
MOV v33,#23
MOV v1,v32
MOV v2,v33
ADD v0,v1,v2
MOV v34,v0
```

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.

preference graph

```
MOV v32,#19
MOV v33,#23
MOV v1,v32
MOV v2,v33
ADD v0,v1,v2
MOV v34,v0
```

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.

preference graph   clash graph

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.

```
MOV  r1,#19
MOV  r2,#23
MOV  r1,r1
MOV  r2,r2
ADD  r0,r1,r2
MOV  r0,r0
```
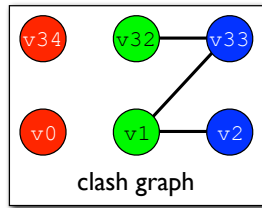
clash graph

Non-orthogonal instructions

And finally,

- When we know an instruction is going to corrupt the contents of an architectural register, we insert an edge on the clash graph between the corresponding virtual register and all other virtual registers live at that instruction — this prevents the register allocator from trying to store any live values in the corrupted register.

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
⋮
```
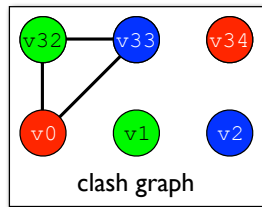
clash graph

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.

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 `MOV`s can be deleted again.

Procedure calling standards

```
x = 7;
y = 11;
z = 13;
a = f(x,y)+z;
```

```
MOV v32,#7
MOV v33,#11
MOV v34,#13
MOV v0,v32
MOV v1,v33
CALL f
MOV v35,v0
ADD v36,v34,v35
```

Procedure calling standards

```
MOV v32,#7
MOV v33,#11
MOV v34,#13
MOV v0,v32
MOV v1,v33
CALL f
MOV v35,v0
ADD v36,v34,v35
```

Procedure calling standards


```
MOV r0,#7
MOV r1,#11
MOV r4,#13
MOV r0,r0
MOV r1,r1
CALL f
MOV r0,r0
ADD r0,r4,r0
```
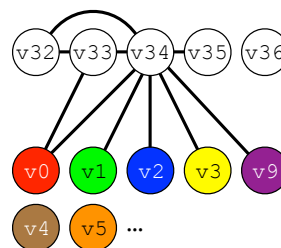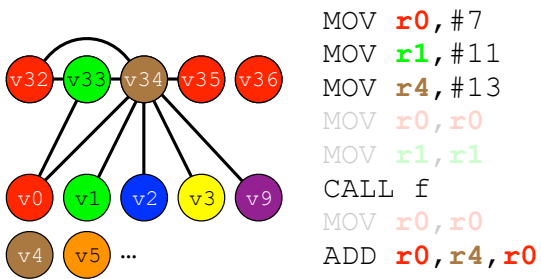
Summary

- 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

Lecture 7
Redundancy elimination

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

Common-subexpression elimination is a transformation which is enabled by available-expression analysis (AVAIL), in the same way as LVA enables dead-code elimination.

Since AVAIL discovers which expressions will have been computed by the time control arrives at an instruction in the program, we can use this information to spot and remove redundant computations.

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.

```
x = (a*b)+c;
⋮
print a * b;    a*b AVAILABLE
```

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

Algorithm


```
a = y * z      b = y * z      c = y * z
```

```
x = t      y*z AVAILABLE
```

Algorithm

```
t = y * z
a = t
```
```
t = y * z
b = t
```
```
t = y * z
c = t
```

```
x = t
```
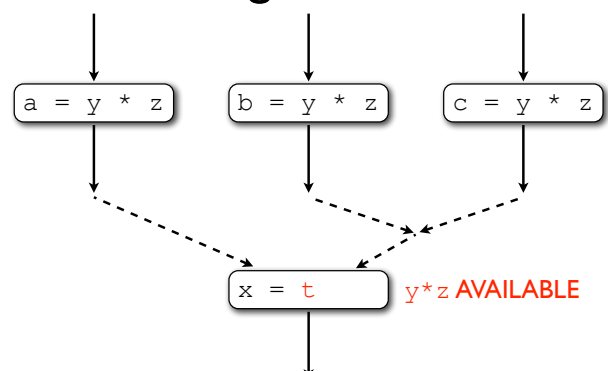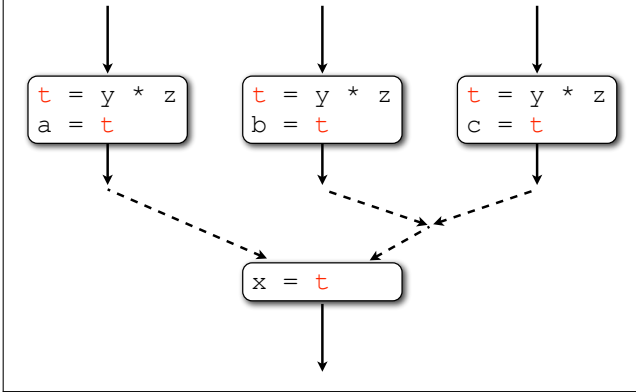
Common subexpressions

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.

Common subexpressions

The program might have "got worse" as a result of performing common-subexpression elimination.

In particular, introducing a new variable increases register pressure, and might cause spilling.

Memory loads and stores are much more expensive than multiplication of registers!

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
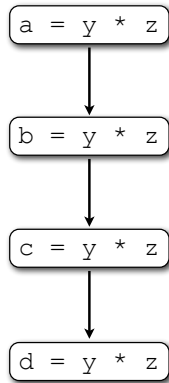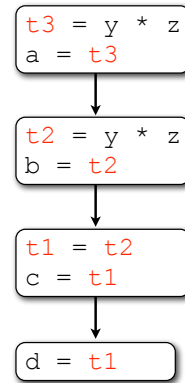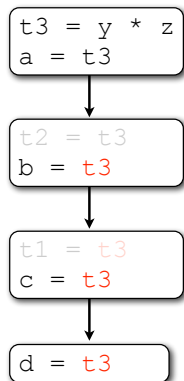
```
a = y * z
```
```
b = y * z
```
```
c = y * z
```
```
d = y * z
```

Copy propagation

```
t3 = y * z
a = t3
```
```
t2 = y * z
b = t2
```
```
t1 = t2
c = t1
```
```
d = t1
```

Copy propagation

```
t3 = y * z
a = t3
```
```
t2 = t3
b = t3
```
```
t1 = t3
c = t3
```
```
d = t3
```

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

x+y VERY BUSY

```
x = 19
y = 23
```

```
a = x + y
```

```
b = x + y
```

Code hoisting

```
x = 19
y = 23
t1 = x + y
```

```
a = t1
```

```
b = t1
```

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;
```

Loop-invariant code motion

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.

Partial redundancy

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

Partial redundancy

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

Putting it all together

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

Putting it all together

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

Putting it all together

```
ADD t3,x,y
MOV t2,t3
MOV t1,t2     COPIES OF t3
MOV a,t1
MOV b,t1
MOV r,z
```
```
ADD r,a,b          ADD s,a,b
MOV s,t2
```
```
ADD t,b,r
MOV u,t3
```

Putting it all together

```
ADD t3,x,y
MOV t2,t3
MOV t1,t3
MOV a,t3
MOV b,t3
MOV r,z
```
```
ADD r,a,b          ADD s,a,b
MOV s,t2
```
```
ADD t,b,r     COPIES OF t3
MOV u,t3
```

Putting it all together

```
ADD t3,x,y
MOV t2,t3     t1,t2 DEAD
MOV t1,t3
MOV a,t3
MOV b,t3
MOV r,z
```
```
ADD r,t3,t3          ADD s,t3,t3
MOV s,t3
```
```
ADD t,t3,r
MOV u,t3
```

Putting it all together

```
ADD t3,x,y
MOV a,t3
MOV b,t3
MOV r,z
```

t3+t3 VERY BUSY

```
ADD r,t3,t3
MOV s,t3
```

```
ADD s,t3,t3
```

```
ADD t,t3,r
MOV u,t3
```

Putting it all together

```
ADD t3,x,y
MOV a,t3
MOV b,t3
MOV r,z
ADD t4,t3,t3
```

a,b DEAD

```
MOV r,t4
MOV s,t3
```

```
MOV s,t4
```

```
ADD t,t3,r
MOV u,t3
```

Putting it all together

```
ADD t3,x,y
MOV r,z
ADD t4,t3,t3
```

```
MOV r,t4
MOV s,t3
```

```
MOV s,t4
```

```
ADD t,t3,r
MOV u,t3
```

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

Lecture 8
Static single assignment and strength reduction

Motivation

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*.

Live ranges

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.

Live ranges

```
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);
}
```

Live ranges

```
a = f(1);
b = f(2);        a, b CLASH
h(a,b);

b = f(3);
c = f(4);        b, c CLASH
h(b,c);

c = f(5);
a = f(6);        c, a CLASH
h(c,a);
```

Live ranges

```
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);
```
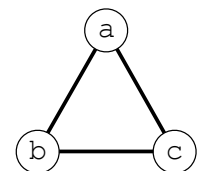

3 registers needed

Live ranges

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.

Live ranges

```
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);
}
```

Live ranges

```
a1 = f(1);
b2 = f(2);    a1,b2 CLASH
h(a1,b2);

b1 = f(3);
c2 = f(4);    b1,c2 CLASH
h(b1,c2);

c1 = f(5);
a2 = f(6);    c1,a2 CLASH
h(c1,a2);
```
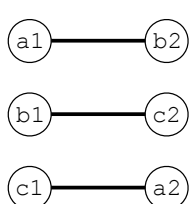
Live ranges

```
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);
```

(a1)——(b2)

(b1)——(c2)

(c1)——(a2)

2 registers needed

Static single-assignment

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.

Static single-assignment

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.

$$v_1 = 3;$$
$$v_2 = v_1 + 1;$$
$$v_3 = v_2 + w_1;$$
$$w_2 = v_3 + 2;$$

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.

Static single-assignment

```
v1 = 3;
```
```
v2 = v1 + 1;
v3 = v2 + w;
```
```
v4 = v1 - 1;
```
```
w = v * 2;
```
?

Static single-assignment

```
v1 = 3;
```
```
v2 = v1 + 1;
v3 = v2 + w;
```
```
v4 = v1 - 1;
```
```
v5 = φ(v3, v4);
w2 = v5 * 2;
```
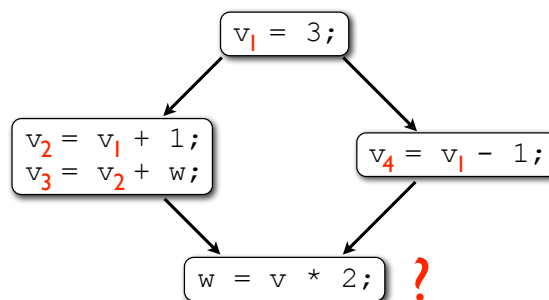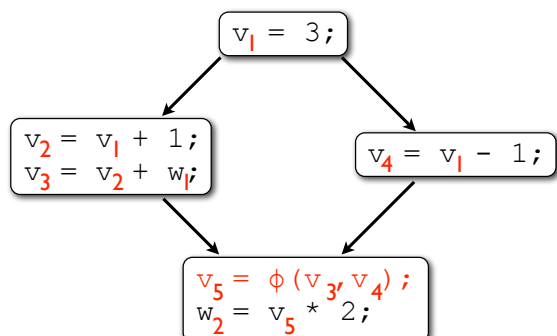
Static single-assignment

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).

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.

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

Higher-level optimisations
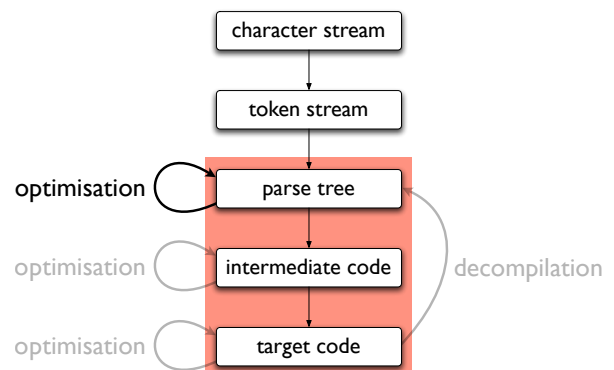
Higher-level optimisations

- 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

... e + 0 ...

↓

... e ...

... (e + n) + m ...

↓

... e + (n + m) ...

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.

Algebraic identities

In a lazy functional language,

```
let x = e in if e′ then … x … else e′′
```

if e′ then let x = e in … x … else e′′

provided e′ and e′′ do not contain x.

This is still *quite* boring.

Strength reduction

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.

Strength reduction

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.

Strength reduction

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

Strength reduction

```
int i; char *p;
for (i = 0; i < 100; i++)
{
   p = (char *)v + 4*i;
   p[0] = 0; p[1] = 0;
   p[2] = 0; p[3] = 0;
}
```

Strength reduction

```
int i; char *p;
for (i = 0; i < 100; i++)
{
   p = (char *)v + 4*i;
   p[0] = 0; p[1] = 0;
   p[2] = 0; p[3] = 0;
}
```

Strength reduction

```
int i; char *p;
p = (char *)v;
for (i = 0; i < 100; i++)
{
   p[0] = 0; p[1] = 0;
   p[2] = 0; p[3] = 0;
   p += 4;
}
```

Strength reduction

```
int i; char *p;
p = (char *)v;
for (i = 0; p < (char *)v + 400; i++)
{
   p[0] = 0; p[1] = 0;
   p[2] = 0; p[3] = 0;
   p += 4;
}
```

Strength reduction

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

Strength reduction

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

Strength reduction

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

Multiplication 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

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

- induction variable: $i = i \oplus c$

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

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

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.

Summary

- Live range splitting reduces register pressure
- In SSA form, each variable is assigned to only once
- SSA uses φ-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

Lecture 9
Abstract interpretation

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

Warning: this will be a heavily simplified view of abstract interpretation; there is only time to give a brief introduction to the ideas, not explore them with depth or rigour.

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.

Abstract interpretation

For example, to plan a trip, you might use a map.

- A road map sacrifices a lot of detail —
  - trees, road conditions, individual buildings;
  - an entire dimension —
- but it retains most of the information which is important for planning a journey:
  - place names;
  - roads and how they are interconnected.

Abstract interpretation

Crucially, a road map is a useful abstraction because the route you plan is probably still valid back in reality.

- A cartographer creates an abstraction of reality (a map),

- you perform some computation on that abstraction (plan a route),

- and then you transfer the result of that computation back into the real world (drive to your destination).

Abstract interpretation

Trying 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 -1515×37 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 $\otimes$.

Multiplying integers

In this example the magnitudes of the numbers are insignificant; we care only about their sign, so we can use this information to design our abstraction.

$$(-) = \{\, z \in \mathbb{Z} \mid z < 0 \,\}$$
$$(0) = \{\, 0 \,\}$$
$$(+) = \{\, z \in \mathbb{Z} \mid z > 0 \,\}$$

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

Multiplying integers

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

$\otimes$	(−)	(0)	(+)
(−)	(+)	(0)	(−)
(0)	(0)	(0)	(0)
(+)	(−)	(0)	(+)

Multiplying integers

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

$$abs(-1515) = (-)$$
$$abs(37) = (+)$$
$$(-) \otimes (+) = (-)$$

So, without doing any *concrete* computation, we have discovered that −1515 × 37 has a negative result.

Multiplying 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.

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*.

Safety

We consider a particular abstraction to be safe if, whenever a property is true in the abstract world, it must also be true in the concrete world.

Our multiplication example is actually quite precise, and therefore trivially safe: the magnitudes of the original integers are irrelevant, so when the abstraction says that the result of a multiplication will be negative, it definitely will be.

In general, however, abstractions will be more approximate than this.

Adding integers

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

$\oplus$	(−)	(0)	(+)
(−)	(−)	(−)	(?)
(0)	(−)	(0)	(+)
(+)	(?)	(+)	(+)

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: (?).
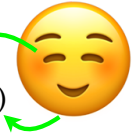
$$(-) = \{\, z \in \mathbb{Z} \mid z < 0 \,\}$$
$$(0) = \{\, 0 \,\}$$
$$(+) = \{\, z \in \mathbb{Z} \mid z > 0 \,\}$$
$$(?) = \mathbb{Z}$$

Adding integers

(?) 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

$abs(19 + 23) = abs(42) = (+)$

$abs(19) \oplus abs(23) = (+) \oplus (+) = (+)$

$$
\begin{array}{ccc}
19, 23 & \xrightarrow{\ +\ } & 42 \\
abs \downarrow & & \downarrow abs \\
(+), (+) & \xrightarrow[\oplus]{} & (+)
\end{array}
$$
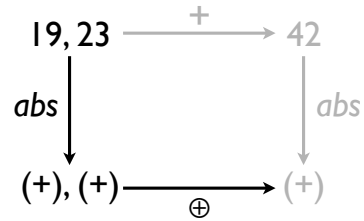
Adding integers

$abs(-1515 + 37) = abs(-1478) = (-)$

$abs(-1515) \oplus abs(37) = (-) \oplus (+) = (?)$

$$
\begin{array}{ccc}
-1515, 37 & \xrightarrow{\ +\ } & -1478 \\
abs \downarrow & & \downarrow abs \\
(-), (+) & \xrightarrow[\oplus]{} & \mathbf{(?)}
\end{array}
$$
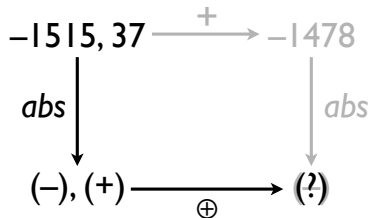
Safety

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 $\alpha : D \to D^{\#}$ (e.g. abs), and
- a concretisation function $\gamma : D^{\#} \to D$, e.g.:
  - $(-) \mapsto \{\, z \in \mathbb{Z} \mid z < 0 \,\}$,
  - $(0) \mapsto \{\, 0 \,\}$, etc.

Abstraction
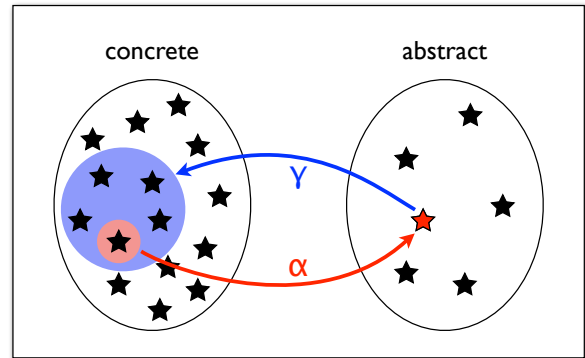
Abstraction

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

$$\hat{+}(\{\, 2, 5 \,\}, \{\, -3, -7 \,\}) = \{\, -1, -5, 2, -2 \,\}$$

$$\oplus((+), (-)) = (?)$$

Abstraction

So, for $D = \wp(\mathbb{Z})$ and $D^{\#} = \{\, (-), (0), (+), (?) \,\}$, we have:

$$
\begin{array}{ccc}
\mathbb{Z} \times \mathbb{Z} & \xrightarrow{\ +\ } & \mathbb{Z} \\
\downarrow & & \downarrow \\
D \times D & \xrightarrow{\ \hat{+}\ } & D \\
\alpha_1 \updownarrow \gamma_1 & & \alpha_2 \updownarrow \gamma_2 \\
D^{\#} \times D^{\#} & \xrightarrow{\ \oplus\ } & D^{\#}
\end{array}
$$
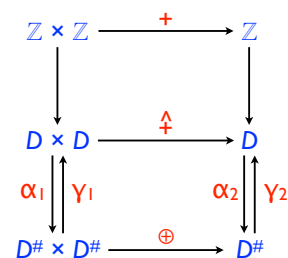
where $\alpha_{1,2}$ and $\gamma_{1,2}$ are the appropriate abstraction and concretisation functions.

Abstraction

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

Abstract interpretation can get very theoretical, but what's significant is the idea of using an abstraction to safely model reality.

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) ↦ if 3=0 then 4 else plus(3-1,4+1)
          ↦ plus(2,5)
          ↦ plus(1,6)
          ↦ plus(0,7)
          ↦ 7
```

Call-by-need evaluation

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

Using call-by-need:

```
plus(3,4) ↦ if 3=0 then 4 else plus(3-1,4+1)
          ↦ plus(3-1,4+1)
          ↦ plus(2-1,4+1+1)
          ↦ plus(1-1,4+1+1+1)
          ↦ 4+1+1+1
          ↦ 5+1+1
          ↦ 6+1
          ↦ 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.

Neededness

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

Neededness

In fact, neededness is too conservative:

λx,y,z. if x then y 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. if x then y else Ω

is strict in *x* and strict in *y*.

Strictness

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

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

$$e ::= x_i \mid A_i(e_1, \ldots, e_{r_i}) \mid F_i(e_1, \ldots e_{k_i})$$

where each A_i is a symbol representing a built-in (predefined) function of arity r_i.

94

Recursion equations

For our earlier example,

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

we can write the recursion equation

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

where $cond$, eq, 0, $sub1$ and $add1$ are built-in functions.

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: $\bot$.

Our concrete domain D is therefore $\mathbb{Z}_\bot = \mathbb{Z} \cup \{\, \bot \,\}$.

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:

$$
\begin{aligned}
cond(\bot, x, y) &= \bot \\
cond(0, x, y) &= y \\
cond(p, x, y) &= x \quad \text{otherwise} \\[6pt]
eq(\bot, y) &= \bot \\
eq(x, \bot) &= \bot \\
eq(x, y) &= x =_\mathbb{Z} y \quad \text{otherwise}
\end{aligned}
$$

Standard interpretation

The standard interpretation f_i of a user-defined function F_i is constructed from the built-in functions by composition and recursion according to its defining equation.

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

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 Ω), or *may* it eventually terminate and return a result?

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

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

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.

Abstract interpretation

x	y	$eq^\#(x,y)$
0	0	0
0	1	0
1	0	0
1	1	1

Abstract interpretation

p	x	y	$cond^\#(p,x,y)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Abstract interpretation

These functions may be expressed more compactly as boolean expressions, treating 0 and 1 from $D^\#$ as *false* and *true* respectively.

We can use Karnaugh maps (from IA DigElec) to turn each truth table into a simple boolean expression.

Abstract interpretation

$cond^\#$	0, 0	0, 1	1, 1	1, 0
0	0	0	0	0
1	0	1	1	1

$cond^\#(p, x, y) = (p \wedge y) \vee (p \wedge x)$

Abstract interpretation

$eq^\#$	0	1
0	0	0
1	0	1

$eq^\#(x, y) = x \wedge y$
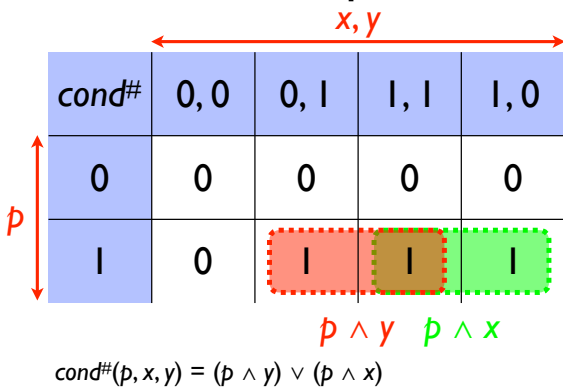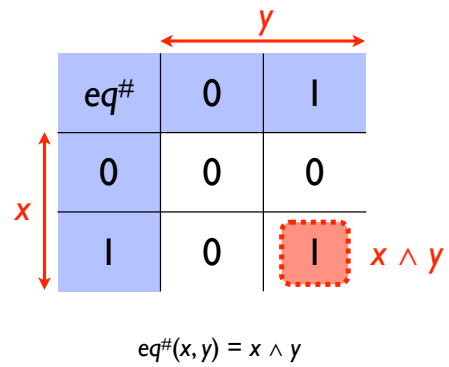
Strictness analysis

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.

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.

Strictness analysis

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:

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

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

$$plus^\#(x, y) = cond^\#(eq^\#(x, 0^\#), y, plus^\#(sub1^\#(x), add1^\#(y)))$$

Strictness analysis

$$plus^\#(x, y) = cond^\#(eq^\#(x, 0^\#), y, plus^\#(sub1^\#(x), add1^\#(y)))$$

We already know all the other strictness functions:

$$
\begin{aligned}
cond^\#(p, x, y) &= p \wedge (x \vee y) \\
eq^\#(x, y) &= x \wedge y \\
0^\# &= 1 \\
sub1^\#(x) &= x \\
add1^\#(x) &= x
\end{aligned}
$$

So we can use these to simplify the expression for *plus*$^\#$.

Strictness analysis

$$
\begin{aligned}
plus^\sharp(x,y) &= cond^\sharp(eq^\sharp(x,0^\sharp),y,plus^\sharp(sub1^\sharp(x),add1^\sharp(y))) \\
&= eq^\sharp(x,0^\sharp) \wedge (y \vee plus^\sharp(sub1^\sharp(x),add1^\sharp(y))) \\
&= eq^\sharp(x,1) \wedge (y \vee plus^\sharp(x,y)) \\
&= x \wedge 1 \wedge (y \vee plus^\sharp(x,y)) \\
&= x \wedge (y \vee plus^\sharp(x,y))
\end{aligned}
$$

Strictness analysis

$$
plus^\sharp(x,y) = x \wedge (y \vee plus^\sharp(x,y))
$$

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

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

Algorithm

```
for i = 1 to n do f#[i] := λx⃗.0
while (f#[] changes) do
  for i = 1 to n do
    f#[i] := λx⃗.eᵢ#
```

$e_i\#$ means "e_i (from the recursion equations) with each A_j replaced with $a_j\#$ and each F_j replaced with $f\#[j]$".

Algorithm

We have only one user-defined function, *plus*, and so only one recursion equation:

$$
plus(x,y) = cond(eq(x,0),y,plus(sub1(x),add1(y)))
$$

We initialise the corresponding element of our f#[] array to contain the always-0 strictness function of the appropriate arity:

$$
\texttt{f\#[1]} := \lambda x,y.\,0
$$

Algorithm

On the first iteration, we calculate $e_1\#$:

- The recursion equations say
  $e_1 = cond(eq(x,0),y,plus(sub1(x),add1(y)))$

- The current contents of f#[] say $f_1\#$ is $\lambda x,y.\,0$

- So:

$e_1\# = cond\#(eq\#(x,0\#),y,(\lambda x,y.\,0)\,(sub1\#(x),add1\#(y)))$

Algorithm

$e_1\# = cond\#(eq\#(x,0\#),y,(\lambda x,y.\,0)\,(sub1\#(x),add1\#(y)))$

Simplifying:
$e_1\# = cond\#(eq\#(x,0\#),y,0)$

Using definitions of $cond\#$, $eq\#$ and $0\#$:
$e_1\# = (x \wedge 1) \wedge (y \vee 0)$

Simplifying again:
$e_1\# = x \wedge y$

Algorithm

So, at the end of the first iteration,

$$
\texttt{f\#[1]} := \lambda x,y.\,x \wedge y
$$

Algorithm

On the second iteration, we recalculate $e_1\#$:

- The recursion equations still say
  $e_1 = cond(eq(x,0),y,plus(sub1(x),add1(y)))$

- The current contents of f#[] say $f_1\#$ is $\lambda x,y.\,x \wedge y$

- So:

$e_1\# = cond\#(eq\#(x,0\#),y,(\lambda x,y.\,x \wedge y)\,(sub1\#(x),add1\#(y)))$

Algorithm

$e_1^\# = cond^\#(eq^\#(x, 0^\#), y, (\lambda x, y.\; x \wedge y)\; (sub1^\#(x), add1^\#(y)))$

Simplifying:
$e_1^\# = cond^\#(eq^\#(x, 0^\#), y,\; sub1^\#(x) \wedge add1^\#(y))$

Using definitions of $cond^\#, eq^\#, 0^\#, sub1^\#$ and $add1^\#$:
$e_1^\# = (x \wedge 1) \wedge (y \vee (x \wedge y))$

Simplifying again:
$e_1^\# = x \wedge y$

Algorithm

So, at the end of the second iteration,
$$\texttt{f\#[1]} \;:=\; \lambda x, y.\; x \wedge y$$

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

Algorithm

$$plus^\#(x, y) = x \wedge y$$

Optimisation

So now, finally, we can see that

$$plus^\#(1, 0) = 1 \wedge 0 = 0$$
and
$$plus^\#(0, 1) = 0 \wedge 1 = 0$$

which means our concrete *plus* function is strict in its first argument and strict in its second argument: we may always safely use CBV when passing either.

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

Lecture 11
Constraint-based analysis

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 naïve strategy is very imprecise.

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

Imprecise control flow
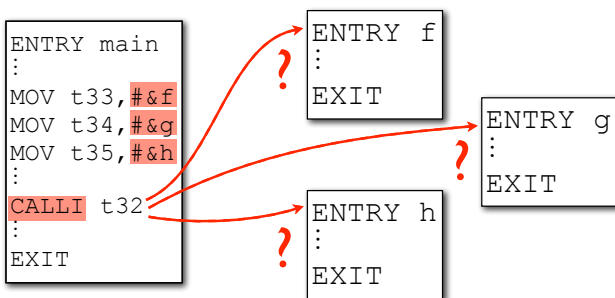
Constraint-based analysis

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:

$$\textit{in-live}(m) = (\textit{out-live}(m) \smallsetminus \textit{def}(m)) \cup \textit{ref}(m)$$
$$\textit{out-live}(m) = \textit{in-live}(n) \cup \textit{in-live}(o)$$
$$\textit{in-live}(n) = (\textit{out-live}(n) \smallsetminus \textit{def}(n)) \cup \textit{ref}(n)$$
$$\vdots$$

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

$$\text{let id} = \lambda x.\, x \text{ in id id } 7$$

Program points

$$\text{let id} = \lambda x.\, x \text{ in id id } 7$$
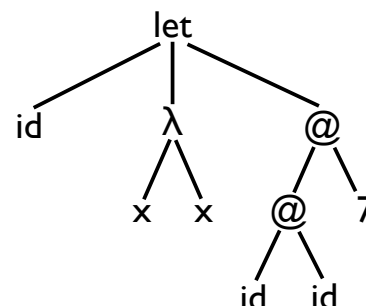
Program points

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$
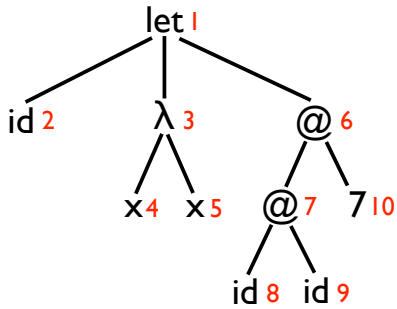
Program points

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

Each program point i has an associated *flow variable* α_i.

Each α_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 $(\lambda x^4. x^5)^3$.

Program points

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

The precise value of each α_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.

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$c^a \longrightarrow \alpha_a \supseteq \{\, c^a \,\}$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$7^{10} \longrightarrow \alpha_{10} \supseteq \{\, 7^{10} \,\}$

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$(\lambda x^a. e^b)^c \longrightarrow \alpha_c \supseteq \{\, (\lambda x^a. e^b)^c \,\}$

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$(\lambda x^4. x^5)^3 \longrightarrow \alpha_3 \supseteq \{\, (\lambda x^4. x^5)^3 \,\}$

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$
$\alpha_3 \supseteq \{\, (\lambda x^4. x^5)^3 \,\}$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$\lambda x^b. \dots x^a \dots$
$\text{let } x^b = \dots x^a \dots \quad \longrightarrow \quad \alpha_a \supseteq \alpha_b$

$\alpha_{10} \supseteq \{ 7^{10} \}$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$\lambda x^4. \dots x^5 \dots \quad \longrightarrow \quad \alpha_5 \supseteq \alpha_4$

$\text{let id}^2 = \dots \text{id}^8 \dots \quad \longrightarrow \quad \alpha_8 \supseteq \alpha_2$

$\text{let id}^2 = \dots \text{id}^9 \dots \quad \longrightarrow \quad \alpha_9 \supseteq \alpha_2$

$\alpha_{10} \supseteq \{ 7^{10} \} \qquad \alpha_8 \supseteq \alpha_2$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \} \qquad \alpha_9 \supseteq \alpha_2$
$\alpha_5 \supseteq \alpha_4$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$(\text{let } \_^a = \_^b \text{ in } \_^c)^d$

$\alpha_d \supseteq \alpha_c$
$\alpha_a \supseteq \alpha_b$

$\alpha_{10} \supseteq \{ 7^{10} \} \qquad \alpha_8 \supseteq \alpha_2$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \} \qquad \alpha_9 \supseteq \alpha_2$
$\alpha_5 \supseteq \alpha_4$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$(\text{let } \_^2 = \_^3 \text{ in } \_^6)^1$

$\alpha_1 \supseteq \alpha_6$
$\alpha_2 \supseteq \alpha_3$

$\alpha_{10} \supseteq \{ 7^{10} \} \qquad \alpha_8 \supseteq \alpha_2 \qquad \alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \} \qquad \alpha_9 \supseteq \alpha_2$
$\alpha_5 \supseteq \alpha_4 \qquad \alpha_1 \supseteq \alpha_6$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$(\_^a \ \_^b)^c \qquad (\alpha_b \mapsto \alpha_c) \supseteq \alpha_a$

$\alpha_{10} \supseteq \{ 7^{10} \} \qquad \alpha_8 \supseteq \alpha_2 \qquad \alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \} \qquad \alpha_9 \supseteq \alpha_2$
$\alpha_5 \supseteq \alpha_4 \qquad \alpha_1 \supseteq \alpha_6$

Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$(\_^8 \ \_^9)^7 \quad \longrightarrow \quad (\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$(\_^7 \ \_^{10})^6 \quad \longrightarrow \quad (\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_{10} \supseteq \{ 7^{10} \} \qquad \alpha_8 \supseteq \alpha_2 \qquad \alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \} \qquad \alpha_9 \supseteq \alpha_2 \qquad (\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4 \qquad \alpha_1 \supseteq \alpha_6 \qquad (\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
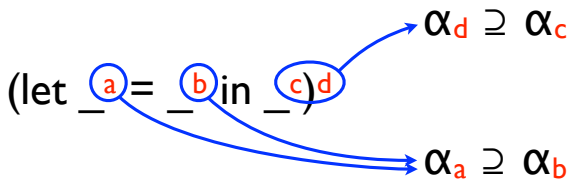
Generating constraints

$(\text{let id}^2 = (\lambda x^4. x^5)^3 \text{ in } ((\text{id}^8 \text{ id}^9)^7 \ 7^{10})^6)^1$

$\alpha_{10} \supseteq \{ 7^{10} \}$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$
$\alpha_5 \supseteq \alpha_4$
$\alpha_8 \supseteq \alpha_2$
$\alpha_9 \supseteq \alpha_2$

$\alpha_1 \supseteq \alpha_6$
$\alpha_2 \supseteq \alpha_3$
$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \} \qquad \alpha_8 \supseteq \alpha_2 \qquad \alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \} \qquad \alpha_9 \supseteq \alpha_2 \qquad (\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4 \qquad \alpha_1 \supseteq \alpha_6 \qquad (\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \} \qquad \alpha_6 = \{ \}$
$\alpha_2 = \{ \} \qquad \alpha_7 = \{ \}$
$\alpha_3 = \{ \} \qquad \alpha_8 = \{ \}$
$\alpha_4 = \{ \} \qquad \alpha_9 = \{ \}$
$\alpha_5 = \{ \} \qquad \alpha_{10} = \{ \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ \}$ $\alpha_7 = \{ \}$
$\alpha_3 = \{ \}$ $\alpha_8 = \{ \}$
$\alpha_4 = \{ \}$ $\alpha_9 = \{ \}$
$\alpha_5 = \{ \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ \}$ $\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ \}$
$\alpha_4 = \{ \}$ $\alpha_9 = \{ \}$
$\alpha_5 = \{ \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ \}$
$\alpha_4 = \{ \}$ $\alpha_9 = \{ \}$
$\alpha_5 = \{ \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_4 = \{ \}$ $\alpha_9 = \{ \}$
$\alpha_5 = \{ \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_4 = \{ \}$ $\alpha_9 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_5 = \{ \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_4 = \{ \}$ $\alpha_9 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_5 = \{ \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_5 = \{ \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\boxed{\alpha_7 \supseteq \alpha_5}$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_3 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $\boxed{(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7}$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_7 = \{ \boxed{(\lambda x^4.x^5)^3} \}$
$\alpha_3 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\boxed{\alpha_4 \supseteq \alpha_{10}}$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_3 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_9 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\alpha_4 \supseteq \alpha_{10}$ $\boxed{\alpha_6 \supseteq \alpha_5}$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_2 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_3 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_9 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\boxed{\alpha_5 \supseteq \alpha_4}$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\alpha_4 \supseteq \alpha_{10}$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_2 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_3 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_9 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\boxed{\alpha_7 \supseteq \alpha_5}$ $\alpha_4 \supseteq \alpha_{10}$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{ \}$ $\alpha_6 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_2 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$
$\alpha_3 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_9 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\boxed{\alpha_1 \supseteq \alpha_6}$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\alpha_4 \supseteq \alpha_{10}$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_6 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_2 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$
$\alpha_3 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_9 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4.x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\alpha_4 \supseteq \alpha_{10}$ $\boxed{\alpha_6 \supseteq \alpha_5}$

$\alpha_1 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_6 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$
$\alpha_2 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$
$\alpha_3 = \{ (\lambda x^4.x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_9 = \{ (\lambda x^4.x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4.x^5)^3, 7^{10} \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$ $\boxed{\alpha_1 \supseteq \alpha_6}$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\alpha_4 \supseteq \alpha_{10}$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{ (\lambda x^4 . x^5)^3, 7^{10} \}$ $\alpha_6 = \{ (\lambda x^4 . x^5)^3, 7^{10} \}$

$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4 . x^5)^3, 7^{10} \}$

$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$

$\alpha_4 = \{ (\lambda x^4 . x^5)^3, 7^{10} \}$ $\alpha_9 = \{ (\lambda x^4 . x^5)^3 \}$

$\alpha_5 = \{ (\lambda x^4 . x^5)^3, 7^{10} \}$ $\alpha_{10} = \{ 7^{10} \}$

Using solutions

$\alpha_{10} = \{ 7^{10} \}$

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

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

$(\text{let } id^2 = (\lambda x^4 . x^5)^3 \text{ in } ((id^8 \ id^9)^7 \ 7^{10})^6)^1$

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

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $\boxed{(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8}$
$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $\boxed{(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7}$
$\boxed{\alpha_4 \supseteq \alpha_9}$ $\alpha_7 \supseteq \alpha_5$ $\boxed{\alpha_4 \supseteq \alpha_{10}}$ $\alpha_6 \supseteq \alpha_5$

$\alpha_7 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$

$\alpha_9 = \{ (\lambda x^4 . x^5)^3 \}$ $\alpha_{10} = \{ 7^{10} \}$

$(\text{let } id^2 = (\lambda x^4 . x^5)^3 \text{ in } ((id^8 \ id^9)^7 \ 7^{10})^6)^1$

1CFA

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.

A better approximation is given by 1CFA ("first-order..."), in which a function has a separate flow variable for each call site in the program; this isolates separate calls to the same function, and so produces a more precise result.

1CFA

1CFA is a *polyvariant* approach.

Another alternative is to use a *polymorphic* approach, in which the values themselves are enriched to support specialisation at different call sites (cf. ML polymorphic types).

It's unclear which approach is "best".

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

Lecture 12
Inference-based analysis

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)
- Γ is a set of assumptions about free variables of e
- ϕ is a program property

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 Γ, the expression e has type t".

Type systems

We will avoid the more complicated ML typing issues (see Types course for details) and just consider the expressions in the lambda calculus:

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2$$

Our program properties are types *t*:

$$t ::= \alpha \mid int \mid t_1 \rightarrow t_2$$

Type systems

Γ 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 Γ with the additional assumption that *x* has type *t* (overriding any other assumption about *x*).

Type systems

In all inference systems, we use a set of *rules* to inductively define which judgements are valid.

In a type system, these are the *typing rules*.

Type systems

$$\frac{}{\Gamma[x:t] \vdash x:t} \quad (\textsc{Var})$$

$$\frac{\Gamma[x:t] \vdash e:t'}{\Gamma \vdash \lambda x.e : t \rightarrow t'} \quad (\textsc{Lam})$$

$$\frac{\Gamma \vdash e_1 : t \rightarrow t' \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : t'} \quad (\textsc{App})$$

Type systems

$\Gamma = \{\ 2 : int, add : int \rightarrow int \rightarrow int, multiply : int \rightarrow int \rightarrow int\ \}$

$e = \lambda x.\,\lambda y.\,add\ (multiply\ 2\ x)\ y$

$t = int \rightarrow int \rightarrow int$

$$\frac{\frac{\Gamma[x:int][y:int] \vdash add : int \rightarrow int \rightarrow int \quad \Gamma[x:int][y:int] \vdash multiply\ 2\ x : int}{\Gamma[x:int][y:int] \vdash add\ (multiply\ 2\ x) : int \rightarrow int} \quad \Gamma[x:int][y:int] \vdash y : int}{\frac{\Gamma[x:int][y:int] \vdash add\ (multiply\ 2\ x)\ y : int}{\frac{\Gamma[x:int] \vdash \lambda y.\,add\ (multiply\ 2\ x)\ y : int \rightarrow int}{\Gamma \vdash \lambda x.\,\lambda y.\,add\ (multiply\ 2\ x)\ y : int \rightarrow int \rightarrow int}}}$$

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.

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.

Odds and evens

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

This time, the program property φ has the form

$$\phi ::= odd \mid even \mid \phi_1 \rightarrow \phi_2$$

Odds and evens

$$\frac{}{\Gamma[x:\phi] \vdash x:\phi} \quad (\textsc{Var})$$

$$\frac{\Gamma[x:\phi] \vdash e:\phi'}{\Gamma \vdash \lambda x.e : \phi \rightarrow \phi'} \quad (\textsc{Lam})$$

$$\frac{\Gamma \vdash e_1 : \phi \rightarrow \phi' \quad \Gamma \vdash e_2 : \phi}{\Gamma \vdash e_1 e_2 : \phi'} \quad (\textsc{App})$$

Odds and evens

$\Gamma = \{\ 2 : even, add : even \rightarrow even \rightarrow even,$
$\qquad multiply : even \rightarrow odd \rightarrow even\ \}$

$e = \lambda x.\,\lambda y.\,add\ (multiply\ 2\ x)\ y$

$\phi = odd \rightarrow even \rightarrow even$

$$\frac{\frac{\Gamma[x:odd][y:even] \vdash add : even \rightarrow even \rightarrow even \quad \Gamma[x:odd][y:even] \vdash multiply\ 2\ x : even}{\Gamma[x:odd][y:even] \vdash add\ (multiply\ 2\ x) : even \rightarrow even} \quad \Gamma[x:odd][y:even] \vdash y : even}{\frac{\Gamma[x:odd][y:even] \vdash add\ (multiply\ 2\ x)\ y : even}{\frac{\Gamma[x:odd] \vdash \lambda y.\,add\ (multiply\ 2\ x)\ y : even \rightarrow even}{\Gamma \vdash \lambda x.\,\lambda y.\,add\ (multiply\ 2\ x)\ y : odd \rightarrow even \rightarrow even}}}$$

Safety

The safety condition for this inference system is

$$(\{\} \vdash e : \phi) \Rightarrow (\llbracket e \rrbracket \in \llbracket \phi \rrbracket)$$

where $\llbracket \phi \rrbracket$ is the denotation of ϕ:

$$\llbracket odd \rrbracket = \{ z \in \mathbb{Z} \mid z \text{ is odd} \},$$
$$\llbracket even \rrbracket = \{ z \in \mathbb{Z} \mid z \text{ is even} \},$$
$$\llbracket \phi_1 \rightarrow \phi_2 \rrbracket = \llbracket \phi_1 \rrbracket \rightarrow \llbracket \phi_2 \rrbracket$$

Richer properties

Note that if we want to show a judgement like

$$\Gamma \vdash \lambda x.\, \lambda y.\, add\ (multiply\ 2\ x)\ (multiply\ 3\ y) : even \rightarrow even \rightarrow even$$

we need more than one assumption about *multiply*:

$$\Gamma = \{ \dots, multiply : even \rightarrow even \rightarrow even,$$
$$multiply : odd \rightarrow even \rightarrow even, \dots \}$$

Richer properties

This might be undesirable, and one alternative is to enrich our properties instead; in this case we could allow *conjunction* inside properties, so that our single assumption about *multiply* looks like:

$$multiply : even \rightarrow even \rightarrow even \wedge$$
$$even \rightarrow odd \rightarrow even \wedge$$
$$odd \rightarrow even \rightarrow even \wedge$$
$$odd \rightarrow odd \rightarrow odd$$

We would need to modify the inference system to handle these richer properties.

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.

Side-effects

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

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \xi?x.e \mid \xi!e_1.e_2$$

- ξ represents some channel name.
- $\xi?x.e$ reads an integer from the channel named ξ, 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 ξ, and returns the result of evaluating e_2.

Side-effects

Some example expressions:

$$\xi?x.\ x$$
read an integer from channel ξ and return it

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

$$\xi?x.\ \zeta!x.\ x$$
read an integer from channel ξ, write it to channel ζ and return it

Side-effects

Ignoring their side-effects, the typing rules for these new operations are straightforward.

Side-effects

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

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

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

$$R_\xi$$ read from channel ξ

$$W_\xi$$ write to channel ξ

Effect systems

For example:

$$\xi?x.\ x \qquad F = \{\ R_\xi\ \}$$

$$\xi!x.\ y \qquad F = \{\ W_\xi\ \}$$

$$\xi?x.\ \zeta!x.\ x \qquad F = \{\ R_\xi, W_\zeta\ \}$$

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 W_ξ may occur *later*, whenever this newly-created function is applied.

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:

$$t ::= int \mid t_1 \xrightarrow{F} t_2$$

Effect systems

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

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

is

$$int \xrightarrow{\{\ W_\xi\ \}} int$$

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

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

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

Effect systems

$$\frac{\Gamma[x : int] \vdash e : t, F}{\Gamma \vdash \xi?x.e : t, \{R_\xi\} \cup F} \quad \text{(READ)}$$

$$\frac{\Gamma \vdash e_1 : int, F \quad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash \xi!e_1.e_2 : t, F \cup \{W_\xi\} \cup F'} \quad \text{(WRITE)}$$

Effect systems

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

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

$$\frac{\Gamma \vdash e_1 : t \xrightarrow{F''} t', F \quad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash e_1 e_2 : t', F \cup F' \cup F''} \quad \text{(APP)}$$

Effect systems

$$\frac{\dfrac{\{x : int, y : int\} \vdash x : int, \{\}}{\{x : int, y : int\} \vdash \xi!x.\, x : int, \{W_\xi\}}}{\dfrac{\{y : int\} \vdash \lambda x.\xi!x.\, x : int \xrightarrow{\{W_\xi\}} int, \{\}\quad \overline{\{y : int\} \vdash y : int, \{\}}}{\{y : int\} \vdash (\lambda x.\xi!x.\, x)\, y : int, \{W_\xi\}}}$$

Effect subtyping

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

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

$e ::= x \mid \lambda x.e \mid e_1\, e_2 \mid \xi?x.e \mid \xi!e_1.e_2 \mid$ if e_1 then e_2 else e_3

Effect subtyping

$$\frac{\Gamma \vdash e_1 : int, F \quad \Gamma \vdash e_2 : t, F' \quad \Gamma \vdash e_3 : t, F''}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t, F \cup F' \cup F''} \quad \text{(COND)}$$

Effect subtyping

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

Effect subtyping

$$int \xrightarrow{\{W_\xi\}} int \qquad\qquad int \xrightarrow{\{\}} int$$

if x then $\lambda x.\, \xi!3.\, x + 1$ else $\lambda x.\, x + 2$

$$\frac{\Gamma \vdash e_1 : int, F \quad \Gamma \vdash e_2 : t, F' \quad \Gamma \vdash e_3 : t, F''}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t, F \cup F' \cup F''} \quad \text{(COND)}$$

Effect subtyping

✗

$$int \xrightarrow{\{W_\xi\}} int \qquad\qquad int \xrightarrow{\{\}} int$$

if x then $\lambda x.\, \xi!3.\, x + 1$ else $\lambda x.\, x + 2$

Effect subtyping

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

Effect subtyping

$$\frac{\Gamma \vdash e : t \xrightarrow{F'} t', F \quad F' \subseteq F''}{\Gamma \vdash e : t \xrightarrow{F''} t', F} \quad (\textsc{Sub})$$

Effect subtyping

$$int \xrightarrow{\{W_\xi\}} int$$

$$(\textsc{Sub}) \Bigl\uparrow$$

$$int \xrightarrow{\{W_\xi\}} int \qquad int \xrightarrow{\{\}} int$$

if *x* then $\lambda x.\, \xi!3.\, x + 1$ else $\lambda x.\, x + 2$

Effect subtyping

✓

$$int \xrightarrow{\{W_\xi\}} int$$

$$int \xrightarrow{\{W_\xi\}} int \qquad int \xrightarrow{\{\}} int$$

if *x* then $\lambda x.\, \xi!3.\, x + 1$ else $\lambda x.\, x + 2$

Optimisation

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.

Safety

$$\bigl(\{\} \vdash e : t, F\bigr) \Rightarrow$$

$$\bigl(v \in [\![t]\!] \wedge f \subseteq F \text{ where } (v, f) = [\![e]\!]\bigr)$$

Extra structure

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.

$\xi?x.\, \zeta!x.\, x$	$F = \{\, R_\xi,\, W_\zeta\,\}$
$\zeta!y.\, \xi?x.\, x$	$F = \{\, R_\xi,\, W_\zeta\,\}$
$\zeta!y.\, \xi?x.\, \zeta!x.\, x$	$F = \{\, R_\xi,\, W_\zeta\,\}$

Extra structure

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.

Extra structure

$$\frac{\Gamma[x : int] \vdash e : t, F}{\Gamma \vdash \xi?x.e : t, \langle R_\xi \rangle @ F} \quad (\textsc{Read})$$

$$\frac{\Gamma \vdash e_1 : int, F \quad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash \xi!e_1.e_2 : t, F @ \langle W_\xi \rangle @ F'} \quad (\textsc{Write})$$

Extra structure

In the new system, these expressions all have different effects:

$$\xi?x.\ \zeta!x.\ x \qquad F = \langle\ R_\xi;\ W_\zeta\ \rangle$$

$$\zeta!y.\ \xi?x.\ x \qquad F = \langle\ W_\zeta;\ R_\xi\ \rangle$$

$$\zeta!y.\ \xi?x.\ \zeta!x.\ x \qquad F = \langle\ W_\zeta;\ R_\xi;\ W_\zeta\ \rangle$$

Extra structure

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

Lecture 13a
Alias and points-to analysis

Motivation

We've seen a number of different analyses that are affected by ambiguity in variables accessed (e.g. in LVA we assume all address-taken variables are referenced).

Alongside this, in modern machines we would like to exploit parallelism where possible, either by running code in separate threads on a multi-core, or in separate lanes using short vector (SIMD) instructions.

Our ability to do this depends on us being able to tell whether memory-access instructions alias (i.e. access the same memory location).

Example

As a simple example, consider some MP3 player code:

```
for (channel = 0; channel < 2; channel++)
  process_audio(channel);
```

Or even

```
process_audio_left();
process_audio_right();
```

Can we run these two calls in parallel?
In other words, when is it safe to do so?

Memory accessed

In general we can parallelise if neither call writes to a memory location read or written by the other.

We therefore want to know, at compile time, what memory locations a procedure might read from and write to at run time.

Essentially, we're asking what locations the procedure's instructions access at run time.

Memory accessed

We can reduce this problem to finding locations accessed by each instruction, then combining for all instructions within a procedure.

So, given a pointer value, we are interested in finding a *finite description* of the locations it *might* point to.

If two such descriptions have an empty intersection then we can parallelise / reorder the instructions / …

Andersen's analysis

Andersen's points-to analysis is an $O(n^3)$ analysis—the underlying problem is the same as 0-CFA.

We'll only look at the intra-procedural case.

We won't consider pointer arithmetic or functions returning pointers.

All object fields are conflated, although a 'field-sensitive' analysis is possible too.

Andersen's analysis

Assume the program has been re-written so that all *pointer-typed* operations are of the form:

$x := \text{new}_\ell$	ℓ is a program point
$x := \text{null}$	optional, a variant of `new`
$x := \&y$	C-like languages, also like `new`
$x := y$	copy
$x := *y$	field access of an object
$*x := y$	field access of an object

Andersen's analysis

We first define a set of abstract values:

$$V = Var \cup \{\text{new}_\ell \mid \ell \in Prog\} \cup \{\text{null}\}$$

Note that all allocations at program point ℓ are conflated, which makes things finite but loses precision.

We create the *points-to* relation as a function:

$$pt : V \to \mathcal{P}(V)$$

Some analyses have a different pt at each program point (like LVA); Andersen keeps one per function.

Andersen's analysis

We could use type-like constraints (one per source-level assignment):

$$\overline{\vdash x := \&y : y \in pt(x)} \qquad \overline{\vdash x := \texttt{null} : \texttt{null} \in pt(x)}$$

$$\overline{\vdash x := \texttt{new}_\ell : \texttt{new}_\ell \in pt(x)} \qquad \overline{\vdash x := y : pt(y) \subseteq pt(x)}$$

$$\frac{z \in pt(y)}{\vdash x := *y : pt(z) \subseteq pt(x)} \qquad \frac{z \in pt(x)}{\vdash *x := y : pt(y) \subseteq pt(z)}$$

Andersen's analysis

Or use the style of 0-CFA:

$$x \ := \ \& y$$

$$pt(x) \supseteq \{\, y \,\}$$

Andersen's analysis

Or use the style of 0-CFA:

$$x \ := \ y$$

$$pt(x) \supseteq pt(y)$$

Andersen's analysis

Or use the style of 0-CFA:

$$x \ := \ * y$$

$$pt(y) \supseteq \{\, z \,\} \implies pt(x) \supseteq pt(z)$$

Andersen's analysis

Or use the style of 0-CFA:

$$* x \ := \ y$$

$$pt(x) \supseteq \{\, z \,\} \implies pt(z) \supseteq pt(y)$$

Note that this is just stylistic, it's the same constraint system but no obvious deep connections between 0-CFA and Andersen's points-to analysis.

Andersen's analysis

The algorithm is flow-insensitive—it only considers the statements and not the order in which they occur. This is faster but less precise.

Property inference rules are then essentially:

$$(\text{ASS})\frac{}{\vdash x := e : \ldots} \qquad (\text{SEQ})\frac{\vdash C : S \quad \vdash C' : S'}{\vdash C; C' : S \cup S'}$$

$$(\text{COND})\frac{\vdash C : S \quad \vdash C' : S'}{\vdash \texttt{if } e \texttt{ then } C \texttt{ else } C' : S \cup S'}$$

$$(\text{WHILE})\frac{\vdash C : S}{\vdash \texttt{while } e \texttt{ do } C : S}$$

Andersen example

Consider the following code:

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

pt(a) = {}   pt(c) = {}
pt(b) = {}   pt(d) = {}
          pt(e) = {}

Andersen example

```
a = &b;
c = &d;
d = &a;      ⟶     pt(a) ⊇ { b }
e = c;
c = *e;
*a = d;
```

pt(a) = { b }   pt(c) = {}
pt(b) = {}   pt(d) = {}
              pt(e) = {}

Andersen example

```
a = &b;
c = &d;
d = &a;      ⟶     pt(c) ⊇ { d }
e = c;
c = *e;
*a = d;
```

pt(a) = { b }   pt(c) = { d }
pt(b) = {}   pt(d) = {}
              pt(e) = {}

Andersen example

```
a = &b;
c = &d;
d = &a;      ⟶     pt(d) ⊇ { a }
e = c;
c = *e;
*a = d;
```

pt(a) = { b }   pt(c) = { d }
pt(b) = {}   pt(d) = { a }
              pt(e) = {}

Andersen example

```
a = &b;
c = &d;
d = &a;      ⟶     pt(e) ⊇ pt(c)
e = c;
c = *e;
*a = d;
```

pt(a) = { b }   pt(c) = { d }
pt(b) = {}   pt(d) = { a }
              pt(e) = { d }

Andersen example

```
a = &b;
c = &d;
d = &a;      ⟶     pt(e) ⊇ { d }
e = c;                 ⟹ pt(c) ⊇ pt(d)
c = *e;
*a = d;
```

pt(a) = { b }   pt(c) = { a, d }
pt(b) = {}   pt(d) = { a }
              pt(e) = { d }

Andersen example

```
a = &b;
c = &d;
d = &a;      ⟶     pt(a) ⊇ { b }
e = c;                 ⟹ pt(b) ⊇ pt(d)
c = *e;
*a = d;
```

pt(a) = { b }   pt(c) = { a, d }
pt(b) = { a }   pt(d) = { a }
              pt(e) = { d }

Andersen example

```
a = &b;
c = &d;
d = &a;      ⟶     pt(e) ⊇ pt(c)
e = c;
c = *e;
*a = d;
```

pt(a) = { b }   pt(c) = { a, d }
pt(b) = { a }   pt(d) = { a }
              pt(e) = { a, d }

Andersen example

```
a = &b;
c = &d;
d = &a;      ⟶     pt(e) ⊇ { a, d }
e = c;                 ⟹ pt(c) ⊇ pt(a)
c = *e;                 ⟹ pt(c) ⊇ pt(d)
*a = d;
```

pt(a) = { b }   pt(c) = { a, b, d }
pt(b) = { a }   pt(d) = { a }
              pt(e) = { a, d }

Andersen example

```
a = &b;
c = &d;
d = &a;          →    pt(e) ⊇ pt(c)
e = c;
c = *e;
*a = d;
```

pt(a) = { b }   pt(c) = { a, b, d }
pt(b) = { a }   pt(d) = { a }
pt(e) = { a, b, d }

Andersen example

pt(a) = { b }   pt(c) = { a, b, d }
pt(b) = { a }   pt(d) = { a }
pt(e) = { a, b, d }

Note that a flow-sensitive algorithm would give
pt(c) = { a, d } and pt(e) = { d }
assuming the statements appear in the given
order in a single basic block that is not in a loop.

Other approaches

Steensgaard's algorithm merges a and b if any pointer
can reference both. This is less accurate but runs in
almost linear time.

Shape analysis (Sagiv, Wilhelm, Reps) models abstract
heap nodes and edges between them representing *must*
or *may* point-to. More accurate but the abstract heaps
can get very large.

In general, coarse techniques give poor results whereas
more sophisticated techniques are expensive.

Summary

- Points-to analysis identifies which memory
  locations variables (and other memory locations)
  point to

- We can use this information to improve data-
  dependence analysis

- This allows us to reorder loads and stores, or
  parallelise / vectorise parts of the code

- Andersen's analysis is a flow-insensitive algorithm
  that works in $O(n^3)$

Lecture 14
Instruction scheduling

Part C
Instruction scheduling

Instruction scheduling

```
character stream
      ↓
  token stream
      ↓
   parse tree
      ↓
intermediate code
      ↓
  target code
```

optimisation

optimisation

optimisation

decompilation
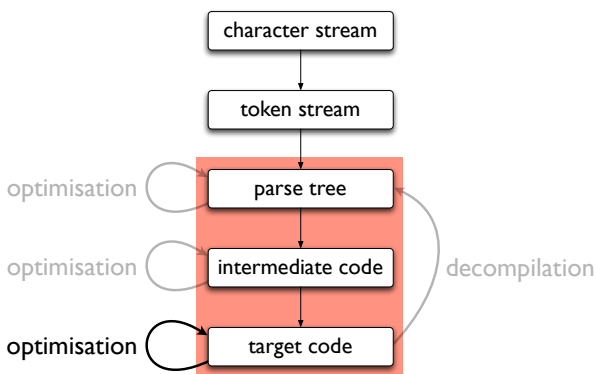
Motivation

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.

Single-cycle implementation

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:

Instruction fetch (IF)	Register fetch (RF)	Execute (EX)	Memory access (MEM)	Register write-back (WB)

For example, a load instruction uses all five.

Single-cycle implementation

lw $1,0($0)	lw $2,4($0)	lw $3,8($0)
IF RF EX MEM WB	IF RF EX MEM WB	IF RF EX MEM WB
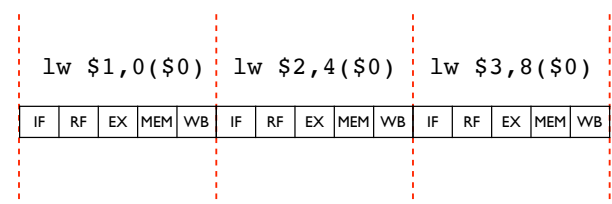
Single-cycle implementation

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 naïvely 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

```
lw $1,0($0)   | IF | RF | EX | MEM | WB |
lw $2,4($0)        | IF | RF | EX | MEM | WB |
lw $3,8($0)             | IF | RF | EX | MEM | WB |
```
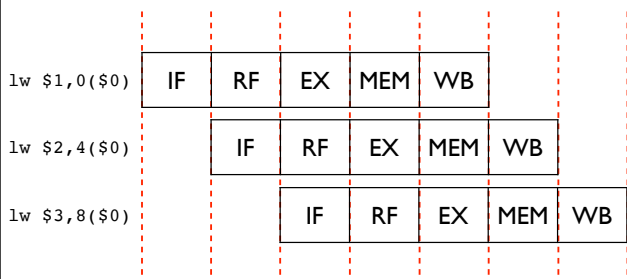
Pipelined implementation

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.

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

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

Pipeline hazards

```
add $3,$1,$2   | IF | RF | EX | MEM | WB |
add $5,$3,$4        |  STALL  | IF | RF | EX |
```
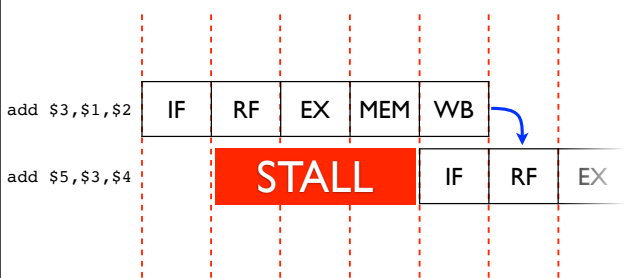
Pipeline hazards

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.

Pipeline hazards

```
add $3,$1,$2   | IF | RF | EX | MEM | WB |
add $5,$3,$4        | IF | RF | EX | MEM | WB |
```
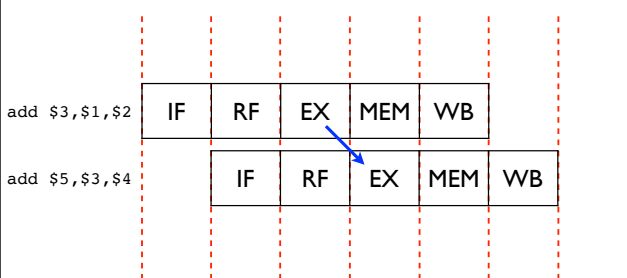
Pipeline hazards

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

Pipeline hazards

	IF	RF	EX	MEM	WB		
lw $1,0($0)

```
lw $1,0($0)     IF  RF  EX  MEM  WB
add $3,$1,$2   STALL  IF  RF  EX  MEM  WB
```
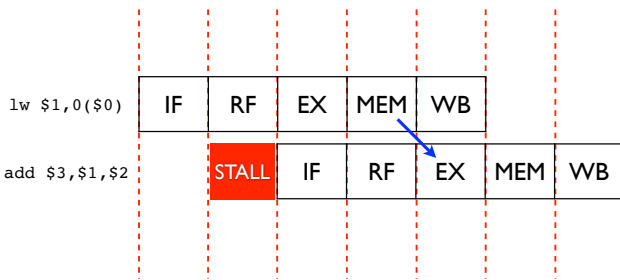
Instruction order

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
```
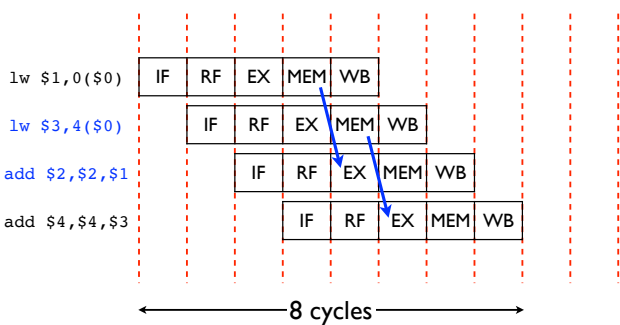
Instruction order
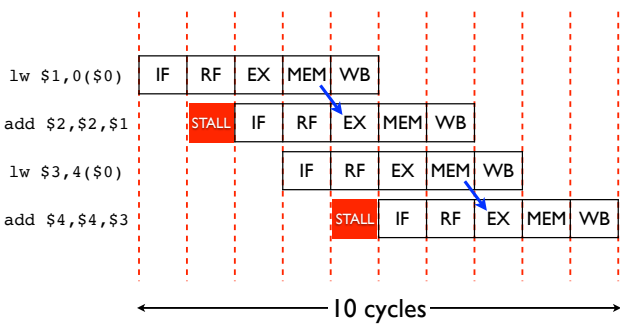
```
lw $1,0($0)     IF  RF  EX  MEM  WB
add $2,$2,$1   STALL  IF  RF  EX  MEM  WB
lw $3,4($0)         IF  RF  EX  MEM  WB
add $4,$4,$3       STALL  IF  RF  EX  MEM  WB
```

← 10 cycles →

Instruction order

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

Instruction order

```
lw $1,0($0)     IF  RF  EX  MEM  WB
lw $3,4($0)         IF  RF  EX  MEM  WB
add $2,$2,$1           IF  RF  EX  MEM  WB
add $4,$4,$3              IF  RF  EX  MEM  WB
```

← 8 cycles →

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.

Instruction dependencies

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 dependencies

Read after write:
An instruction reads from a location after an earlier instruction has written to it.

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

Reads old value ✗
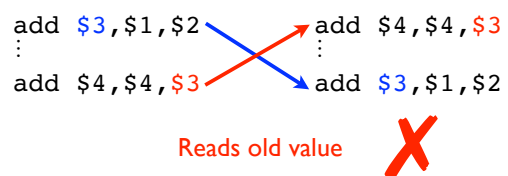
Instruction dependencies

Write after read:
An instruction writes to a location
after an earlier instruction has read from it.

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

Reads new value ✗
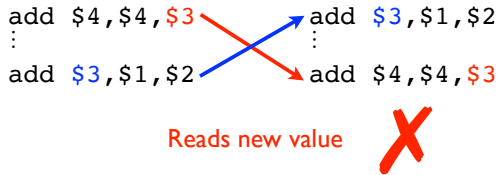
Instruction dependencies

Write after write:
An instruction writes to a location
after an earlier instruction has written to it.

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

Writes old value ✗
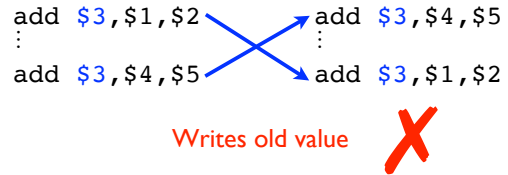
Instruction scheduling

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.

Preserving dependencies

```
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)
```
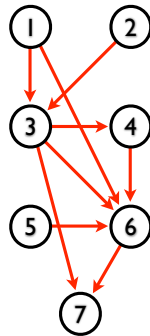

Preserving dependencies

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.

Preserving dependencies
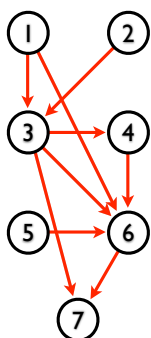

```
1, 2, 3, 4, 5, 6, 7
2, 1, 3, 4, 5, 6, 7

1, 2, 3, 5, 4, 6, 7
1, 2, 5, 3, 4, 6, 7
1, 5, 2, 3, 4, 6, 7
5, 1, 2, 3, 4, 6, 7

2, 1, 3, 5, 4, 6, 7
2, 1, 5, 3, 4, 6, 7
2, 5, 1, 3, 4, 6, 7
5, 2, 1, 3, 4, 6, 7
```

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.

Minimising stalls

Each time we're emitting the next instruction, we should try to choose one which:

- does not conflict with the previous emitted instruction

- is most likely to conflict if first of a pair (e.g. prefer `lw` to `add`)

- is as far away as possible (along paths in the DAG) from an instruction which can validly be scheduled last

Algorithm

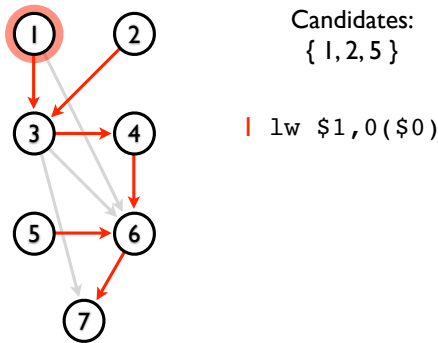Armed with the scheduling DAG and the static scheduling heuristics, we can now devise an algorithm to perform instruction scheduling.

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.

Algorithm

- 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
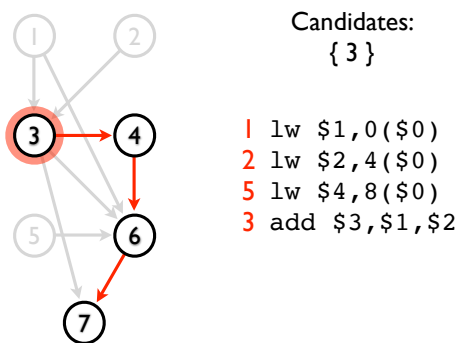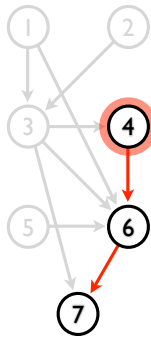

Candidates:
{ 1, 2, 5 }

1 `lw $1,0($0)`

Algorithm

Candidates:
{ 2, 5 }

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

Algorithm

Candidates:
{ 3, 5 }

1 `lw $1,0($0)`
2 `lw $2,4($0)`
5 `lw $4,8($0)`

Algorithm

Candidates:
{ 3 }
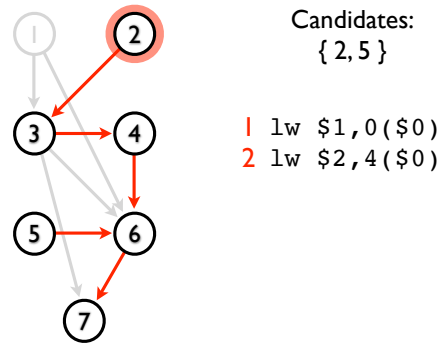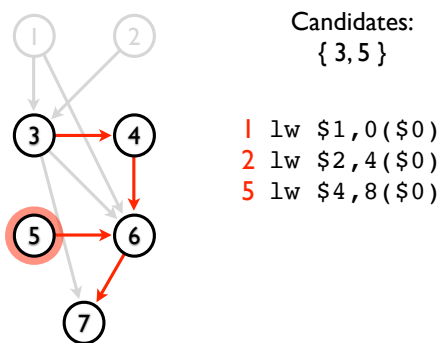
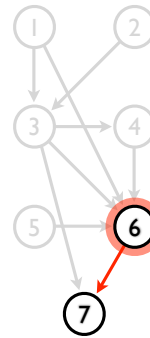1 `lw $1,0($0)`
2 `lw $2,4($0)`
5 `lw $4,8($0)`
3 `add $3,$1,$2`

Algorithm

Candidates:
{ 4 }

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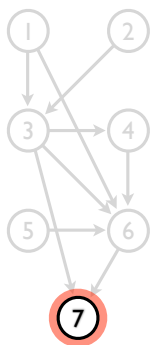
Algorithm

Candidates:
{ 6 }

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

Algorithm

Candidates:
{ 7 }

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

Algorithm

Original code:

```
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)
```

2 stalls
13 cycles

Scheduled code:

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

no stalls
11 cycles

Dynamic scheduling

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
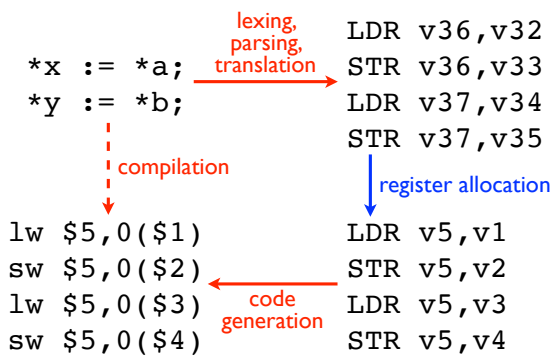
Lecture 15
Register allocation vs instruction scheduling, reverse engineering
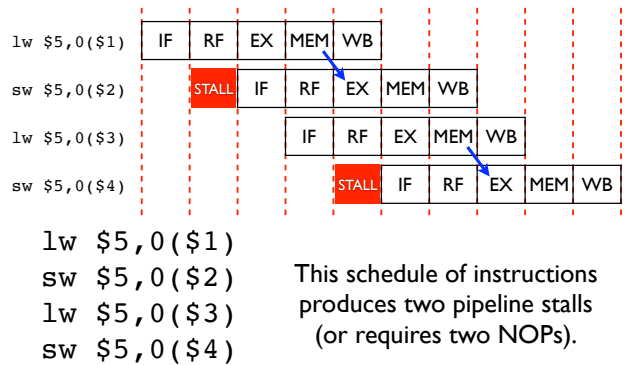
Allocation vs. scheduling

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.
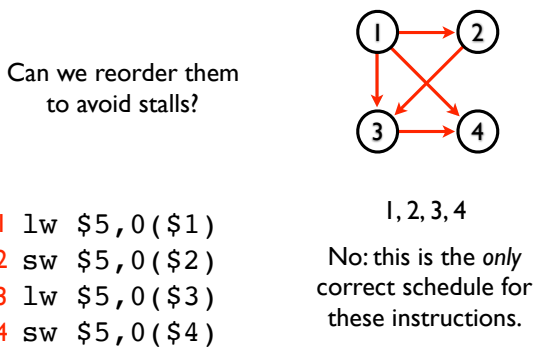
Allocation vs. scheduling

```
*x := *a;        lexing,        LDR v36,v32
                 parsing,       STR v36,v33
*y := *b;       translation     LDR v37,v34
                                STR v37,v35

     compilation            register allocation

lw $5,0($1)      LDR v5,v1
sw $5,0($2)      STR v5,v2
lw $5,0($3)      LDR v5,v3
sw $5,0($4)   code    STR v5,v4
           generation
```

Allocation vs. scheduling

```
lw $5,0($1)    IF  RF  EX  MEM WB
sw $5,0($2)   STALL IF  RF  EX  MEM WB
lw $5,0($3)          IF  RF  EX  MEM WB
sw $5,0($4)          STALL IF  RF  EX  MEM WB
```

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

This schedule of instructions produces two pipeline stalls (or requires two NOPs).

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)
```

1, 2, 3, 4

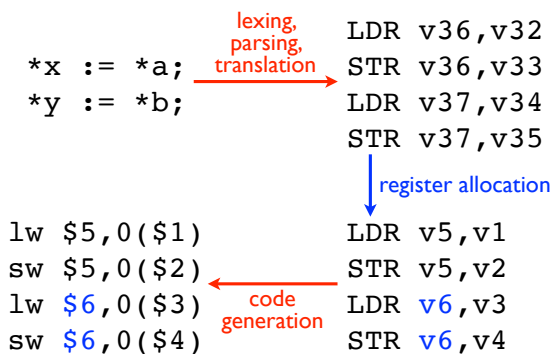No: this is the *only* correct schedule for these instructions.
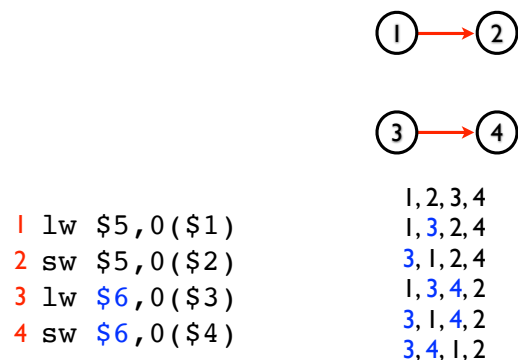
Allocation vs. scheduling

We might have done better if register $5 wasn't so heavily used.
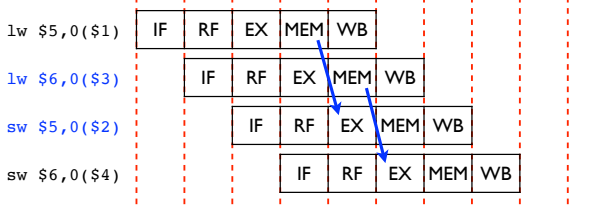
If only our register allocation had been less aggressive!

Allocation vs. scheduling

```
*x := *a;        lexing,        LDR v36,v32
                 parsing,       STR v36,v33
*y := *b;       translation     LDR v37,v34
                                STR v37,v35

                          register allocation

lw $5,0($1)      LDR v5,v1
sw $5,0($2)      STR v5,v2
lw $6,0($3)   code    LDR v6,v3
sw $6,0($4) generation STR v6,v4
```

Allocation vs. scheduling

```
1 lw $5,0($1)
2 sw $5,0($2)
3 lw $6,0($3)
4 sw $6,0($4)
```

1, 2, 3, 4
1, 3, 2, 4
3, 1, 2, 4
1, 3, 4, 2
3, 1, 4, 2
3, 4, 1, 2

Allocation vs. scheduling

```
lw $5,0($1)   | IF | RF | EX | MEM | WB |
lw $6,0($3)        | IF | RF | EX | MEM | WB |
sw $5,0($2)             | IF | RF | EX | MEM | WB |
sw $6,0($4)                  | IF | RF | EX | MEM | WB |
```

This schedule of the new instructions produces no stalls.

```
lw $5,0($1)
lw $6,0($3)
sw $5,0($2)
sw $6,0($4)
```

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

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

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

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

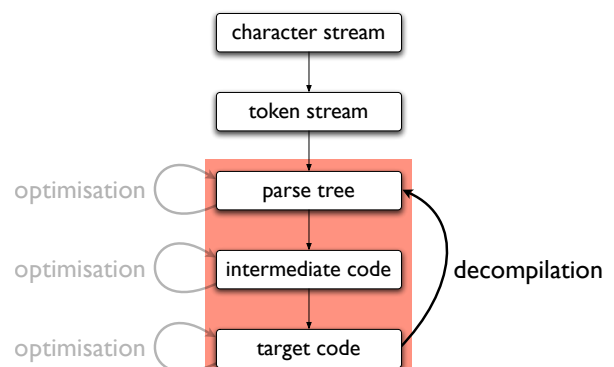The same problem also shows up in dynamic scheduling done by hardware.

Executable x86 code, for example, has lots of register reuse because of the small number of architectural 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
Decompilation and reverse engineering

Decompilation

character stream → token stream → parse tree → intermediate code → target code

optimisation (parse tree, intermediate code, target code)

decompilation

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.

Reverse engineering
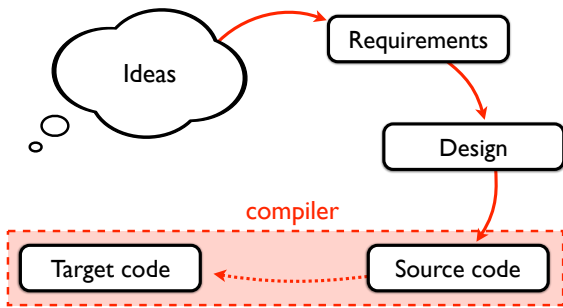
Reverse engineering

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.

Reverse engineering
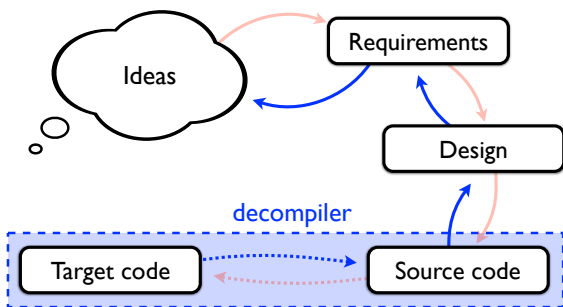
Legality and ethics

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?

Legality and ethics

Companies and individuals responsible for creating software generally consider source code to be their confidential intellectual property; they will not make it available, and they do not want you to reconstruct it.

(There are some well-known exceptions.)

Usually this desire is expressed via an end-user license agreement, either as part of a shrink-wrapped software package or as an agreement to be made at installation time ("click-wrap").

Legality and ethics

However, the European Union Software Directive of 1991 (91/250/EC) says:

Article 4 Restricted Acts

Subject to the provisions of Articles 5 and 6, the exclusive rights of the rightholder within the meaning of Article 2, shall include the right to do or to authorize:

(a) the permanent or temporary reproduction of a computer program by any means and in any form, in part or in whole. Insofar as loading, displaying, running, transmission or storage of the computer program necessitate such reproduction, such acts shall be subject to authorization by the rightholder;

(b) the translation, adaptation, arrangement and any other alteration of a computer program and the reproduction of the results thereof, without prejudice to the rights of the person who alters the program;

(c) any form of distribution to the public, including the rental, of the original computer program or of copies thereof. The first sale in the Community of a copy of a program by the rightholder or with his consent shall exhaust the distribution right within the Community of that copy, with the exception of the right to control further rental of the program or a copy thereof.

Article 5 Exceptions to the restricted acts

1. In the absence of specific contractual provisions, the acts referred to in Article 4 (a) and (b) shall not require authorization by the rightholder where they are necessary for the use of the computer program by the lawful acquirer in accordance with its intended purpose, including for error correction.

2. The making of a back-up copy by a person having a right to use the computer program may not be prevented by contract insofar as it is necessary for that use.

3. The person having a right to use a copy of a computer program shall be entitled, without the authorization of the rightholder, to observe, study or test the functioning of the program in order to determine the ideas and principles which underlie any element of the program if he does so while performing any of the acts of loading, displaying, running, transmitting or storing the program which he is entitled to do.

Article 6 Decompilation

1. The authorization of the rightholder shall not be required where reproduction of the code and translation of its form within the meaning of Article 4 (a) and (b) are indispensable to obtain the information necessary to achieve the interoperability of an independently created computer program with other programs, provided that the following conditions are met:

(a) these acts are performed by the licensee or by another person having a right to use a copy of a program, or on their behalf by a person authorized to to so;

(b) the information necessary to achieve interoperability has not previously been readily available to the persons referred to in subparagraph (a); and (c) these acts are confined to the parts of the original program which are necessary to achieve interoperability.

2. The provisions of paragraph 1 shall not permit the information obtained through its application:

(a) to be used for goals other than to achieve the interoperability of the independently created computer program;

(b) to be given to others, except when necessary for the interoperability of the independently created computer program; or (c) to be used for the development, production or marketing of a computer program substantially similar in its expression, or for any other act which infringes copyright.

Legality and ethics

"The authorization of the rightholder shall not be required where [...] translation [of a program is] necessary to achieve the interoperability of [that program] with other programs, provided [...] these acts are performed by [a] person having a right to use a copy of the program"

Legality and ethics

The more recent European Union Copyright Directive of 2001 (2001/29/EC, aka "EUCD") is the EU's implementation of the 1996 WIPO Copyright Treaty.

It is again concerned with the ownership rights of technological IP, but Recital 50 states that: "[this] legal protection does not affect the specific provisions [of the EUSD]. In particular, it should not apply to [...] computer programs [and shouldn't] prevent [...] the use of any means of circumventing a technological measure [allowed by the EUSD]."

Legality and ethics

And the USA has its own implementation of the WIPO Copyright Treaty: the Digital Millennium Copyright Act of 1998 (DMCA), which contains a similar exception for reverse engineering:

"This exception permits circumvention [...] for the sole purpose of identifying and analyzing elements of the program necessary to achieve interoperability with other programs, to the extent that such acts are permitted under copyright law."

Legality and ethics

Predictably enough, the interaction between the EUSD, EUCD and DMCA is complex and unclear, particularly at the increasingly-blurred interfaces between geographical jurisdictions (cf. Dmitry Sklyarov), and between software and other forms of technology (cf. Jon Johansen).

Get a lawyer.

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.

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.

Control reconstruction

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

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

In a flowgraph, we say 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* ≠ *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*.
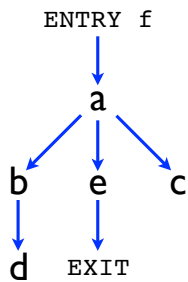
Intuitively this is the set of nodes where *m*'s dominance stops.

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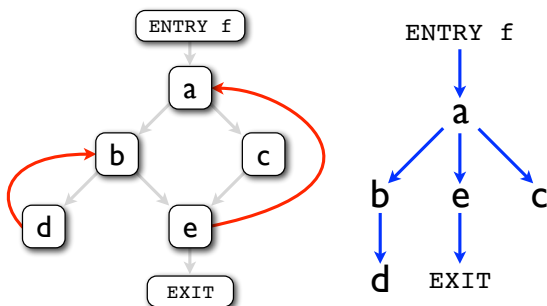
Dominance
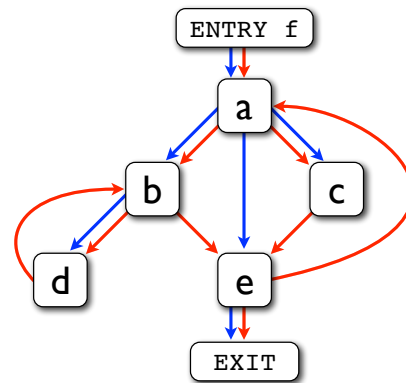
Dominance

Back edges

We can now define the concept of a *back edge*.

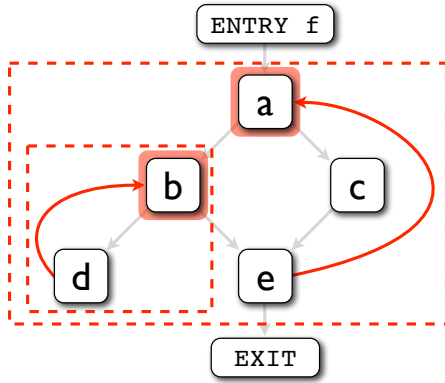In a flowgraph, a back edge is one whose head dominates its tail.

Back edges

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.
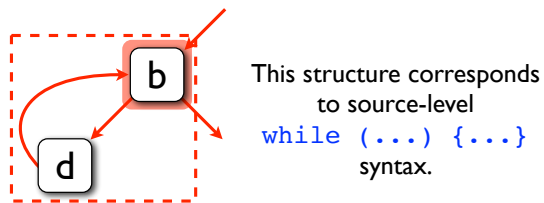
Finding loops

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.
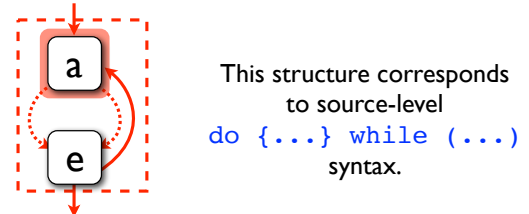
Finding conditionals

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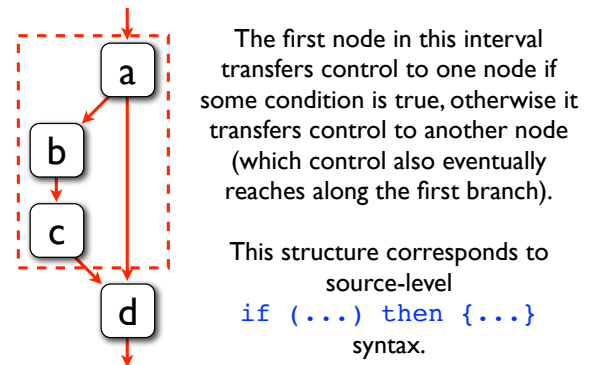
The first node in this interval transfers control to one node if some condition is true, otherwise it transfers control to another node (which control also eventually reaches along the first branch).

This structure corresponds to source-level
`if (...) then {...}`
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.
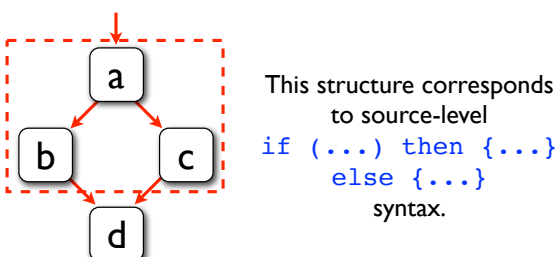
Control reconstruction

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).

Type reconstruction

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.

Type reconstruction

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;              MOV r3,#42
⋮                        ⋮
char *y = "42";          MOV r3,#0xFF34
```

Type reconstruction

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 r3a,#42
⋮                       ⋮
MOV r3,#0xFF34          MOV r3b,#0xFF34
```

Type reconstruction

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

↓ compile

ARM
```
f: ldr r0,[r0,#4]
   add r0,r0,#2
   mov r15,r14
```

Type reconstruction

C
```
int f (int r0) {
    r0 = *(int *)(r0 + 4);
    r0 = r0 + 2;
    return r0;
}
```

↑ decompile

ARM
```
f: ldr r0,[r0,#4]
   add r0,r0,#2
   mov r15,r14
```

Type reconstruction

```
int f (int r0) {
    r0 = *(int *)(r0 + 4);
    r0 = r0 + 2;
    return r0;
}
```

↓ SSA

```
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;
}
```

↑ reconstruct types

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

130

Type reconstruction

```
int f (int *r0_a) {
   int r0_b = *(r0_a + 1);
   int r0_c = r0_b + 2;
   return r0_c;
}
```

$\downarrow$ reconstruct syntax

```
int f (int *r0_a) {
   int r0_b = r0_a[1];
   int r0_c = r0_b + 2;
   return r0_c;
}
```

Type reconstruction

```
int f (int *r0_a) {
   return r0_a[1] + 2;
}
```

$\uparrow$ propagate copies

```
int f (int *r0_a) {
   int r0_b = r0_a[1];
   int r0_c = r0_b + 2;
   return r0_c;
}
```

Type reconstruction

```
int f (int *r0_a) {
   return r0_a[1] + 2;
}
```

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

```
T f (T *r0_a) {
   return r0_a[1] + 2;
}
```

Type reconstruction

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