UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Computer Science Tripos Part II

Optimising Compilation

http://www.cl.cam.ac.uk/Teaching/2001/OptComp/

Alan Mycroft am@cl.cam.ac.uk

2001–2002 (Lent Term)

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) Dead-code elimination

Lecture 3: (Analysis) Live variable analysis

Lecture 5: (Analysis) Available expressions

Lecture 5: (Transformation) Uses of LVA

Lecture 6: (Continuation) Register allocation by colouring

Lecture 7: (Transformation) Uses of Avail

Lecture 8: Single Static Assignment

Lecture 9: (Framework) Abstract interpretation

Lecture 10: (Instance) Strictness analysis

Lecture 11: Control-flow analysis (for λ-terms)

Lecture 12: (Framework) Inference-based program analysis

Lecture 13: (Instance) Effect systems

Lecture 14: Instruction scheduling

Lecture 15: Same continued, slop

Lecture 16: Decompilation.

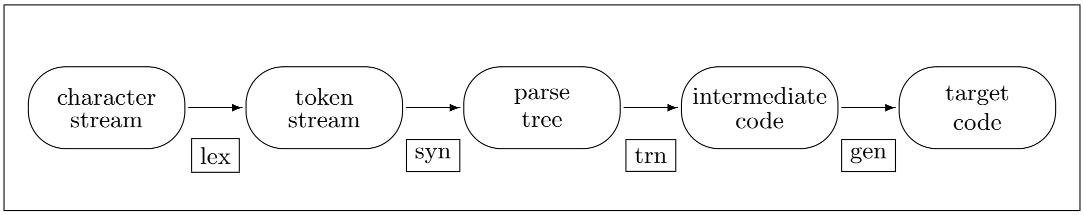
Books

- 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.
- Appel A. *Modern compiler implementation in C/ML/Java* (2nd edition). CUP 1997.
  See `http://www.cs.princeton.edu/~appel/modern/index.html`
- 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://www.mkp.com/books_catalog/1-55860-320-4.asp`
- Wilhelm, R. *Compiler design.* Addison-Wesley, 1995.
  See `http://cseng.aw.com/bookpage.taf?ISBN=0-201-42290-5`

Part A: Classical 'Dataflow' Optimisations

1 Introduction

Recall the structure of 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

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

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 and f is a procedure name):

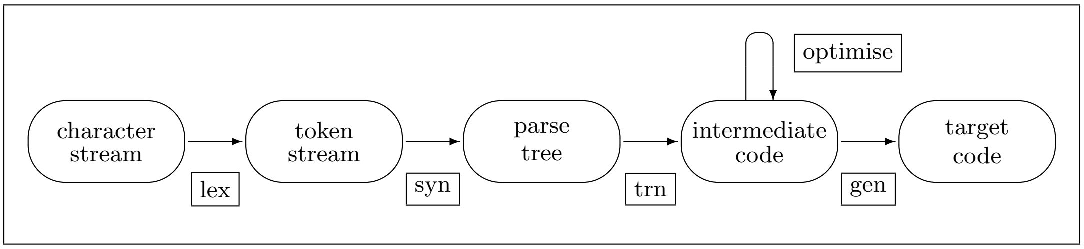
- ENTRY f: no predecessors;
- EXIT: no successors;
- ALU a, b, c: one successor;
- CMP$\langle \mathit{cond} \rangle$ b, c: two successors.

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` $\langle \mathit{exp} \rangle$) 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.

Slogan: Optimisation = Analysis + Transformation

Transformations are often simple (e.g. delete this instruction) but may need complicated analysis to show 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 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)

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 ‘dead 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: dead-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:

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

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 `goto` ⟨*exp*⟩ (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: dead-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* 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.

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

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*:

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

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

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

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

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 = &y;
if (...) p = &y;
*p = 7;
if (...) p = &x;
t = *p;
print z+t;
```

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 $ref = \{\mathtt{p}\}$ and $def = \{\}$ whereas `t = *p;` has $ref = \{\mathtt{p}, \mathtt{x}, \mathtt{y}\}$ and $def = \{\mathtt{t}\}$.

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

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 using bit p being set to represent that variable $\mathtt{x}_p$ (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

Available expressions analysis (AVAIL) 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.

This leads to dataflow equations:

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

Here $gen(n)$ gives the expressions freshly computed at n, i.e. $\{\mathtt{y} + \mathtt{z}\}$ for the instruction `x = y+z;` but $\{\}$ for the instruction `x = x+z`. 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] := $\bigcap_{p \in pred(\mathtt{i})}$ (avail[$p$] $\setminus$ $kill(\mathtt{p}) \cup gen(\mathtt{p})$).
```

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 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 naive 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 typically 32. Derive a graph (the 'clash graph') whose nodes are 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 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[1]) 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 an 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 reasonable 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 physical registers are a subset of virtual registers by arranging that (say) virtual registers `v0`–`v31` are pre-allocated to physical registers `r0`–`r31` and virtual registers allocated for temporaries and user variables start from 32. Now

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

[1] Of course this 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.

- similarly when an instruction produces a result in a given physical 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 physical 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

```
f:    mov   r0,v32
      call  g
      mov   r0,v33
      mov   v32,r0
      call  h
      add   v33,r0,v34
      add   #1,v34,r0
      ret
```

which, noting that v32 and v33 are clash with all non-preserved registers (being live over a procedure call), might generate code (on a machine where r8 upwards are specified to be preserved over procedure call)

```
f:    push  {r8,r9}
      mov   r0,r8
      call  g
      mov   r0,r9
      mov   r8,r0
      call  h
      add   r9,r0,r0
      add   #1,r0,r0
      pop   {r8,r9}
      ret
```

Note that r8 and r9 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.

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 details (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 subexpression is big enough to justify the LD/ST cost of spilling of whether the common subexpression is small enough that ignoring it by re-computing is cheaper.

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:

```
t1 := a*b;
x := t1+c;
t2 := a*b;
y := t2+c.
```

CSE as presented converts this to

```
t3 := a*b;
t1 := t3;
x := t1+c;
t2 := t3;
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 `t1:=t3` and `t2:=t3` by the expedient of renaming `t2` and `t3` as `t1`. 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

```
    t3 := a*b;
    t4 := t3+c;
    x := t4;
    y := t4.
```

which is just about optimal for input to register allocation (remember that x or y may be spilled 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 Single Static Assignment 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) r8, 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 is 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 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 Single Static Assignment (SSA) form and the transformation to it.

The Single Static Assignment (SSA) form (see e.g. [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

$$\texttt{v}_7\texttt{ = 3; v}_8\texttt{ = v}_7\texttt{+1; v}_9\texttt{ = v}_8\texttt{+w}_3\texttt{; w}_4\texttt{ = v}_9\texttt{*2;}$$

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 (single static) 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)

$$\texttt{\{ if (p) \{ v}_4\texttt{ = v}_3\texttt{+1; v}_5\texttt{ = v}_4\texttt{+w; \} else v}_6\texttt{=v}_3\texttt{-1; \} v}_7\texttt{ = }\phi\texttt{(v}_5\texttt{,v}_6\texttt{); w = v}_7\texttt{*2;}$$

Part B: Higher-Level Optimisations

This second half 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 communicate sufficiently for unified theory and notation to have developed.

7 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 $e + 0 \to e$ or $(e + n) + m \to e + (n + m)$ 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.

8 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 × positive is negative’ from school algebra. We formalise this as a table

$\otimes$	$(-)$	$(=)$	$(+)$
$(-)$	$(+)$	$(=)$	$(-)$
$(=)$	$(=)$	$(=)$	$(=)$
$(+)$	$(-)$	$(=)$	$(+)$

Here there are two calculation routes: one is to calculate in the real world (according to the *standard interpretation* of operators (e.g. × 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. × 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

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

where (?) represents an additional abstract value conveying no knowledge (the always-true property). Abstract addition $\oplus$ operates on (?) by $(?) \oplus x = (?) = x \oplus (?)$. Thus we find that, writing *abs* for the abstraction from concrete (real-world) to abstract values we have

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

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 $\{z \in \mathbb{Z} \mid z < 0\}$).

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. A good example for the rule-of-signs is the example $(x + 1) \times (x + 1) + 1$ which gives (+) when $x = (-)$ whereas $x \times x + 2 \times x + 2$ gives (?).

Abstract interpretation has been used 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.

9 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
          ↦ 6+1
          ↦ 7.
```

The language we consider here is that of recursion equations:

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

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 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 $D = \mathbb{Z}_\perp$ be the space of integer values (for terminating computations of expressions e) augmented with a value $\perp$ (to represent non-termination). The 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

$$\begin{array}{rcl} +(\perp,y) & = & \perp \\ +(x,\perp) & = & \perp \\ +(x,y) & = & x +_{\mathbb{Z}} y \quad \text{otherwise} \\ \mathit{cond}(\perp,x,y) & = & \perp \\ \mathit{cond}(0,x,y) & = & y \\ \mathit{cond}(p,x,y) & = & x \quad \text{otherwise} \end{array}$$

(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 $D^\sharp = 2 \overset{\text{def}}{=} \{0,1\}$ 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

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

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

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

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

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

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 is 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

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

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 $f^\natural(\vec{x}) \le f^\sharp(\vec{x})$.

Returning to our running example

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

we derive equation

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

Simplifying with built-ins

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

gives

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

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')

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

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 $\mathit{plus}^\sharp$ 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] := $\lambda\vec{x}.0$
while (f#[] $changes$) do
   for i=1 to $n$ do
      f#[i] := $\lambda\vec{x}.e_i^\sharp$.
```

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[2] 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.

10 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

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

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

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

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

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

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

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 an formulation to get safe approximations (here possibly over-estimates) for the α_i.[3] 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):

- 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)

$$\text{whenever } \beta \supseteq \{(\lambda x^j.e^k)^i\} \text{ we have } \alpha_j \supseteq \gamma \wedge \delta \supseteq \alpha_k.$$

Such constraint sets are always finitely satisfiable and have a unique least solution (w.r.t. $\subseteq$) because of reasons concerning monotonicity of the above implication (outside this course).

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

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

[3] 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.

Again all solutions are safe, but the least solution is

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

You may verify that this solution is safe, but note that is imprecise because $7^{33} \in \alpha_1$. 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 $\{7, \lambda\mathtt{x}.\mathtt{x}\}$ whereas we 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 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.

11 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 (to be covered in more detail the 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

$$t ::= \alpha \mid \mathit{int} \mid t \to t'.$$

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:

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

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

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

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

$$\phi ::= \mathit{odd} \mid \mathit{even} \mid \phi \to \phi'.$$

We would then have rules:

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

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

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

Under the assumptions

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

we could then show

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

but note that showing

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

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

$$\begin{array}{rl} \times : & \mathit{even} \to \mathit{even} \to \mathit{even} \ \wedge \\ & \mathit{even} \to \mathit{odd} \to \mathit{even} \ \wedge \\ & \mathit{odd} \to \mathit{even} \to \mathit{even} \ \wedge \\ & \mathit{odd} \to \mathit{odd} \to \mathit{odd}. \end{array}$$

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

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

for some Γ.

12 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:

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

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

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

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

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

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

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

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

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

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

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

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

and

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

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:

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

However, this means that

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

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:

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

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

13 Introduction

In this part we 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 feed-forwarding 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 has 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. Intel Pentium architecture goes 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; moreover, even if all scheduling were to be done dynamically in hardware, someone (now hardware designers) still has to understand scheduling principles!

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. We present the $O(n^2)$ algorithm due to Gibbons and Muchnick (attached in the paper version of the notes) in these notes.

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 the 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 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.

14 Antagonism of register allocation and instruction scheduling

Register allocation by colouring results 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; 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[4] 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

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

which then executes in only 4 cycles.

At the moment there does not appear to be 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. 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).

This problem also poses dynamic scheduling problems in pipelines for corresponding 80x86 instruction sequences which need to reuse registers as much as possible because their limited number. Processors such as the Intel Pentium achieve effective dynamic rescheduling by having a larger register set in the computational engine than the 8-register based (ax,bx,cd etc.) 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.

[4] 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.

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 Digitial Millenium 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 overridding law such as the above.

A good source of information is the *Decompilation Page* [5] on the web, in particular the subpage

`http://www.csee.uq.edu.au/~csmweb/decompilation/ethics.html` is attached to the (paper) course notes and mirrored on the course web site

```
http://www.cl.cam.ac.uk/Teaching/2001/OptComp
```

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 [6] for much more detail. In particular pages 123–130 are attached to the (paper) course notes and mirrored on the course web site

```
http://www.cl.cam.ac.uk/Teaching/2001/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 [7] attached to the (paper) notes or via the course web site

```
http://www.cl.cam.ac.uk/Teaching/2001/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] 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.

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

[5] Cifuentes, C. et al. "The decompilation page".
URL http://www.csee.uq.edu.au/~csmweb/decompilation/

[6] Cifuentes, C. "Reverse compilation techniques". PhD thesis, University of Queensland, 1994.
URL http://www.csee.uq.edu.au/~csmweb/dcc.html

[7] Mycroft, A. Type-Based Decompilation. Lecture Notes in Computer Science: Proc. ESOP'99, vol. 1576: 208–223, Springer-Verlag, 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.]

Advertisement

In addition to the usual sources of money, various tame companies have funded Ph.D. research at the higher CASE level in areas which interest them. For more details see our research group site `http://www.cl.cam.ac.uk/Research/CPRG` and come and talk to me if interested.