UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Computer Science Tripos Part II

# Optimising Compilers (Parts B C D)

http://www.cl.cam.ac.uk/Teaching/1011/OptComp/

**Alan Mycroft** am@cl.cam.ac.uk

**2010–2011 (Michaelmas Term)**

# 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 communicate 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 $2{*}e \longrightarrow \texttt{let}\ \ x = e\ \texttt{in}\ x + x$. 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 is 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 (depends 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 is 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

$$\begin{array}{ll} \mathit{abs}((-1515) + 37) & = \mathit{abs}(-1478) = (-), \quad \text{but} \\ \mathit{abs}(-1515) \oplus \mathit{abs}(37) = (-) \oplus (+) & = (?). \end{array}$$

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,

$$\begin{array}{ll}(abs(-1515)\otimes abs(37))\oplus abs(42) = ((-)\otimes(+))\oplus(+) = (?), & \text{or}\\ (abs(-1515)\oplus abs(37))\otimes abs(0) \;= ((-)\oplus(+))\otimes(0) \;= (0). & \end{array}$$

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
```

```
↦ 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 $\lambda$-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 $D = \mathbb{Z}_\bot (= \mathbb{Z} \cup \{\bot\})$ be the space of integer values (for terminating computations of expressions $e$) augmented with a value $\bot$ (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

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

$$\begin{array}{rcl} \mathit{cond}(\bot,x,y) & = & \bot \\ \mathit{cond}(0,x,y) & = & y \\ \mathit{cond}(p,x,y) & = & x \quad \text{otherwise} \end{array}$$

(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 $i$th parameter (recall earlier we said that this was if the parameter had necessarily been evaluated whenever the function returns). This happens precisely when

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

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

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

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 $i$th argument iff

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

(where the ‘0’ is in the $i$th 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 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 $f^\natural(\vec{x}) \le f^\sharp(\vec{x})$.

Returning to our running example

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

we derive equation

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

Simplifying with built-ins

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

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[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 $t_1 = (t_2 \rightarrow t_3)$ 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 analysis below, the constraints are inequations, but they again have the property that a minimal solution can be reached by initially assuming that all sets $\alpha_i$ 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 $\alpha$ to be $\phi$ 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 $\lambda$-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 $\lambda$-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

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

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 $\alpha_i$ 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 $\alpha_i$.[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 $\alpha_i$ 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. $\alpha_i$ 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 (`let` $x^l = e_1^j$ `in` $e_2^k)^i$ we get the constraints $\alpha_i \supseteq \alpha_k$ and $\alpha_l \supseteq \alpha_j$;
- for a term (`if` $e_1^j$ `then` $e_2^k$ `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 $\beta$ (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 $\gamma$, it yields results contained within the result specification $\delta$. (Of course $\delta$ 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

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

Now note this implication can also be written as two implications

$$\begin{aligned} \alpha_j \supseteq \{(\lambda x^q.e^r)^p\} &\Longrightarrow \alpha_q \supseteq \alpha_k \\ \alpha_j \supseteq \{(\lambda x^q.e^r)^p\} &\Longrightarrow \alpha_i \supseteq \alpha_r \end{aligned}$$

Now, if you know about Prolog/logic programming then you can see that 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:

$$\begin{aligned} \alpha_q \supseteq \alpha_k &\Longleftarrow \alpha_j \supseteq \{(\lambda x^q.e^r)^p\} \\ \alpha_i \supseteq \alpha_r &\Longleftarrow \alpha_j \supseteq \{(\lambda x^q.e^r)^p\} \end{aligned}$$

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:

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

Again all solutions are safe, but the least solution is

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

You may verify that this solution is safe, but note that is imprecise because $(\lambda \mathtt{x}^{20}.\mathtt{x}^{21})^{22} \in \alpha_1$ 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 $\lambda$-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 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

I might say something more about this in lectures, but formally this section (at least for 2006/07) 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 $\phi$ is a program property and $\Gamma$ 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 $\lambda$-calculus:

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

and (assuming $\alpha$ to range over type variables) types $t$ of the syntax

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

Now let $\Gamma$ 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 $\Gamma$ with any assumption about $x$ removed and with $x : t$ additionally assumed. We then have inference rules:

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

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

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

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:

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

where $[\![e]\!]$ represents the result of evaluating $e$ (its denotation) and $[\![t]\!]$ 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:

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

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

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

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

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

**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 $\Gamma$.

## 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 $\lambda$-calculus augmented with expressions $\xi?x.e$ which reads an *int* from a channel $\xi$ 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 $\xi$ 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 $\xi$ but also a *latent effect* of writing to $\zeta$ via the resulting $\lambda$-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 $\lambda$-abstraction we need to augment the type system to

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

Letting $\mathit{one}(f) = \{f\}$ 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

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

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:

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

## 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

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

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 Anderson's analysis in detail

Define a set of abstract values

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

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 $pt$:

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

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

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

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):

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

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

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

**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 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. 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 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.

## 20 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[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

```
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 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.

[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.

# 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/
```

### A Research Project

One potentially interesting future PhD topic is to extend the notion of decompilation to hardware, so that we can decompile (say) structural descriptions of a circuit in VHDL or Verilog into behavioural descriptions (and yes, there are companies with legacy structural descriptions which they would dearly like to have in more readable/modifiable form!).

## 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.]