1 Introduction

1.1 The structure of a compiler

The compiling process has two parts. The *analysis* part breaks up the source program and imposes a grammatical structure on them. In addition to this it will alert the user about illegal input (grammar, syntactical etc.) and fill inn the *symbol table*. The *synthesis* part constructs the target program from the intermediate representation and the *symbol table*. The parts are called *front end* and *back end* respectively.

1.1.1 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency. It also gathers type information and saves it to the symbol table. An important part here is *type checking* where it checks that operators have matching operands. It might also handle *coercions*, i.e. transforming INT + FLOAT to FLOAT + FLOAT.

1.2 Programming Language Basics

1.2.1 Aliasing

There is as interesting consequence of call-by-reference (references passed by value). Two formal parameters can refer to the same location, and they are alias of each other. You need to understand this in order to optimize a program, because certain optimizations may only be done if one can be sure there is no aliasing.

2 A Simple Syntax-Directed Translator

2.1 Introduction

Intermediate code-generation comes in two forms, *abstract syntax tree* which represent the hierarchical syntactic structure of the source program. The other one is *three address instructions*

2.2 Syntax-Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar.

2.2.1 Synthesized Attributes

The idea of associating quantities with programming constructs can be expressed in terms of grammars. We attach rules to productions of the grammar. A *syntax-directed definition* associates:

1. With each grammar symbol, a set of attributes
2. With each production, a set of *semantic rules* for computing values of the attributes associated with the symbols appearing in the production.

An attribute is *synthesized* if its value at a parse-tree node N is determined from attribute values at the children of N and N itself, and can be evaluated by a single bottom-up traversal.

2.3 A Translator for Simple Expressions

A syntaxt-directed translation scheme often serves as the specification for a translator. We might come up in a conflict between a translatable and parsable grammar.

2.3.1 Abstract and Concrete Syntax

Useful starting point for creating a translator. AST resemble parse trees to an extent, but these are conceptually different. Parse trees have the interior nodes representing non-terminals, and a syntax tree represent programming constructs. Many productions represent grammar constructs, but many are just there to simplify parsing. The helpers are dropped in the syntax tree.

2.4 Lexical Analysis

A lexical analyzer reads characters from the input and groups them into "token objects" with attribute values. A sequence that comprises a single token is called a *lexeme*. An **num** can have a value and an **id** can have a lexeme.

2.4.1 Removal of White Space and Comments

White space and comments can be removed by the lexer, so the parser wont have to worry.

2.5 Symbol Tables

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phase of a compiler and used in the synthesis part to generate code. Entities in the symbol table contain information about an identifier such as its character string (or lexeme), dits type, its position in storage and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program. We set up symbol tables for each scope in the program.

2.5.1 Symbol Table Per Scope

The term "scope of identifier x" really refers to the scope of a particular decalarion of x. Scopes are important, because common names have multiple uses. The *most closely nested* rule for blocks says that an identifier x is in the scope of the most-closely declaration of x. This behavior can be implementet by chaining symbol tables. It can be implemented with:

- *Create a new symbol table.* Create new table with another symbol table as parent.
- *Put.* Will put key-value pair into current symbol table.
- *Get.* Will try to find the symbol, by scanning itself or recursively scan its ancestors.

2.5.2 The Use of Symbol Tables

The role of a symbol table is to pass information from declaration to uses.

2.5.3 Data structures used for symbol tables

Array Implementation This implementation gives one entry per symbol, and the array must be scanned linearly. This is not ideal because of the linear lookup procedure, and because the table has a fixed size (which must be known beforehand)

List Implementation One structure per entry with pointer to the next entry. This can grow dynamically. This has a slow look-up speed, as half of the elements on average has to be checked.

Hash Table Implementation Uses an hash table that maps a string identifier to a bucket. This gives constant look-up time. I am not familiar with any disadvantages, as long as you have a good hash function that distributes the entries evenly.

2.6 Intermediate Code Generation

2.6.1 Construction of Syntax Trees

Syntax trees can be created for any construct, not just expressions. Each construct is represented by a node, with children for the semantically meaningful components of the construct. These nodes can be represented in a class structure with an arbitrary amount of children.

3 Lexical Analysis

We can write by hand or get a generator (like *lex*) to produce lexical analysers for us.

3.1 The Role of the Lexical Analyzer

The main task of the lexical analyzer is to read input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program, which is then again sent to the syntax analyzer. The syntax analyzer calls *getNextToken* until end of token stream. Lexers might remove whitespace and comments.

3.1.1 Lexical Analysis Versus Parsing

Why split these?

1. Simplicity. Grammars will be much more complicated if it were to handle the lexers job
2. Efficiency. We may use specialized tasks for lexing.
3. Protability.

3.1.2 Tokens, Patterns, and Lexemes

We use terms:

- A *token* is a pair consisting of a token name and an optional value. The name is an abstract symbol.
- A *pattern* is a descritpion of the form that the lexemes of a token may take
- A *lexeme* is a sequence of characters in teh source program that matches the pattern of a token and is identifiyed by the lexical analyzer as an intance of that token.

3.2 The Lexical-Analyzer Generator Lex

This tool allows one to specify a lexical analyzer by specifying regular exrpessions to describe tokens. The expressions are transformed to deterministic finite automatas. Use:

- Lex source program $\rightarrow$ lex compiler $\rightarrow$ c source program
- C source program $\rightarrow$ C compiler $\rightarrow$ Lexer executable
- Input stream $\rightarrow$ Lexer executable $\rightarrow$ Sequence of tokens

3.3 Finite Automata

Essentially graphs, but:

- Finite automata are *recognizers*, says yes or no about input
- Two flavours: *Nondeterministic finitie automata* (NFA) (has no edge restrictions, have empty string and multiple edges for same input) and *Deterministic finite automata* (DFA) (exactly one edge for each input).

Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same langauges, called the *regular languages*, that regular expressions can describe.

3.4 From Regular Expressions to Automata

3.4.1 Conversion of an NFA to a DFA

The general idea is that each state in the DFA reprenents a set of states in the NFA. The number of DFA states is possibly exponential in the number of NFA states, but this behavior is not seen in practice.

4 Syntax Analysis

Every program has rules that describes the syntactic structure. These rules are normally expressed as context-free grammars. Grammars have significant benefits:

- A grammar gives a precise, yet easy-to-understand syntactic specification of a programming language
- Certain classes of grammars can be deterministicly and efficiently parsed, and the process of making the parser might reveal ambigous constructs in the language.

4.1 Writing a grammar

4.1.1 Elimination of Left Recursion

A grammar is *left recursive* if it has a nonterminal A such that $A \xrightarrow{+} A\alpha$. You can remove this from $A \to A\alpha|\beta$ to:

$$\begin{aligned} A &\to \beta A' \\ A' &\to \alpha A'|\epsilon \end{aligned} \tag{1}$$

4.2 Top-Down Parsing

The problem of constructiong a parse tree for the input string, starting from the root and creating the nodes of the porse tree in preorder(depth-first). *Will generate the leftmost derivation.* At each step the problem is to determine the production to apply for each non-terminal. A predictive recursive-descent parser can choose productions by looking at the next symbol. We call thiss the LL(k) class.

4.3 Bottom-Up Parsing

A bottom-up parse corresponds to the construction of a parse tree for an input string that begins at the leaves (the bottom) and working up towards the root (the top). A general style of bottom-up parsing is shift-reduce parsing, which can be built by using LR grammars. It need not to see as much of the parse tree as the top-down parsers.

4.3.1 Reductions

We can thing of bottom-up parsing as the process of reducing a string w to the start symbol of the grammar. Each step matches a substring and replaces it with a non-terminal. By definition, a reduction is the reverse of a step in a derivation. The goal is to construct a derivation in reverse, which in fact will be a rightmost derivation.

4.4 Introduction to LR Parsing: Simple LR

The mot prevalent type of bottom-up parser today is LR(k) parsing; the "L" is for left-to-right, "R" is for rightmost derivation (in reverse) and k is number of lookahead symbols. This section introduces the easiest way to construct shift/reduce parsers.

4.4.1 Why LR Parsers?

LR-parsers are table driven. Attractive because:

- LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
- The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet implemented efficiently.
- Can detect syntax errors as soon as possible
- The class of grammars that can be parsed using LR methods **is a proper superset** of grammars that can be parsed with predictive LL methods.

Typical drawback is that it is too much work to construct an LR parser by hand for a typical language grammar.

4.4.2 Items and the LR(0) automaton

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse, states we call *LR(0)items*. Like:

$$\begin{aligned} A &\to .XY \\ A &\to X.Y \\ A &\to XY. \end{aligned} \tag{2}$$

It indicates how much we have seen. One collection of these LR(0) items is called the *canonical LR(0) collection* forms the basis of the *LR(0) automaton*. To create this we define $FIRST$ and $FOLLOW$, and augment the grammar with a new start symbol S' and a production $S \to S$.

$CLOSURE(I)$

1. Add all items in I to $CLOSURE(I)$
2. If $A \to \alpha.B\beta$ is in $CLOSURE(I)$ and $B \to \gamma$ is a production, then add item $B \to .\gamma$ to $CLOSURE(I)$. Rinse repeat.

You can devide between kernel and nonkernel items, where the latter is everything but the start symbol that has a dot to the left.

$GOTO(I, X)$ Is defined to be the closure of the set of all items $[A \to \alpha X.\beta]$ such that $[A \to \alpha.X\beta]$ is in I

Use of the LR(0) Automaton The central idea behind "Simple LR", or SLR, parsing is the construction from the grammar of the LR(0) automaton. Shift reduce decisions can be made: If automata has transition, shift. If it does not, reduce. There will always only be one applicable reduction.

4.4.3 The LR-Parsing Algorithm

The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The LR parser shifts state when the parser reduces a symbol.

Structure of the LR Parsing Table Two parts:

1. The $ACTION$ function takes I and terminal a which can shift, reduce, accept or error.
2. The $GOTO[I_i, A] = I_j$ is expanded to states. A is a nonterminal.

Behaviour of the LR Parser The next move of the parser from the configuration is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack and lookup $ACTION[s_m, a_i]$

1. If $ACTION$ is shift, pop input off the stack and go to the state in action.
2. If $ACTION$ is reduce $A \to \beta$, pop the length of β off the stack, then perform $GOTO[S_{m-r}, A]$.

Accept and error might also happen.

4.4.4 Constructing SLR-Parsing Tables

Begins with the LR(0) automaton. After constructed:

1. If $[A \to \alpha.a\beta]$ with $GOTO = I_j$, set action to "shift j". a must be terminal.
2. If $[A \to \alpha.]$ is in I_i, then set $ACTION[i, a]$ to "reduce $A \to \alpha$" for all a in $FOLLOW(A)$. If A is S', set accept instead.

4.4.5 Viable Prefixes

Why can LR(0) automata be used to make shift-reduce decisions? Not all prefixes of right-sentential forms can appear on the stack. The parser must make sure to reduce before putting another symbol on stack (in some occasions). The prefixes of right sentinental forms that can appear on the stack of a shift-reduce parser are called *viable prefixes*. They are defined as follows: A viable prefix is a prifx of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form. SLR parsing is pased on the fact that the LR(0) automata recognize viable prefixes.

4.5 More Powerful LR Parsers

This section extends SLR by utilizing lookahead symbols. Two models:

1. The *canonical-LR* or just *LR* method, which makes full use of lookahead. This method uses a large set of items (LR(1) items)
2. The *Lookahead-LR* or LALR, based on LR(0) items. It introduces lookaheads into the LR(0) itemns, and thereby handle more grammars, but keeping the table no bigger than the SLR method.

4.5.1 Constructing LALR Parsing Tables

lookahead-LR. The SLR and LALR tables for a grammar always have the same number of states.

4.6 Using Ambigous Grammars

Any ambigous grammar fails to be in LR. Ambigous grammars might still be useful, because of it's expressiveness. We can specify disambigouating rules, although these should be used iwth care.

4.6.1 Precedence and Associativity to Resolve Conflicts

We might alter productions to give precedence (E $\rightarrow$ E + E — E * E — (E) — id). This might not always be the optimal apporach, because the parser will have more states, and it is hard to modify precedence. We can express conflicts with precedence if we have different operators, associativity when considering equal operators.

4.7 Parser Generators

yacc (yet another compiler-compiler) can construct a translator for a programming language using a specification. It will parse with the LALR method. It is used the same way as *lex* (from specification to program that can read an input stream). The yacc specification file has three parts:

- Declarations
- Translation rules
- Supporting C routines

The Declarations Part Consists of two parts, normal C declarations and declarations of grammar tokens. The grammar docens will then be available for the second and third part. You can make the tokens available for Lex as well.

The Translation Rules Part Put rules here, where each rule consists of grammar productions and associated semantic actions. Unquoted means non-terminals, quoted means terminals. *$$* means the attribute value associated with the head, *$i* means the ith gramar symbol.

The Supporting C-Routines Part Must provide *yylex()* (might come from lex/flex) which produces tokens.

4.7.1 Using Yacc with Ambigous Grammars

Some grammars might produce conflicts, and it will report so. Unless instructed otherwise, it will resolve all conflics with following rules:

- A reduce/reduce is resolved by choosing the conflicting production listed first in the specification.
- A shift/reduce conflict is resolved in favour of shift (will resolve dangling-else ambigouity correctly).

These rules might not be what the compiler writer wants, so we can chose associativity (*%left '+' '-'*) or force it to be nonnassociative. You can also force precedence.

5 Syntax-Directed Translation

This section develops translation guided by contex-free grammars. We associate information with a language construct by attaching *attributes* to the grammar symbols. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree, and the n to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree.

5.1 Syntax-Directed Definitions

A *syntax-directed definition (SDD)* is a context-free grammar together with attributes and rules.

5.1.1 Inherited and Synthesized Attributes

Two kinds of attributes for non-terminals:

1. A *Synthezised attribute*. An attribute at node N that is defined only in terms of attribute values at the children of N and at N itself.
2. An *inherited attribute* at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings.

An SDD that involves only synthesized attributes is called *S-attributed*. Head is computed from the body of the production.

5.1.2 Evaluating an SDD at the Nodes of a Parse Tree

We may visit the nodes in any bottom-up order. We may use an annotated parse tree for this.

5.2 Evaluation Orders for SDD's

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. We introduce *S-attributed* and *L-attributed* SDD's

5.2.1 Dependency Graphs

A *dependency graph* depicts the flow of information among the attribute instances in a particular parse tree. An endge from one attribute to another means that the value of the first is needed to compute the second.

5.2.2 Ordering the Evaluation of Attributes

You may sort the dependency graph topologically. If cycles exist, there is no way to evaluate the SDD.

5.2.3 S-Attributed Definitions

We need to guarantee that there is an evaluation order. *An SDD is S-attributed if every attribute is synthesized*. Can be implemented buring bottom-up parsing, since it corresponds to a postorder traversal.

5.2.4 L-Attributed Definitions

Another way to guarantee evaluation order is to let only the dependency graph have edges that can go from left to right.

6 Intermediate-Code Generation

6.1 Type Checking

To do *type checking* a compiler needs to assign a type expression to each component of the source program. It must then determine that these types conform to the rules (type system) of a program. It has the potential of caching errors. A *sound type system* eliminates the need for dynamic checking for type erors. A language is *strolgy typed* if a compler guarantees that the program will run without type errors.

- Static/dynamic typed is about WHEN type information is aquired (compile-time or run-time)
- Strongly/weakly typed is about how stricly types are distinguished.

6.1.1 Rules for Type Checking

Two forms, *synthesis* which builds up the type of an expression from the types of its subexpressions and *type inference* which determines the type of a language construct from the way it is used. The latter can be used for languages that permits names to be used without declaration, but still uses type checking. *void* is a special type that denote the absence of a value.

6.1.2 Type Conversions

You have *widening* (preserve information) and *narrowing* (may loose information). If type conversions are done automatically by the compiler, it is *implicit*. These conversions are also known as *coercions*, and is normally limited to widening conversions. Two functions, *max(a,b)* finds the highest type in the hierarchy and *widen()* which widens a type.

6.2 Control Flow

6.3 Backpatching

A key problem when generating code for boolean expressions and flow of control statements is that of matching a jump intstruction with the target of the jump. AN approach to this is called *backpatching*, in which lists of jumps are passed as synthesized attributes. When a jump is generated, the target of the jump is tempoarily left unspecified.

6.3.1 One-Pass Code Generation using Backpatching

Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass. We use synthesized attributes *truelist* and *falselist* to manage labels in jumping code for bool expressions.

6.3.2 Backpatching for Boolean Expressions

6.3.3 Flow-of-Control Statements

7 Run-Time Environments

A compiler must implement the abstarctions in the source language. In addition to names, scopes, bindings etc, it must also cooperate with the operating system and other system software. To do this, the compiler creates an *run-time environment* which it assumes the program are being executed. This deals with storage, namning, mechanisms to access variables, parameters, return values, interface to the operating system etc.

7.1 Storage Organization

In the perspective of the compiler writer, the program runs in its own logical address space, where the OS maps from logical to physical. Normal setup is Code $\rightarrow$ Static $\rightarrow$ Heap $\rightarrow$ Free Memory $\rightarrow$ Stack. The amount of storage needed is determined by the types in the program. Many machines requires the data to be *aligned*, that all addresses must be divisible by 4. The compiler may omit this limitation by packing data. Code size and static data is known compile-time.

You have two areas for dynamic storage allocation, the *stack* and the *heap*, which sizes can change during program execution. Some programs allocate the heap automatically and has a garbage collection, some must have the programmer explicitly allocate space on the heap.

7.1.1 Stacit Versus Dynamic Storage Allocation

A storage allocation is static (compile time) if the decicion can be made at looking at the text of a program. A storage allocation is *dynamic* if it can be decided only when the program is running. Two techniques:

- Stack
- Heap. For data that might outlive the call to the procedure that created it.

7.2 Stack Allocation of Space

Almost all compilers for procedures/methods will use the stack. Will make sure no variables overlap in time and the relative addresses are always the same.

7.2.1 Activation Records

Procedure calls and returns are usually managed by a run-time stack (*Control stack*). Each live activation has an activation record (frame). Might include:

- Temporary values
- Local data
- A saved machine status
- Access link
- Control link, pointing to the activation record of the caller.
- Space for return value
- The actual parameters used by the calling procedure.

7.3 Heap Management

For data that must be explicitly deleted.

7.3.1 The Memory Manager

The memory manager keeps track of all the free space in heap storage at all times. Performs:

- *Allocation*. Must provide a memory region in requested size.
- *Deallocation*. For reusing space.

8 Code Generation

The final phase in the compiler is the code generator. It takes IR and input and produces a semantically equivalent target program. It must

- Preserve semantic meaning
- Make efficient use of resources on target machine
- The code generator itself must run efficiently

Making optimal code are undecidable, but many useful methods with heuristics exists which will improve program performance by many factors.

8.1 Addresses in the Target Code

We will have a look at how names in the IR can be converted to addresses in the target code. We have four areas:

1. *Code*. Holds executable target code.
2. *Static*. Hold global constants and other data generated by compiler.
3. *Heap* for holding data that are allocated and freed during execution.
4. *Stack* for data that are created and destroyed during procedure calls

8.2 Instruction Selection by Tree Rewriting

Instruction selection can be a large combinatorial task, especially on CISC. We treat instruction selection as a tree-rewriting problem. Here we can make efficient algorithms.

8.2.1 Tree-Translation Schemes

The input of code-generation will be a sequence of trees at the semantic level. We need to apply a sequence of tree-rewriting rules to reduce the input tree to a single node. Each rule represents the translation of a portion of the tree given by the template. The replacement is called *tiling* of the subtree

8.2.2 Code Generation by Tiling an Input Tree

Given an input tree, the templates in the tree-rewriting rules are applied to tile its subtrees. When matchi is found, the subtree is replaced with replacement node of a rule and action associated with the rule is done (generate code). Two issues:

- How is the tree-pattern matching to be done?
- What if we have more matches?

Must always have a match and not inifite amount of matches. You can create a postfix representation of a tree and create a grammar for it. By reduce-reduce, take largest production, by shift-reduce, shift. This is known as *maximal munch*[1]. This is a good and well understood method, but it has its challenges.

8.3 Register Allocation and Assignment

Instructions involving registers are faster than those referencing memory, so we need to utilitize registers efficiently. Two problems:

- *Register Allocation*. What values?
- *Register Assignment*. Which value should go where?

8.3.1 Global Register Allocation

We would like to keep registers consistent across all block boundaries (live variables). Loops are important. One approach is to have some fixed number of registers holding the inner loop variables, like you could in C by using the *register* keyword.

[1] From wikipedia: In computer programming and computer science, "maximal munch" or "longest match" is the principle that when creating some construct, as much of the available input as possible should be consumed.

8.3.2 Usage Counts

Count how much savings you get by keeping a variable in a register during a loop.

8.3.3 Register Assignment for Outer Loops

Having applied the teqnique from section **??**, the same idea may be applied to progressively larger enclosing loops.

8.3.4 Register Allocation By Graph Coloring

When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (*spilled*). Graph coloring can be used for allocating and managing register spills. Two passes:

The first pass, it assumes it has an infinite number of symbolic registers and selects instructions, and we assume we have reserved registers for stack pointer, frame pointer etc.

The second pass, for each procedure a *register-interference graph* is constructed in which the nodes are symbolic registers and an edge connects swo nodes if one is live at a point where the other is defined. An attempt is made to color the register-interference graph using k colors, where k is the number of available registers. No adjacent nodes have the same color. A color represents a register. Although NP hard problem, we can use the following heuristic technique:

- Remove node n if it has less than k from G and get G'. If G' can be k-colored, G can too (we know we have at least one color available).
- Rinse, repeat. Two things might happen: You get an empty graph (which means success) or you do not (which means you have to spill).

9 Machine-Independent Optimizations

High-level language constructs can introduce substantial run-time overhead if we naively translate each construct independently. We have three forms:

- Local code optimization - within a block
- Global code optimization - across basic blocks
- Interprocedural code optimization - across functions and procedures

Most global optimization are based on *data-flow analyses*.

9.1 The Principal Sources of Optimization

A compiler optimization must presere the semanticts of the original program, and the optimizer is usually only able to apply low-level semantic transformations.

9.1.1 Causes of redundancy

Sometimes redundancy is available at source level, but often introduced because the source is written in a high level language (array accesses will cause redundancy). By having a compiler eliminate the redundancies, we get the best of both worlds: the programs are both efficient and easy to maintain.

9.1.2 A Running Example: Quicksort

blabla

9.1.3 Semantics-Preserving Transformations

A compiler can improve a program without changing the function it computes.

9.1.4 Global Common Subexpressions

An occurence of an expression E is called a *common subexpression* if E was previously computed and the values of the variables in E have not changed since the previous computation.

9.1.5 Copy Propagation

A *copy statement* is an assignment on the form $u = v$. The idea is to use v for u in other statements using u. It may not appear to be an improvement, but dead-code elimination gives the opportunity to get rid of the copy statement.

9.1.6 Dead-Code Elimination

A variable is *live* at a point in a program if its value can be used subesquently, otherwise *dead*. We would like to remove dead statements. Copy propagation makes dead code elimination able to delete many copy statements.

Deducing at compile time that a value of an expression is known as *constant folding*.

9.1.7 Code Motion

Loops are very important place for optimizations. *Code motion* takes an expression that evaluates to the same each time and lifts it out of the loop.

9.1.8 Induction Variables and Reduction in Strength

A variable x is said to be an induction variable if thees is a positive or negative constant c such that each time x is assigned, its value increases by c. They can be computed with a single increment per loop iteration. Transforming from an expensive to a cheap operation is *strength reduction*, which you can do on induction variables. It's normal to work inside-out when optimizing loops. When there ar two or more induction variables in a loop, it might be possible to get rid of all but one.

9.2 Introduction to Data-Flow Analysis

Data-flow analysis refers to a body of techniques that derive information about the flow of data along program execution paths. You can answer many questions about optimization, like global common subexpression and dead code ellimination.

9.2.1 The Data-Flow Abstraction

You can view the program as a series of transformations, where the states are between the statements. Control flow will go from statement to statement in single blocks, and go between blocks if there is an edge between. *Execution path* is the instructions executed, which there normally is an infinite number of. Different analyses may choose to abstract out different information.

The definitions that may reach a program point along some path are known as reaching definitions.

9.2.2 The Data-Flow Analysis Schema

At each program point we associate a *data-flow value* which is the set of all possible program states that can be observed for that point. We denote the data-flow values before and after each statement s by $IN[s]$ and $OUT[s]$, and the *data-flow problem* is to find a solution to a set of constraints on these definitions for all statements s.

Transfer Functions The relationship between the data-flow values before and after the assignment statement is known as a *transfer function*. Might have two directions (forward and backward analysis).

Control-Flow Constraints You can create $IN[B]$ and $OUT[B]$ by running through all the statements.

9.2.3 Data-Flow Schemas on Basic Blocks

Going through a block is easy, just follow each statement, and we get:

$$OUT[B] = f_b(IN[B]) \tag{3}$$

If we do constant propagation, we are interested in the sets of constats that *may* be assigned to a variable, then we have forward flow with union operator.

$$IN[B] = \cup_{P a predecessor of B} OUT[P] \tag{4}$$

No unique solution, but find the most precise that satisfies control-flow and transfer constraints.

9.2.4 Reaching Definitions

One of the most common and useful data-flow schemas. We say a definition d *reaches* a point p if there is a path from the point immediatly following d to p, such that d is not killed along that path. We might have aliases, and that is why we must be conservative.

Transfer Equation for Reaching Definitions

$$f_d(x) = gen_d \cup (x - kill_d) \tag{5}$$

The gen-set contains all definitions that are directly visible after the block (will only generate last assignment of each variable), but the kill-set kills all definitions of assignment (even those inside the same block).

Control-Flow Equations Union is the meet operator for reaching definitions. It creates a summary fo the contributions from different paths at the confluecnce of those paths.

9.2.5 Live-Variable Analysis

eful for register allocation for basic blocks (dead values need not be stored). We use **bacwkard analysis** and we define:

1. def_B as the set of variables defined in the block.
2. use_B as the set of variables used in the block.

About the same as the reaching definitions, except opposite direction:

$$IN[B] = use_b \cup (OUT[B] - def_B) \tag{6}$$

9.2.6 Available Expressions

An expression $x + y$ is *available* at a point p if every path from the entry node to p evaluates $x + y$, and after the last such evaluation prior to reaching p, there are no subesquent assignments to x or y. A block *kills* expression $x + y$ if it assigns (or may assign) x or y and does not subesquently recompute $x + y$. A block *generates* an expression $x + y$ if it evaluates it and not subsequently define x or y. The primary use of available expressions is detecting global common subexpressions. At each step ($x = y + z$):

1. Add to S the expression $y + z$.
2. Delete from S any expression involving x.

Must be done in correct order. Data-flow can be done the same way as reaching definitions (forward analysis), but the meet operator is *intersection rather than union*. An expression is available at the top of a block if it is available at the end of each predecessor block.

In reaching definitions, we start with a set too small and work up, in available expression, we work with a set too large and work down. We start with the assumption that everything is available everywhere, to keep the optimization conservative.

9.3 Foundations of Data-Flow Analysis

After showing the use of data-flow analysis, we present the schemas abstractly. Will cover

1. Under what circumstances is the iterative algorithm used in data-flow analysis correct?
2. How precise is the solution obtained by the iterative algorithm?
3. Will the algorithm converge?
4. What is the meaning of the solution?

We present a general approach. The framework helps us identify reusable components, and therefore programming errors will most likely reduce. A *data-flow analysis framework (D, V, $\wedge$, F)* consists of

1. A direction of the data flow D (forward or backward).
2. A semilattice, which includes a domain of values V and a meet-operator $\wedge$.
3. A family F of transfer functions from V to V. Must contain boundary conditions (transfer functions for ENTRY and EXIT).

9.3.1 Semilattices

A semilattice is a set of values V and a binary meet operator $\wedge$ such for all x, y, z:

1. $x \wedge x = x$ (idempotent)
2. $x \wedge y = y \wedge x$ (commutative)
3. $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (associative)

You also need a *top element*, detoned $\top$ such that $\top \wedge x = x$. Optionally, you may have a *bottom element* detoned $\bot$ such that $\bot \wedge x = \bot$.

Partial orders We define a partial operator on the values, with a relation $\leq$ which is *reflexive*, *antisymmetric* and *transitive*.

The partial Order for a Semilattice For all x and y in V, we define:

$$x \leq y \text{ if and only if } x \wedge y = x \tag{7}$$

We might get into situations where *more* elements are considered to be smaller in the partial order. To say that a set is larger in size but smaller in partial order is counterintuitive, but it is an unavoidable consequence of the definitions.

Greates Lower Bound You can prove that the meet of x and y is their only greatest lower bound.

Lattice Diagrams You can draw the domain V as a lattice diagram, starting from the top element. We do not draw all edges, because of transitivity.

9.3.2 Transfer Functions

The family of transfer function $F : V \to V$ has following properties:

1. F has an identity function I, such that $I(x) = x$ for all x in V
2. F is closed under composition; that is for any two functions f and g in F, the function h defined by $h(x) = g(f(x))$ is in F.

Monotone Frameworks Formally, a data-flow framework is *monotone* if

$$\text{For all } x \text{ and } y \text{ in } V \text{ and } f \text{ in } F, x \leq y \text{ implies } f(x) \leq f(y) \tag{8}$$

This can be proven. You can also define monotonicity as:

$$\text{For all } x \text{ and } y \text{ in } V \text{ and } f \text{ in } F,\ f(x \wedge y \leq f(x) \wedge f(y) \tag{9}$$

Distributive Frameworks Stronger condition than equation **??**:

$$f(x \wedge y) = f(x) \wedge f(y) \tag{10}$$

Distributivity implies monotonicity, converse is not true.

9.3.3 The Iterative Algorithm for General Frameworks

We can use the abstracted methods to create an iterative algorithm, much like the one used for reaching definitions, available expressions and live-variable analysis. We can prove the following:

1. If the algorithm converge, the result is a solution to the data-flow equations
2. If the framework is monotone, the solution found is the *maximum fixedpoint (MFP)* of the data-flow equations. That is a solution where all $IN[B]$ and $OUT[B]$ is smaller $\geq$ than the values of the MFP.
3. If the semilattice of the framework is monotone and of finite height, then the algorithm is guaranteed to converge.

9.3.4 Meaning of a Data-Flow Solution

We know our solution is MPF, but what does that mean? To describe the solution obtained by the iterative algorithm, first describe the ideal solution.

The Ideal Solution Uhm... We claim that:

- Any answer that is greater than ideal is incorrect. This is because if you get a greater solution, a path has been ignored, and we cannot be sure that there is not some effect along that path to invalidate ony program improvement we might make based on the greater solution.
- Any value smaller than or equal to the ideal is conservative, i.e., safe. This is because any solution less than ideal include paths that does not exist is a flow graph, or exists but the program can never follow.

The Meet-Over-Paths Solution Finding all paths is an undecidable problem, therefore we must approximate. We assume that every path in the flow graph can be taken. The paths considered using MOP is a superset of all the paths that can possibly be executed. We can therefore conclude:

$$MOP \leq IDEAL \tag{11}$$

which means MOP is a conservative solution.

The Maximum Fixedpoint Versus the MOP Solution If the graph contains cycles, the MOP solution is still unbounded. The iterative algorithm does not first find all the paths leading to a basic block before applying the meet operator. Rather

1. The algorithms visits basic blocks, not necessarily in the order of execution.
2. At each point, the algorithm applies the meet operator to the values obtained so far. Some of the values might be artificially made in the initialization process, and not represent the path.

So what is the relationship between the MOP solution and the MFP produced by the algorithm? The answer will be the same if the framework is distributive. If it is monotone, we have $IN[B] \leq MOP[B]$. This happens because we apply the meet operator early.

9.4 Loops in Flow Graphs

Loops when considering optimizations are important because programs spend most of their time executing them. Additionally, program analysis can happen faster if it contains no loops, because you only need one pass through the source code.

9.4.1 Dominators

A node d of a flow graph *dominates* node n, written d dom n, if every path from the entry node of the flow graph to n goes through d. Every node dominates itself. The information can be presented in a *dominator tree*, where entry is the root and each node dominates its descendants. The problem can be formulated as a forward data-flow analysis.

9.4.2 Edges in a Depth-First Spanning Tree

We can traverse it with DFS, getting following endges

- Advancing edges
- Retreating edges
- Cross edges

10 Interprocedural Analysis

An interprocedural analysis operates across an entire program. One simple and useful technique is *inlining* of functions, where you replace the function invocation with the body of the function itself. You cannot immideatly inline recursive methods, and inlining may expand code size exponentially (cache miss). Advantages of function inlining is improved speed, due to no method invocation overhead.

11 Cornell Slides

11.1 Lecture 13: Static Semantics

Type inference systems are a declarative formal system used to define typings for legal programs in a language.

$$\vdash E : T \tag{12}$$

means "E is a well-typed construct of type T". Type checking demonstrate the validity. *Hypothetical type judgement*:

$$A \vdash E : T \tag{13}$$

means "In type of context A expression E is well-typed with type T".

11.2 Lecture 17: Types and Type-Checking

Types describes values possibly computed during execution of the program. Type-safety: absence of type errors at run time.

How to ensure Type-Safety? Bind language constructs, then static checkt to enforce type safety.

Statically typed languages Types are defined and checked compile time, and do not change.

Dynamically typed language Types defined and checked at run-time during program execution.

11.2.1 Strong vs. weak typing

Refer to how much type consistency is enforced. **Strongly typed languages:** guarantees that accepted programs are type-safe. **Weakly typed languages** allow programs to contain type errors. Can archieve strong typing with both static and dynamic typing.

Sondness Sound type systems can statically ensure that the program is type safe. Soundness implies strong typing. May reject type-safe programs, but reject as few as possible.

11.3 Lecture 22: Implementing Objects

About classes. Fields have values that are different between objects and usually mutable. Methods are shared by all objects of a class and usually immutable.

11.3.1 Methods

You have implicit receiver argument, and you must use dispatch vector instead of jumping to absolute address (compiler cannot tell which method to call). All methods has its own integer index, used to look up in dispatch vector. Interfaces has no dispatch vector, only a dispatch vector layout. Abstract classes are halfway. DV only needed for concrete classes. Static methods in java are not really methods. The pointer to the dispatch vector is saved as a field in the class.

11.4 Lecture 23: Introduction to Optimizations

We have IR code, and want to optimize at this level. We can optimize in time and space, but all transformations must be safe. It is normally a tradeoff between space and time.

Constant propagation If a variable is known to be a constant, replace the use of variable with constants.

Constant Folding Evaluate operands if they are known compile-time (constants)

Algebraic Simplification Use algebraic rules:

$$\frac{y * 1 + 0}{1} \rightarrow y * 1 + 0 \rightarrow y * 1 \rightarrow y \tag{14}$$

Copy Propagation For $x = y$, replace uses of x with y. Apply transitvely.

Common Subexpression Elimination If program computes same expression mulitple times, we can reuse the computed value.

Ureachable Code elimination Eliminate code that is never used (loop condition always false or if-statement always true/false.

Dead Code Elimination Remove statements that define variables that are never used. Other optimizations might create those

11.4.1 Loop Optimization

Most time (90/10) spent in loop.

Loop-Invariant Code motion Hoist computation that does not change during the loop out. Must be safe!

Strength Reduction Replaces expensive operations by cheap ones. *Induction variable* is a loop variable whose value depend linearly on the itarion number. Apply strength reduction here. Also ohter strength reduction ($x * 2 = x + x$, $i * 2^c = i << c$).

Induction Variable Elimination If there are multiple induction variables in the loop, eliminate the ones that are only used in the test. Will need to rewrite test.

Loop Unrolling Execute loop body multiple times at each iteration, get rid of conditional branches. Space-time tradeoff: program size increases.

Funciton Inlining Replace function call with the body of the function. Not applicable for recursive procedures, and hard to inline methods.

Function cloning Create specialized verions of fuctions that are called from different call sites with different arguments.

11.5 Lecture 33: Register Allocation

Low IR manipulate data in local/temp variables, assembly in memory/registers. The compiler backend must allocate variables to memory or registers in the generated code.

Register allocation: Keep variable in registers as long as possible, ideally as long as the lifetime of the variable (never need to store it to memory). Main idea: *cannot allocate two variables to the same register if they are both live at some program point.* Algorithm:

1. Perform live variable analysis (over abstract assembly code)
2. Get live variables in each program point
3. Variables in the same set can't be allocated in the same register, they interfere.
4. If they dont interfere, they can have the same register.

Draw interferences in a graph. Same algorithm as in section **??**. Add nodes back one by one and assign colors (registers).

If you have a graph left, it might be colorable (but NP hard). *Optimistic Coloring* is to try to color the spilled node. If not possible, you get *actual spill.*

11.5.1 Accessing Spilled Variables

- *Simple*: Reserve extra registers for shuttling data
- *Better*: Rewrite code introducing a new temporary and rerun liveness analysis and register allocation.

Example: assume:

```
add r1, r1, r2
```

If $r2$ assigned for spilling, rewrite to:

```
ldr t1, [fp, #-8]
add r1, r1, t1
```

$t1$ has very short lifetime. Rerun algorithm.

11.5.2 Optimizing Move Instructions

You can get rid of move instruction

```
mov t5, t9
```

if $t5$ and $t9$ are not connected in the interference graph. Assign them to the same register. This might make a graph uncolorable. *Conservative Coalescing*: Ensure that coalescing doesn't make the graph non-colorable.

12 Other definitions

Loop Unrolling Loop unrolling creates more instructions in the loop body, permitting global scheduling algorithms to find more parallelism. Will increase code size, and hereby introduce more cache misses and memory references.