Optimising Compilers

University of Cambridge

Ashwin Ahuja

Computer Science Tripos
Part II

May 2020

Contents

1 Introduction

1. Character Stream

   lexing

2. Token Stream

   parsing

3. Parse Tree

   translation

4. Intermediate Code: normally a stack-oriented abstract machine code. To ease optimisation (make things (1) smaller, (2) faster, (3) cheaper - lower power consumption), we use 3-address code as an intermediate code which eases moving computation around. This intermediate code is also stored as a flowgraph, where the nodes are labelled with 3-address instructions where nodes are labelled with 3-address instructions.

   code generation

5. Target Code

Optimisation is composed of: (1) **Transformation** - do something dangerous and (2) **Analysis** - determine whether it's safe. Analysis shows that a program has some property and the transformation is designed to be safe for all programs with that property - therefore would be safe to do the transformation.
Flowgraph: Each edge represents potential control flow.

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

Forms of instructions are:

1. ENTRY f: no predecessors
2. EXIT: no successors
3. ALU a, b, c (ADD, MUL, etc): one successor. Procedure calls and indirect calls are treated as atomic instructions
4. CMP<cond> a, b, lab (CMPNE, CMPEQ, etc): two successors. Multi-way branches can be considered as a cascade of CMP instructions

Basic Block: Maximal sequence of instructions which have: (1) exactly one predecessor (except possibly the first instruction) and (2) exactly one successor (except possibly the last instruction). We can reduce the time and space requirements for analysis by calculating the data flow information once per block instead of once per instruction. Flowgraph where basic blocks are combined is in normal form. Therefore, we have a few areas of analysis and optimisation:

1. Within basic blocks - local

   Peephole: Combine moves and additions, etc. E.g. 'MOV x, y', 'MOV y, x' gets replaced by 'MOV x, y'.

2. Between basic blocks - global, intra-procedural - **Live Variable Analysis**
3. Whole program - inter-procedural. Collected from instructions of each procedure and then propagated between procedures

   Unreachable-procedure elimination: aim is to find the callable procedures (using a call graph) where the transformation is to delete the procedures which analysis does not mark as callable.

   (a) Mark procedure main as callable
   (b) Mark every successor of a marked node as callable and repeat until no further marks are required

   This is safe, but has a safe overestimation of reachability, as we assume that a procedure containing an indirect call has all address-taken procedures as successors in the call graph - ie it could take all of them.

Information Types: (1) Control flow - control structure (basic blocks, loops, calls between procedures) and (2) Data flow - data flow structure (variable uses, expression evaluation)
To find basic blocks, we find all the instructions which are leaders - for each leader, its basic block consists of itself and all instructions upto next leader.

1. First instruction is a leader
2. Target of any branch is a leader
3. Any instruction following a branch is a leader

2 Unreachable-code Elimination

Dead Code: code where we compute unused values. This is a data-flow property. **Unreachable Code**: code that can never be run. This is a control-flow property. In general analysis of whether a code statement is reachable is an undecidable problem. However, we can simplify and assume that (1) branches of a conditional are reachable and (2) loops always terminate, conservatively overestimating the reach-ability.

1. Mark entry node of every procedure as reachable
2. Mark every success for of a marked node as reachable and repeat until no more marks required

This analysis is safe, but does not fully optimise - eg. when branch-address computation completely unrestricted - indirect branch forces to assume all instructions are potentially reachable. For example, doesn't solve for copy propagation errors.

2.1 Straightening

Optimisation is an optimisation which can eliminate jumps between basic blocks by coalescing them, therefore meaning the new program will execute faster.

3 Live Variable Analysis

This is a form of backwards data-flow analysis - based around the motivations: (1) code exists which gets executed but has no useful effect, (2) variables being used before being defined, (3) variables need to be allocated registers / memory locations for compilation. Live-ness in general asks 'is the variable needed'. Each instruction has an associated set of live variables.

Semantically Live: x is live at node n if there is some execution sequence starting at n show I/O behaviour can be affected by changing the value of x. This is undecidable - may depend on arithmetic.

Syntactically Live: variable is syntactically live if there is a path to the exit of the flow-graph along which its value may be used before it is redefined. Concerned with properties of syntactic structure of the program. This is decidable and is a computable approximation of semantic liveness. In particular it does the following for safety:

1. Overestimates ambiguous references
2. Underestimates ambiguous definitions

sem-live$(n) \subseteq$syn-live(n)

Usage information from future instructions must be propagated backwards through future instructions must be propagated backwards through the program to discover which variables are live. Each instruction has an effect on the liveness information as it flows past (**makes a variable live when it references it** and **makes a variable dead when it defines or assigns to it**). Therefore, each instruction has a ref and def set, which can be used to declare the dataflow equations:

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

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

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

3.1 Algorithm

```
for i=1 to N do live[i] := {}
while (live[] changes) do
   for i=1 to N do
      live[i] := ( ⋃_{s∈succ(i)} live[s] ) \ def(i) ∪ ref(i)
```

Algorithm is guaranteed to give the smallest solution. We can each element of live[] with n-bit value (where we have n values), with each bit indicating liveness of one variable. Store liveness once per basic block and recompute inside a block where necessary.

3.2 Available Expressions

Availability: Has the value of the expression already been computed. Each instruction has an associated set of available expressions. As before, we have semantic and syntactic availability:

1. **Semantic Availability**: Expression available at node n if value gets computed and not subsequently invalidated along every execution sequence ending at n.
2. **Syntactic Availability**: Expression available at node n if value gets computed and not subsequently invalidated along every path from the entry of the flowgraph to n. This is decidable. Safety in assuming that fewer expressions are available.

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

This is a forwards data-flow analysis - information from past instructions propagated forwards through the program to discover which expressions are available. An instruction makes an expression available when it generates (gen set) its current value and unavailable when it kills (kill set) its current value (assignment to variable x kills all expressions in the program which contains occurrences of x).

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

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

avail(n) = {} if pred(n) = {}

$$\text{avail}(n) = \bigcap_{p \in \text{pred}(n)} (\text{avail}(p) \cup \textit{gen}(p) \setminus \textit{kill}(p)) \quad \text{if } \textit{pred}(n) \neq \{\}$$

$$\text{avail}\ (n) = \{\} \quad \text{if pred}\ (n) = \{\}$$

Can rewrite by redefining gen and kill:

- Node generates e if it must compute value of e and does not subsequently redefine any of the variables in e
- Node kills e if it may redefine some of variables in e and does not subsequently recompute value of e

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

3.2.1 Algorithm

Set avail[] to store the available expressions for each node - initially each node has all expressions available.

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

Can do better is we assume flowgraph has single entry node (first node in avail[]). Can then initialise avail[1] to empty set and not recalculate avail[1]'s availability at the first node during each iteration.

Solution is safe and guaranteed to terminate since effect of an iteration is monotonic.

As with LVA, n bits for each element of avail and store only once per basic block.

With address taken variables, we must again underestimate ambiguous generation and overestimate ambiguous killing.

3.2.2 Analysis Framework

- LVA is union over successors
- *RD (Reaching Definition) is union over predecessors. Where things might have been defined.*
- AVAIL is intersection over predecessors
- *VBE is very busy expressions*

Therefore, with a single algorithm for iterative solution of data-flow equations, we can compute all the analyses and others which fit into framework.

3.3 Uses of LVA

Data-Flow Anomalies: Can break program or just make it perform worse. Therefore, helps in both optimisation and bug elimination.

- For dead code, simple fix to remove the dead code.
- For uninitialised variables which can be easily found by LVA, can warn a user - since there could be spurious warnings.
- Write-write anomalies: occur when variable written twice with no intervening read - first write can be removed. Can be solved using LVA in reverse.

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

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

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

However, second write to variable may turn earlier write into dead code, doesn't necessarily do this, therefore need a different analysis. However, overly safe, therefore, should warn the programmer rather than automatically fix the system.

4 Register Allocation (by colouring)

Clash Graphs: In generating intermediate code, we invent many variables to hold results of computations. Leads to "normal form", where temporary variables used on occasion without being reused. This generates 3-address codes but assumes unlimited virtual registers. LVA can tell us which variables are simultaneously live and hence must be kept in separate virtual registers. One vertex for each virtual register with edge when two registers are simultaneously live.

Register Allocation: Take virtual registers and allocate to small finite number of actual hardware registers. Find minimal colouring for the clash graph - however, this problem is NP-hard and therefore difficult to do efficiently, and needs a heuristic (may not always give the best colouring). Additionally, need one or two free registers to be able to load and operate on items from spilled memory.

4.1 Spilling

Occasionally need to spill excess values into memory, therefore consider this in heuristic.

- Choose vertex with fewest incident edges
- If fewer edges than colours:
  1. Remove vertex and push onto LIFO stack
  2. Otherwise choose register to spill and remove vertex
- Repeat until graph is empty
- Pop each vertex from the stack and colour it in the most conservative way which avoids the colour of already-coloured neighbours.
- Any uncoloured vertices have to be spilled

Notes

- This doesn't necessarily work well - doesn't take into account loops and therefore consider one static access inside loop as 4 accesses.

- There are often a number of options for what colour to label things with - therefore construct a preference graph to show which pairs of registers appear together.
- Some architectures have limitations on what is required in particular registers for various operations. Therefore, for example, we require MOV command to move items to the right register for an ALU operation and produce a trailing MOV to transfer the result into a new virtual register. The MOV operations can be eliminated by using a preference graph.
- When we know an instruction is going to corrupt the contents of an architectural register, insert an edge on the clash graph between corresponding virtual register and all other virtual registers at that instruction. This prevents register allocator from storing any live values in the corrupted register.
- Also, some standards dictate that procedure calls require certain registers for arguments and results - therefore need synthesised MOV instructions as before.

5 Code Motion

5.1 Common-subexpression elimination

Transformation enabled by AVAIL, using it to identify which expressions will have been computed by the time control arrives at an instruction in the program. Therefore, can spot and remove redundant computations. Transformation is to eliminate common subexpression by storing value into temporary variable and reusing variable the next time this is required. This is a form of code motion transformation: operate by moving instructions and computation around programs to take advantage of opportunities identified by control and data-flow analysis.

5.1.1 Algorithm

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

Note, this might note be better, since we need a lot of register copy instructions.

5.2 Copy Propagation

Scan forwards from an x=y instruction and replace x with y wherever it appears as long as neither x nor y has been modified.

5.3 Code Hoisting

Reduce size of program by moving duplicated expression computation to a single place where it can be combined into a single instruction. Relies on data low analysis called **very busy expressions** - backwards version of AVAIL. Finds expressions that are definitely going to be evaluated later in the program. Not guaranteed to make a program faster.

5.4 Loop-Invariant Code Motion

Some expressions redundant in the sense that they get recomputed on every iteration though their value never changed within the loop. Relies on data-flow analysis to find which assignments may affect value of a variable (**reaching definitions**)

5.5 Partial Redundancy Elimination

Combines common subsequence elimination and loop-invariant code motion into one optimisation. Can be completed on its own using complex combination of forwards and backwards data-flow analyses.

Partially Redundant Expression: when can be computed more than once on some paths through a flowgraph.

6 Static Single Assignment

- **Live Range**: The range of instruction lines where a single user variable in the program
- **Live Range Splitting**: Live ranges are made smaller by using a different virtual register to store variable value at different times

SSA Form allows register allocation to do a better job - when virtual register is only assigned to once. Therefore, live ranges are already as small as possible. This works by renaming each variable using subscripts which is incremented ever time that a variable is assigned to. When control-flow edges meet, two differently-named variables have to be merged together using (non-executable) ϕ-functions.

By representing data dependencies as precisely as possible, it makes many optimising transformation simpler and more effective.

Need to be aware that it is generally a difficult problem to decide what order to perform optimisations, since certain optimisations are antagonistic.

7 Strength Reduction

Higher-level optimisations operate on syntax of source language. Functional languages are used to make optimisations more clear.

Algebraic Identities: apply peephole optimisation to abstract syntax trees:

1. a + 0 = a
2. a + (b + c) = (a + b) + c
3. let $x = e$ in if e' then ...x... else e'' $\implies$ if e' then let $x = e$ in ...x... else e'', provided e' and e'' do not contain x

Strength Reduction is an optimisation which replaces expensive operations (multiplication and division) with less expensive ones (addition and subtraction). At syntax tree level, all loop structure is apparent, therefore can more easily do strength reduction.

8 Abstract Interpretation

Idea is that it is tough to make predictions about dynamic behaviour statically, therefore find things out by running a simplified version - by using a model of a reality. This model:

- Discards enough detail that model becomes manageable
- Retains enough detail to provide useful insight into the real world

Multiplication of integers - try to find if positive or negative: (1) Compute in concrete world or (2) Compute in abstract world as follows:

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

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

When you have abstraction, you discard details, therefore important to ensure the imprecision is safe. Abstraction can also be more difficult, for example with addition of integers:

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

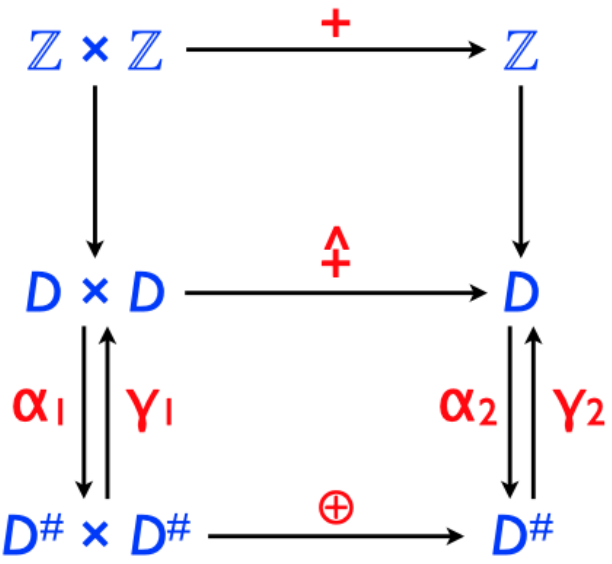
Whilst abstract addition is less precise, it is still safe and hasn't missed anything.

8.1 Formal Definition

Abstraction of some concrete domain D ($\wp(\mathbb{Z})$) consists of:

- Abstract domain $D^{\#}$
- Abstraction function $\alpha : D \longrightarrow D^{\#}$

  Given a function f from one concrete domain to another, we require an abstract function $f^{\#}$ between the corresponding abstract domain
- Concretisation function $\gamma : D^{\#} \longrightarrow D$



where $\alpha_{1,2}$ and $\gamma_{1,2}$ are the appropriate abstraction and concretisation functions

9 Strictness Analysis

Helps to improve the efficiency of compiled functional code.

- Strict functional languages use a **call-by-value** evaluation strategy. This is efficient in space and time, but might evaluate more arguments than necessary.

$$\frac{e_2 \downarrow v_2 \quad e_1[v_2/x] \downarrow v_1}{(\lambda x \cdot e_1) e_2 \downarrow v_1}$$

- Non-strict functional languages use **call-by-name** evaluation strategy. This only evaluates arguments when necessary but it copies and redundantly re-evaluates arguments.

$$\frac{e_1[e_2/x] \downarrow v}{(\lambda x \cdot e_1) e_2 \downarrow v}$$

Optimisation is use call-by-need. If language has no side-effects, duplicated instances of an argument can be shared and evaluated only once. This never changes the semantics of the original program, whereas replacing by call-by-value does (particularly changing the program's termination). Intuitively, safe to use CBV in place of CBN whenever an argument is definitely going to be evaluated.

- **Neededness**: Too conservative - for example: $\lambda x,y,z.$ if x then y else Ω. This function might not evaluate y so only x is needed. But safe to pass y by value, since if y doesn't terminate, then doesn't matter.
- **Strictness**: Safe to pass an argument by value when the function fails to terminate whenever the argument fails to terminate. When this holds, function is strict in that argument. Can use that information to selectively replace CBN with CBV to obtain a more efficient program.

Can perform strictness analysis by abstract interpretation, using recursion equations:

$$F_1(x_1,...,x_{k_1}) = e_1$$
$$... = ...$$
$$F_n(x_1,...,x_{k_n}) = e_n$$
$$e ::= x_i | A_i(e_1,...,e_{r_i}) | F_i(e_1,...e_{k_i})$$

where each A_i is a symbol representing a built-in function of arity r_i

Built in functions are:

- cond(a, b, c)
- eq(a, b)
- plus(a, b)
- add1(a)
- sub1(a)

All that we care about whether an evaluation definitely doesn't terminate or whether it may terminate. Can find strictness function $f_i^{\#}$ for a user dexfined function. Use composition and recursion from built in functions and then:

- $cond^{\#}(p,x,y) = p \wedge (x \vee y)$
- $eq^{\#}(x,y) = x \wedge y$
- $0^{\#} = 1$
- $sub1^{\#}(x) = x$
- $add1^{\#}(x) = x$
- $plus^{\#}(x,y) = \quad x \wedge \Big(y \vee plus^{\#}(x,y)\Big)$. This is clearly a recursive definition, which isn't very useful. We want a definition of $plus^{\#}$ which satisfies this equation - actually want least fixed point of the equation which can be computed iteratively.

  From below: plus $(x,y) = x \wedge y$. plus is strict in both arguments, so can use CBV when passing arguments.

Algorithm

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

$e_i^{\#}$ means "e_i (from the recursion equations) with each A_j replaced with $a_j^{\#}$ and each F_j replaced with f#[j]".

f#[1] := λx,y.0

- On the first iteration, we calculate $e_1^{\#}$. Therefore:

$$e_1^{\#} = \text{cond}^{\#}\big(eq^{\#}(x,0^{\#}), y, (\lambda x,y.0)\big(\text{sub1}^{\#}(x), \text{add1}^{\#}(y)\big)\big)$$

$$e_1^{\#} = x \wedge y$$
$$\text{f\#[1]} \quad := \lambda x,y.x \wedge y$$

- On second iteration, recalculate $e_1^{\#}$, getting:

$$e_1^{\#} = x \wedge y$$

  Therefore, the same, and hence we stop.

10 Constraint-based Analysis

Constraint-based analysis (0CFA) can compute a precise estimate of a prediction of how control can flow at execution time. This control-flow information is required for intra-procedural analysis. This is a hard problem as we have to deal with imprecise control flow where we don't know what code is being referred to.

LVA: Generate equality constraints from each instruction in the program. Then iteratively compute their minimal solution.

0CFA - Zeroth-Order Control-Flow Analysis: Constraint-based analysis for discovering which values may reach different places in a program. When functions (or pointers to functions) are present, this provides information about which functions may potentially be called at each call site. This can be used to build a more precise call graph.

Do this analysis for functional languages with this minimal syntax which can generate programs (closed expressions):

$$e ::= x | c | \lambda x.e | e_I e_2 | \text{let} x = e_1 \text{ in } e_2$$

Can break this down into a graph structure, with a number of program points where:

- Each program point i has an associated flow variable α_i
- Each α_i represents the set of flow values which may be yielded at program point i during execution
- For this language, the flow values are integers and function closures.
- The precise value of each α_i is undecidable in general, so 0CFA over-approximates this.

  Therefore, generate set of constraints on the flow variables which we then treat as data-flow inequalities and iteratively compute their least solution.

$$c^a \longrightarrow \alpha_a \supseteq \{c^a\}$$

$$\left(\lambda x^a.e^b\right) \longrightarrow \quad \alpha_c \supseteq \left\{\left(\lambda x^a.e^b\right)^c\right\}$$

$$(let\_^a = \_^b in\ \_^c)^d \longrightarrow (\alpha_d \supseteq \alpha_c) \cap (\alpha_a \supseteq \alpha_b)$$

$$(\_^a\ \_^b)^c \longrightarrow (\alpha_b \mapsto \alpha_c) \supseteq \alpha_a$$

We then solve the constraints, therefore go through all the elements with no reliance on other elements, and then work backwards. This is still imprecise because it is monovariant: each expression has only one flow variable associated with it so multiple calls to the same function allow multiple values into the single flow variable for the function body and these values leak out at all potential call sites.

1CFA: Better expression thatn 0CFA. Function has a separate flow variable for each call site in the program. This is polyvariant. Can also use polymorphic approach where values themselves are enriched to support specialisation at different call sites (like ML polymorphic types).

11 Inference-based Analysis

Inference systems consist of sets of rules for determining program properties. These properties depend recursively upon the properties of the program's sub-expressions. Inference systems express this relationship and hence compute the property, thereby specifying judgements:

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

where: e is an expression, Γ is a set of assumptions about the free variables of e and ϕ is a program property. For $\phi = t$, means: under the assumptions in Γ, the expression e has type t.

Define:

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

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

Γ of form: $\{x_1 : t_1, ..., x_n : t_n\}$, writing $\Gamma[x:t]$ means that gamma with the additional assumption that x has type t.

Define a set of rules (such as typing rules), to inductively define which judgements are valid. We define these typing rules:

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

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

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

All values must be tagged with their types and use run-time checks since no compile-time checker. If we can check a program is well-types, we can avoid these and tag the final result with the correct type.

Safety condition for inference is:

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

where [e] is value obtained by evaluating e and [t] is the set of values of type t.

Odds and Evens

$$\phi ::= \text{ odd} \mid \text{even} \mid \phi_1 \to \phi_2$$

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

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

$$\frac{\Gamma \vdash e_1 : \phi \to \phi' \quad \Gamma \vdash e_2 : \phi}{\Gamma \vdash e_1 e_2 : \phi'}$$

$[\phi]$: (1) [odd] = $\{\mathbf{z} \in \mathbb{Z} | \mathbf{z} \text{ is odd }\}$, (2) [even] = $\{z \in \mathbb{Z} | z \text{ is even }\}$, (3) $[\phi_1 \to \phi_2] = [\phi_1] \to [\phi_2]$

Can allow conjunctions inside properties, eg: **multiply**: $even \to even \to even \Lambda even \to odd \to even \Lambda odd \to even \to even \Lambda odd \to odd \to odd$

12 Effect Systems

Used to analyse and safely transform programs which contain operations with **side-effects**: Change of state which occurs as a result of evaluating an expression. To do this, we use lambda calculus with read and write operations on channels.

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

- ξ represents a channel name
- $\xi?x.e$ reads an integer from the channel named ξ, binds it to x and returns result of evaluating e
- $\xi?e_1.e_2$ evaluates e_1, writes the resulting integer to channel ξ and returns the result of evaluating e_2
- $\xi!x \cdot x$: Read integer from channel ξ and return it. $F = \{R\xi\}$
- $\xi!x \cdot y$: Write value of x to channel ξ and return the value of y. $F = \{W\xi\}$
- $\xi!x.\zeta!x.x$: Read integer from ξ, write it to channel ζ and return it. $F = \{R\xi, W\zeta\}$

Typing Rules:

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

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

To do transformations on a program in this language, need to pay attention to potential side-effects. Do this by modifying our existing type system to create an effect. F is a set containing elements of the form: $R\xi$ (read from channel ξ) and Wξ (write to channel ξ). Example of F is above. Also need to be able to handle expressions like: $\lambda x.\xi!x.x$. Here, the evaluation doesn't have any immediate effects, but the effect can occur later whenever the newly-created function is applied. To extend the latent effects, extend syntax of types:

$$t ::= int | t_1 \xrightarrow{F} t_2$$

Hence, type of $\lambda x.\xi!x.x$ is: int $\xrightarrow{\{w_\xi\}}$ int. Therefore type system is: $\Gamma \vdash e : t, F$

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

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

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

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

$$\frac{\Gamma\vdash e_1:t\xrightarrow{F''}t',F \quad \Gamma\vdash e_2:t,F'}{\Gamma\vdash e_1e_2:t',F\cup F'\cup F''}(APP)$$

Can also add an if-then-else: if e_1 then e_2 else e_3 with effect rule:

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

However, this does not work (see Lecture 13, slide 22). This is fixed by adding rule to deal with subtyping:

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

This information discovered by the effect system can be used to decide whether particular transformations are safe. Can say that if we have an expression with no side-effects, can be replaced with another expression with no change to the semantics of the program. Referentially transparent expressions may be removed if LVA says they are dead.

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

Since we are using sets of effects, we don't have any information about how many times things happen / what order. Instead can use sequences, appending new items every time.

13 Points-to and Alias Analysis

Want to be able to run calls in parallel, but need to know when the transformations are safe. Can only parallelise if neither call writes to a memory location is read to or written by the other. Therefore want to know at compile time what locations a procedure might write to at run time.

Hence, this means for a given pointer value, we want to find a finite description of what locations it might point to, or for a procedure, a description of what locations it might read from or write to. If the two descriptions have an empty intersection then can parallelise.

- For simple variables, this is simple, but harder for multi-level pointers.
- **Course solution**: treat all allocations done at a single program point as being aliased - as if they all return a pointer to a single piece of memory.

13.1 Andersen's Points-to analysis

All programs re-written so that all pointer-typed operations of the form (no pointer arithmetic):

- x := new_l : l is a program point
- x := y : copy
- x := *y (*x := y) : field access of object

Hence, points-to relation is function: pt: V $\rightarrow P(V)$. This is defined for each function. This has type-like constraints (one per source-level assignment):

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

$$\frac{}{\vdash x:=y:pt(y)\subseteq pt(x)}$$

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

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

Alternatively:

- For command x:=&y, emit constraint pt(x) $\supseteq \{y\}$
- For command x:=y, emit constraint pt(x) $\supseteq$ pt(y)
- For x:=*y, $pt(y)\supseteq\{z\}\Longrightarrow pt(x)\supseteq pt(z)$

- For *x := y, $pt(x) \supseteq \{z\} \Longrightarrow pt(z) \supseteq pt(y)$

This is flow insensitive, we only look at assignments, not in order they occur. Faster but less precise - syntax-directed rules all use the same set-like combination of constraints. This flow-insensitivity means the property inference rules are of the form:

$$(\text{ASS})\frac{}{\vdash x := e : \ldots} \qquad (\text{SEQ})\frac{\vdash C : S \quad \vdash C' : S'}{\vdash C; C' : S \cup S'}$$

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

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

13.2 Other Approaches

1. **Steensgaard's Algorithm**: Treat e:=e' and e':=e identically. It is less accurate than Andersen's algorithm, but runs in almost linear time.
2. **Shape Analysis**: Program analysis with elements being abstract heap nodes and edges between them being must or may point-to. Nodes are labelled with variables and fields which may point to them. This is more accurate but abstract heaps can become large.

Tradeoff between poor results and cost for large programs.
In general alias analysis is undecidable in theory and intractable in practise. Also discontinuous - small changes in program produce global changes in analysis of aliasing.
Fix this using structured programming (and structured data) - restrictions on where pointers can point to.

14 Instruction Scheduling

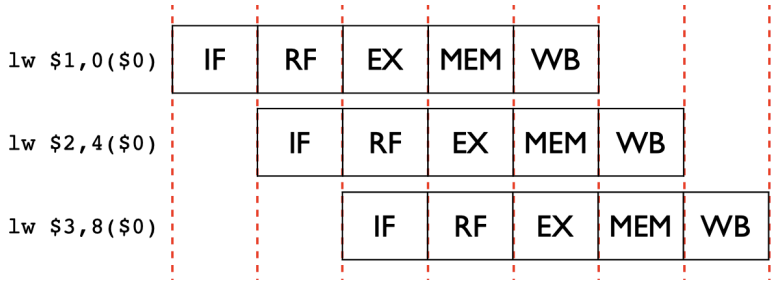
This is a part of the optimisation of the target code, taking target code into more efficient target code.

14.1 Single-Cycle Implementation

Entire instruction executed in a single clock cycle, using some of the processing stages: (1) Instruction Fetch, (2) Register Fetch, (3) Execute, (4) Memory Access, (5) Register Write-Back
Order of instructions doesn't make any difference to execution time. Therefore, can reorder to yield possible benefits.

14.2 Pipelined Implementation

Each processing stage works independently and does its job in a single clock cycle. So different stages can be handling different instructions simultaneously. Results from each unit passed via pipeline registers. Hence:



Hence, clock cycle is much shorter (one pipeline stage vs one complete instruction). However, there are occasionally pipeline hazards. These cause a pipeline stalls; on non-interlocked hardware, the compiler has to generate NOPs to avoid errors.

- **Data hazards**: occur when instruction depends upon result of earlier one. Pipeline has to stall.

  This can be reduced in severity by adding extra paths between functional units. Allows data to be used before written back to registers. Can reorder instructions to reduce the amount of stalls, by considering data dependencies. If one of these exists between two instructions, cannot safely permute them.

- Read after write
- Write after read
- Write after write

Aim of instruction reordering:

1. **Preserve dependencies between instructions**: Construct Directed Acyclic Graph - represent dependencies between instructions. Any topological sort of this DAG maintains dependencies and preserve correctness of the program.
2. **Achieve minimum number of pipeline stalls**: NP-Complete problem, but devise static scheduling heuristics. Each time we emit a new instruction, choose one which:
   - Does not conflict with previous emitted instruction
   - Most likely to conflict if first of a pair
   - As far away as possible from instruction which can be validly scheduled last

14.3 Algorithm

- Construct scheduling DAG: scan backwards through basic block and add edges where dependencies arise
- Initialise candidate list to contain the minimal elements of the DAG.
- While candidate list is non-empty:
  - If possible, emit candidate instruction satisfying all three of the static scheduling heuristics
  - Else, either emit NOP or an instruction satisfying only the last two heuristics
  - Remove instruction from DAG and insert newly minimised elements into the candidate list.

14.4 Dynamic Scheduling

Modern processors have dedicated hardware for performing instruction scheduling dynamically as code is executing. Effectively compiler technology implemented in hardware.

14.5 Allocation vs Scheduling

Register allocation makes instruction scheduling more difficult. In particular, if register allocation is less aggressive, instruction scheduling could be better. In particular, register allocation reduces spills by using fewer registers and instruction scheduling reduces stalls when more registers are used.

One option is to try and allocate architectural registers cyclically rather than re-using them at the earliest opportunity. Hence, to do register allocation by colouring for a basic block, aims are:

1. Satisfy all important constraints as usual
2. See how many spare architectural registers we have
3. For every unallocated virtual register, choose architectural register distinct from all others allocated in the same basic block

Often, withhold registers, then do dynamic recolouring with this larger register set.

15 Decompilation and Reverse Engineering

Take target code to source code. Increase level of abstraction of a system, making it less suitable for implementation but more suitable for comprehension and modification. In general, it is legally complicated - may not be legal to reconstruct source code from target code. But EU law (and DMCA) says that it may not illegal, see Lecture 15.

Impossible to recover lots of information that is changed due to optimisations that occur. Important to note that compilation is not injective - many different source programs results in the same compiled code, so the best we can do is pick a reasonable representative source program.

Can assemble flowgraph from assembler program and locate basic blocks, dealing with semantics of target instructions rather than intermediate 3-address code. Beneficial to convert target instructions back into 3-address code when storing to flowgraph.

Presents problems: Architectures include instructions which test or set condition flags in a status register - necessary to laboriously reconstruct this behaviour with extra virtual registers. Then use dead-code elimination to remove unnecessary generated instructions.

Get high-level control structure by matching intervals of flowgraph with fixed set of familiar syntactic forms from our high-level language. To find high-level structure of loops, use dominance.

Dominance

- Node m dominates node n if control must go through m before it can reach n
- m strictly dominates another node n if m dominates n and $m \neq n$
- Immediate dominator of node n is unique node that strictly dominates n but doesn't dominate any other strict dominator of n.
- Node n is dominance frontier of a node m if m does not strictly dominate n but does dominate an immediate predecessor of n. Set of nodes where m's dominance stops.
- Can represent dominance relation with a dominance tree in which each edge connects a node with its immediate dominator

Back Edges: Edge where head dominates tail. Each back edge has an associated loop where head of the back edge points to the loop header and the loop body consists of all nodes from which the tail can be reached without passing through the loop header.

Once a loop has been identified, can examine structure to determine type of loop (or other structure) and represent it in source code:

- Header contains conditional and last node of body unconditionally transfers control - while()
- Header unconditionally allows body to execute and last node tests whether loop should execute again - do while()
- First node in interval transfers control based on condition - if() then else

15.1 Control Reconstruction

Do as above for whatever other control-flow constructs are available in source language. Once interval matched against higher level control structure, entire subgraph can be replaced with a single node that represents all information.

15.2 Type Reconstruction

Very hard to regain information about the type of variables, since at target code level, no variables, only registers and memory locations. Made even more difficult by SSA (splits user variable into many variables - one for each static assignment) and Register Allocation. But can assign each single register a different type if necessary and reflect this in the source code. Then, we reconstruct the syntax, before propagating the copies and then work out the type (it could be generic based on the known target code). To do this, we use constraint-based analysis with generating constraints for each target instruction. This is based on the types of the registers. These constraints are then solved using the constraints in order to assign types at the source level. Typing information is often incomplete intra-procedurally with constraints generated at call sites helping to fill gaps, For an example, see Lecture 16.