Lecture Notes on Compiler Design: Overview

15-411: Compiler Design
Frank Pfenning

Lecture 1
August 27, 2013

1 Introduction

This course is a thorough introduction to compiler design, focusing on more low-level and systems aspects rather than high-level questions such as polymorphic type inference or separate compilation. You will be building several complete end-to-end compilers for successively more complex languages, culminating in a mildly optimizing compiler for a safe variant of the C programming language to x86-64 assembly language. For the last project you will have the opportunity to optimize more aggressively, to implement a garbage collector, or retarget the compiler to an abstract machine.

In this overview we review the goals for this class and give a general description of the structure of a compiler. Additional material can be found in the optional textbook [App98, Chapter 1].

2 Goals

After this course you should know how a compiler works in some depth. In particular, you should understand the structure of a compiler, and how the source and target languages influence various choices in its design. It will give you a new appreciation for programming language features and the implementation challenges they pose, as well as for the actual hardware architecture and the runtime system in which your generated code executes. Understanding the details of typical compilation models will also make you a more discerning programmer.

You will also understand some specific components of compiler technology, such as lexical analysis, grammars and parsing, type-checking, intermediate representations, static analysis, common optimizations, instruction selection, register allocation, code generation, and runtime organization. The knowledge gained

should be broad enough that if you are confronted with the task of contributing to the implementation of a real compiler in the field, you should be able to do so confidently and quickly.

For many of you, this will be the first time you have to write, maintain, and evolve a complex piece of software. You will have to program for correctness, while keeping an eye on efficiency, both for the compiler itself and for the code it generates. Because you will have to rewrite the compiler from lab to lab, and also because you will be collaborating with a partner, you will have to pay close attention to issues of modularity and interfaces. Developing these software engineering and system building skills are an important goal of this class, although we will rarely talk about them explicitly.

3 Compiler Requirements

As we will be implementing several compilers, it is important to understand which requirement compilers should satisfy. We discuss in each case to what extent it is relevant to this course.

Correctness. Correctness is absolutely paramount. A buggy compiler is next to useless in practice. Since we cannot formally prove the correctness of your compilers, we use extensive testing. This testing is end-to-end, verifying the correctness of the generated code on sample inputs. We also verify that your compiler rejects programs as expected when the input is not well-formed (lexically, syntactically, or with respect to the static semantics), and that the generated code raises an exception as expected if the language specification prescribes this. We go so far as to test that your generated code fails to terminate (with a time-out) when the source program should diverge.

Emphasis on correctness means that we very carefully define the semantics of the source language. The semantics of the target language is given by the GNU assembler on the lab machines together with the semantics of the actualy machine. Unlike C, we try to make sure that as little as possible about the source language remains undefined. This is not just for testability, but also good language design practice since an unambiguously defined language is portable. The only part we do not fully define are precise resource constraints regarding the generated code (for example, the amount of memory available).

Efficiency. In a production compiler, efficiency of the generated code and also efficiency of the compiler itself are important considerations. In this course, we set very lax targets for both, emphasizing correctness instead. In one of the later labs in the course, you will have the opportunity to optimize the generated code.

The early emphasis on correctness has consequences for your approach to the design of the implementation. Modularity and simplicity of the code are important for two reasons: first, your code is much more likely to be correct, and, second, you will be able to respond to changes in the source language specification from lab to lab much more easily.

Interoperability. Programs do not run in isolation, but are linked with library code before they are executed, or will be called as a library from other code. This puts some additional requirements on the compiler, which must respect certain interface specifications.

Your generated code will be required to execute correctly in the environment on the lab machines. This means that you will have to respect calling conventions early on (for example, properly save callee-save registers) and data layout conventions later, when your code will be calling library functions. You will have to carefully study the ABI specification [MHJM09] as it applies to C and our target architecture.

Usability. A compiler interacts with the programmer primarily when there are errors in the program. As such, it should give helpful error messages. Also, compilers may be instructed to generate debug information together with executable code in order help users debug runtime errors in their program.

In this course, we will not formally evaluate the quality or detail of your error messages, although you should strive to achieve at least a minimum standard so that you can use your own compiler effectively.

Retargetability. At the outset, we think of a compiler of going from one source language to one target language. In practice, compilers may be required to generate more than one target from a given source (for example, x86-64 and ARM code), sometimes at very different levels of abstraction (for example, x86-64 assembly or LLVM intermediate code).

In this course we will deemphasize retargetability, although if you structure your compiler following the general outline presented in the next section, it should not be too difficult to retrofit another code generator. One of the options for the last lab in this course is to retarget your compiler to produce code in a low-level virtual machine (LLVM). Using LLVM tools this means you will be able to produce efficient binaries for a variety of concrete machine architectures.

4 The Structure of a Compiler

Certain general common structures have arisen over decades of development of compilers. Many of these are based on experience and sound engineering princi-

ples rather than any formal theory, although some parts, such as parsers, are very well understood from the theoretical side. The overall structure of a typical compiler is shown in Figure 1.

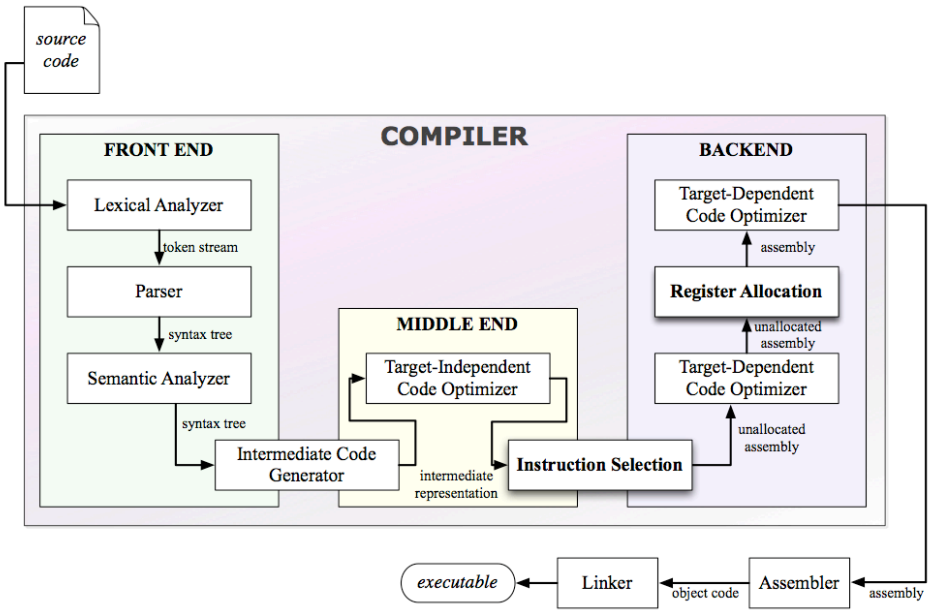


Figure 1: Structure of a typical compiler[2]

In this course, we will begin by giving you the front and middle ends of a simple compiler for a very small language, and you have to write the back end, that is, perform instruction selection and register allocation. Consequently, Lectures 2 and 3 will be concerned with instruction selection and register allocation, respectively, so that you can write your own.

We then turn to the front end and follow through the phases of a compiler in order to complete the picture, while incrementally complicating the language features you have to compile. Roughly, we will proceed as follows, subject to adjustment throughout the course:

1. A simple expression language
2. Loops and conditionals
3. Functions
4. Structs and arrays

5. Memory safety and basic optimizations

The last lab is somewhat open-ended and allows either to implement further optimizations, a garbage collector, or a new back end which uses the low-level virtual machine (LLVM)[1].

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[MHJM09] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V application binary interface, AMD64 architecture processor supplement. Available at http://www.x86-64.org/documentation/abi.pdf, May 2009. Draft 0.99.

[1] See http://llvm.org

[2] Thanks to David Koes for this diagram.

Lecture Notes on Instruction Selection

15-411: Compiler Design
Frank Pfenning

Lecture 2
August 29, 2013

1 Introduction

In this lecture we discuss the process of instruction selection, which typcially turns some form of intermediate code into a pseudo-assembly language in which we assume to have infinitely many registers called "temps". We next apply register allocation to the result to assign machine registers and stack slots to the temps before emitting the actual assembly code. Additional material regarding instruction selection can be found in the textbook [App98, Chapter 9].

2 A Simple Source Language

We use a very simple source language where a program is just a sequence of assignments terminated by a return statement. The right-hand side of each assignment is a simple arithmetic expression. Later in the course we describe how the input text is parsed and translated into some intermediate form. Here we assume we have arrived at an intermediate representation where expressions are still in the form of trees and we have to generate instructions in pseudo-assembly. We call this form *IR Trees* (for "Intermediate Representation Trees").

We describe the possible IR trees in a kind of pseudo-grammar, which should not be read as a description of the concrete syntax, but the recursive structure of the data.

Programs	$\vec{s}$	::=	$s_1, \ldots, s_n$	sequence of statements
Statements	s	::=	$t = e$	assignment
		|	return e	return, always last
Expressions	e	::=	c	integer constant
		|	t	temp (variable)
		|	$e_1 \oplus e_2$	binary operation
Binops	$\oplus$	::=	$+ \mid - \mid * \mid / \mid \ldots$	

3 Abstract Assembly Target Code

For our very simple source, we use an equally simple target. Our target language has fixed registers and also arbitrary temps, which it shares with the IR trees.

Programs	$\vec{i}$	::=	$i_1, \ldots, i_n$	
Instructions	i	::=	$d \leftarrow s$	
		|	$d \leftarrow s_1 \oplus s_2$	
Operands	d, s	::=	r	register
		|	c	immediate (integer constant)
		|	t	temp (variable)

We use d to denote operands of instructions that are *destinations* of operations and s for *sources* of operations. There are some restrictions. In particular, immediate operands cannot be destinations. More restrictions arise when memory references are introduced. For example, it may not be possible for more than one operand to be a memory reference.

4 Maximal Munch

The simplest algorithm for instruction selection proceeds top-down, traversing the input tree and recursively converting subtrees to instruction sequences. For this to work properly, we either need to pass down or return a way to refer to the result computed by an instruction sequence. In lecture, it was suggest to pass down a *destination* for the result of an operation. We therefore have to implement a function

cogen(d, e)	a sequence of instructions implementing e, putting the result into destination d.

e	$\mathsf{cogen}(d, e)$	proviso
c	$d \leftarrow c$	
t	$d \leftarrow t$	
$e_1 \oplus e_2$	$\mathsf{cogen}(t_1, e_1), \mathsf{cogen}(t_2, e_2), d \leftarrow t_1 \oplus t_2$	(t_1, t_2 new)

If our target language has more specialized instructions we can easily extend this translation by matching against more specialized patterns and matching against them first. For example: if we want to implement multiplication by the constant 2 with a left shift, we would add one or two patterns for that.

e	$\mathsf{cogen}(d, e)$	proviso
c	$d \leftarrow c$	
t	$d \leftarrow t$	
$2 * e$	$\mathsf{cogen}(t, e), d \leftarrow t \mathsf{<<} 1$	(t new)
$e * 2$	$\mathsf{cogen}(t, e), d \leftarrow t \mathsf{<<} 1$	(t new)
$e_1 \oplus e_2$	$\mathsf{cogen}(t_1, e_1), \mathsf{cogen}(t_2, e_2), d \leftarrow t_1 \oplus t_2$	(t_1, t_2 new)

Since $*$ is a binary operation (that is, $\oplus$ can be $*$), the patterns for e now need to be matched in order to avoid ambiguity and to obtain the intended more efficient implementation. If we always match the deepest pattern first at the root of the expression, this algorithm is called *maximal munch*. This is also a first indication where the built-in pattern matching capabilities of functional programming languages can be useful for implementing compilers.

Now the translation of statements is straightforward. We write $\mathsf{cogen}(s)$ for the sequence of instructions implementing statement s. We assume that there is a special return register r_{ret} so that a return instruction is translated to a move into the return register.

s	$\mathsf{cogen}(s)$
$t = e$	$\mathsf{cogen}(t, e)$
$\mathsf{return}\ e$	$\mathsf{cogen}(r_{\mathrm{ret}}, e)$

Now a sequence of statements constituting a program is just translated by appending the sequences of instructions resulting from their translations. Maximal munch is easy to implement (especially in a language with pattern matching) and gives acceptable results in practice.

5 Optimal Instruction Selection

If we have a good cost model for instructions, we can often find better translations if we apply dynamic programming techniques to construct instruction sequences

of minimal cost, from the bottom of the tree upwards. In fact, one can show that we get "optimal" instruction selection in this way if we start with tree expressions.

On modern architectures it is very difficult to come up with realistic cost models for the time of individual instructions. Moreover, these costs are not additive due to features of modern processors such as pipelining, out-of-order execution, branch predication, hyperthreading, etc. Therefore, optimal instruction selection is more relevant when we optimize code size, because then the size of instructions is not only unambiguous but also additive. Since we do not consider code-size optimizations in this course, we will not further discuss optimal instruction selection.

6 x86-64 Considerations

Assembly code on the x86 or x86-64 architectures is not as simple as the assumptions we have made here, even if we are only trying to compile straight-line code. One difference is that the x86 family of processors has two-address instructions, where one operand will function as a source as well as destination of an instruction, rather than three-address instructions as we have assumed above. Another is that some operations are tied to specific registers, such as integer division, modulus, and some shift operations. We briefly show how to address such idiosyncracies.

To implement a three-address instruction we replace it by a move and a two-address instruction. For example:

3-address form	2-address form	x86-64 assembly
$d \leftarrow s_1 + s_2$	$d \leftarrow s_1$	`MOVL` s_1, d
	$d \leftarrow d + s_2$	`ADDL` s_2, d

Here we use the GNU assembly language conventions where the destination of an operation comes last, rather than the Intel assembly language format where it comes first.

In order to deal with operations tied to particular registers we have to make similar transformations. It is important to keep the live range of these registers short, so they interfere with other registers as little as possible, as explained in Lecture 3 on register allocation. As an example, we consider integer division. On the left is the simple three-address form. In the middle is a reasonable approximation in two-address form. On the right is the actual x86 assembly.

3-address form	2-address form (approx.)	x86-64 assembly
$d \leftarrow s_1 \;/\; s_2$	`%eax` $\leftarrow s_1$	`MOVL` s_1`,%eax`
		`CLTD`
	`%eax` $\leftarrow$ `%eax` $/\; s_2$	`IDIVL` s_2
	$\|\|$ `%edx` $\leftarrow$ `%eax` $\%\; s_2$	
	$d \leftarrow$ `%eax`	`MOVL` `%eax`, d

Here, `CLTD` sign-extends `%eax` into `%edx`. In the Intel Instruction Set Reference, this instruction is called `CDQ`. This is one of relatively few places where the Intel and GNU assembler names of instructions differ. The `IDIVL` s_2 instruction divides the 64-bit number represented by [`%edx`, `%eax`] by s_2, storing the quotient in `%eax` and the remainder in `%edx`. Note that the `IDIVL` instruction will raise a division by zero exception when s_2 is 0, or if there is an overflow (if we divide the smallest 32 bit integer in two's complement representation, -2^{31}, by -1).

7 Extensions

In general, there will be interdependencies of instruction selection and register allocation. The register allocation depends on which instructions are executed, especially for special instructions on x86-64. Also some of the analysis needed for register allocation may depend on the selected instructions. Conversely, however, optimal instructions may depend on the register assignment. For these and similar reasons, recent advanced compilers, especially those following the so-called SSA intermediate representation combine register allocation and code generation into a joint phase.

Questions

1. How can you implement the data structures for an intermediate representation as defined in this lecture?
2. What are the advantages of working with a 3-address intermediate representation compared to a 2-address representation and vice versa?
3. What is the advantage and disadvantage of using macro expansion for instruction selection, i.e., to associate exactly one instruction sequence to each individual piece of the intermediate language?
4. Why do many CPUs provide such an asymmetric set of instructions? Why do they not just provide us with all useful instructions and no special register requirements?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

Lecture Notes on Register Allocation

15-411: Compiler Design
Frank Pfenning, André Platzer

Lecture 3
September 3, 2013

1 Introduction

In this lecture we discuss register allocation, which is one of the last steps in a compiler before code emission. Its task is to map the potentially unbounded numbers of variables or "temps" in pseudo-assembly to the actually available registers on the target machine. If not enough registers are available, some values must be saved to and restored from the stack, which is much less efficient than operating directly on registers. Register allocation is therefore of crucial importance in a compiler and has been the subject of much research. Register allocation is also covered thoroughly in the textbook [App98, Chapter 11], but the algorithms described there are complicated and difficult to implement. We present here a simpler algorithm for register allocation based on *chordal graph coloring* due to Hack [Hac07] and Pereira and Palsberg [PP05]. Pereira and Palsberg have demonstrated that this algorithm performs well on typical programs even when the interference graph is not chordal. The fact that we target the x86-64 family of processors also helps, because it has 16 general registers so register allocation is less "crowded" than for the x86 with only 8 registers (ignoring floating-point and other special purpose registers).

Most of the material below is based on Pereira and Palsberg [PP05][1], where further background, references, details, empirical evaluation, and examples can be found.

2 Building the Interference Graph

Two variables need to be assigned to two different registers if they need to hold two different values at some point in the program. This question is undecidable in

[1] Available at http://www.cs.ucla.edu/~palsberg/paper/aplas05.pdf

general for programs with loops, so in the context of compilers we reduce this to *liveness*. A variable is said to be *live* at a given program point if it will be used in the remainder of the computation. Again, we will not be able to able to accurately predict at compile time whether this will be the case, but we can approximate liveness through a particular form of *dataflow analysis* discussed in the next lecture. If we have (correctly) approximated liveness information for variables then two variables cannot be in the same register wherever their live ranges overlap, because they may both be then used at the same time.

In our simple straight-line expression language, this is particularly easy. We traverse the program backwards, starting at the last line. We note that the return register, `%eax`, is live after the last instruction. If a variable is live on one line, it is live on the preceding line unless it is assigned to on that line. And a variable that is used on the right-hand side of an instruction is live for that instruction.[2]

As an example, we consider the simple straight-line computation of the fifth Fibonacci number, in our pseudo-assembly language. We list with each instruction the variables that are live *before* the line is executed. These are called the variables *live-in* to the instruction.

			live-in	
f_1	$\leftarrow$	1	$\cdot$	
f_2	$\leftarrow$	1	f_1	
f_3	$\leftarrow$	$f_2 + f_1$	f_2, f_1	
f_4	$\leftarrow$	$f_3 + f_2$	f_3, f_2	
f_5	$\leftarrow$	$f_4 + f_3$	f_4, f_3	
`%eax`	$\leftarrow$	f_5	f_5	
`return`			`%eax`	return register

The nodes of the *interference graph* are the variables and registers of the program. There is an (undirected) edge between two nodes if the corresponding variables interfere and should be assigned to different registers. There are never edges from a node to itself, because, at any particular use, variable x is put in the same register as variable x. We distinguish the two forms of instructions.

- For an $t \leftarrow s_1 \oplus s_2$ instruction we create an edge between t and any different variable $t_i \neq t$ live after this line, i.e., live-in at the successor. t and t_i should be assigned to different registers, because otherwise the assignment to t could destroy the proper contents of t_i.
- For a $t \leftarrow s$ instruction (move) we create an edge between t and any variable t_i live after this line different from t *and* s. We omit the potential edge between

[2]Note that we do not always have to put the same variable in the same register at all places, but could possibly choose different registers for the same variables at different instructions (given suitable copying back and forth). But SSA already takes care of this issue as we will see later.

t and s because if they happen to be assigned to the same register, they still hold the same value after this (now redundant) move. Of course, there may be other occurrences of t and s which force them to be assigned to different registers.

For the above example, we obtain the following interference graph.

Here, the register `%eax` is special, because, as a register, it is already predefined and cannot be arbitrarily assigned to another register. Special care must be taken with predefined registers during register allocation; see some additional remarks in Section 9.

3 Register Allocation via Graph Coloring

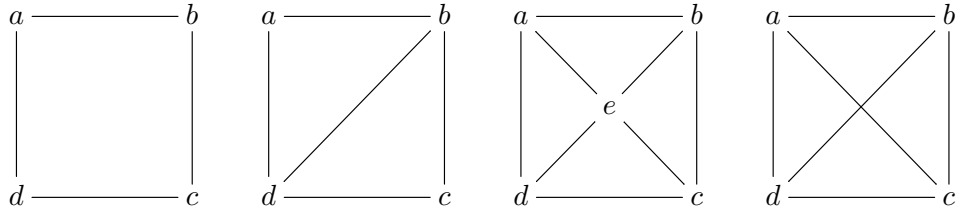
Once we have constructed the interference graph, we can pose the register allocation problem as follows: construct an assignment of K colors (representing K registers) to the nodes of the graph (representing variables) such that no two connected nodes are of the same color. If no such coloring exists, then we have to save some variables on the stack which is called *spilling*.

Unfortunately, the problem whether an arbitrary graph is K-colorable is NP-complete for $K \geq 3$. Chaitin [Cha82] has proved that register allocation is also NP-complete by showing that for any graph G there exists some program which has G as its interference graph. In other words, one cannot hope for a theoretically optimal and efficient register allocation algorithm that works on all machine programs.

Fortunately, in practice the situation is not so dire. One particularly important intermediate form is *static single assignment* (SSA). Hack [Hac07] observed that for programs in SSA form, the interference graph always has a specific form called *chordal*. Coloring for chordal graphs can be accomplished in time $O(|V| + |E|)$ (hence at most quadratic in size) and is quite efficient in practice. Better yet, Pereira and Palsberg [PP05] noted that as much as 95% of the programs occurring in practice have chordal interference graphs anyhow. Moreover, using the algorithms designed for chordal graphs behaves well in practice even if the graph is not quite chordal, which will just lead to unnecessary spilling, not incorrectness. Finally, the algorithms needed for coloring chordal graphs are quite easy to implement compared, for example, to the complex algorithm in the textbook. You are, of course, free to choose any algorithm for register allocation you like, but we would suggest one based on chordal graphs explained in the remainder of this lecture.

4 Chordal Graphs

An undirected graph is *chordal* if every cycle with 4 or more nodes has a chord, that is, an edge not part of the cycle connecting two nodes on the cycle. Consider the following three examples:



Only the second and fourth are chordal (how many cycles need to be checked for chords?). In the other two, the cycle $abcd$ does not have a chord. In particular, the effect of the non-chordality is that a and c as well as b and d, respectively, can safely use the same color, unlike in the chordal case.

On chordal graphs, optimal coloring can be done in two phases, where optimal means using the minimum number of colors. In the first phase we determine a particular ordering of the nodes in which we proceed when coloring the nodes. This order is called *simplicial elimination ordering*. In the second phase we apply *greedy coloring* based on this order. These are explained in the next two sections.

5 Simplicial Elimination Ordering

A node v in a graph is *simplicial* if its neighborhood forms a clique, that is, all neighbors of v are connected to each other, hence all need different colors. An ordering $v_1, \ldots, v_n$ of the nodes in a graph is called a *simplicial elimination ordering* if every node v_i is simplicial in the subgraph $v_1, \ldots, v_i$. Interestingly, a graph has a simplicial elimination ordering if and only if it is chordal. That is, we will not be making a suboptimal decision on those graphs by pretending that all previously occurring neighbors need to be assigned different colors. Furthermore, the number of colors needed for a chordal graph is at most the size of its largest clique.

We can find a simplicial elimination ordering using *maximum cardinality search*, which can be implemented to run in $O(|V| + |E|)$ time (so at most quadratic in the size of the program). The algorithm associates a weight $\mathrm{wt}(v)$ with each vertex which is initialized to 0 updated by the algorithm. The weight $w(v)$ represents how many neighbors of v have been chosen earlier during the search. We write $N(v)$ for the neighborhood of v, that is, the set of all adjacent nodes.

If the graph is not chordal, the algorithm will still return some ordering although it will not be simplicial. Such an ordering from a non-chordal graph can

still be used correctly in the coloring phase, but does not guarantee that only the minimal numbers of colors will be used. Essentially, for non-chordal graphs, generating an elimination ordering in the way described here amounts to pretending that all nodes of the neighborhood are in conflict, which is conservative but suboptimal. For chordal graphs the assumption is actually justified and the correctly allocated registers are also optimal.

Algorithm: Maximum cardinality search
Input: $G = (V, E)$ with $|V| = n$
Output: A simplicial elimination ordering $v_1, \ldots, v_n$
For all $v \in V$ set $\mathrm{wt}(v) \leftarrow 0$
Let $W \leftarrow V$
For $i \leftarrow 1$ to n do
    Let v be a node of maximal weight in W
    Set $v_i \leftarrow v$
    For all $u \in W \cap N(v)$ set $\mathrm{wt}(u) \leftarrow \mathrm{wt}(u) + 1$
    Set $W \leftarrow W \setminus \{v\}$

In our example,

$$f_1 \text{ —— } f_2 \text{ —— } f_3 \text{ —— } f_4 \qquad f_5 \qquad \boxed{\texttt{\%eax}}$$

if we pick f_1 first, the weight of f_2 will become 1 and has to be picked second, followed by f_3 and f_4. Only f_5 is left and will come last, ignoring here the node `%eax` which is already colored into a special register. It is easy to see that this is indeed a simplicial elimination ordering.

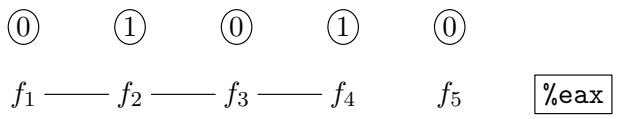
In contrast, $f_2, f_4, f_3, \ldots$ is not, because the neighborhood of f_3 in the subgraph f_2, f_4, f_3 does not form a clique. Indeed, when giving arbitrary (let's say different) colors to f_2 and f_4 in this order, they would require f_3 to assume a third color, which is suboptimal.

6 Greedy Coloring

Given an ordering, we can apply greedy coloring by simply assigning colors to the vertices in this order, always using the lowest available color. Initially, no colors are assigned to nodes in V. We write $\Delta(G)$ for the maximum out-degree of a node in G. The algorithm will always assign at most $\Delta(G) + 1$ colors. If the ordering is a simplicial elimination ordering, the result is furthermore guaranteed to be optimal, i.e., use the fewest possible colors.

Algorithm: Greedy coloring
Input: $G = (V, E)$ and ordered sequence $v_1, \ldots, v_n$ of nodes.
Output: Assignment $\mathrm{col} : V \to \{0, \ldots, \Delta(G)\}$.
For $i \leftarrow 1$ to n do
    Let c be the lowest color not used in $N(v_i)$
    Set $\mathrm{col}(v_i) \leftarrow c$

In our example, we would just alternate color assigments:



Of course, `%eax` is represented by one of the colors. Assuming this color is 0 and `%edx` is the name of register 1, we obtain the following program:

```
%eax ← 1
%edx ← 1
%eax ← %edx + %eax
%edx ← %eax + %edx
%eax ← %edx + %eax
%eax ← %eax                    // redundant self move
```

It should be apparent that some optimizations are possible. Some are immediate, such as the redundant move of a register to itself. We discuss another one called *register coalescing* in Section 8.

7 Register Spilling

So consider that we have applied the above coloring algorithm and it turns out that there are more colors needed than registers available. In that case we need to save some temporary values. In our runtime architecture, the stack is the obvious place. One convenient way to achieve this is to simply assign stack slots instead of registers to some of the colors. The choice of which colors to spill can have a drastic impact on the running time. Pereira and Palsberg suggest two heuristics: (i) spill the least-used color, and (ii) spill the highest color assigned by the greedy algorithm. For programs with loops and nested loops, it may also be significant *where* in the programs the variables or certain colors are used: keeping variables used frequently in inner loops in registers may be crucial for certain programs.

Once we have assigned stack slots to colors, it is easy to rewrite the code using temps that are spilled if we reserve a register in advance for moves to and from the stack when necessary. For example, if `%r11` on the x86-64 is reserved to implement save and restore when necessary, then

$$t \leftarrow t + s$$

where t is assigned to stack offset 8 and s to `%eax` can be rewritten to

$$\begin{array}{lcl} \texttt{\%r11} & \leftarrow & 8(\texttt{\%rsp}) \\ \texttt{\%r11} & \leftarrow & \texttt{\%r11} + \texttt{\%eax} \\ 8(\texttt{\%rsp}) & \leftarrow & \texttt{\%r11} \end{array}$$

Sometimes, this is unnecessary because some operations can be carried out directly with memory references. So the assembly code for the above could be shorter

```
ADDL %eax, 8(%rsp)
```

although it is not clear whether and how much more efficient this might be than a 3-instruction sequence

```
MOVL 8(%rsp), %r11
ADDL %eax, %r11
MOVL %r11, 8(%rsp)
```

We recommend generating the simplest uniform instruction sequences for spill code.

Extensions Heuristic factors that are used for register allocation especially for breaking ties in deciding which temps to spill into the memory include

- values that rematerialize easily, i.e., that can be recomputed easily (say with 1 or 2 instructions) from other registers or at least loaded from or recomputed easily from few memory accesses. When rematerializing from memory, the placement of the instruction needs to be scheduled appropriately for cache and pipeline efficiency reasons.
- values that (approximately) will not be used quickly again when following the (likely) control flow, counting loop bodies as "closer" than loop exits.
- values that interfere with many others.

Especially on SSA programs, deciding on register spilling can sometimes be more efficient before final register allocation, which can help the interplay with instruction selection. On SSA programs, register allocation can be done without explicitly constructing the interference graph (based on a postfix order of the dominance tree). The reason is that the central SSA relation called dominance tree defines a simplicial elimination order by doing a prefix traversal order of the dominance tree, such that register allocation is immediate. It, thus, makes sense to reconsider register allocation and interference graph construction for possible simplifications in case you later choose to implement SSA.

8 Register Coalescing

After register allocation, a common further optimization is used to eliminate register-to-register moves called *register coalescing*. Algorithms for register coalescing are usually tightly integrated with register allocation. In contrast, Pereira and Palsberg describe a relatively straightforward method that is performed entirely after graph coloring called *greedy coalescing*.

Greedy coalescing follows the principle

1. Consider each move between variables $t \leftarrow s$ occurring in the program in turn.
2. If t and s are the same color, the move can be eliminated without further action.
3. If there is an edge between them, that is, they interfere, they cannot be coalesced.
4. Otherwise, if there is a color c which is not used in the neighborhoods of t and s, i.e., $c \notin N(t) \cup N(s)$, and which is smaller than the number of available registers, then the variables t and s are coalesced into a single new variable u with color c. Then create edges from u to any vertex in $N(t) \cup N(s)$ and remove t and s from the graph.

Because of the tested condition, the resulting graph is still K-colored, where K is the number of available registers. Of course, we also need to eventually rewrite the program appropriately to maintain a correspondence with the graph.

This simple greedy coalescing will eliminate the redundant self move in the example above. Optimal register coalescing can be done using a reduction to integer linear programming, which can be too slow.

9 Precolored Nodes

Some instructions on the x86-64, such as integer division `IDIV`, require their arguments to be passed in specific registers and return their results also in specific registers. There are also `call` and `ret` instructions that use specific registers and must respect caller-save and callee-save register conventions. We will return to the issue of calling conventions later in the course. When generating code for a straight-line program as in the first lab, some care must be taken to save and restore callee-save registers in case they are needed.

First, for code generation, the live range of the fixed registers should be limited to avoid possible correctness issues and simplify register allocation.

Second, for register allocation, we can construct an elimination ordering as if all precolored nodes were listed first. This amounts to the initial weights of the

ordinary vertices being set to the number of neighbors that are precolored before the maximum cardinality search algorithm starts. The resulting list may or may not be a simplicial elimination ordering, but we can nevertheless proceed with greedy coloring as before.

10 Summary

Register allocation is an important phase in a compiler. It uses liveness information on variables to map unboundedly many variables to a finite number of registers, spilling temporaries onto stack slots if necessary. The algorithm described here is due to Hack [Hac07] and Pereira and Palsberg [PP05]. It is simpler than the one in the textbook and appears to perform comparably. It proceeds through the following passes:

1. **Build** the interference graph from the liveness information.
2. **Order** the nodes using maximum cardinality search.
3. **Color** the graph greedily according to the elimination ordering.
4. **Spill** if more colors are needed than registers available.

5^* **Coalesce** non-interfering move-related nodes greedily.

The last step, coalescing, is an optimization which is not required to generate correct code. Variants such as a separate spilling pass before coloring are described in the references above can further improve the efficiency of the generated code.

On chordal graphs, which come from SSA programs and often arise directly, register allocations is polynomial and efficient in practice. Optimal register coalescing and optimal spilling, however, are still NP-complete. Even when using heuristics, register allocation may consume the most time during a compiler run.

Questions

1. Why does register allocation take such a long time? It is polynomial isn't it?
2. Is it safe to restrict the interference graph definition for the instruction $t \leftarrow s_1 \oplus s_2$ to the case where t is live after that line?
3. What is the advantage of working with the intuition "overlapping live ranges" compared to the construction given in Section 2?
4. Does it make a difference where we start our register allocation, i.e., where we start the construction of a simplicial order?

5. Is register allocation for programs with mixed data types more difficult than for programs with uniform types? Why or why not?

6. Why is chordality of a graph interesting for register allocation?

7. Why should one worry about allocating half registers of lower data width? Isn't accessing words out of double words etc. inefficient? Is accessing bytes out of words inefficient?

8. Will register coalescing work better on 2-address or 3-address instruction forms?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the Symposium on Compiler Construction*, pages 98–105, Boston, Massachusetts, June 1982. ACM Press.

[Hac07] Sebastian Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In K.Yi, editor, *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 315–329, Tsukuba, Japan, November 2005. Springer LNCS 3780.

Lecture Notes on Liveness Analysis

15-411: Compiler Design
Frank Pfenning, André Platzer

Lecture 4
September 5, 2013

1 Introduction

We will see different kinds of program analyses in the course, most of them for the purpose of program optimization. The first one, *liveness analysis*, is required for register allocation. A variable is *live* at a given program point if it will be used during the remainder of the computation, starting at this point. We use this information to decide if two variables could safely be mapped to the same register, as detailed in the last lecture.

Is liveness decidable? Like many other properties of programs, liveness is undecidable if the language we are analyzing is Turing-complete. The approximation we describe here is standard, although its presentation is not. Chapter 10 of the textbook [App98] has a classical presentation.

2 Liveness by Backward Propagation

Consider a 3-address instruction applying a binary operator $\oplus$:

$$x \quad \leftarrow \quad y \oplus z$$

There are two reasons a variable may be live at this instruction, by which we mean *live just before the instruction is executed*. The first is immediate: if a variable (here: y and z) is used at an instruction, it is used in the computation starting from here. The second is slightly more subtle: since we execute the following instruction next, anything we determine is live at the next instruction is also live here. There is one exception to this second rule: because we assign to x, the value of x coming into this instruction does not matter (unless it is y or z), even if it is live at the next instruction. In summary,

1. y and z are live at an instruction $x \leftarrow y \oplus z$.

2. u is live at $x \leftarrow y \oplus z$ if u is live at the next instruction and $u \neq x$.

Similarly, for an instruction $x \leftarrow c$ with a constant c, we find that u is live at this instruction if u is live at the next instruction and $u \neq x$.

As a last example, x is live at a return instruction $\mathsf{return}\ x$, and nothing else is live there.

If we have a straight-line program, it is easy to compute liveness information by going through the program backwards, starting from the return instruction at the end. In that case, it is also precise rather than an approximation. As an example, one can construct the set of live variables at each line in this simple program bottom-up, using the two rules above.

$\uparrow$

Instructions			Live-in Variables
x_1	$\leftarrow$	1	$\cdot$
x_2	$\leftarrow$	$x_1 + x_1$	x_1
x_3	$\leftarrow$	$x_2 + x_1$	x_1, x_2
y_2	$\leftarrow$	$x_1 + x_2$	x_1, x_2, x_3
y_3	$\leftarrow$	$y_2 + x_3$	y_2, x_3
return		y_3	y_3

For example, looking at the 4th line, we see that x_1 and x_2 are live because of the first rule (they are used) and x_3 is live because it is live at the next instructions and different from y_2.

3 Liveness Analysis in Logical Form

Before we generalize to a more complex language of instructions, we try to specify the rules for liveness analysis in a symbolic form to make them more concise and to avoid any potential ambiguity. For this we give each instruction in a program a line number or *label*. If an instruction has label l, we write $l + 1$ for the label of the next instruction.

We also introduce the predicate $\mathsf{live}(l, x)$ which should be true when variable x is live at line l. We then turn the rules stated informally in English into logical rules.

$$\frac{l : x \leftarrow y \oplus z}{\begin{array}{l}\mathsf{live}(l, y)\\ \mathsf{live}(l, z)\end{array}}\ L_1 \qquad \frac{\begin{array}{l}l : x \leftarrow y \oplus z\\ \mathsf{live}(l+1, u)\\ x \neq u\end{array}}{\mathsf{live}(l, u)}\ L_2$$

Here, the formulas above the line are premises of the inference rule and the formulas below the line are the conclusions. If all premises are true, we know all conclusions must be true. To the right of the line we write the name of the inference rule. For example, we can read rule L_1 as: "*If line l has the form $x \leftarrow y \oplus z$ then y is live at l and z is live at l.*"

This is somewhat more abstract than the backward propagation algorithm because it does not specify in which order to apply these rules. We can now add more rules for different kinds of instructions.

$$\frac{l : \mathsf{return}\ x}{\mathsf{live}(l, x)}\ L_3 \qquad \frac{\begin{array}{l} l : x \leftarrow c \\ \mathsf{live}(l+1, u) \\ x \neq u \end{array}}{\mathsf{live}(l, u)}\ L_4$$

If we only have binary operators, moves of constants into variables, and return instructions, then these four rules constitute a complete specification of when a variable should be live at any point in a program.[1]

This specification also gives rise to an immediate, yet somewhat nondeterministic implementation. We start with a database of facts, consisting only of the original program, with each line properly labeled. Then we apply rules in an arbitrary order — whenever the premises are all in the database we add the conclusion to the database. Applying one rule may enable the application of another rule and so on, but eventually this process will not gain us any more information. At this point, we can still apply rules but all conclusions are already in the database of facts. We say that the database is *saturated*. Since the rules are a complete specification of our liveness analysis, by definition a variable x is deemed lived at line l if and only if the fact $\mathsf{live}(l, x)$ is in the saturated database.

This may seem like an unreasonable expensive way to compute liveness, but in fact it can be quite efficient, both in theory and practice.

In theory, we can look at the rules and determine their theoretical complexity by (a) counting so-called *prefix firings* of each rule, and (b) bounding the size of the completed database. We will return to prefix firings, a notion due to McAllester [McA02], in a later lecture. Bounding the size of the completed database is easy. We can infer at most $L \cdot V$ distinct facts of the form $\mathsf{live}(l, x)$, where L is the number of lines and V is the number of variables in the program. Counting prefix firings does not change anything here, and we get a theoretical complexity of $O(L \cdot V)$ for the analysis so far.

In practice, there are a number of ways logical rules and saturation can be implementation efficiently. One uses Binary Decision Diagrams (BDD's). Whaley, Avots, Carbin, and Lam [WACL05] have shown scalability of global program

[1] As pointed out in lecture, we should really also have a pure move instruction $x \leftarrow y$. We leave it to the reader to write out the additional rules.

analyses using inference rules, transliterated into so-called Datalog programs. See Smaragdakis and Bravenboer's work on Doop [SB10] for a different technique. Unfortunately, there is no Datalog library that we can easily tie into our compilers, so while we specify and analyze the structure of our program analyses via the use of inference rules, we generally do not implement them in this manner. Instead, we use other implementations that follow the ideas that are identified precisely and concisely by the logical rules. Because our logical rules identify the fundamental principles, this presentation makes it easier to understand what the important things of liveness analysis are. This also helps capturing the implementation-independent commonality among different styles of implementation. We will see throughout this whole course, that logical rules can capture many other important concepts in a similarly simple way.

4 Loops and Conditionals

The nature of liveness analysis changes significantly when the language permits loops. This will also be the case for most other program analyses.

Here, we add two new forms of instructions, and unconditional jump l : goto l', and a conditional branch l : if $(x \ ? \ c)$ goto l', where "?" is a relational operator such as equality or inequality.

We now discuss how liveness analysis should be extended for these two forms of instructions. A variable u is live at l : goto l' if it is live at l'. We capture this with the following inference rule, which is the only rule pertaining to goto

$$\frac{\begin{array}{l} l : \mathsf{goto}\ l' \\ \mathsf{live}(l', u) \end{array}}{\mathsf{live}(l, u)}\ L_5$$

When executing a conditional branch l : if $(x \ ? \ c)$ goto l' we have two potential successor instructions: we may go to the next $l + 1$ if the condition is false or to l' if the condition is true. In general, we will not be able to predict at compile time whether the condition will be true or false and usually it will sometimes be true and sometimes be false during the execution of the program. Therefore we have to consider a variable live at l if it is live at the possible successor $l + 1$ or it is live at the possible successor l'. Also, the instruction uses x, so x is live. Summarizing this as rules we obtain

$$\frac{l : \mathsf{if}\ (x \ ? \ c)\ \mathsf{goto}\ l'}{\mathsf{live}(l, x)}\ L_6 \qquad \frac{\begin{array}{l} l : \mathsf{if}\ (x \ ? \ c)\ \mathsf{goto}\ l' \\ \mathsf{live}(l + 1, u) \end{array}}{\mathsf{live}(l, u)}\ L_7 \qquad \frac{\begin{array}{l} l : \mathsf{if}\ (x \ ? \ c)\ \mathsf{goto}\ l' \\ \mathsf{live}(l', u) \end{array}}{\mathsf{live}(l, u)}\ L_8$$

These rules are straightforward enough, but if we have backwards branches we will not be able to analyze in a single backwards pass. As an example to illus-

trate this point, we will use a simple program for calculating the greatest common divisor of two positive integers. We assume that at the first statement labeled 1, variables x_1 and x_2 hold the input, and we are supposed to calculate and return $\gcd(x_1, x_2)$.

		Instructions	Live variables, initially
1	:	if $(x_2 = 0)$ goto 8	
2	:	$q \leftarrow x_1/x_2$	
3	:	$t \leftarrow q * x_2$	
4	:	$r \leftarrow x_1 - t$	
5	:	$x_1 \leftarrow x_2$	
6	:	$x_2 \leftarrow r$	
7	:	goto 1	
8	:	return x_1	

If we start at line 8 we see x_1 is live there, but we can conclude nothing (yet) to be live at line 7 because nothing is known to be live at line 1, the target of the jump. After one pass through the program, listing all variables we know to be live so far we arrive at:

		Instructions	Live variables, after pass 1
1	:	if $(x_2 = 0)$ goto 8	x_1, x_2
2	:	$q \leftarrow x_1/x_2$	x_1, x_2
3	:	$t \leftarrow q * x_2$	x_1, x_2, q
4	:	$r \leftarrow x_1 - t$	x_1, x_2, t
5	:	$x_1 \leftarrow x_2$	x_2, r
6	:	$x_2 \leftarrow r$	r
7	:	goto 1	$\cdot$
8	:	return x_1	x_1

At this point, we can apply the rule for goto to line 7, once with variable x_1 and once with x_2, both of which are now known to be live at line 1. We list the variables that are now further to the right, and make another pass through the program, applying more rules.

		Instructions	Live-in variables, after pass 1	after pass 2	saturate
1	:	if $(x_2 = 0)$ goto 8	x_1, x_2		
2	:	$q \leftarrow x_1/x_2$	x_1, x_2		
3	:	$t \leftarrow q * x_2$	x_1, x_2, q		
4	:	$r \leftarrow x_1 - t$	x_1, x_2, t		
5	:	$x_1 \leftarrow x_2$	x_2, r		
6	:	$x_2 \leftarrow r$	r	x_1	
7	:	goto 1	$\cdot$	x_1, x_2 (from 1)	
8	:	return x_1	x_1		

At this point our rules have saturated and we have identified all the live variables at all program points. From this we can now build the interference graph and from that proceed with register allocation.

The algorithm which saturates the inference rules implies that a variable is designated live at a given line only if we have definitive reason to believe it might be live. Consider the program

```
1 : u1 ← 1
2 : y ← y * x
3 : z ← y + y          (z not used, redundant)
4 : x ← x − u1
5 : if (x > 0) goto 2
6 : return y
```

which has a redundant assignment to z in line 3. Since z is never used, z is not found to be live anywhere in this program. Nevertheless, unless we eliminate line 3 altogether, we have to be careful to note that z interferes with x, u_1, and y because those variables are live on line 4. If not, z might be assigned the same register as x, y, or u_1 and the assignment to z would overwrite one of their values.

In the slightly different program

```
1 : u1 ← 1
2 : y ← y * x
3 : z ← z + z          (z live but never needed)
4 : x ← x − u1
5 : if (x > 0) goto 2
6 : return y
```

the variable z will actually be inferred to be live at lines 1 through 5. This is because it is used at line 3, although the resulting value is eventually ignored. To capture redundancy of this kind is the goal of *dead code elimination* which requires *neededness analysis* rather than liveness analysis. We will present this in a later lecture.

5 Refactoring Liveness

Figure 1 has a summary of the rules specifying liveness analysis.

$$\frac{l : x \leftarrow y \oplus z}{\begin{array}{l}\mathsf{live}(l, y)\\ \mathsf{live}(l, z)\end{array}}\ L_1 \qquad \frac{\begin{array}{l}l : x \leftarrow y \oplus z\\ \mathsf{live}(l+1, u)\\ x \neq u\end{array}}{\mathsf{live}(l, u)}\ L_2$$

$$\frac{l : \mathsf{return}\ x}{\mathsf{live}(l, x)}\ L_3 \qquad \frac{\begin{array}{l}l : x \leftarrow c\\ \mathsf{live}(l+1, u)\\ x \neq u\end{array}}{\mathsf{live}(l, u)}\ L_4$$

$$\frac{\begin{array}{l}l : \mathsf{goto}\ l'\\ \mathsf{live}(l', u)\end{array}}{\mathsf{live}(l, u)}\ L_5$$

$$\frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'}{\mathsf{live}(l, x)}\ L_6 \qquad \frac{\begin{array}{l}l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'\\ \mathsf{live}(l+1, u)\end{array}}{\mathsf{live}(l, u)}\ L_7 \qquad \frac{\begin{array}{l}l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'\\ \mathsf{live}(l', u)\end{array}}{\mathsf{live}(l, u)}\ L_8$$

Figure 1: Summary: Rules specifying liveness analysis (non-refactored)

This style of specification is precise and implementable, but it is rather repetitive. For example, L_2 and L_4 are similar rules, propagating liveness information from $l+1$ to l, and L_1, L_3 and L_6 are similar rules recording the usage of a variable. If we had specified liveness procedurally, we would try to abstract common patterns by creating new auxiliary procedures. But what is the analogue of this kind of restructuring when we look at specifications via inference rules? The idea is to identify common concepts and distill them into new predicates, thereby abstracting away from the individual forms of instructions.

Here, we arrive at three new predicates.

1. $\mathsf{use}(l, x)$: the instruction at l *uses* variable x.
2. $\mathsf{def}(l, x)$: the instruction at l *defines* (that is, writes to) variable x.
3. $\mathsf{succ}(l, l')$: the instruction executed after l *may be* l'.

Now we split the set of rules into two. The first set analyzes the program and generates the use, def and succ facts. We run this first set of rules to saturation.

Afterwards, the second set of rules employs these predicates to derive facts about liveness. It does not refer to the program instructions directly—we have abstracted away from them.

We write the second program first. It translates the following two, informally stated rules into logical language:

1. If a variable is used at l it is live at l.
2. If a variable is live at a possible next instruction and it is not defined at the current instruction, then it is live at the current instruction.

$$\frac{\mathsf{use}(l, x)}{\mathsf{live}(l, x)}\ K_1 \qquad \frac{\begin{array}{l}\mathsf{live}(l', u)\\ \mathsf{succ}(l, l')\\ \neg\mathsf{def}(l, u)\end{array}}{\mathsf{live}(l, u)}\ K_2$$

Here, we use $\neg$ to stand for negation, which is an operator that deserves more attention when using saturation via logic rules. For this to be well-defined we need to know that def does not depend on live. Any implementation must first saturate the facts about def before applying any rules concerning liveness, because the absence of a fact of the form $\mathsf{def}(l, -)$ does not imply that such a fact might not be discovered in a future inference—unless we first saturate the def predicate. Here, we can easily first apply all rules that could possibly conclude facts of the form $\mathsf{def}(l, u)$ exhaustively until saturation. If, after saturation with those rules ($J_1 \ldots J_5$ below), $\mathsf{def}(l, u)$ has not been concluded, then we know $\neg\mathsf{def}(l, u)$, because we have exhaustively applied all rules that could ever conclude it. Thus, after having saturated all rules for $\mathsf{def}(l, u)$, we can saturate all rules for $\mathsf{live}(l, u)$. This simple saturation in stages would break down if there were a rule concluding $\mathsf{def}(l, u)$ that depends on a premise of the form $\mathsf{live}(l', v)$, which is not the case.

We return to the first set of rules. It must examine each instruction and extract the use, def, and succ predicates. We could write several subsets of rules: one subset to generate def, one to generate use, etc. Instead, we have just one rule for each instruction with multiple conclusions for all required predicates.

$$\frac{l : x \leftarrow y \oplus z}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{use}(l, y)\\ \mathsf{use}(l, z)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_1 \qquad \frac{l : \mathsf{return}\ x}{\mathsf{use}(l, x)}\ J_2 \qquad \frac{l : x \leftarrow c}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_3$$

$$\frac{l : \mathsf{goto}\ l'}{\mathsf{succ}(l, l')}\ J_4 \qquad \frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'}{\begin{array}{l}\mathsf{use}(l, x)\\ \mathsf{succ}(l, l')\\ \mathsf{succ}(l, l+1)\end{array}}\ J_5$$

It is easy to see that even with any number of new instructions, this specification can be extended modularly. The main definition of liveness analysis in rules K_1 and K_2 will remain unchanged and captures the essence of liveness analysis.

The theoretical complexity does not change, because the size of the database after each phase is still $O(L \cdot V)$. The only point to observe is that even though the successor relation looks to be bounded by $O(L \cdot L)$, there can be at most two successors to any line l so it is only $O(L)$.

6 Control Flow

Properties of the control flow of a program are embodied in the succ relation introduced in the previous section. The *control flow graph* is the graph whose vertices are the lines of the program and with an edge between l and l' whenever $\mathsf{succ}(l, l')$. It captures the possible flows of control without regard to the actual values that are passed.

The textbook [App98] recommends an explicit representation of the control flow graph, together with the use of *basic blocks* to speed up analysis. A *basic block* is a simple fragment of straight-line code that is always entered at the beginning and exited at the end. That is

- the first statement may have a label,
- the last statement terminates the control flow of the current block (with a goto, conditional branch, or a return), and
- all other statements in between have no labels (entry points) and no gotos or conditional branches (exit points).

From a logical perspective, basic blocks do not change anything, because they just accumulate a series of simple statements into one compound code block. Hence, it is not clear if a logical approach to liveness and other program analyses would actually benefit from basic block representations. But depending on the actual implementation technique, basic blocks can help surprisingly much, because the number of nodes that need to be considered in each analysis is reduced somewhat. Basic blocks basically remove trivial control flow edges and assimilate them into a single basic block, exposing only more nontrivial control flow edges. Basic blocks are an example of an engineering decision that looks like a no-op, but can still pay off. They are also quite useful for SSA intermediate language representations and LLVM code generation.

Control flow information can be made more precise if we analyze the possible values that variables may take. Since control flow critically influences other analyses in a similar way to liveness analysis, it is almost universally important. Our

current analysis is not sensitive to the actual values of variables. Even if we write

$$
\begin{array}{lcll}
l & : & x \leftarrow 0 & \\
l+1 & : & \mathsf{if}\ (x < 0)\ \mathsf{goto}\ l+3 & \\
l+2 & : & \mathsf{return}\ y & \\
l+3 & : & \mathsf{return}\ z & \text{(unreachable in this program due to values)}
\end{array}
$$

we deduce that both y and z may be live at $l + 1$ even though only return y can actually be reached. This and similar patterns may seem unlikely, but in fact they arise in practice in at least two ways: as a result of other optimizations and during array bounds checking. We may address this issue in a later lecture.

7 Summary

Liveness analysis is a necessary component of register allocation. It can be specified in two logical rules which depend on the control flow graph, $\mathsf{succ}(l, l')$, as well as information about the variables used, $\mathsf{use}(l, x)$, and defined, $\mathsf{def}(l, x)$, at each program point. These rules can be run to saturation in an arbitrary order to discover all live variables. On straight-line programs, liveness analysis can be implemented in a single backwards pass, on programs with jumps and conditional branches some iteration is required until no further facts about liveness remain to be discovered. Liveness analysis is an example of a *backward dataflow analysis*; we will see more analyses with similar styles of specifications throughout the course.

Questions

1. Can liveness analysis be faster if we execute it out of order, i.e., not strictly backwards?
2. Is there a program where liveness analysis gives imperfect information?
3. Is there a class of programs where this does not happen? What is the biggest such class?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[SB10] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, pages 245–251, Oxford, UK, March 2010. Springer LNCS 6702. Revised selected papers.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In K.Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 97–118. Springer LNCS 3780, November 2005.

Lecture Notes on Dataflow Analysis

15-411: Compiler Design
Frank Pfenning, André Platzer

Lecture 5
September 10, 2013

1 Introduction

In this lecture we first extend liveness analysis to handle memory references and then consider *neededness analysis* which is similar to liveness and used to discover *dead code*. Both liveness and neededness are backwards dataflow analyses. We then describe *reaching definitions*, a forwards dataflow analysis which is an important component of optimizations such as constant propagation or copy propagation.

2 Memory References

Recall the rules specifying liveness analysis from the previous lecture.

$$\frac{\mathsf{use}(l, x)}{\mathsf{live}(l, x)}\ K_1 \qquad \frac{\begin{array}{l}\mathsf{live}(l', u)\\ \mathsf{succ}(l, l')\\ \neg\mathsf{def}(l, u)\end{array}}{\mathsf{live}(l, u)}\ K_2$$

We do not repeat the rules for extracting def, use, and succ from the program. They represent the following:

- $\mathsf{use}(l, x)$: the instruction at l *uses* variable x.
- $\mathsf{def}(l, x)$: the instruction at l *defines* (that is, writes to) variable x.
- $\mathsf{succ}(l, l')$: the instruction executed after l may be l'.

In order to model the store in our abstract assembly language, we add two new forms of instructions

- Load: $y \leftarrow M[x]$.
- Store: $M[x] \leftarrow y$.

All that is needed to extend the liveness analysis is to specify the def, use, and succ properties of these two instructions.

$$\frac{l : x \leftarrow M[y]}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{use}(l, y)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_6 \qquad \frac{l : M[y] \leftarrow x}{\begin{array}{l}\mathsf{use}(l, x)\\ \mathsf{use}(l, y)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_7$$

The rule J_7 for storing register contents to memory does not define any value, because liveness analysis does not track memory, only variables which then turn into registers. Tracking memory is indeed a difficult task and subject of a number of analyses of which *alias analysis* is the most prominent. We will consider this in a later language.

The two rules for liveness itself do not need to change! This is an indication that we refactored the original specification in a good way.

3 Dead Code Elimination

An important optimization in a compiler is *dead code elimination* which removes unneeded instructions from the program. Even if the original source code does not contain unnecessary code, after translation to a low-level language dead code often arises either just as an artefact of the translation itself or as the result of optimizations. We will see an example of these phenomena in Section 5; here we just use a small example.

In this code, we compute the factorial $x!$ of x. The variable x is live at the first line. This would typically be the case of an input variable to a program.

		Instructions	Live variables	
1	:	$p \leftarrow 1$	x	
2	:	$p \leftarrow p * x$	p, x	
3	:	$z \leftarrow p + 1$	p, x	(z not live $\Rightarrow$ dead code?)
4	:	$x \leftarrow x - 1$	p, x	
5	:	if $(x > 0)$ goto 2	p, x	
6	:	return p	p	

The only unusual part of the loop is the unnecessary computation of $p + 1$.

We may suspect that line 3 is dead code, and we should be able to eliminate it, say, by replacing it with some nop instruction which has no effect, or perhaps eliminate it entirely when we finally emit the code. The reason to suspect that line

3 is dead code is that z is not live at the point where we define it. While this may be sufficient reason to eliminate the assignment here, this is not true in general. For example, we may have an assignment such as $z \leftarrow p/x$ which is required to raise an exception if $x = 0$, or if an overflow occurs, because the result is too large to fit into the allotted bits on the target architecture (division by -1). Another example is a memory reference such as $z \leftarrow M[x]$ which is required to raise an exception if the address x has actually not been allocated or is not readable by the executing process. We will come back to these exceptions in the next section. First, we discuss another phenomenon exhibited in the following small modification of the program above.

		Instructions	Live variables	
1	:	$p \leftarrow 1$	x, z	
2	:	$p \leftarrow p * x$	p, x, z	
3	:	$z \leftarrow z + 1$	p, x, z	(live but not needed)
4	:	$x \leftarrow x - 1$	p, x, z	
5	:	if $(x > 0)$ goto 2	p, x, z	
6	:	return p	p	

Here we see that z is live in the loop (and before it) even though the value of z does not influence the final value returned. To see this yourself, note that in the first backwards pass we find z to be used at line 3. After computing p, x, and z to be live at line 2, we have to reconsider line 5, since 2 is one of its successors, and add z as live to lines 5, 4, and 3.

This example shows that liveness is not precise enough to eliminate even simple redundant instructions such as the one in line 3 above.

4 Neededness

In order to recognize that assignments as in the previous example program are indeed redundant, we need a different property we call *neededness*. We will structure the specification in the same way as we did for liveness: we analyze each instruction and extract the properties that are necessary for neededness to proceed without further reference to the program instructions themselves.

The crucial first idea is that the some variables are needed because an instruction they are involved in may have an *effect*. Let's call such variable *necessary*. Formally, we write $\mathsf{nec}(l, x)$ to say that x is necessary at instruction l. We use the notation $\oslash$ for a binary operator which may raise an exception, such as division or the modulo operator. For our set of instructions considered so far, the following

are places where variables are necessary because of the possiblity of effects.

$$\frac{l : x \leftarrow y \oslash z}{\begin{array}{l}\mathsf{nec}(l, y)\\ \mathsf{nec}(l, z)\end{array}}\ E_1 \qquad \frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'}{\mathsf{nec}(l, x)}\ E_2$$

$$\frac{l : \mathsf{return}\ x}{\mathsf{nec}(l, x)}\ E_3 \qquad \frac{l : y \leftarrow M[x]}{\mathsf{nec}(l, x)}\ E_4 \qquad \frac{l : M[x] \leftarrow y}{\begin{array}{l}\mathsf{nec}(l, x)\\ \mathsf{nec}(l, y)\end{array}}\ E_5$$

Here, x is flagged as necessary at a return statement because that is the final value returned, and a conditional branch because it is necessary to test the condition. The effect here is either the jump, or the lack of a jump.

A side remark: on many architectures including the x86 and x86-64, apparently innocuous instructions such as $x \leftarrow x + y$ have an effect because they set the condition code registers. This makes optimizing unstructured machine code quite difficult. However, in compiler design we have a secret weapon: we only have to optimize the code that we generate! For example, if we make sure that when we compile conditionals, the condition codes are set immediately before the branching instruction examines them, then the implicit effects of other instructions that are part of code generation are benign and can be ignored. However, such "benign effects" may be lurking in unexpected places and may perhaps not be so benign after all, so it is important to reconsider them especially as optimizations become more aggressive. Possible downsides of such convention choices can partially be optimized away in the post optimization phase that we will discuss later.

Now that we have extracted when variables are immediately *necessary* at any given line, we have to exploit this information to compute neededness. We write $\mathsf{needed}(l, x)$ if x is needed at l. The first rule captures the motivation for designing the rules for necessary variables.

$$\frac{\mathsf{nec}(l, x)}{\mathsf{needed}(l, x)}\ N_1$$

This seeds the neededness relation and we need to consider how to propagate it. Our second rule is an exact analogue of the way we propagate liveness.

$$\frac{\begin{array}{l}\mathsf{needed}(l', u)\\ \mathsf{succ}(l, l')\\ \neg\mathsf{def}(l, u)\end{array}}{\mathsf{needed}(l, u)}\ N_2$$

The crucial rule is the last one. In an assignment $x \leftarrow y \oplus z$ the variables y and z are needed if x is needed in the remaining computation. If x cannot be shown to

be needed, then y and z are not needed if $\oplus$ is an effect free operation. Abstracting away from the particular instruction, we get the following:

$$\frac{\begin{array}{l}\mathsf{use}(l, y)\\ \mathsf{def}(l, x)\\ \mathsf{succ}(l, l')\\ \mathsf{needed}(l', x)\end{array}}{\mathsf{needed}(l, y)}\ N_3$$

We see that neededness analysis is slightly more complex than liveness analysis: it requires three rules instead of two, and we need the new concept of a variable necessary for an instruction due to effects. We can restructure the program slightly and could unify the formulas $\mathsf{nec}(l, x)$ and $\mathsf{needed}(l, x)$. This is mostly a matter of taste and modularity. Personally, I prefer to separate local properties of instructions from those that are propagated during the analysis, because local properties are more easily re-used. The specification of neededness is actually an example of that: it re-uses $\mathsf{use}(l, x)$ in rule N_3 which we first introduced for liveness analysis. If we had structured liveness analysis so that the rules for instructions generate $\mathsf{live}(l, x)$ directly, it would not have worked as well here.

We can now perform neededness analysis on our example program. We have indexed each variable with the numbers of all rules that can be used to infer that they are needed (N_1, N_2, or N_3).

		Instructions	Needed variables
1	:	$p \leftarrow 1$	x^2
2	:	$p \leftarrow p * x$	$p^3, x^{2,3}$
3	:	$z \leftarrow z + 1$	p^2, x^2
4	:	$x \leftarrow x - 1$	p^2, x^3
5	:	$\mathsf{if}\ (x > 0)\ \mathsf{goto}\ 2$	$p^2, x^{1,2}$
6	:	$\mathsf{return}\ p$	p^1

At the crucial line 3, z is defined but not needed on line 4, and consequently it is not needed at line 3 either.

Since the right-hand side of $z \leftarrow z + 1$ does not have an effect, and z is not needed at any successor line, this statement is dead code and can be optimized away.

5 Optimization Example

The natural direction for both liveness analysis and neededness analysis is to traverse the program backwards. In this section we present another important analysis whose natural traversal directions is *forward*. As motivating example for this kind of analysis we use an array access with bounds checks.

In our source language C0 we will have an assignment $x = A[0]$ where A is an array. We also assume there are (assembly language) variables n with the number of elements in array A, variable s with the size of the array elements, and a with the base address of the array. We might then translate the assignment to the following code:

$$\begin{array}{rcl} 1 & : & i \leftarrow 0 \\ 2 & : & \mathsf{if}\ (i < 0)\ \mathsf{goto\ error} \\ 3 & : & \mathsf{if}\ (i \geq n)\ \mathsf{goto\ error} \\ 4 & : & t \leftarrow i * s \\ 5 & : & u \leftarrow a + t \\ 6 & : & x \leftarrow M[u] \\ 7 & : & \mathsf{return}\ x \end{array}$$

The last line is just to create a live variable x. We notice that line 2 is redundant because the test will always be false. Computationally, we can figure this out in two steps. First we apply *constant propagation* to replace $(i < 0)$ by $(0 < 0)$ and then apply *constant folding* to evaluate the comparison to 0 (representing falsehood). Line 3 is necessary unless we know that $n > 0$. Line 4 performs a redundant multiplication: because i is 0 we know t must also be 0. This is an example of an arithmetic optimization similar to constant folding. And now line 5 is a redundant addition of 0 and can be turned into a move $u \leftarrow a$, again a simplification of modular arithmetic.

At this point the program has become

$$\begin{array}{rcl} 1 & : & i \leftarrow 0 \\ 2 & : & \mathsf{nop} \\ 3 & : & \mathsf{if}\ (i \geq n)\ \mathsf{goto\ error} \\ 4 & : & t \leftarrow 0 \\ 5 & : & u \leftarrow a \\ 6 & : & x \leftarrow M[u] \\ 7 & : & \mathsf{return}\ x \end{array}$$

Now we notice that line 4 is dead code because t is not *needed*. We can also apply *copy propagation* to replace $M[u]$ by $M[a]$, which now makes u not needed so we can apply *dead code elimination* to line 4. Finally, we can again apply *constant propagation* to replace the *only* remaining occurrence of i in line 3 by 0 followed by *dead code elimination* for line 1 to obtain

$$\begin{array}{rcl} 1 & : & \mathsf{nop} \\ 2 & : & \mathsf{nop} \\ 3 & : & \mathsf{if}\ (0 \geq n)\ \mathsf{goto\ error} \\ 4 & : & \mathsf{nop} \\ 5 & : & \mathsf{nop} \\ 6 & : & x \leftarrow M[a] \\ 7 & : & \mathsf{return}\ x \end{array}$$

which can be quite a bit more efficient than the first piece of code. Of course, when emitting machine code we can delete the nop operations to reduce code size.

One important lesson from this example is that many different kinds of optimizations have to work in concert in order to produce efficient code in the end. What we are interested in for this lecture is what properties we need for the code to ensure that the optimization are indeed applicable.

We return to the very first optimization of constant propagation. We replaced the test $(i < 0)$ with $(0 < 0)$. This looks straightforward, but what happens if some other control flow path can reach the test? For example, we can insert an increment and a conditional to call this optimization into question.

1	:	$i \leftarrow 0$	1	:	$i \leftarrow 0$
2	:	if $(i < 0)$ goto error	2	:	if $(i < 0)$ goto error
3	:	if $(i \geq n)$ goto error	3	:	if $(i \geq n)$ goto error
4	:	$t \leftarrow i * s$	4	:	$t \leftarrow i * s$
5	:	$u \leftarrow a + t$	5	:	$u \leftarrow a + t$
6	:	$x \leftarrow M[u]$	6	:	$x \leftarrow M[u]$
7	:	return x	**7**	:	$i \leftarrow i + 1$
			8	:	if $(i < n)$ goto 2
			9	:	return x

Even though lines 1–6 have not changed, suddenly we can no longer replace $(i < 0)$ with $(0 < 0)$ because the second time line 2 is reached, i is 1. With arithmetic reasoning we may be able to recover the fact that line 2 is redundant, but pure constant propagation and constant folding is no longer sufficient.

What we need to know for copy propagation is that the definition of i in line 1 is the only definition of i that can reach line 2. This is true in the program on the left, but not on the right since the definition of i at line 7 can also reach line 2 if the condition at line 9 is true.

6 Reaching Definitions

We say a definition $l : x \leftarrow \ldots$ *reaches* a line l' if there is a path of control flow from l to l' during which x is not redefined. In logical language:

- reaches(l, x, l') if the definition of x at l reaches l' (especially x has not been redefined since).

We only need two inference rules to define this analysis. The first states that a variable definition reaches any immediate successor. The second expresses that we can propagate a reaching definition of x to all successors of a line l' we have already

reached, unless this line also defines x.

$$\frac{\mathsf{def}(l, x) \quad \mathsf{succ}(l, l')}{\mathsf{reaches}(l, x, l')}\ R_1 \qquad \frac{\mathsf{reaches}(l, x, l') \quad \mathsf{succ}(l', l'') \quad \neg\mathsf{def}(l', x)}{\mathsf{reaches}(l, x, l'')}\ R_2$$

Analyzing the original program on the left, we see that the definition of i at line 1 reaches lines 2–7, and this is (obviously) the only definition of i reching lines 2 and 4. We can therefore apply the optimizations sketched above.

In the program on the right hand side, the definition of i at line 7 also reaches lines 2–8 so neither optimization can be applied.

Inspection of rule R_2 confirms the intuition that reaching definitions are propagated *forward* along the control flow edges. Consequently, a good implementation strategy starts at the beginning of a program and computes reaching definitions in the forward direction. Of course, saturation in the presence of backward branches means that we may have to reconsider earlier lines, just as in the backwards analysis.

A word on complexity: we can bound the size of the saturated database for reaching definitions by L^2, where L is the number of lines in the program. This is because each line defines at most one variable (or, in realistic machine code, a small constant number). Counting prefix firings (which we have not yet discussed) does not change this estimate, and we obtain a complexity of $O(L^2)$. This is not quite as efficient as liveness or neededness analysis (which are $O(L \cdot V)$), so we may need to be somewhat circumspect in computing reaching definitions.

7 Summary

We have extended the ideas behind liveness analysis to neededness analysis which enables more aggressive dead code elimination. Neededness is another example of a program analysis proceeding naturally backward through the program, iterating through loops.

We have also seen reaching definitions, which is a forward dataflow analysis necessary for a number of important optimizations such as constant propagation or copy propagation. Reaching definitions can be specified in two rules and do not require any new primitive concepts beyond variable definitions ($\mathsf{def}(x, l)$) and the control flow graph ($\mathsf{succ}(l, l')$), both of which we already needed for liveness analysis.

Another important observation from the need for dataflow analysis information during optimization is that dataflow analysis may have to be rerun after an optimization transformed the program. Rerunning all analysis exhaustively all the time after each optimization may be time-consuming. Adapting the dataflow

analysis information during optimization transformations is sometimes possible as well, but correctness is less obvious. SSA alleviates this problem somewhat, because some (but not all) dataflow analysis informations are readily read off from SSA.

For an alternative approach to dataflow analysis via *dataflow equations*, see the textbook [App98], Chapters 10.1 and 17.1–3. Notes on implementation of dataflow analyses are in Chapter 10.1–2 and 17.4. Generally speaking, a simple iterative implementation with a library data structure for sets which traverses the program in the natural direction should be efficient enough for our purposes. We would advise against using bitvectors for sets. Not only are the sets relatively sparse, but bitvectors are more time-consuming to implement. An interesting alternative to iterating over the program, maintaining sets, is to do the analysis one variable at a time (see the remark on page 216 of the textbook). The implementation via a saturating engine for Datalog is also interesting, yet a bit more difficult to tie into the infrastructure of a complete compiler. The efficiency gain noted by Whaley et al. [WACL05] becomes only critical for interprocedural and whole program analyses rather than for the intraprocedural analyses we have presented so far.

Questions

1. Why does or liveness analysis not track memory? Should it?
2. Why is neededness different from liveness? Could we reuse part of one analysis for the other? Should we?
3. Why should it be a problem if a single dataflow analysis is slow? We only run it once, don't we?
4. How can the def/use/succ information be made accessible conveniently in a programming language? Does it improve the structure of the code if we do that?
5. Should our intermediate representation have an explicit representation of the control flow graph? What are the benefits and downsides?
6. Why should we care about dead code elimination? Nobody writes dead code down anyways, because that'd be a waste of time.
7. Where do the arithmetic optimizations alluded to in this lecture play a role in compiling? When are they important?
8. Suppose $x = y/z$ is computed but x never used later. That would make the statement not needed and dead code if it wasn't for the fact that the division could have side effects. So it is needed. But what would liveness analysis

do about it? How does this impact register allocation? What is the interplay with the special register requirements of integer division?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In K.Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 97–118. Springer LNCS 3780, November 2005.

Lecture Notes on
Static Single Assignment Form

15-411: Compiler Design
Frank Pfenning

Lecture 6
September 12, 2013

1 Introduction

In abstract machine code of the kind we have discussed so far, a variable of a given name can refer to different values even in straight-line code. For example, in a code fragment such as

$$\begin{array}{lcl} 1 & : & i \leftarrow 0 \\ \ldots & & \\ k & : & \mathsf{if}\ (i < 0)\ \mathsf{goto\ error} \end{array}$$

we can apply *constant propagation* of 0 to the condition only if we know that the definition of i in line 1 is the only one that reaches line k. It is possible that i is redefined either in the region from 1 to k, or somewhere in the rest of the program followed by a backwards jump. It was the purpose of the *reaching definitions* analysis in Lecture 5 to determine whether this is the case.

An alternative is to relabel variables in the code so that each variable is defined only once in the program text. If the program has this form, called *static single assignment* (SSA), then we can perform constant propagation immediately in the example above without further checks. There are other program analyses and optimizations for which it is convenient to have this property, so it has become a de facto standard intermediate form in many compilers and compiler tools such as the LLVM.

In this lecture we develop SSA, first for straight-line code and then for code containing loops and conditionals. Our approach to SSA is not entirely standard although the results are the same on control flow graphs that can arise from source programs in the language we compile.

2 Basic Blocks

A *basic block* is a sequence of instructions with one entry point and one exit point. In particular, from nowhere in the program do we jump into the middle of the basic block, nor do we exit the block from the middle. In our language, the last instruction in a basic block should therefore be a return, goto, or if, where we accept the pattern

$$\begin{array}{l}\mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l_1\\ \mathsf{goto}\ l_2\end{array}$$

at the end of a basic block. On the inside of a basic block we have what is called *straight-line code*, namely, a sequence of moves or binary operations.

It is easy to put basic blocks into SSA form. For each variable, we keep a *generation counter* to track which definition of a variable is currently in effect. We initialize this to 0 for any variable live at the beginning of a block. Then we traverse the block forward, replacing every use of a variable with its current generation. When we see a redefinition of variable we increment its generation and proceed.

As an example, we consider the following C0 program on the left and its translation on the right.

```
int dist(int x, int y) {
  x = x * x;
  y = y * y;
  return isqrt(x+y);
}
```

```
dist(x,y):
  x <- x * x
  y <- y * y
  t0 <- x + y
  t1 <- isqrt(t0)
  return t1
```

Here `isqrt` is an integer square root function previously defined. We have assumed a new form of instruction

$$d \leftarrow f(s_1, \ldots, s_n)$$

where each of the sources s_i is a constant or variable, and the destination d is another variable. We have also marked the beginning of the function with a parameterized label that tracks the variables that may be live in the body of the function.

The parameters x and y start at generation 0. They are *defined implicitly* because they obtain a value from the arguments to the call of dist.

```
dist(x0,y0):
  ------------ x/0, y/0
  x <- x * x
  y <- y * y
  t0 <- x + y
  t1 <- isqrt(t0)
  return t1
```

We mark where we are in the traversal with a line, and indicate there the current generation of each variable. The next line uses x, which becomes x_0, but is also defines x, which therefore becomes the next generation of x, namely x_1.

```
dist(x0,y0):
  x1 <- x0 * x0
  ------------- x/1, y/0
  y <- y * y
  t0 <- x + y
  t1 <- isqrt(t0)
  return t1
```

The next line is processed the same way.

```
dist(x0,y0):
  x1 <- x0 * x0
  y1 <- y0 * y0
  ------------- x/1, y/1
  t0 <- x + y
  t1 <- isqrt(t0)
  return t1
```

At the following line, t_0 is a new temp. The way we create instructions, temps are defined only once. We therefore do not have to create a new generation for them. If we did, it would of course not change the outcome of the conversion. Skipping ahead now, we finally obtain

```
dist(x0,y0):
  x1 <- x0 * x0
  y1 <- y0 * y0
  t0 <- x1 + y1
  t1 <- isqrt(t0)
  return t1
```

We see that, indeed, each variable is defined (assigned) only once, where the parameters x_0 and y_0 are implicitly defined when the function is called and the others explicity in the body of the function. It is easy to see that the original program and its SSA form will behave identically.

3 Loops

To appreciate the difficulty and solution of how to handle more complex programs, we consider the example of the exponential function, where $\mathsf{pow}(b, e) = b^e$ for $e \geq 0$.

```
int pow(int b, int e)
//@requires e >= 0;
{
  int r = 1;
  while (e > 0)
    //@loop_invariant e >= 0;
    // r*b^e remains invariant
    {
      r = r * b;
      e = e - 1;
    }
  return r;
}
```

We translate this to the following abstract machine code.

```
pow(b,e):
  r <- 1
loop:
  if (e <= 0) goto done
  r <- r * b
  e <- e - 1
  goto loop
done:
  return r
```

We can transform this into basic blocks, except that we take a small shortcut with the conditional branch by not following it with an explicit goto to save on space.

```
pow(b,e):
  r <- 1
  goto loop
loop:
  if (e <= 0) goto done
  r <- r * b
  e <- e - 1
  goto loop
done:
  return r
```

Now we note that there are two ways to reach the label loop: when we first enter the loop, or from the end of the loop body. This means the variable e in the conditional branch really could refer to either the procedure argument, or the value of e after the decrement operation in the loop body. Therefore, our straightforward idea for SSA conversion of straight line code no longer works.

The key idea is to parameterize labels (the jump targets) with the variables that are live in the block that follows.[1] Labels l occuring as targets in goto l or if $(-)$ goto l are then given matching arguments.

```
pow(b,e):
  r <- 1
  goto loop(b,e)

loop(b,e,r):
  if (e <= 0) goto done(r)
  r <- r * b
  e <- e - 1
  goto loop(b,e,r)

done(r):
  return r
```

Next, we convert each block into SSA form with the previous algorithm, but using a global generation counter throughout. An occurrence in a label in a jump goto $l(\ldots, x, \ldots)$ is seen as a *use* of x, while an occurrence of a variable in in a jump target $l(\ldots, x, \ldots)$ is seen as a *definition* of x. Applying this to the first block we obtain

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

-------------------- b/0, e/0, r/0

loop(b,e,r):
  if (e <= 0) goto done(r)
  r <- r * b
  e <- e - 1
  goto loop(b,e,r)

done(r):
  return r
```

Since we encounter a new definition of b, e, and r we advance all three generations and proceed with the next block.

[1] One can also safely, but redundantly approximate this by using *all* variables.

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 <= 0) goto done(r1)
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)

------------------- b/1, e/2, r/2

done(r):
  return r
```

Completing the conversion with the last block, we obtain:

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 <= 0) goto done(r1)
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)

done(r3):
  return r3
```

First, we verify that this code does indeed have the SSA property: each variable is assigned at most once, even counting implicit definitions at the parameterized labels pow(b_0, e_0), loop(b_1, e_1, r_1), and done(r_3). The operational reading of this program should be evident. For example, if we reach goto loop(b_0, e_0, r_0) we pass the current values of b_0, e_0 and r_0 and move them into variables b_1, e_1, and r_1. That fact that labeled jumps correspond to moving values from arguments to label parameters will be the essence of how to generate assembly code from the SSA intermediate form in Section 7.

4 SSA and Functional Programs

We can notice that at this point the program above can be easily interpreted as a *functional program* if we read assignments as bindings and labeled jumps as function calls. We show the functional program below on the right in ML-like form.

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 <= 0) goto done(r1)
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)

done(r3):
  return r3
```

```
fun pow(b0,e0) =
    let r0 = 1
     in loop(b0,e0,r0)

and loop(b1,e1,r1) =
    if e1 <= 0 then done(r1)
    else let r2 = r1 * b1
         and e2 = e1 - 1
          in loop(b1,e2,r2)

and done(r3) =
      r3
```

There are several reasons this works in general. First, in SSA form each variable is defined only once, which means it can be modeled by a let binding in a functional language. Second, each goto is at the end of a block, which translates into a tail call in the functional language. Third, because all jumps become tail calls, a return instruction can be modeled simply be returning the corresponding value.

We conclude that translation into SSA form is just translating abstract machine code to a functional program! Because our language does not have first-class functions, the target of this translation also does not have higher-order functions. Interestingly, this observation also works in reverse: a (first-order) functional program with tail calls can be translated into abstract machine code where tail calls become jumps.

While this is clearly an interesting observation, it does not directly help our compiler construction effort (although it might if we were interested in compiling a functional language).

5 Optimization and Minimal SSA Form

At this point we have constructed clean and simple abstract machine code with parameterized labels. But are all the parameters really necessary? Let's reconsider:

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 <= 0) goto done(r1)
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)
```

```
done(r3):
  return r3
```

For example, done(r_3) is only targeted from one location, with a goto done(r_1). There is no need to pass r_1 and assign its value to r_3. We can instead remove this argument from the label done and substitute r_1 for r_3. This yields:

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 <= 0) goto done()
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)

done():
  return r1
```

We see this is still in SSA form. Next we can ask if all the arguments to loop are really necessary. We have two gotos and one definition:

```
goto loop(b0,e0,r0)
goto loop(b1,e3,r2)

loop(b1,e1,r1):
```

Let's consider the first argument. In the first call it is b_0 and in the second b_1. Since we have SSA form, we know that the b_1 will always hold the same value. In fact, the only call with a different value is with b_0, so b_1 will in fact always have the value b_0. This means the first argument to loop is not needed and we can erase it, substituting b_0 for b_1. This yields:

```
pow(b0,e0):
  r0 <- 1
  goto loop(e0,r0)

loop(e1,r1):
  if (e1 <= 0) goto done()
  r2 <- r1 * b0
  e2 <- e1 - 1
  goto loop(e2,r2)

done():
  return r1
```

It is easy to check this is still in SSA form. The remaining arguments to loop are all different, however (e_0 and e_3 for e_1 and r_0 and r_2 for r_1), so we cannot optimize further.

This code is now in *minimal SSA form* in the sense that we cannot remove any label arguments by purely syntactic considerations.

The general case for this optimization is as follows: assume we have a parameterized label $l(\ldots, x_i, \ldots)$: where all gotos have the form goto $l(\ldots, x_i, \ldots)$ (for the same generation i) or goto $l(\ldots, x_k, \ldots)$ (all at the same generation k). Then the x argument to l is redundant, and x_i can be replaced by x_k everywhere in the program.

6 Extended Basic Blocks and Conditionals

As a special case of the rule at the end of the last section, we see that if a label is the target of exactly one jump, then this condition is automatically satisfied for all of its parameters. This was the case for the label 'done' in our example. In such a case, *all* parameters of this label can be removed.

We can then go a step further and not generate parameters to labels that are targeted only once. An *extended basic block* is a collection of basic blocks with one label at the beginning (that may be the target of multiple jumps) and internal labels, each of which is the target of only one internal jump and no external jumps.

When converting to SSA form, we can treat extended basic blocks as a single unit, since we do not have to create fresh parameterized labels within them. We have already applied this idea tacitly, because in our exponential function, strictly speaking, the loop should be decomposed into basic blocks as

```
loop:
  if (e <= 0) goto done
  goto body
body:
  r <- r * b
  e <- e - 1
  goto loop
```

However, the label 'body' is targeted only by one jump, so we contracted the two instructions. The optimization discussed above is a post-hoc justification for this.

Next we consider conditionals as a new language feature. We change our program to a "fast" power function which exploits the equations $b^{2e} = (b * b)^e$ and $b^{2e+1} = b * (b * b)^e$. The C0 program is on the left, its abstract assembly form on the right.

```
int fastpow(int b, int e)
//@requires e >= 0;
{
  int r = 1;
  while (e > 0)
    //@loop_invariant e >= 0;
    // r * b^e remains invariant
    {
      if (e % 2 != 0)
        r = r * b;
      b = b * b;
      e = e / 2;
    }
  return r;
}
```

```
fastpow(b,e):
  r <- 1
  goto loop
loop:
  if (e <= 0) goto done
  t0 <- e % 2
  if (t0 == 0) goto next
  r <- r * b
  goto next
next:
  b <- b * b
  e <- e / 2
  goto loop
done:
  return r
```

We see that the compiling the conditional creates another situation where we have one label (next) is the target of two jumps. This is often the case for conditionals, because the control flow graph has edges from each branch of the conditional to the statement following the conditional.

Next, we parameterize labels that are the target of more than one jump with the variables live at that program point.

```
fastpow(b,e):
  r <- 1
  goto loop(b,e,r)

loop(b,e,r):
  if (e <= 0) goto done
  t0 <- e % 2
  if (t0 == 0) goto next(b,e,r)
  r <- r * b
  goto next(b,e,r)

next(b,e,r):
  b <- b * b
  e <- e / 2
  goto loop(b,e,r)

done:
  return r
```

Now we convert to SSA form by generating multiple generations of variables, as in the previous example. Make sure you understand the process on this code.

```
fastpow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next(b1,e1,r1)
  r2 <- r1 * b1
  goto next(b1,e1,r2)

next(b2,e2,r3):
  b3 <- b2 * b2
  e3 <- e2 / 2
  goto loop(b3,e3,r3)

done:
  return r1
```

Next we minimize. We see the following parameterized labels and labeled jumps:

```
loop(b1,e1,r1):

goto loop(b0,e0,r1)
goto loop(b3,e3,r3)

next(b2,e2,r3):

goto next(b1,e1,r1)
goto next(b1,e1,r2)
```

We see that all arguments to loop are necessary, but that the b and e arguments in the calls to next are the same and can be eliminated (substituting b_1 for b_2 and e_1 for e_2). This yields the SSA at the top of the next page.

If we want a minimal SSA (which we should), we now need to re-examine the calls to loop, because it is possible that the substitution of b_1 for b_2 and e_1 for e_2 has unified arguments that were previously distinct. That might in turn render other parameters redundant. Here, we observe that we have already reached the minimal SSA form.

```
fastpow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next(r1)
  r2 <- r1 * b1
  goto next(r2)

next(r3):
  b3 <- b1 * b1
  e3 <- e1 / 2
  goto loop(b3,e3,r3)

done:
  return r1
```

7 Assembly Code Generation from SSA Form

Of course, actual assembly code does not allow parametrized labels. To recover lower level code, we need to implement labeled jumps by moves followed by plain jumps. We show this again on the first example, with SSA and the left and the de-SSA form on the right.

```
pow(b0,e0):
  r0 <- 1
  goto loop(e0,r0)


loop(e1,r1):
  if (e1 <= 0) goto done()
  r2 <- r1 * b0
  e2 <- e1 - 1
  goto loop(e2,r2)


done():
  return r1
```

```
pow(b0,e0):
  r0 <- 1
  e1 <- e0
  r1 <- r0
  goto loop

loop:
  if (e1 <= 0) goto done
  r2 <- r1 * b0
  e2 <- e1 - 1
  e1 <- e2
  r1 <- r2
  goto loop

done:
  return r1
```

In some cases of conditional jumps, there may be no natural place for the additional move instructions and we may have to introduce a new jump target. This is illustrated in the next example, which continues fastpow from above. All the variable to variable moves in this program arise from resolving the labeled jump shown on their left.

```
fastpow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)


loop(b1,e1,r1):
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next(r1)
  r2 <- r1 * b1
  goto next(r2)


next(r3):
  b3 <- b1 * b1
  e3 <- e1 / 2
  goto loop(b3,e3,r3)


done:
  return r1
```

```
fastpow(b0,e0):
  r0 <- 1
  b1 <- b0
  e1 <- e0
  r1 <- r0

loop:
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next0
  r2 <- r1 * b1
  r3 <- r2
  goto next

next0:
  r3 <- r1
  goto next

next:
  b3 <- b1 * b1
  e3 <- e1 / 2
  b1 <- b3
  e1 <- e3
  r1 <- r3
  goto loop

done:
  return r1
```

Either way, we retain here the parameters at the function boundary; we will talk about how the implementation of function calls in a later lecture.

The new form on the right is of course no longer in SSA form. Therefore one cannot apply any SSA-based optimization. Conversion out of SSA should therefore be one of the last steps before code emission. At this point register allocation, possibly with register coalescing, can do a good job of eliminating redundant moves.

8 Φ Functions

Our presentation of SSA using parameterized labels is not standard in the literature. Instead, the standard representation of SSA form uses so-called Φ functions. A Φ function is not actually a function, but represents the assignment of the label parameter from all the jumps that target it. In order for this to be precise, we indicate for each jump which number of argument in the Φ function this particular jump represents. This perhaps best seen in the fastpow example, with the parameterized label form on the left and Φ functions on the right.

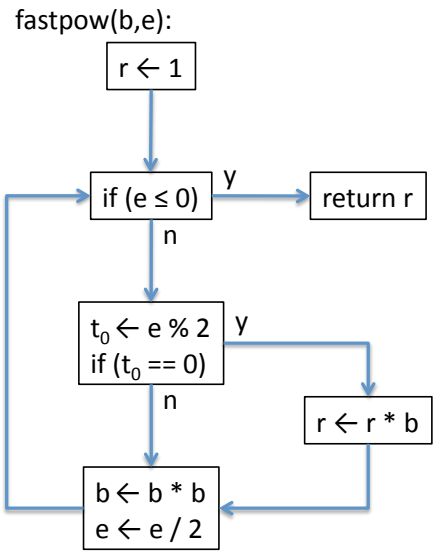
```
fastpow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):


  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next(r1)
  r2 <- r1 * b1
  goto next(r2)

next(r3):

  b3 <- b1 * b1
  e3 <- e1 / 2
  goto loop(b3,e3,r3)

done:
  return r1
```

```
fastpow(b0,e0):
  r0 <- 1
  goto loop/0

loop:
  b1 <- phi(b0,b3)
  e1 <- phi(e0,e3)
  r1 <- phi(r0,r3)
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next/0
  r2 <- r1 * b1
  goto next/1

next:
  r3 <- phi(r1,r2)
  b3 <- b1 * b1
  e3 <- e1 / 2
  goto loop/1

done:
  return r1
```

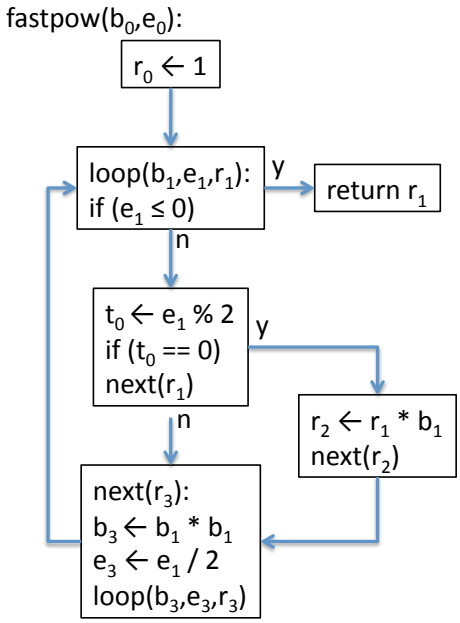
Φ functions only make sense at the beginning of blocks, and they should always have exactly as many arguments as jumps targeting the beginning of the block. Sometimes, the argument number indicators are omitted from from jumps, in which case the textual representation of the abstract assembly code does not have enough information to unambiguously determine its meaning. Then we need a global convention, such as the textually first jump supplies the first argument to the Φ functions, the second jump the second, etc.

9 Graphical Representation

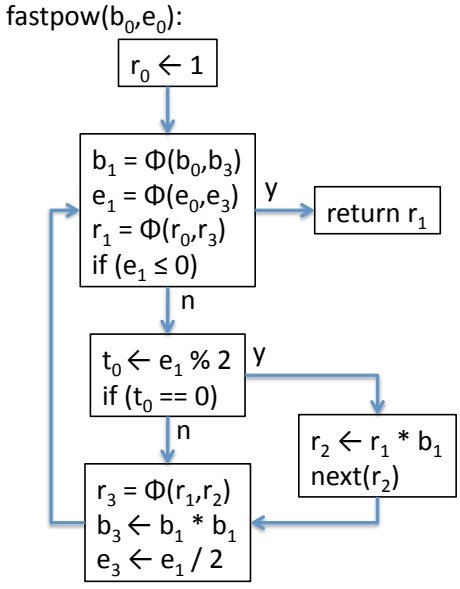
A control flow graph is often represented visually as a graph where the nodes are basic blocks and the directed edges are jump (whether conditional or not). For example, the fastpow function might be drawn as:



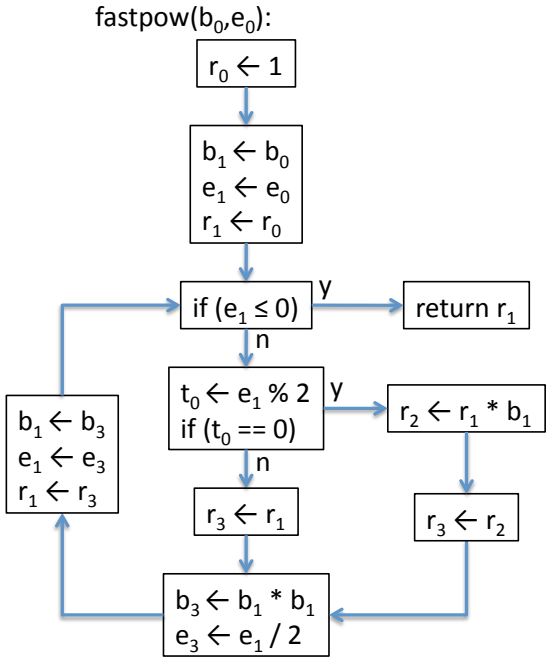
Although not commonly used, SSA form with parameterized labels might look like this:



If we use Φ-functions instead, they would be inserted at the beginning basic blocks.



When we de-SSA, the necessary register moves are added along the edges going into each node which starts with Φ-functions.



10 Conclusion

Static Single Assignment (SSA) form is a quasi-functional form of abstract machine code, where variable assignments are variable bindings, and jumps are tail calls. It was devised by Cytron et al. [CFR+89] and simplifies many program analyses and optimization. Of course, you have to make sure that program transformations maintain the property. The particular algorithm for conversion into SSA form we describe here is to due Aycock and Horspool [AH00]. Hack has shown that programs in SSA form generate chordal interference graphs which means register allocation by graph coloring is particularly efficient [Hac07]. For further reading and some different algorithms related to SSA, you can also consult the Chapter 19 of the textbook [App98].

Questions

1. Can you think of an example of minimal SSA that nevertheless has redundant label arguments?
2. Can you think of situations where the control flow graph for a conditional does not have a subsequent basic block with two incoming control flow edges?

References

[AH00] John Aycock and R. Nigel Horspool. Simple generation of static single-assignment form. In D. Watt, editor, *Proceedings of the 9th International Conference on Compiler Construction (CC'00)*, pages 110–124. Springer LNCS 1781, 2000.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the 16th Annual Symposium on Principles of Programming Languages (POPL 1989)*, pages 25–35, Austin, Texas, January 1989. ACM Press.

[Hac07] Sebastian Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.

Lecture Notes on Lexical Analysis

15-411: Compiler Design
André Platzer*

Lecture 7
September 17, 2013

1 Introduction

Lexical analysis is the first phase of a compiler. Its job is to turn a raw byte or character input stream coming from the source file into a token stream by chopping the input into pieces and skipping over irrelevant details. The primary benefits of doing so include significantly simplified jobs for the subsequent syntactical analysis, which would otherwise have to expect whitespace and comments all over the place. The job of the lexical analysis is also to classify input tokens into types like `INTEGER` or `IDENTIFIER` or `WHILE`-keyword or `OPENINGBRACKET`. Another benefit of the lexing phase is that it greatly compresses the input by about 80%. A lexer is essentially taking care of the first layer of a regular language view on the input language. We follow a presentation similar to a recent book [WSH12, Ch. 2]. Further presentations can be found in [WM95, Ch. 7] and [App98, Ch. 2].

2 Lexer Specification

We fix an alphabet Σ, i.e., a finite set of input symbols, e.g., the set of all letters a-z and digits 0-9 and brackets and operators +,- and so on.[1] The the set Σ^* of words or strings is defined as the set of all finite sequences of elements of Σ. For instance, `ifah5+xy-+` is a string, but not necessarily a very interesting one, from a grammatical perspective (which is what lexers will not have to know about, because that's the parser's job). The empty string with no characters is denoted by ϵ, but you will

*Fall 2013 lecture given by Sri Raghavan

[1] Real lexers also have to deal with capital letters, but we simply pretend to be ignorant about capitalization in these lecture notes to make things easier.

sometimes also find the name λ for it, which we don't use here in order to not get confused with Church's λ-calculus.

A lexer specification has to say what kind of input it accepts and which token type it will associate with a particular input. For example, the fragment `15411` of the input string should be tokenized as an INTEGER. For reasons of representational efficiency, it is a very good idea to specify the input that a lexer accepts by regular expressions. On a side note, regular expressions and their extensions [Sal66, Koz97, HKT00, Pla12] actually turn out to be very useful for many purposes.

Regular expressions r, s are expressions that are recursively built of the following form:

regex	**matches**
a	matches the specific character a from the input alphabet
$[a-z]$	matches a character in the specified range of letters a to z
ϵ	matches the empty string
$r\|s$	matches a string that matches r or one that matches s
rs	matches a string that can somehow be split into two parts, the first matching r, the second matching s
r^*	matches a string that consists of n parts where each part matches r, for any natural number $n \geq 0$

For instance, the set of strings over the alphabet $\{a, b\}$ with no two or more consecutive a's is described by the regular expression $b^*(abb^*)^*(a|\epsilon)$. Other common regular expressions are

regex	**defined**	**matches**
r^+	rr^*	matches a string that consists of n parts where each part matches r, for any natural number $n \geq 1$
$r?$	$r\|\epsilon$	optionally matches r, i.e., matches the empty string or a string matching r

To specify a lexical analyzer we can use a sequence of regular expressions along with the token type that they recognize (the last one, `LPAREN`, for instance, recognizes a single opening parenthesis, whose occurrence on the right hand side we need to quote to distinguish it from brackets used to describe the regular expression, likewise for space):

`IF`	$\equiv if$
`GOTO`	$\equiv goto$
`FOR`	$\equiv for$
`IDENTIFIER`	$\equiv [a-z]([a-z]\|[0-9])^*$
`INT`	$\equiv [0-9][0-9]^*$
`REAL`	$\equiv ([0-9][0-9]^*.[0-9]^*)\|(.[0-9][0-9]^*)$
`LPAREN`	$\equiv$ `"("`
`ASSIGN`	$\equiv$ `"="`
`SKIP`	$\equiv$ `" "`*

In addition, we would say that tokens matching the `SKIP` whitespace recognizer are to be skipped and filtered away from the input, because the parser does not want to see whitespace. Likewise with comments. Note, however, that whitespaces and comments are still significant for the lexer because they separate tokens. For example, `if xyz` gives `IF IDENTIFIER`, while `ifxyz` gives `IDENTIFIER`, even if the `SKIP` token in between is never shown to the parser.

Regular expressions themselves are not unambiguous for splitting an input stream into a token sequence. The input `goto5` could be tokenized as `IDENTIFIER` or as the sequence `GOTO INT`. The input sequence `if 5` could be tokenized as `IF INT` or as `IDENTIFIER INT`.

As disambiguation rule we will use the principle of the *longest possible match*. The longest possible match from the beginning of the input stream will be matched as a token. And if there are still multiple regular expression rules that match the same length, then the first rule with longest match takes precedence over others.

Why do we choose the longest possible match as a disambiguation rule instead of the shortest? The shortest would be easier to implement. But with the shortest match, `ifo = ford_trimotor` would be tokenized as `IF IDENTIFIER ASSIGN FOR IDENTIFIER` and not as `IDENTIFIER ASSIGN IDENTIFIER`. And, of course, the latter is what one would have meant by assigning the identifier for the 1925 Ford Trimotor aircraft "Tin Goose" to the identified flying object (ifo).

3 Lexer Implementation

Lexers are specified by regular expressions. Classically, however, they are implemented by finite automata.

Definition 1 *A* finite automaton *for a finite alphabet Σ consists of*

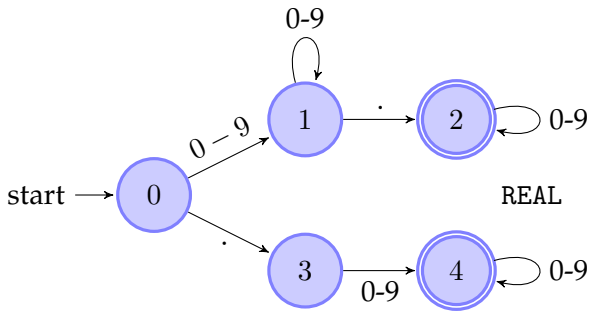
- *a finite set Q of states,*
- *a set $\Delta \subseteq Q \times \Sigma \times Q$ of edges from states to states that are labelled by letters from the input alphabet Σ. We also allow ϵ as a label on an edge, which then means that (q, ϵ, q') is a spontaneous transition from q to q' that consumes no input.*
- *an initial state $q_0 \in Q$*
- *a set of accepting states $F \subseteq Q$.*

The finite automaton accepts an input string $w = a_1 a_2 \ldots a_k \in \Sigma^*$ iff there is an $n \in \mathbb{N}$ and a sequence of states $q_0, q_1, q_2, \ldots, q_n \in Q$ where q_0 is the initial state and $q_n \in F$ is an accepting state such that $(q_{i-1}, a_i, q_i) \in \Delta$ for all $i = 1, \ldots, n$.

In pictures, this condition corresponds to the existence of a set of edges in the automaton labelled by the appropriate input:

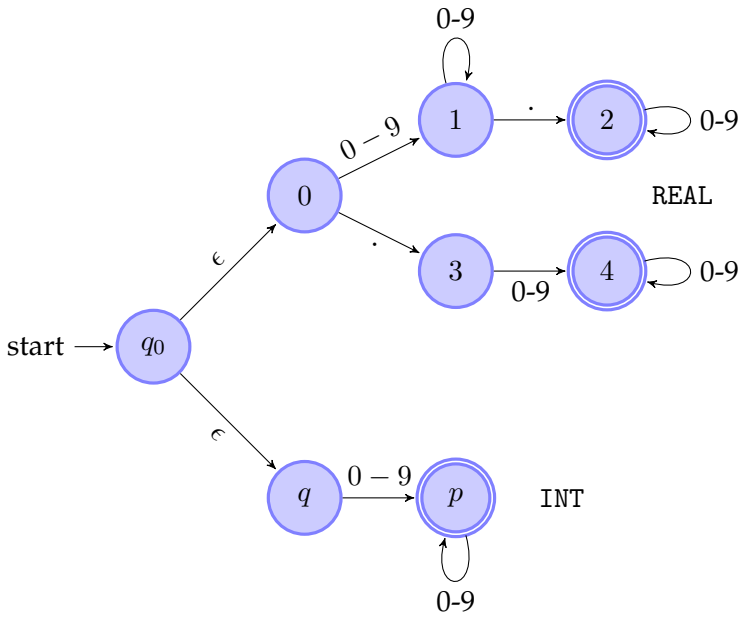
$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \xrightarrow{a_4} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n \in F$$

As an abbreviation for this situation, we also write $(q_0, w) \rightarrow^* (q_n, \epsilon)$. We also write $(q_{i-1}, a_i w_i) \rightarrow (q_i, w_i)$ when $(q_{i-1}, a_i, q_i) \in \Delta$. By that we mean that the automaton, when starting in state q_0 can consume all input of word w with a series of transitions and end up in state q_n with no remaining input to read (ϵ). For instance, an automaton for accepting `REAL` numbers is



Of course, when we use this finite automaton to recognize the number `3.1415926` in the input stream `3.1415926-3+x;if`, then we do not only want to know that a token of type `REAL` has been recognized and that the remaining input is `-3+x;if`. We also want to know what the value of the token of type `REAL` has been, so we store it's value along with the token type.

The above automaton is a *deterministic finite automaton* (DFA). At every state and every input there is at most one edge enabling a transition. But in general, finite automata can be *nondeterministic finite automata* (NFA). That is, for the same input, one path may lead to an accepting state while another attempt fails. That can happen when for the same input letter there are multiple transitions from the same state. In particular, in order to be able to work with the longest possible match principle, we have to keep track of the last accepting state and reset back there if the string cannot be accepted anymore. Consider, for instance, the nondeterministic automaton that accepts both `REAL` and `INT` and starts of by a nondeterministic choice between the two lexical rules.

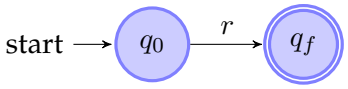


In the beginning, this poor NFA needs to guess which way the future input that he hasn't seen yet will end up. That's hard. But NFAs are quite convenient for specification purposes (just like regular expressions), because the user does not need to worry about these choices.

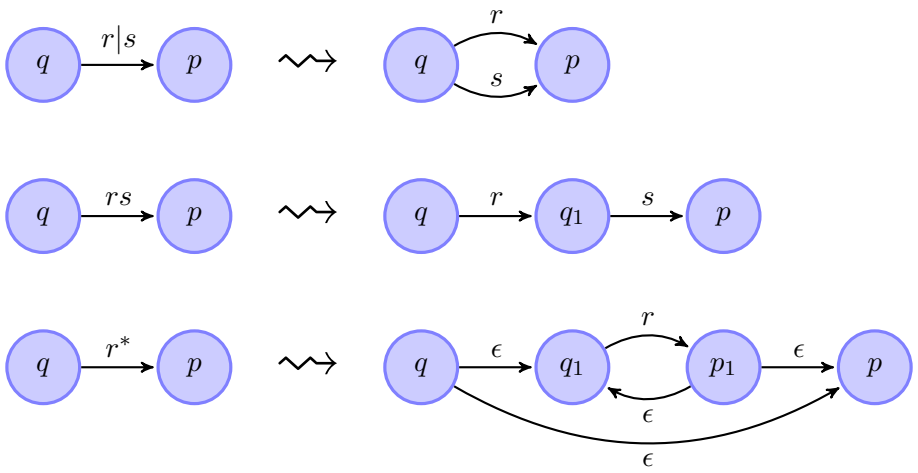
4 Regular Expressions $\rightsquigarrow$ Nondeterministic Finite Automata

Regular expressions are very nice for representing what a lexer is supposed to read. Fortunately, the regular expressions can be converted into a finite automaton (and also backwards, which we will not need here).

For converting a regular expression r into a nondeterministic finite automaton (NFA), we define a recursive procedure. We start with an extended NFA that still has regular expressions as input labels on the edges.



Then we successively transform edges that still have regular expressions as improper input labels by their defining automata patterns. That is, whenever we find a regular expression on an edge that is not just a single letter from the input alphabet then we use the transformation rule to get rid of it



When applying the rule we match on the pattern on the left in the current candidate for an NFA and replace it by the right, introducing new states q_1, q_2 as necessary.

5 Nondeterministic Finite Automata $\rightsquigarrow$ Deterministic Finite Automata

The conversion from regular expressions to NFAs is quite simple. NFAs are convenient for specification purposes, but bad for implementation. It is easy to implement a DFA, however. We just store the current state in a program variable, initialized to q_0, and depending on the next input character, we transition to the next state according to the transition table Δ. Whenever there is an accepting state, we notice that this would be a token that we recognized. But in order to find the longest possible match, we still keep going. If we ultimately find an input character that is not recognized or accepted, then we just backtrack to the last possible match that we have remembered (and unconsume the input characters we have read from the input stream so far). But how would we implement an NFA? There are so many choices that we do not know which one to choose. There is no canonical last accepting choice in an NFA even.

What we could do to implement an NFA is to follow the input like in a DFA implementation, but whenever there is a choice, we follow all options at once. That will branch quickly and will require us to do a lot of work at once, which is inefficient. Nevertheless, it gives us the right intuition about what has to be done. We just need to turn it around and follow the same principle in a precomputation step instead of at runtime. We follow all options and keep the set of choices of where we could be around.

This is the principle behind the *powerset construction* that turns an NFA into a DFA by following all options at once. That is, instead of a single state, we now

consider the set of states in which we could be. We, of course, want to start in the initial powerset state $\{q_0\}$ that only consists of the single initial state q_0. But, first we have to follow all possible ϵ-transitions that lead us from q_0 to other states. When $S \subseteq Q$ is a set of states, we define $Cl^\epsilon(S)$ to be the ϵ-closure of S, i.e., the set of states we can go to by following arbitrarily many ϵ-transitions from states of S, which do not consume any input.

$$Cl^\epsilon(S) := \bigcup_{q \in S} \{q' \ : \ (q, \epsilon) \to^* (q', \epsilon)\}$$

Now from a set of states $S \subseteq Q$ we make a transition, say with input letter a and figure out the set of all states to which we could get to by following a-transitions from any of the S states, again following ϵ-transitions:

$$N(S, a) := Cl^\epsilon(\{q' \in Q \ : \ (q, a) \to (q', \varepsilon) \text{ and } q \in S\})$$

The condition $(q, a) \to (q', \varepsilon)$ is equivalent to $(q, a, q') \in \Delta$. We can summarize all these transitions by just a single a-transition from S to successor $N(S, a)$. Repeating this process results in a DFA that accepts exactly the same language as the original NFA. The complexity of the algorithm could be exponential, though, because there are exponentially many states in the powerset that we could end up using during the DFA construction.

Definition 2 (NFA$\rightsquigarrow$DFA) *Given an NFA* finite automaton (Q, Δ, q_0, F)*, the corresponding DFA* (Q', Δ', q_0', F') *accepting the same language is defined by*

- Q' *is a subset of the sets of all subsets of* Q*, i.e., a part of the powerset* $Q' \subseteq 2^Q$
- $\Delta' := \{(S, a, N(S, a)) \ : \ a \in \Sigma\}$.
- $q_0' := Cl^\epsilon(q_0)$
- $F' := \{S \subseteq Q \ : \ S \cap F \neq \emptyset\}$

After turning the NFA into a DFA, we can directly implement it to recognize tokens from the input stream.

It should be noted that there are direct ways of obtaining DFAs from regular expressions, without going through the construction of NFAs. Those techniques are very algebraic and elegant using Brzozowski derivatives [Brz64].

6 Minimizing Deterministic Finite Automata

Another operation that is often done by lexer generator tools is to minimize the resulting DFA by merging states and reducing the number of states and transitions in the automaton. This is an optimization and we will not pursue it any further.

7 Regular Expression $\rightsquigarrow$ Deterministic Finite Automata

It turns out that there is a very elegant and purely algebraic way of directly translating regular expressions into DFAs without having to go through explicit automata construction, determinization, and possibly minimization. This algebraic approach uses Brzozowski derivatives [Brz64] and *Antimirov's partial derivatives* [Ant96]. For this, we identify regular expressions by the set of words that they match. So instead of saying that regular expression r matches the word w, we simply write $w \in r$. The derivative, $D_a(r)$ of a regular expression r by alphabet letter a is defined as

$$D_a(r) = \{w \ : \ aw \in r\}$$

The derivative represents the set of continuations after letter a that the regular expression r can match. The derivative of a regular expression can be computed syntactically in a very similar way as the usual derivatives of functions. The result is a regular expression.

$$\begin{aligned}
D_a(\emptyset) &= \emptyset \\
D_a(\epsilon) &= \emptyset \\
D_a(a) &= \epsilon \\
D_a(b) &= \emptyset \qquad (b \neq a) \\
D_a(r|s) &= D_a(r) \mid D_a(s) \\
D_a(rs) &= D_a(r)s \mid \delta(r)D_a(s) \\
D_a(r^*) &= D_a(r)r^*
\end{aligned}$$

If we tilt our head a little bit and pretend $|$ was addition $(+)$ and pretend that rs would be multiplication, this looks very much like a standard derivative of functions with ϵ playing the role of 1 and $\emptyset$ playing the role of 0. The primary difference is the occurrence of operator $\delta(r)$ in $D_a(rs)$, which we still need to define. The operator $\delta(r)$ is supposed to detect whether r matches the empty word ϵ. Thus, $\delta(r)$ is defined as follows

$$\delta(r) = \begin{cases} \varepsilon & \text{if } \epsilon \in r \\ \emptyset & \text{otherwise} \end{cases}$$

This operator can be computed entirely syntactically as well

$$\begin{aligned}
\delta(\emptyset) &= \emptyset \\
\delta(\epsilon) &= \epsilon \\
\delta(a) &= \emptyset \\
\delta(r|s) &= \delta(r) \mid \delta(s) \\
\delta(rs) &= \delta(r)\delta(s) \\
\delta(r^*) &= \epsilon
\end{aligned}$$

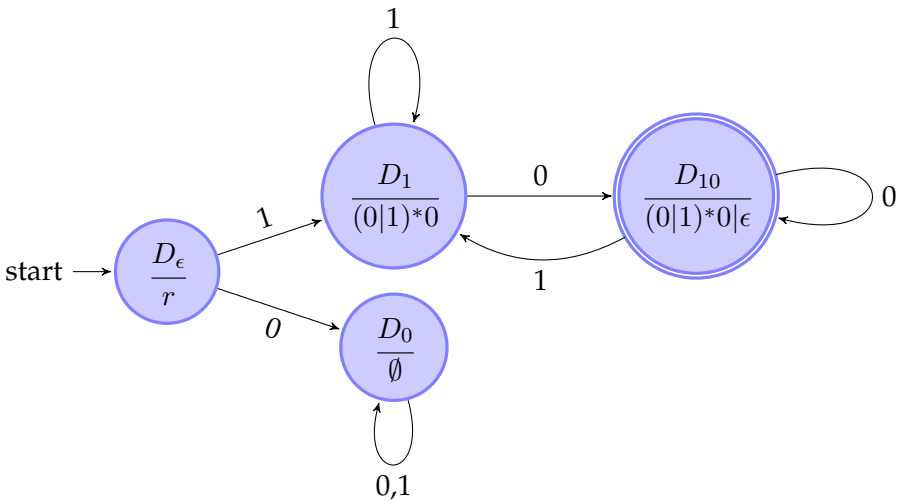
For ordinary functions, higher derivatives can be defined by deriving multiple times. The same thing makes sense for derivatives of regular expressions where we define $D_w(r)$ for a word w by a simple inductive definition on w in which we derive successively by the next letter:

$$D_\epsilon(r) = r$$
$$D_{wa}(r) = D_a(D_w(r))$$

A number of very interesting theoretical and practical results can be proved about Brzozowski derivatives and their extensions. Here we only show how an automaton can be constructed systematically using successive derivatives. It can be shown that this process terminates.

The idea is that $D_a(r)$ represents the "remainder" regular expression of r after input a has been read. Thus, there is a transition with input a from the state r to the state $D_a(r)$. We simply use regular expressions as the states of an automaton (not as their actions like in Section 4).

As an example, consider the regular expression $r = 1(0|1)^*0$. Thus, we construct a DFA for it by starting from a state $D_\epsilon(r) = r$ and successively following all letters a_1 to states $D_{a_1}(r)$ and then on following all letters a_2 to states $D_{a_1a_2}(r)$ and so on. The states where the regular expression matches the empty word ϵ are the ones that are final states. State s is a final state iff $\delta(s) = \epsilon$. In fact, it can be shown that $w \in r$ iff $\delta(D_w(r)) = \epsilon$.



In this automaton graph, we use the notation $\frac{D_w}{s}$ to say that $D_w(r) = s$. It can also be shown that every regular expression can be written in the following linear form

$$r = \delta(r) + \sum_{a \in \Sigma} aD_a(r)$$

8 Summary

Lexical analysis reduces the complexity of subsequent syntactical analysis by first dividing the raw input stream up into a shorter sequence of tokens, each classified by its type (`INT`, `IDENTIFIER`, `REAL`, `IF`, ...). The lexer also filters out irrelevant whitespace and comments from the input stream so that the parser does not have to deal with that anymore. The steps for generating a lexer are

1. Specify the token types to be recognized from the input stream by a sequence of regular expressions
2. Bear in mind that the longest possible match rule applies and the first production that matches longest takes precedence.
3. Lexical analysis is implemented by DFA.
4. Convert the regular expressions into NFAs (or directly into DFAs using derivatives).
5. Join them into a master NFA that chooses between the NFAs for each regular expression by a spontaneous ϵ-transition
6. Determinize the NFA into a DFA
7. Optional: minimize the DFA for space
8. Implement the DFA for a recognizer. Respect the longest possible match rule by storing the last accepted token and backtracking the input to this one if the DFA run cannot otherwise complete.

Questions

1. Why do compilers have a lexing phase? Why not just do without it?
2. Should a lexer return whitespaces and comments?
3. Why do we categorize tokens into token classes, instead of just working with the particular piece of the input string they represent?
4. Why are there programming languages that do not accept inputs like `x----y`?
5. What aspects of the programming language does a lexer not know about?
6. Do lexer tools work with regular expressions or automata internally? Should they?

7. Why can lexers not work with nondeterministic finite automata? They are so useful for description purposes.

8. Should a reserved keyword of a programming language be a token class of its own? What are the benefits and downsides?

References

[Ant96] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic*. MIT Press, Cambridge, 2000.

[Koz97] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

[Pla12] André Platzer. Logics of dynamical systems. In *LICS*, pages 13–24. IEEE, 2012.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[WSH12] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design: Syntactic and Semantic Analysis*. Addison-Wesley, 2012.

Lecture Notes on Context-Free Grammars

15-411: Compiler Design
Frank Pfenning*

Lecture 7
September 19, 2013

1 Introduction

Grammars and parsing have a long history in linguistics. Computer science built on the accumulated knowledge when starting to design programming languages and compilers. There are, however, some important differences which can be attributed to two main factors. One is that programming languages are designed, while human languages evolve, so grammars serve as a means of specification (in the case of programming languages), while they serve as a means of description (in the case of human languages). The other is the difference in the use of grammars and parsing. In programming languages the meaning of a program should be unambiguously determined so it will execute in a predictable way. Clearly, this then also applies to the result of parsing a well-formed expression: it should be unique. In natural language we are condemned to live with its inherent ambiguities, as can be seen from famous examples such as "*Time flies like an arrow*".

In this lecture we review an important class of grammars, called *context-free grammars* (Chomsky-2 in the Chomsky hierarchy [Cho59]) and the associated problem of parsing. They end up to be too awkward for direct use in a compiler, mostly due to the problem of ambiguity, but also due to potential inefficiency of parsing. Alternative presentations of the material in this lecture can be found in the textbook [App98, Chapter 3] and in a seminal paper by Shieber et al. [SSP95]. In the next lecture we will consider more restricted forms of grammars, whose definition, however, is much less natural.

*With edit by André Platzer

2 Context-Free Grammars

Grammars are designed to describe languages, where in our context a *language* is just a set of strings. Abstractly, we think of strings as a sequence of so-called *terminal symbols*. Inside a compiler, these terminal symbols are most likely *lexical tokens*, produced from a bare character string by lexical analysis that already groups substrings into tokens of appropriate type and skips over whitespace.

A *context-free grammar* consists of a set of productions of the form $X \longrightarrow \gamma$, where X is a *non-terminal symbol* and γ is a potentially mixed sequence of terminal and non-terminal symbols. It is also sometimes convenient to distinguish a *start symbol* traditionally named S, for *sentence*. We will use the word *string* to refer to any sequence of terminal and non-terminal symbols. We denote strings by $\alpha, \beta, \gamma, \ldots$. non-terminals are generally denoted by X, Y, Z and terminals by $\mathtt{a}, \mathtt{b}, \mathtt{c}$

For example, the following grammar generates all strings consisting of matching parentheses.

$$\begin{array}{lcl} S & \longrightarrow & \\ S & \longrightarrow & \mathtt{[}S\mathtt{]} \\ S & \longrightarrow & S\,S \end{array}$$

The first rule looks somewhat strange, because the right-hand side is the empty string. To make this more readable, we usually write the empty string as ϵ.

A *derivation* of a sentence w from start symbol S is a sequence $S = \alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_n = w$, where w consists only of terminal symbols. In each step we choose exactly one occurrence of a non-terminal X in α_i and one production $X \longrightarrow \gamma$ and replace this occurrence of X in α_i by γ.

We usually label the productions in the grammar so that we can refer to them by name. In the example above we might write

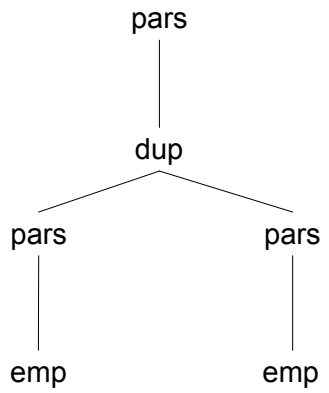
$$\begin{array}{lcll} [\mathsf{emp}] & S & \longrightarrow & \\ [\mathsf{pars}] & S & \longrightarrow & \mathtt{[}S\mathtt{]} \\ [\mathsf{dup}] & S & \longrightarrow & S\,S \end{array}$$

Then the following is a derivation of the string `[[][]]`, where each transition is labeled with the production that has been applied.

$$\begin{array}{lcll} S & \longrightarrow & \mathtt{[}S\mathtt{]} & [\mathsf{pars}] \\ & \longrightarrow & \mathtt{[}SS\mathtt{]} & [\mathsf{dup}] \\ & \longrightarrow & \mathtt{[[}S\mathtt{]}S\mathtt{]} & [\mathsf{pars}] \\ & \longrightarrow & \mathtt{[[]}S\mathtt{]} & [\mathsf{emp}] \\ & \longrightarrow & \mathtt{[[]\,[}S\mathtt{]]} & [\mathsf{pars}] \\ & \longrightarrow & \mathtt{[[]\,[]]} & [\mathsf{emp}] \end{array}$$

We have labeled each derivation step with the corresponding grammar production that was used.

Derivations are clearly not unique, because when there is more than one non-terminal, then we can replace it in any order in the string. In order to avoid this kind of harmless ambiguity in rule order, we like to construct a *parse tree* in which the nodes represents the non-terminals in a string, with the root being S. In the example above we obtain the following tree:



While the parse tree removes some ambiguity, it turns out the sample grammar is ambiguous in another way. In fact, there are infinitely many parse trees of every string in the above language. This can be seen by considering the cycle

$$S \longrightarrow SS \longrightarrow S$$

where the first step is dup and the second is emp, applied either to the first or second occurrence of S. We can get arbitrarily long parse trees for the same string with this.

Whether a grammar is ambiguous in the sense that there are sentences permitting multiple different parse trees is an important question for the use of grammars for the specification of programming languages. The basic problem is that it becomes ambiguous in which grammatical function a specific terminal occurs in the source program. This could lead to misinterpretations. We will see an example shortly.

3 Parse Trees are Deduction Trees

We now present a formal definition of when a terminal string w matches a string γ. We write:

$[r]X \longrightarrow \gamma$	production r maps non-terminal X to string γ
$w : \gamma$	terminal string w matches string γ

The second judgment is defined by the following four simple rules. Here we use string concatenation, denoted by juxtaposing to strings. Note that the empty string ϵ satisfies $\gamma\,\epsilon = \epsilon\,\gamma = \gamma$ and that concatenation is associative (mathematically speaking, strings form a monoid, which is like a group that does not have inverse elements).

$$\frac{}{\epsilon : \epsilon}\ P_1 \qquad \frac{w_1 : \gamma_1 \quad w_2 : \gamma_2}{w_1\,w_2 : \gamma_1\,\gamma_2}\ P_2 \qquad \frac{}{\texttt{a} : \texttt{a}}\ P_3 \qquad \frac{\begin{array}{c}[r]X \longrightarrow \gamma \\ w : \gamma\end{array}}{w : X}\ P_4(r)$$

We have labeled the fourth rule by the name of the grammar production, while the others remain unlabeled. This allows us to omit the actual grammar rule from the premises since it can be looked up in the grammar directly by its name. Then the earlier derivation of `[[][]]` becomes the following deduction.

$$\dfrac{\dfrac{\texttt{[} : \texttt{[}\ P_3 \qquad \dfrac{\dfrac{\dfrac{}{\epsilon : \epsilon}\ P_1}{\epsilon : S}\ P_4(\mathsf{emp}) \qquad \dfrac{}{\texttt{]} : \texttt{]}}\ P_3}{\texttt{[]} : \texttt{[}\,S\,\texttt{]}}\ P_2^2}{\texttt{[]} : S}\ P_4(\mathsf{pars}) \qquad \vdots}{\texttt{[]}\,\texttt{[]} : S\,S}\ P_2 \qquad \dfrac{}{\texttt{]} : \texttt{]}}\ P_3}{\texttt{[[]}\,\texttt{[]]} : \texttt{[}\,S\,\texttt{]}}\ P_2^2$$
$$\texttt{[[]}\,\texttt{[]]} : S\ P_4(\mathsf{pars})$$

The one omitted subdeduction (marked $\vdots$) is identical to its sibling on the left. We observe that the labels have the same structure as the parse tree, except that it is written upside-down. Parse trees are therefore just deduction trees.

4 CYK Parsing

The rules above that formally define when a terminal string matches an arbitrary string can be used to immediately give an algorithm for parsing.

Assume we are given a grammar with start symbol S and a terminal string w_0. Start with a databased of assertions $\epsilon : \epsilon$ and $a : a$ for any terminal symbol occurring in w_0. Now arbitrarily apply the given rules in the following way: if the premises of the rules can be matched against the database, and the conclusion $w : \gamma$ is such that w is a substring of the input w_0 and γ is a string occurring in the grammar, then add $w : \gamma$ to the database. The side conditions are used to focus the parsing process to the facts that may matter during the parsing (i.e., that talk

about the actual input string w_0 being parsed and that fit to the actual grammatical productions in the grammar).

We repeat this process until we reach *saturation*: any further application of any rule leads to conclusion are already in the database. We stop at this point and check if we see $w_0 : S$ in the database. If we see $w_0 : S$, we succeed parsing w_0; if not we fail.

This process must always terminate, since there are only a fixed number of substrings of the grammar, and only a fixed number of substrings of the query string w_0. In fact, only $O(n^2)$ terms can ever be derived if the grammar is fixed and $n = |w|$. Using a meta-complexity result by Ganzinger and McAllester [McA02, GM02] we can obtain the complexity of this algorithm as the maximum of the size of the saturated database (which is $O(n^2)$) and the number of so-called *prefix firings* of the rule. We count this by bounding the number of ways the premises of each rule can be instantiated, when working from left to right. The crucial rule is the splitting rule

$$\frac{w_1 : \gamma_1 \quad w_2 : \gamma_2}{w_1\, w_2 : \gamma_1\, \gamma_2}\ P_2$$

There are $O(n^2)$ substrings, so there are $O(n^2)$ ways to match the first premise against the database. Since $w_1\, w_2$ is also constrained to be a substring of w_0, there are only $O(n)$ ways to instantiate the second premise, since the left end of w_2 in the input string is determined, but not its right end. This yields a complexity of $O(n^2 * n) = O(n^3)$.

The algorithm we have just presented is an abstract form of the Cocke-Younger-Kasami (CYK) parsing algorithm invented in the 1960s. It originally assumes the grammar is in a normal form, and represents substring by their indices in the input rather than directly as strings. However, its general running time is still $O(n^3)$.

As an example, we apply this algorithm using an n-ary concatenation rule as a short-hand. We try to parse `[[][]]` with our grammar of matching parentheses. We start with three facts that derive from rules P_1 and P_3. When working forward it is important to keep in mind that we only infer facts $w : \gamma$ where w is a substring of $w_0 =$ `[[][]]` and γ is a substring of the grammar.

1	`[`	:	`[`	
2	`]`	:	`]`	
3	ϵ	:	ϵ	
4	ϵ	:	S	P_4(emp) 3
5	`[]`	:	`[` S `]`	P_2^2 1, 4, 2
6	`[]`	:	S	P_4(pars) 5
7	`[][]`	:	$S\, S$	P_2 6, 6
8	`[][]`	:	S	P_4(dup) 7
9	`[[][]]`	:	`[` S `]`	P_2^2 1, 8, 2
10	`[[][]]`	:	S	P_4(pars) 9

A few more redundant facts might have been generated, such as `[]` : $S\,S$, but otherwise parsing is remarkably focused in this case. From the justifications in the right-hand column is it easy to generate the same parse tree we saw earlier.

5 Recursive Descent Parsing

For use in a programming language parser, the cubic complexity of the CYK algorithm is unfortunately unacceptable. It is also not so easy to discover potential ambiguities in a grammar (except for a particular input) or give good error messages when parsing fails. What we would like an algorithm that scans the input left-to-right (because that's usually how we design our languages!) and works in one pass through the input.

Unfortunately, some languages that have context-free grammars cannot be specified in the form of a grammar satisfying the above specification. So now we turn the problem around: considering the kind of parsing algorithms we would like to have, can we define classes of grammars that can be parsed with this kind of algorithm? The other property we would like is that we can look at a grammar and decide if it is ambiguous in the sense that there are some strings admitting more than one parse tree. Such grammars should be rejected as specifications for programming languages. Fortunately, the goal of efficient parsing and the goal of detecting ambiguity in a grammar work hand-in-hand: generally speaking, unambiguous grammars are easier to parse.

We now rewrite our rules for parsing to work exclusively from left-to-right instead of being symmetric. This means we do not use general concatenation of strings that are split arbitrarily. Instead, we just consider the left-most terminal or left-most non-terminal. We just prepend a single non-terminal to the beginning of a string. This left non-terminal is then the only part where we allow expansion by a production. We also have to change the nature of the rule for non-terminals so it can handle a non-terminal at the left end of the string.

$$\frac{}{\epsilon : \epsilon}\ R_1 \qquad \frac{w : \gamma}{\mathtt{a}\,w : \mathtt{a}\,\gamma}\ R_2 \qquad \frac{[r]X \longrightarrow \beta \quad w : \beta\,\gamma}{w : X\,\gamma}\ R_3(r)$$

Rule R_2 *compares* the first terminal `a` of the actual input string $\mathtt{a}w$ with the first terminal `a` of the currently parsed expression $\mathtt{a}\gamma$. For grammar production $[r]X \rightarrow \beta$, rule $R_3(r)$ *generates* or *expands* the righthand side β for the left-most non-terminal X in the currently parsed expression $X\gamma$. Rule $R_3(r)$ uses the grammar production forward to produce the result β. Of course, ultimately, the parse derivation will only be successful if the compare rule R_2 can also match the ultimately generated terminals in the input and the generated parse expression.

At this point the rules are entirely linear (each rule has zero or one premises, note that we count the static grammar productions $[r]X \to \beta$ as part of the rule $R_3(r)$ here) and decompose the string left-to-right (we only proceed by stripping away a terminal symbol `a`).

Rather than blindly using these rules from the premises to the conclusions (which wouldn't be analyzing the string from left to right), couldn't we use them the other way around from the desired conclusions to the premisses? After all, we know what we are trying to get at. Recall that we are starting with a given goal, namely to derive $w_0 : S$, if possible, or explicitly fail otherwise. Now could we use the rules in a goal-directed way? The first two rules certainly do not present a problem. Using the compare rules R_1 and R_2 from conclusions to premisses just successively simplifies the strings by consuming the first token (or ϵ). But the expansion rule R_3 presents a problem, since we may not be able to determine which production we should use if there are multiple productions for a given non-terminal X.

The difficulty then lies in the third rule: how can we decide which production to use? Guessing which expansion β of X in R_3 will enable us to parse w as $\beta\gamma$ could be difficult. Yet, we can turn the question around: for which grammars can we always decide which grammar expansion r to use for $R_3(r)$?

We return to an example to explore this question. We use a simple grammar for an expression language similar one to the one used in Lab 1. We use id and num to stand for identifier and number tokens produced by the lexer.

[assign]	S	$\longrightarrow$	id `=` E `;` S
[return]	S	$\longrightarrow$	`return` E
[plus]	E	$\longrightarrow$	E `+` E
[times]	E	$\longrightarrow$	E `*` E
[ident]	E	$\longrightarrow$	id
[number]	E	$\longrightarrow$	num
[parens]	E	$\longrightarrow$	`(` E `)`

As an example string, consider

```
x = 3; return x;
```
$id(\texttt{"x"}) \;=\; num(3)\;;\; \texttt{return}\; id(\texttt{"x"})$

After lexing, `x` and `3` are replaced by tokens $id(\texttt{"x"})$ and $num(3)$ as indicated in the second line. We write just write those tokens as id and num, for short.

If we always guess right, we would construct the following deduction *from the bottom to the top*. That is, we start with the last line, either determine or guess which rule to apply to get the previous line, etc. until we reach $\epsilon : \epsilon$ (successful parse) or get stuck (syntax error, or wrong guess).

$$
\begin{array}{rcl}
\epsilon & : & \epsilon \\
; & : & ; \\
id\ ; & : & id\ ; \\
id\ ; & : & E\ ; \qquad [\mathsf{ident}] \\
\texttt{return}\ id\ ; & : & \texttt{return}\ E\ ; \\
\texttt{return}\ id\ ; & : & S \qquad [\mathsf{return}] \\
;\ \texttt{return}\ id\ ; & : & ;\ S \\
num\ ;\ \texttt{return}\ id\ ; & : & num\ ;\ S \\
num\ ;\ \texttt{return}\ id\ ; & : & E\ ;\ S \qquad [\mathsf{number}] \\
=\ num\ ;\ \texttt{return}\ id\ ; & : & =\ E\ ;\ S \\
id\ =\ num\ ;\ \texttt{return}\ id\ ; & : & id\ =\ E\ ;\ S \\
id\ =\ num\ ;\ \texttt{return}\ id\ ; & : & S \qquad [\mathsf{assign}]
\end{array}
$$

This parser (assuming all the guesses are made correctly) evidently traverses the input string from left to right. It also produces a *left-most* derivation (always expand the left-most nonterminal first), which we can read off from this deduction by reading the right-hand side from the bottom to top.

We have labeled the inference that potentially involved a choice with the chosen name of the chosen grammar production. If we restrict ourselves to look only at the first token in the input string on the left, which ones could we have predicted correctly? Which grammar production choices could we predict by looking ahead at the first input token?

In the last line (the first guess we have to make) we are trying to parse an S and the first input token is id. There is only one production that would allow this, namely [assign]. So we do not have to guess but just choose deterministically based on the first token id.

In the fourth-to-last line (our second potential choice point), the first token is num and we are trying to parse an E. It is tempting to say that this must be the production [number]. But this is wrong! For example, the string num + id also starts with token num, but we must use production [plus] to parse it correctly. This is bad news, because we cannot decide which production rule to use based on the first token.

In fact, no input token can disambiguate expression productions for us here. The problem is that the rules [plus] and [times] are *left-recursive*, that is, the right-hand side of the production starts with the non-terminal on the left-hand side. But this non-terminal could produce a lot of different strings. We can never decide by a finite token look-ahead which rule to choose, because any token which can start an expression E could arise via the [plus] and [times] productions. We cannot decide if we will need the [plus] or [times] production just based on the first token before we have fully understood what the first E is. Yet E could have unbounded length.

The only thing we can do at our current state of knowledge is to parse the

input with a recursive descent parser, guess our choices, and backtrack to different choices whenever things don't work out.

In the next lecture we develop some techniques for analyzing the grammar to determine if we can parse its language by searching for a deduction without backtracking, if we are permitted some lookahead to make the right choice. This will also be the key for *parser generation*, the process of compiling a grammar specification to a specialized efficient parser.

Questions

1. What is the benefit of using a lexer before a parser?
2. Why do compilers have a parsing phase? Why not just work without it?
3. Is there a difference between a parse tree and an abstract syntax tree? Should there be a difference?
4. What aspects of a programming language does a parser not know about? Should it know about it?
5. For which programming languages and for which programs is recursive descent parsing slow?
6. What are all the benefits of reading the input from left to right? Are there downsides?
7. Is there a language that CYK can parse but recursive descent cannot parse?
8. What are *all* the difficulties with rule P_2? What are all the difficulties with rule $P_4(r)$?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In P.Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[SSP95] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *J. Log. Program.*, 24(1&2):3–36, 1995.

Lecture Notes on Predictive Parsing

15-411: Compiler Design
Frank Pfenning*

Lecture 9
September 24, 2013

1 Introduction

In this lecture we discuss two parsing algorithms, both of which traverse the input string from left to right. The first, LL(1), makes a decision on which grammar production to use based on the first character of the input string. If that were ambiguous, the grammar would have to be rewritten to fall into this class, which is not always possible. The second, LR(1), can postpone the decision at first by pushing input characters onto a stack and then deciding on the production later, taking into account both the first input character and the stack. It is variations on the latter which are typically used in modern parser generation tools.

Alternative presentations of the material in this lecture can be found in the textbook [App98, Chapter 3] and a paper by Shieber et al. [SSP95].

2 LL(1) Parsing

We have seen in the previous section, that the general idea of recursive descent parsing without restrictions forces us to non-deterministically choose between several productions which might be applied and potentially backtrack if parsing gets stuck after a choice, or even loop (if the grammar is left-recursive). Backtracking is not only potentially very inefficient, but it makes it difficult to produce good error messages in case the string is not grammatically well-formed. Say we try three different ways to parse a given term and all fail. How could we say which of these is the source of the error? This is compounded because nested choices multiply the

*with contributions by André Platzer

number of possibilities. We therefore have to look for ways to disambiguate the choices.

One way is to *require* of the grammar that at each potential choice point we can look at the next input token and based on that token decide which production to take. This is called *1 token lookahead*, and grammars that satisfy this restriction are called *LL(1)* grammars. Here, the first *L* stands for *L*eft-to-right; the second *L* stands for *L*eftmost parse (which a recursive descent parser generates) and *1* stands for *1 token lookahead*. Potentially, we could also define LL(2), LL(3), etc., but these are of limited practical utility.

Since we are restricting ourselves to parsing by a left-to-right traversal of the input string, we will consider only tails, or postfixes of the input strings, and also of the strings in the grammar, when we restrict our inference rules. For short, we will say γ is a *postfix substring* of the grammar, or w is a postfix substring of the input string w_0. For example, in the grammar

$$\begin{array}{lrcl} [\mathsf{emp}] & S & \longrightarrow & \\ [\mathsf{pars}] & S & \longrightarrow & \texttt{[}\, S\, \texttt{]} \\ [\mathsf{dup}] & S & \longrightarrow & S\, S \end{array}$$

the only postfix substrings are ϵ, $\texttt{[}\, S\, \texttt{]}$, $S\, \texttt{]}$, $\texttt{]}$, S, and $S\, S$, but not $\texttt{[}\, S$.

We begin be defining two kinds of predicates (later we will have occasion to add a third), where β is either a non-terminal or postfix substring of the grammar.

$\mathsf{first}(\beta, \mathtt{a})$	Token $\mathtt{a}$ can be first in string β
$\mathsf{null}(\beta)$	String β can produce the empty string ϵ

These predicates must be computed entirely statically, by an analysis of the grammar before any concrete string is ever parsed. This is because we want to be able to tell if the parser can do its work properly with 1 token look-ahead regardless of the string it has to parse.

We define the relation $\mathsf{first}(\beta, \mathtt{a})$ by the following rules.

$$\frac{}{\mathsf{first}(\mathtt{a}\, \beta, \mathtt{a})}\ F_1$$

This rule *seeds* the first predicate. Then is it propagated to other strings appearing in the grammar by the following three rules.

$$\frac{\mathsf{first}(X, \mathtt{a})}{\mathsf{first}(X\ \beta, \mathtt{a})}\ F_2 \qquad \frac{\mathsf{null}(X) \quad \mathsf{first}(\beta, \mathtt{a})}{\mathsf{first}(X\ \beta, \mathtt{a})}\ F_3 \qquad \frac{\begin{array}{c}[r] X \longrightarrow \gamma \\ \mathsf{first}(\gamma, \mathtt{a})\end{array}}{\mathsf{first}(X, \mathtt{a})}\ F_4(r)$$

Even though ϵ may be technically a postfix substring of every grammar, it can never arise in the first argument of the first predicate. The auxiliary predicate null is also

easily defined.

$$\frac{}{\mathsf{null}(\epsilon)}\ N_1 \qquad \frac{\mathsf{null}(X) \quad \mathsf{null}(\gamma)}{\mathsf{null}(X\,\gamma)}\ N_2 \qquad \frac{\begin{array}{l}[r]X \longrightarrow \gamma \\ \mathsf{null}(\gamma)\end{array}}{\mathsf{null}(X)}\ N_3$$

We can run these rules to saturation because there are only $O(|G|)$ possible strings in the first argument to both of these predicates, and at most the number of possible terminal symbols in the grammar, $O(|\Sigma|)$, in the second argument. Naive counting the number of prefix firings (see [GM02]) gives a complexity bound of $O(|G| \times |\Xi| \times |\Sigma|)$ where $|\Xi|$ is the number of non-terminals in the grammar. Since usually the number of symbols is a small constant, this is roughly equivalent to $O(|G|)$ and so is reasonably efficient. Moreover, it only happens once, before any parsing takes place.

Next, we modify the rules for recursive descent parsing from the last lecture to take these restrictions into account. The first two stay the same.

$$\frac{}{\epsilon : \epsilon}\ R_1 \qquad \frac{w : \gamma}{\mathtt{a}\,w : \mathtt{a}\,\gamma}\ R_2$$

The third,

$$\frac{\begin{array}{l}[r]X \longrightarrow \beta \\ w : \beta\,\gamma\end{array}}{w : X\,\gamma}\ R_3(r)$$

is split into two, each of which has an additional precondition.

$$\frac{\begin{array}{l}[r]X \longrightarrow \beta \\ \mathsf{first}(\beta, \mathtt{a}) \\ \mathtt{a}\,w : \beta\,\gamma\end{array}}{\mathtt{a}\,w : X\,\gamma}\ R_3' \qquad \frac{\begin{array}{l}[r]X \longrightarrow \beta \\ \mathsf{null}(\beta) \\ w : \beta\,\gamma\end{array}}{w : X\,\gamma}\ R_3''?$$

We would like to say that a grammer is LL(1) if the additional preconditions in these last two rules make all choices unambiguous when an arbitrary non-terminal X is matched against a string starting with an arbitrary terminal $\mathtt{a}$. Unfortunately, this does not quite work yet in the presence non-terminals that can rewrite to ϵ, because the second rule above does not even look at the input string.

To further refine this we need one additional predicate, again on postfix strings in the grammar and non-terminals.

$\mathsf{follow}(\beta, \mathtt{a})$ Token $\mathtt{a}$ can follow string β in a valid string

We seed this relation with the rules

$$\frac{\begin{array}{l} X\,\gamma\ \mathit{postfix} \\ \mathsf{first}(\gamma, \mathtt{a}) \end{array}}{\mathsf{follow}(X, \mathtt{a})}\ W_1$$

Here, $X\,\gamma$ *postfix* means that the string $X\,\gamma$ appears as a postfix substring on the right-hand side of a production. We then propagate this information applying the following rules from premises to conclusion until saturation is reached.

$$\frac{\mathsf{follow}(\mathtt{b}\,\gamma, \mathtt{a})}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_2 \qquad \frac{\mathsf{follow}(X\,\gamma, \mathtt{a})}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_3 \qquad \frac{\begin{array}{l} \mathsf{follow}(X\,\gamma, \mathtt{a}) \\ \mathsf{null}(\gamma) \end{array}}{\mathsf{follow}(X, \mathtt{a})}\ W_4 \qquad \frac{\begin{array}{l} [r]X \longrightarrow \gamma \\ \mathsf{follow}(X, \mathtt{a}) \end{array}}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_5$$

The first argument here should remain a non-empty postfix or a non-terminal. Now we can refine the proposed R_3'' rule from above into one which is much less likely to be ambiguous.

$$\frac{\begin{array}{l} [r]X \longrightarrow \beta \\ \mathsf{first}(\beta, \mathtt{a}) \\ \mathtt{a}\,w : \beta\,\gamma \end{array}}{\mathtt{a}\,w : X\,\gamma}\ R_3' \qquad \frac{\begin{array}{l} [r]X \longrightarrow \beta \\ \mathsf{null}(\beta) \\ \mathsf{follow}(X, \mathtt{a}) \\ \mathtt{a}\,w : \beta\,\gamma \end{array}}{\mathtt{a}\,w : X\,\gamma}\ R_3''$$

We avoid creating an explicit rule to treat the empty input string by appending a special $ symbol at the end before starting the parsing process. We repeat the remaining rules for completeness.

$$\frac{}{\epsilon : \epsilon}\ R_1 \qquad \frac{w : \gamma}{\mathtt{a}\,w : \mathtt{a}\,\gamma}\ R_2$$

These rules are interpreted as a parser by proof search, applying them from the conclusion to the premise. We say the grammar is LL(1) if for any goal $w : \gamma$ at most one rule applies. If X cannot derive ϵ, this amounts to checking that there is at most one production $X \longrightarrow \beta$ such that $\mathsf{first}(\beta, \mathtt{a})$. For nullable non-terminals the condition is slightly more complicated, but can still easily be read off from the rules.

We now use a very simple grammar to illustrate these rules. We have transformed it in the way indicated above, by assuming a special token $ to indicate the end of the input string.

$$\begin{array}{lcll} [\mathsf{start}] & S & \longrightarrow & S'\,\$ \\ [\mathsf{emp}] & S' & \longrightarrow & \epsilon \\ [\mathsf{pars}] & S' & \longrightarrow & [\,S'\,] \end{array}$$

This generates all string starting with an arbitrary number of opening parentheses followed by the same number of closing parentheses and an end-of-string marker.

We have:

$\mathsf{null}(\epsilon)$	N_1
$\mathsf{null}(S')$	N_3
$\mathsf{first}(\texttt{[}\, S' \,\texttt{]}, \texttt{[})$	F_1
$\mathsf{first}(\texttt{]}, \texttt{]})$	F_1
$\mathsf{first}(S' \,\texttt{]}, \texttt{]})$	F_3
$\mathsf{first}(S', \texttt{[})$	F_4 [pars]
$\mathsf{first}(S' \,\texttt{]}, \texttt{[})$	F_2
$\mathsf{first}(\$, \$)$	F_1
$\mathsf{first}(S' \,\$, \$)$	F_3
$\mathsf{first}(S' \,\$, \texttt{[})$	F_2
$\mathsf{first}(S, \$)$	F_4 [start]
$\mathsf{first}(S, \texttt{[})$	F_4 [start]
$\mathsf{follow}(S', \$)$	W_1
$\mathsf{follow}(S', \texttt{]})$	W_1
$\mathsf{follow}(\texttt{[}\, S' \,\texttt{]}, \$)$	W_5
$\mathsf{follow}(\texttt{[}\, S' \,\texttt{]}, \texttt{]})$	W_5
$\mathsf{follow}(S' \,\texttt{]}, \$)$	W_3
$\mathsf{follow}(S' \,\texttt{]}, \texttt{]})$	W_3
$\mathsf{follow}(\texttt{]}, \$)$	W_4
$\mathsf{follow}(\texttt{]}, \texttt{]})$	W_4

3 Parser Generation

Parser generation is now a very simple process. Once we have computed the null, first, and follow predicates by saturation from a given grammar, we specialize the inference rules $R_3'(r)$ and $R_3''(r)$ by matching the first two and three premises against grammar productions and saturated database. In this case, this leads to the following specialized rules (repeating once again the two initial rules).

$$\frac{}{\epsilon : \epsilon}\ R_1 \qquad \frac{w : \gamma}{\mathtt{a}\, w : \mathtt{a}\, \gamma}\ R_2$$

$$\frac{\texttt{[}\, w : S' \,\$\, \gamma}{\texttt{[}\, w : S\, \gamma}\ R_3'(\mathsf{start}) \qquad \frac{\$\, w : S' \,\$\, \gamma}{\$\, w : S\, \gamma}\ R_3'(\mathsf{start})$$

$$\frac{\texttt{[}\, w : \texttt{[}\, S' \,\texttt{]}\, \gamma}{\texttt{[}\, w : S'\, \gamma}\ R_3'(\mathsf{pars}) \qquad \frac{\texttt{]}\, w : \gamma}{\texttt{]}\, w : S'\, \gamma}\ R_3''(\mathsf{emp}) \qquad \frac{\$\, w : \gamma}{\$\, w : S'\, \gamma}\ R_3''(\mathsf{emp})$$

Recall that these rules are applied from the bottom-up, starting with the goal $w_0\,\texttt{\$} : S$, where w_0 is the input string. It is easy to observe by pattern matching that each of these rules are mutually exclusive: if one of the applies, none of the other rules applies. Moreover, each rule except for R_1 (which accepts) has exactly one premise, so the input string is traversed linearly from left-to-right, without backtracking. When none of the rules applies, then the input string is not in the language defined by the grammar. This proves that our simple language $\texttt{(}^n\texttt{)}^n$ is LL(1).

Besides efficiency, an effect of this approach to parser generation is that it supports good error messages in the case of failure. For example, if we see the parsing goal $\texttt{(}\,w\,:\,\texttt{)}\,\gamma$ we can state: *Found '(' while expecting ')'.*, and similarly for other cases that match none of the conclusions of the rules.

4 Removing Ambiguities

One standard way to deal with ambiguities in grammars is to rewrite them, but under the constraint that they accept the same strings. When designing our own programming language, we sometimes have the immense luxury to actually change the syntax to make it easier to parse (and, hopefully, also easier to read and understand).

As an example, we use the following simple grammar for expressions.

[plus]	E	$\longrightarrow$	E + E
[times]	E	$\longrightarrow$	E * E
[ident]	E	$\longrightarrow$	id
[number]	E	$\longrightarrow$	num
[parens]	E	$\longrightarrow$	(E)

If we see a simple expression such as `3 + 4 * 5` (which becomes the token stream num + num * num), we cannot predict when we see the + symbol which production to use because of the inherent ambiguity of the grammar.

In order the rewrite it to make the parse tree unambiguous we have to analyze how to *rule out* the unintended parse tree. In the expression `3 + 4 * 5` we have to all the parse equivalent to `3 + (4 * 5)` but we have to *rule out* the parse equivalent to `(3 + 4) * 5`. In other words, the left-hand side of a product is *not allowed to be a sum* (unless it is explicitly parenthesized).

Backing up one step, how about `3 + 4 + 5`? We want addition to be *left associative*, so this should parse as `(3 + 4) + 5`. In other words, we have to *rule out* the parse `3 + (4 + 5)`. Instead of

$$E \longrightarrow E + E$$

we want

$$E \longrightarrow E + P$$

where P is a new nonterminal that does not allow a sum. Continuing the above thought, P *is* allowed to be a product, so we proceed

$$P \longrightarrow P * A$$

Since multiplication is also left-associative, we have made up a new symbol A which cannot be a product. In fact, in our language A can only be an identifier, a number, or a parenthesized (arbitrary) expression.

[plus]	E	$\longrightarrow$	E + P
[times]	P	$\longrightarrow$	P * A
[ident]	A	$\longrightarrow$	id
[number]	A	$\longrightarrow$	num
[parens]	A	$\longrightarrow$	(E)

This is not yet complete, because it is in fact empty: it claims an expression must always be a sum. But it could also just be a product. Similarly, products P may just consist of an atom A. This yields:

[plus]	E	$\longrightarrow$	E + P
[e/p]	E	$\longrightarrow$	P
[times]	P	$\longrightarrow$	P * A
[p/a]	P	$\longrightarrow$	A
[ident]	A	$\longrightarrow$	id
[number]	A	$\longrightarrow$	num
[parens]	A	$\longrightarrow$	(E)

You should convince yourself that this grammar is now unambiguous. Unfortunately, it is not LL(1): from the first token we cannot tell which grammar production to choose. In fact any token can start *any production*!

5 LR(1) Parsing

One difficulty with LL(1) parsing is that it is often difficult or impossible to rewrite a grammar so that 1 token look-ahead during a left-to-right traversal becomes unambiguous. The example in the previous section illustrate this: it was relatively easy to rewrite the grammar to be unambiguous, but we need much more work to make it LL(1).

We can react by rewriting the grammar, at significant expense of readability, or we could just specify that (a) addition and multiplication are left-associative, and (b) multiplication has higher precedence than addition, + $<$ *. Obviously, the latter is more convenient, but how can we make it work?

The idea is to put off the decision on which productions to use and just *shift* the input symbols onto a stack until we can make the decision! We write

$\gamma \mid w$ parse input w under stack γ

where, as generally in predictive parsing, the rules are interpreted as transitions from the conclusion to the premises. The parsing attempt succeeds if we can consume all of w and and produce the start symbol S on the left-hand side. That is, the deduction representing a successful parse of terminal string w_0 has the form

$$\frac{}{S \mid \epsilon}\, R_1$$
$$\vdots$$
$$\epsilon \mid w_0$$

Parsing is defined by the following rules:

$$\frac{}{S \mid \epsilon}\, R_1(= \mathsf{accept})$$

$$\frac{\gamma\, \mathsf{a} \mid w}{\gamma \mid \mathsf{a}\, w}\, R_2(= \mathsf{shift}) \qquad \frac{\begin{array}{c}[r]X \longrightarrow \beta \\ \gamma\, X \mid w\end{array}}{\gamma\, \beta \mid w}\, R_3(r)(= \mathsf{reduce}(r))$$

We resume the example above, parsing num + num * num. After one step (reading this bottom-up)

num	$\mid$	+ num * num	?
ϵ	$\mid$	num + num * num	shift

we already have to make a decision: should we shift + or should we reduce num using rule [number]. In this case the action to reduce is forced, because we will never get another chance to see this num as an E.

E	$\mid$	+ num * num	?
num	$\mid$	+ num * num	reduce(number)
ϵ	$\mid$	num + num * num	shift

At this point we need to shift +; no other action is possible. We take a few steps and arrive at

E + E	$\mid$	* num	
E +num	$\mid$	* num	reduce(number)
E +	$\mid$	num * num	shift
E	$\mid$	+ num * num	shift
num	$\mid$	+ num * num	reduce(number)
ϵ	$\mid$	num + num * num	shift

At this point, we have a real conflict. We can either reduce, viewing E + E as a subexpression, or shift and later consider E * E as a subexpression. Since the * has higher precedence than +, we need to shift.

E	ϵ	accept
E + E	ϵ	reduce(plus)
E + E * E	ϵ	reduce(times)
E + E * num	ϵ	reduce(number)
E + E *	num	shift
E + E	* num	shift
E +num	* num	reduce(number)
E +	num * num	shift
E	+ num * num	shift
num	+ num * num	reduce(number)
ϵ	num + num * num	shift

Since E was the start symbol in this example, this concludes the deduction. If we now read the lines from the top to the bottom, ignoring the separator, we see that it represents a *rightmost* derivation of the input string. So we have parsed analyzing the string from left to right, constructing a rightmost derivation. This type of parsing algorithms is called LR-parsing, where the L stands for left-to-right and the R stands for rightmost.

The decisions above are based on the postfix of the stack on the left-hand side and the first token on the right-hand side. Here, the postfix of the stack on the left-hand side must be a *prefix substring* of a grammar production. If not, it would be impossible to complete it in such a way that a future grammar production can be applied in a reduction step: the parse attempt is doomed to failure.

6 LR(1) Parsing Tables

We could now define again a slightly different version of follow(γ, a), where γ is a prefix substring of the grammar or a non-terminal, and then specialize the rules. An alternative, often used to describe parser generators, is to construct a *parsing table*. For an LR(1) grammar, this table contains an entry for every prefix substring of the grammar and token seen on the input. An entry describes whether to shift, reduce (and by which rule), or to signal an error. If the action is ambiguous, the given grammar is not LR(1), and either an error message is issued, or some default rule comes into effect that chooses between the options.

We now construct the parsing table, assuming + $<$ *, that is, multiplication binds more tightly than addition. Moreover, we specify that both addition and multiplication are *left associative* so that, for example, `3 + 4 + 5` should be parsed

as `(3 + 4) + 5`. We have removed id since it behaves identically to num.

$$\begin{array}{lrcl} [\text{plus}] & E & \longrightarrow & E \texttt{ + } E \\ [\text{times}] & E & \longrightarrow & E \texttt{ * } E \\ [\text{number}] & E & \longrightarrow & num \\ [\text{parens}] & E & \longrightarrow & \texttt{(} E \texttt{)} \end{array}$$

As before, we assume that a special end-of-file token `$` has been added to the end of the input string. When the parsing goal has the form $\gamma\,\beta \mid \texttt{a}\,w$ where β is a prefix substring of the grammar, we look up β in the left-most column and `a` in the top row to find the action to take. The non-terminal $\epsilon\,E$ in the last line is a special case in that E must be the only thing on the stack. In that case we can accept if the next token is `$` because we know that `$` can only be the last token of the input string.

$\beta \setminus$ a	+	*	num	(	)	$
E + E	reduce(plus) (+ left assoc.)	shift (+ $<$ *)	error	error	reduce(plus)	reduce(plus)
E * E	reduce(times) (+ $<$ *)	reduce(times) (* left assoc.)	error	error	reduce(times)	reduce(times)
num	reduce(number)	reduce(number)	error	error	reduce(number)	reduce(number)
(E)	reduce(parens)	reduce(parens)	error	error	reduce(parens)	reduce(parens)
E +	error	error	shift	shift	error	error
E *	error	error	shift	shift	error	error
(E	shift	shift	error	error	shift	error
(	error	error	shift	shift	error	error
ϵ	error	error	shift	shift	error	error
$\epsilon\,E$	shift	shift	error	error	error	accept(E)

We can see that the bare grammar has four shift/reduce conflicts, while all other actions (including errors) are uniquely determined. These conficts arise when E + E or E * E is on the stack and either + or * is the first character in the remaining input string. It is called a shift/reduce conflict, because either a shift action or a reduce action could lead to a valid parse. Here, we have decided to resolve the conflicts by giving a precedence to the operators and declaring both of them to be left-associative.

It is also possible to have reduce/reduce conflicts, if more than one reduction could be applied in a given situation, but it does not happen in this grammar.

Parser generators will generally issue an error or warning when they detect a shift/reduce or reduce/reduce conflict. For many parser generators, the default behavior of a shift/reduce conflict is to shift, and for a reduce/reduce conflict to apply the textually first production in the grammar. Particularly the latter is rarely what is desired, so we strongly recommend rewriting the grammar to eliminate any conflicts in an LR(1) parser.

One interesting special case is the situation in a language where the else-clause of a conditional is optional. For example, one might write (among other productions)

$$\begin{array}{lcl} E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E \\ E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E \end{array}$$

Now a statement

```
if b then if c then x else y
```

is ambiguous because it would be read as

```
if b then (if c then x) else y
```

or

```
if b then (if c then x else y)
```

In a shift/reduce parser, typically the default action for a shift/reduce conflict is to shift to extend the current parse as much as possible. This means that the above grammar in a tool such as ML-Yacc will parse the ambiguous statement into the second form, that is, the `else` is match with the most recent unmatched `if`. This is consistent with language such as C (or C0, the language used in this course), so we can tolerate the above shift/reduce conflict, if you wish, instead of rewriting the grammar to make it unambiguous.

We can also think about how to rewrite the grammar so it is unambiguous. What we have to so is rule out the parse

```
if b then (if c then x) else y
```

In other words, the *then* clause of a conditional should be balanced in terms of if-then-else and not have something that is just an if-then without an *else* clause.

$$\begin{array}{lcl} E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E \\ E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E'\ \mathsf{else}\ E \\ \\ E' & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E'\ \mathsf{else}\ E' \\ E' & \longrightarrow & \ldots \end{array}$$

We would also have to repeat all the other clauses for E, or refactor the grammar so the other productions of E can be shared with E'.

Questions

1. What happens if we remove the ϵ from the last entry in the LR parser table? Aren't ϵ's irrelevant and can always be removed?
2. What makes `x*y;` difficult to parse in C and C0? Discuss some possible solutions, once you have identified a problem?
3. Give a very simple example of a grammar with a shift/reduce conflict.
4. Give an example of a grammar with a shift/reduce conflict that occurs in programming language parsing and is not easily resolved using associativity or precedence of arithmetic operators.
5. Give a very simple example of a grammar with a reduce/reduce conflict.
6. Give an example of a grammar with a reduce/reduce conflict that occurs in programming language parsing and is not easily resolved.
7. In the reduce rule, we have used a number of symbols on the top of the stack and the lookahead to decide what to do. But isn't a stack something where we can only read one symbol off of the top? Does it make a difference in expressive power if we allow decisions to depend on 1 or on 10 symbols on the top of the stack? Does it make a difference in expressive power if we allow 1 or arbitrarily many symbols from the top of the stack for the decision?
8. What's wrong with this grammar that was meant to define a program P as a sequence of statements S by $P \to S \mid P; P$

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In P.Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

[SSP95] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.

Lecture Notes on Intermediate Representation

15-411: Compiler Design
Frank Pfenning*

Lecture 10
September 26, 2013

1 Introduction

In this lecture we discuss the "middle end" of the compiler. After the source has been parsed and elaborated we obtain an abstract syntax tree, on which we carry out various static analyses to see if the program is well-formed. In the L2 language, this consists of type-checking (which is rather straightforward), checking that every finite control flow path ends in a return statement, that every variable is initialized before its use along every control flow path. For more specific information you may refer to the Lab 2 specification.

After we have constructed and checked the abstract syntax tree, we transform the program through several forms of intermediate representation on the way to abstract symbolic assembly and finally actual x86-64 assembly form. How many intermediate representations and their precise form depends on the context: the complexity and form of the language, to what extent the compiler is engineered to be retargetable to different machine architectures, and what kinds of optimizations are important for the implementation. Some of the most well-understood intermediate forms are intermediate representation trees (IR trees), static single-assignment form (SSA), quads and triples. Quads (that is, three-address instructions) and triples (two-address instructions) are closer to the back end of the compiler and you will probably want to use one of them, maybe both. In this lecture we focus on IR trees.

*with contributions by André Platzer

2 Abstract Syntax Trees

We describe abstract syntax trees in a BNF form (Backus-Naur Form) which was originally designed for describing grammars. An abstract syntax tree is the output of parsing and is formed by removing immaterial information from the parse tree (e.g., tokens that are not important in the tree structure) and transforming into a more canonical form. Here we use BNF to describe the recursive structure of the abstract syntax trees.

$$\begin{array}{lrcl} \text{Expressions} & e & ::= & n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid f(e_1, \ldots, e_n) \\ & & & \mid e_1 \texttt{ ? } e_2 \mid \texttt{!}e \mid e_1 \texttt{ \&\& } e_2 \mid e_1 \texttt{ || } e_2 \\ \text{Statements} & s & ::= & \mathsf{assign}(x, e) \mid \mathsf{if}(e, s_1, s_2) \mid \mathsf{while}(e, s) \\ & & & \mid \mathsf{return}(e) \mid \mathsf{nop} \mid \mathsf{seq}(s_1, s_2) \end{array}$$

We use n for constants, x for variables, $\oplus$ for effect-free operators, $\oslash$ for potentially effectful operators (such as division, which could raise an exception), '?' for comparison operators returning a boolean, `!`, `&&`, and `||` for logical negation, conjunction, and disjunction, respectively. The latter have the meaning as in C, always returning either 0 or 1, and short-circuiting evaluation if the left-hand side is false (for `&&`) or true (for `||`).

3 IR Trees

In the translation to IR trees we want to achieve several goals. One is to isolate potentially effectful expressions, making their order of execution explicit. This simplifies instruction selection and also means that the remaining pure expressions can be optimized much more effectively. Another goal is to make the control flow explicit in the form of conditional or unconditional branches, which is closer to the assembly language target and allows us to apply standard program analyses based on an explicit control flow graph. The treatment in the textbook achieves this [App98, Chapters 7 and 8] but it does so in a somewhat complicated manner using tree transformations that would not be motivated for our language.

We describe the IR through *pure expressions* p and *commands* c. Programs r are just sequences of commands; typically these would be the bodies of function definitions. An empty sequence of commands is denoted by '$\cdot$', and we write $r_1 \ ; \ r_2$ for the concatenation of two sequences of commands.

$$
\begin{array}{lrcl}
\text{Pure Expressions} & p & ::= & n \mid x \mid p_1 \oplus p_2 \\
\text{Commands} & c & ::= & x \leftarrow p \\
 & & \mid & x \leftarrow p_1 \oslash p_2 \\
 & & \mid & x \leftarrow f(p_1, \ldots, p_n) \\
 & & \mid & \mathsf{if}\ (p_1\ ?\ p_2)\ \mathsf{goto}\ l \\
 & & \mid & \mathsf{goto}\ l \\
 & & \mid & l: \\
 & & \mid & \mathsf{return}(p) \\
\text{Programs} & r & ::= & c_1\ ;\ \ldots\ ;\ c_n
\end{array}
$$

Pure expressions are a subset of all expressions that do not have any side effects. We choose an IR tree representation in which potentially effectful operations and function calls can only appear at the top-level of assignments. The logical operators are no longer present and must be eliminated in the translation in favor of conditionals. These transformations help optimizations and analysis. Function calls only take pure arguments, which guarantees the left-to-right evaluation order prescribed in the C0 language semantics. Since function calls may have effects, we also lift function calls to the command level rather than embedding them inside expression evaluation.

4 Translating Expressions

The first idea may be to translate abstract syntax expressions to pure expressions, but this does not quite work because potentially effectful expressions have to be turned into commands, and commands are not permitted inside pure expressions. Returning just a command, or sequence of commands, is also insufficient because we somehow need to refer to the result of the translation as a pure expression so we can use it, for example, in a conditional jump or return command.

A solution is to translate from an expression e to a pair consisting of a sequence of instructions r and a pure expression p. After executing r, the value of p will the value of e (assuming the computation does not abort). We write

$$\mathsf{tr}(e) = \langle \check{e}, \hat{e} \rangle$$

where $\check{e}$ is a sequence of commands r that we need to *write down* to compute the effects of e and $\hat{e}$ is a pure expression p that we can use to compute the value of e *back up*. Here are the first three clauses in the definition of $\mathsf{tr}(e)$:

$$
\begin{array}{lcl}
\mathsf{tr}(n) & = & \langle \cdot, n \rangle \\
\mathsf{tr}(x) & = & \langle \cdot, x \rangle \\
\mathsf{tr}(e_1 \oplus e_2) & = & \langle (\check{e}_1\ ;\ \check{e}_2), \hat{e}_1 \oplus \hat{e}_2 \rangle
\end{array}
$$

Constants and variables translate to themselves. If we have a pure operation $e_1 \oplus e_2$ it is possible that the subexpressions have effects, so we concatenate the command sequences for these to expressions $\check{e}_1$ and $\check{e}_2$. Now $\hat{e}_1$ and $\hat{e}_2$ are pure expressions referring to the values of e_1 and e_2, respectively, so we can combine them with a pure operation to get a pure expression representing the result.

We can see that the translation of any pure expression p yields an empty sequence of commands followed by the same pure expression p, that is, $\mathsf{tr}(p) = \langle \cdot, p \rangle$. Effectful operations and function calls require us to introduce some commands and a fresh temporary variable to refer to the value resulting from the operation or call.

$$\begin{array}{lcll} \mathsf{tr}(e_1 \oslash e_2) & = & \langle (\check{e}_1 \,;\, \check{e}_2 \,;\, t \leftarrow \hat{e}_1 \oslash \hat{e}_2), t \rangle & (t \text{ new}) \\ \mathsf{tr}(f(e_1, \ldots, e_n)) & = & \langle (\check{e}_1 \,;\, \ldots \,;\, \check{e}_n \,;\, t \leftarrow f(\hat{e}_1, \ldots, \hat{e}_n)), t \rangle & (t \text{ new}) \end{array}$$

We postpone the translation of boolean expressions e_1 `?` e_2, `!`e, e_1 `&&` e_2 and e_1 `||` e_2 to Section 6.

5 Translating Statements

Translating statements is in some ways simpler, because we only need to return a sequence of instructions. It is slightly more complicated in other ways, since we have to manage control flow via jumps and conditional branches. So the statement translation takes three arguments: the statement to translate, and two optional labels. We elide these labels for simplicity: they are absent on the top-level and passed down in recursive calls and change when entering a while loop. We write $\mathsf{tr}(s) = \check{s}$, where $\check{s}$ is a sequence of commands r.

Assigments and conditionals are simple, given the translation of expression from the previous section, as are return, nop and seq.

$$\begin{array}{lcl} \mathsf{tr}(\mathsf{assign}(x, e)) & = & \check{e} \,;\, x \leftarrow \hat{e} \\ \mathsf{tr}(\mathsf{return}(e)) & = & \check{e} \,;\, \mathsf{return}(\hat{e}) \\ \mathsf{tr}(\mathsf{nop}) & = & \cdot \\ \mathsf{tr}(\mathsf{seq}(s_1, s_2)) & = & \check{s}_1 \,;\, \check{s}_2 \end{array}$$

Conditionals require labels and jumps. Below is a first attempt We combine labels with the following statement (where there is one) to make it easier to read.

$$\begin{array}{lcll} \mathsf{tr}(\mathsf{if}(e, s_1, s_2)) & = & \check{e} \,; & \\ & & \mathsf{if}\ (\hat{e} == 0)\ \mathsf{goto}\ l_2 \,; & \\ & l_1 \ : & \check{s}_1 \,; & \\ & & \mathsf{goto}\ l_3 \,; & \\ & l_2 \ : & \check{s}_2 & \\ & l_3 \ : & & (l_1, l_2, l_3 \text{ new}) \end{array}$$

We can unify the presentation a bit more by inserting a redundant jump (assuming it will be optimized away late in the compilation) and combining a few commands involving control on the same line.

$$\begin{array}{rcll} \mathsf{tr}(\mathsf{if}(e, s_1, s_2)) & = & & \check{e}\ ; \\ & & & \mathsf{if}\ (\hat{e}\ \texttt{==}\ 0)\ \mathsf{goto}\ l_2\ ;\ \mathsf{goto}\ l_1\ ; \\ & & l_1 \ : & \check{s}_1\ ;\ \mathsf{goto}\ l_3\ ; \\ & & l_2 \ : & \check{s}_2\ ;\ \mathsf{goto}\ l_3\ ; \\ & & l_3 \ : & \qquad\qquad\qquad\qquad\qquad\qquad (l_1, l_2, l_3 \text{ new}) \end{array}$$

The remaining awkwardness in this code comes from having to compute e to a boolean value and then checking this against 0. While this is correct, it does not lead to particularly efficient machine code. We will present an improved translation in the next section.

Here is a similarly straightforward translation for while.

$$\begin{array}{rcll} \mathsf{tr}(\mathsf{while}(e, s)) & = & l_1 \ : & \check{e}; \\ & & & \mathsf{if}\ (\hat{e}\ \texttt{==}\ 0)\ \mathsf{goto}\ l_3\ ;\ \mathsf{goto}\ l_2\ ; \\ & & l_2 \ : & \check{s}\ ;\ \mathsf{goto}\ l_1; \\ & & l_3 \ : & \qquad\qquad\qquad\qquad\qquad\qquad (l_1, l_2, l_3 \text{ new}) \end{array}$$

For the kind of processor we are compiling for, it is advantageous for branch prediction if the conditional jump in the is *backwards*. We can rotate the loop by replicating the loop guard (often small) before entry into the loop body.

$$\begin{array}{rcll} \mathsf{tr}(\mathsf{while}(e, s)) & = & & \check{e}; \\ & & & \mathsf{if}\ (\hat{e}\ \texttt{==}\ 0)\ \mathsf{goto}\ l_3\ ;\ \mathsf{goto}\ l_1\ ; \\ & & l_1 \ : & \check{s}\ ; \\ & & & \check{e}\ ; \\ & & & \mathsf{if}\ (\hat{e})\ \mathsf{goto}\ l_1\ ;\ \mathsf{goto}\ l_3\ ; \\ & & l_3 \ : & \end{array}$$

6 Translating Boolean Expressions

As indicated above, the code with the translations above does not take advantage of the way conditional branches work in x86 and x86-64, where we can compare two values and then branch based on the outcome of the comparison by testing condition flags. So we may look for ways to translation conditionals ($\mathsf{if}(e, s_1, s_2)$) and loops ($\mathsf{while}(e, s)$) into simpler code.

One insight is that we use booleans mostly so we can branch on them. So we define a new function

$$\mathsf{cp}(b, l, l') = r$$

where b is a boolean expression. The resulting command sequence r should jump to l if b is true and jump to l' if b is false. Boolean expressions here are comparisons, negation, logical *and*, and logical *or*. They can also be function calls returning booleans or constants 0 for false and 1 for true.

We define

$$
\begin{array}{lcl}
\mathsf{cp}(e_1 \texttt{ ? } e_2, l, l') & = & \check{e}_1 \texttt{ ; } \check{e}_2 \texttt{ ;} \\
 & & \texttt{if } (\hat{e}_1 \texttt{ ? } \hat{e}_2) \texttt{ goto } l \texttt{ ; goto } l' \\
\mathsf{cp}(\texttt{!}\, e, l, l') & = & \mathsf{cp}(e, l', l) \\
\mathsf{cp}(e_1 \texttt{ \&\& } e_2, l, l') & = & \mathsf{cp}(e_1, l_2, l') \texttt{ ;} \\
 & = & l_2 \texttt{ : } \mathsf{cp}(e_2, l, l') \qquad (l_2 \text{ new}) \\
\mathsf{cp}(e_1 \texttt{ || } e_2, l, l') & = & \textit{left to the reader} \\
 & & \\
\mathsf{cp}(0, l, l') & = & \texttt{goto } l \\
\mathsf{cp}(1, l, l') & = & \texttt{goto } l' \\
\mathsf{cp}(e, l, l') & = & \check{e} \texttt{ ;} \\
 & & \texttt{if } (\hat{e} \texttt{ != } 0) \texttt{ goto } l \texttt{ ; goto } l' \qquad (e = f(e_1, \ldots, e_n))
\end{array}
$$

This is then used in the translation of statements in a straightforward way

$$
\begin{array}{lcl}
\mathsf{tr}(\texttt{if}(b, s_1, s_2)) & = & \mathsf{cp}(b, l_1, l_2) \\
 & & l_1 \texttt{ : } \mathsf{tr}(s_1) \texttt{ ; goto } l_3 \\
 & & l_2 \texttt{ : } \mathsf{tr}(s_2) \texttt{ ; goto } l_3 \\
 & & l_3 \texttt{ : } \qquad (l_1, l_2, l_3 \text{ new})
\end{array}
$$

We leave while loops using the cp translation to the reader.

We still have to define how to compile an expression that happens to be boolean (for example, as part of return statement).

$$
\begin{array}{lcl}
\mathsf{tr}(e) & = \langle & \mathsf{cp}(e, l_1, l_2) \texttt{ ;} \\
 & & l_1 \texttt{ : } t \leftarrow 1 \texttt{ ; goto } l_3 \\
 & & l_2 \texttt{ : } t \leftarrow 0 \texttt{ ; goto } l_3 \\
 & & l_3 \texttt{ :} \\
 & , \ t \ \rangle & \qquad (l_1, l_2, l_3, t \text{ new})
\end{array}
$$

7 Ambiguity in Language Specification

The C standard explicity leaves the order of evaluation of expressions unspecified [KR88, p. 200]:

> *The precedence and associativity of operators is fully specified, but the order of evaluation of expressions is, with certain exceptions, undefined, even if the subexpressions involve side effects.*

At first, this may seem like a virtue: by leaving evaluation order unspecified, the compiler can freely optimize expressions without running afoul the specification. The flip side of this coin is that programs are almost by definition not portable. They may check and execute just fine with a certain compiler, but subtly or catastrophically break when a compiler is updated, or the program is compiled with a different compiler.

A possible reply to this argument is that a program whose proper execution depends on the order of evaluation is simply wrong, and the programmer should not be surprised if it breaks. The flaw in this argument is that dependence on evaluation order may be a very subtle property, and neither language definition nor compiler give much help in identifying such flaws in a program. No amount of testing with a single compiler can uncover such problems, because often the code *will* execute correctly under the decision made for this compiler. It may even be that all available compilers at the time the code is written may agree, say, evaluating expressions from left to right, but the code could break in a future version.

Therefore I strongly believe that language specifications should be entirely unambiguous. In this course, this is also important because we want to hold all compilers to the same standard of correctness. This is also why the behavior of division by 0 and division overflow, namely an exception, is fully specified. It is not acceptable for an expression such as `(1/0)*0` to be "optimized" to `0`. Instead, it must raise an exception.

The translation to intermediate code presented here therefore must make sure that any potentially effectful expressions are indeed evaluated from left to right. Careful inspection of the translation will reveal this to be the case. On the resulting pure expressions, many valid optimizations can still be applied which would otherwise be impossible, such as commutativity, associativity, or distributivity, all of which hold for modular arithmetic.

8 Translating C0 to C

At this point in time, the `cc0` compiler for C0 performs lexing, parsing, and static semantic checks and then generates corresponsing C code. This translation has to take care of protecting the C0 code against the undefined or unspecified behavior of certain expressions in C. We list here some of them and the compiler's approach to accomodating them.

- Undefined behavior of certain divisions, shifts, and memory accesses. These are handled by protecting the corresponding operations in C with tests and reliably raising the required exceptions.
- Undefined behavior of overflow of signed integer arithmetic. This is currently handled using the `-fwrapv` flag for `gcc` which requires two's complement integer arithmetic for signed quantities. It was previously handled by

declaring C variables as unsigned (for which modular arithmetic is specified) and casting them to corresponding signed quantities before comparisons.

- Unspecified evaluation order. This is handled by a similar translation as shown this lecture, isolating potentially effectful expressions in sequences of assignment statements. This fixes evaluation order since evaluation order of a sequence statements is guaranteed in C even if it is not for expressions.
- Unspecified size of `int` and related integral types. This is currently handled by checking, before invoking the generated binary, that `int` does indeed have 32 bits. At a previous point in time it was handled more portably by translating C0's `int` type to C's `int32_t`.

Questions

1. In the section on abstract syntax trees it looks like we have defined a language instead of an abstract syntax tree. What is the difference? Why is there a difference? What can be represented in the language but not the AST? What can be represented in the AST but not the language?
2. Which choices of $i, j, k, l \in \{1, 2\}$ make the following translation valid?

$$\mathsf{tr}(e_1 + e_2) = \langle (\check{e}_i; \check{e}_j), \hat{e}_k + \hat{e}_l \rangle$$

3. You can make your translation more uniform by requiring all translations to put their results into temp variables using commands, as we did in the lecture on instruction selection. Discuss the difference.
4. Extend the translations to hand `break` and `continue` for while loops under their C semantics.
5. Does each basic block in the intermediate representation for C0 have at most 2 predecessors?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

Lecture Notes on Calling Conventions

15-411: Compiler Design
Frank Pfenning

Lecture 11
October 1, 2013

1 Introduction

In Lab 3 you will be adding functions to the arithmetic language with loops and conditionals. Compiling functions creates some new issues in the front end and the back end of the compiler. In the front end, we need to make sure functions are called with the right number of arguments, and arguments of the right type. In the back end, we need to create assembly code that respects the *calling conventions* of the machine architecture. Strict adherence to the calling conventions is crucial so that your code can interoperate with library routines, and the environment can call functions that you define.

Calling conventions are rather machine-specific and often quite arcane. You must carefully read the Section 3.2 of the AMD64 ABI [MHJM09][1]. Examples and additional information is provided in Section 6 of a handout on x86-64 Machine-Level Programming by Bryant & O'Hallaron.

2 IR Trees

We have already seen in Lecture 10 that function calls should take pure arguments in order to easily guarantee the left-to-right evaluation order prescribed by our language semantics. Moreover, they should be lifted to the level of commands rather than remain embedded inside expressions because functions may have side-effects.

[1] Available at `http://refspecs.linuxfoundation.org/elf/x86-64-abi-0.99.pdf`

3 Low-Level Intermediate Language

In the low level intermediate language of quads that we have used so far in this course, it is convenient to add a new form of instruction

$$d \leftarrow f(s_1, \ldots, s_n)$$

where each s_i is a source operand and d is a destination operand.

The generic $\mathsf{def}(l, x)$, $\mathsf{use}(l, x)$ and $\mathsf{succ}(l, l')$ predicates are easily defined, assuming for simplicity that source and destinations are all temps.

$$\frac{l : d \leftarrow f(s_1, \ldots, s_n)}{\begin{array}{l}\mathsf{def}(l, d) \\ \mathsf{use}(l, s_i) \quad (1 \leq i \leq n) \\ \mathsf{succ}(l, l+1)\end{array}} \; J_8$$

Unfortunately, this is overly simplistic, because calling conventions prescribe the use of certain fixed registers for passing arguments and receiving results, so we will have to extend the above rule further.

4 x86-64 Calling Conventions

In x86-64, the first six arguments are passed in registers, the remaining arguments are passed on the stack. The result is returned in a specific return register `%rax`. These conventions do not count floating point arguments and results, which are passed in the dedicated floating point registers `%xmm0` to `%xmm7` and on the stack only if there are more than eight floating point parameters. Fortunately, our language has only integers at the moment, so you do not have to worry about the conventions for floating point numbers.

On the x86, stack frames were required to have a frame pointer `%ebp` (base pointer) which had to be saved and restored with each function call. It provided a reliable pointer to the beginning of a stack frame for easy calculation of frame offsets to handle references to arguments and local variables. It also allowed tools such as `gdb` to print backtraces of the stack. On the x86-64 this information is maintained elsewhere and a frame pointer is no longer required.

The general organization of stack frames at the time a procedure is called, will be as follows.

Position	Contents	Frame
$\cdots$	$\cdots$	Caller
`16(%rsp)`	argument 8	
`8(%rsp)`	argument 7	
`(%rsp)`	return address	

Note that all arguments take 8 bytes of space on the stack, even if the type of argument would indicate that only 4 bytes need to be passed.

The function that is called, the *callee*, should set up its stack frame, reserving space for local variables, spilled temps that could not be assigned to registers, and arguments passed to functions it calls in turn. We recommend calculating the total space needed and then decrementing the stack pointer `%rsp` by the appropriate amount. By changing the stack pointer only once, at the beginning, references to parameters and local variables remain constant throughout the function's execution. The stack then looks as follows, where the size of the callee's stack frame is n.

Position	Contents	Frame
...	...	Caller
`n + 16(%rsp)`	argument 8	
`n + 8(%rsp)`	argument 7	
`n + 0(%rsp)`	return address	
	local variables	Callee
	...	
	argument build area	
	for function calls	
	...	
`(%rsp)`	end of frame	
	128 bytes	red zone

Note that `%rsp` should be aligned $0 \mod 16$ before another function is called, and may be assumed to be aligned $8 \mod 16$ on function entry. This happens because the `call` instruction saves the 64-bit return address on the stack.

The area below the stack pointer is called the *red zone* and may be used by the callee as temporary storage for data that is not needed across function calls or even to build arguments to be used before a function call. The ABI states that the red zone "shall not be modified by signal or interrupt handlers." This can be tricky, however, because, for example, Linux kernel code may not respect the red zone and overwrite this area. We therefore suggest not using the red zone.

5 Register Convention

We extract from [MHJM09] the relevant information on register usage. In the first column is a suggested numbering for the purpose of register allocation.

Abstract form	x86-64 Register	Usage	Preserved accross function calls
res_0	`%rax`	return value*	No
arg_1	`%rdi`	argument 1	No
arg_2	`%rsi`	argument 2	No
arg_3	`%rdx`	argument 3	No
arg_4	`%rcx`	argument 4	No
arg_5	`%r8`	argument 5	No
arg_6	`%r9`	argument 6	No
ler_7	`%r10`	caller-saved	No
ler_8	`%r11`	caller-saved	No
lee_9	`%rbx`	callee-saved	Yes
lee_{10}	`%rbp`	callee-saved*	Yes
lee_{11}	`%r12`	callee-saved	Yes
lee_{12}	`%r13`	callee-saved	Yes
lee_{13}	`%r14`	callee-saved	Yes
lee_{14}	`%r15`	callee-saved	Yes
	`%rsp`	stack pointer	Yes

The starred registers have a potentially relevant alternative use. `%al` (the lower 8 bits of `%rax`) contains the number of floating point arguments on the stack in a call to varargs functions. `%rbp` is the frame pointer for the stack frame, in an x86-like calling convention (which is optional for the x86-64).

6 Typical Calling Sequence

If we have 6 or fewer arguments, a typical calling sequence for 32-bit arguments with an instruction

$$d \leftarrow f(s_1, s_2, s_3)$$

will have the following form:

$$\begin{array}{l} arg_3 \leftarrow s_3 \\ arg_2 \leftarrow s_2 \\ arg_1 \leftarrow s_1 \\ \mathsf{call}\ f \\ d \leftarrow res_0 \end{array}$$

First we move the temps into the appropriate argument registers, then we call the function f (represented by a symbolic label), and then we move the result register into the desired destination.

This organization, perhaps just before register allocation, has the advantage that the live ranges of fixed registers (called *precolored nodes* in register allocation) is

minimized. This is important to avoid potential conflict. We have already applied a similar technique in the implementation of div and mod operations, which expect their arguments in fixed registers.

Let us state this as a fundamental principle of code generation that you should strive to adhere to:

> The live range of precolored registers should be as short as possible!

We can now see a problem with our previous calculation of def and use information: the above sequence to actually implement the function call will overwrite the argument registers `%edx`, `%esi`, `%edi` as well as the result register `%eax` (the lower 32bits of the return register `%rax`)! In fact, any of the argument registers, the result register, as well as `%r10` (temporary register for passing static function chain pointers) and `%r11` (temporary register) may not be preserved across function calls and therefore have to be considered to be *defined* by the call. If we represent this in the low-level intermediate language, we would *add* to the rule R_8 the following rule R_8':

$$\frac{\begin{array}{l} l : d \leftarrow f(s_1, \ldots, s_n) \\ \mathsf{caller\text{-}save}(r) \end{array}}{\mathsf{def}(l, r)}\ J_8'$$

where $\mathsf{caller\text{-}save}(r)$ is true of register r among `%rax`, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, `%r10`, and `%r11`.

Here we assume that register aliasing is handled correctly, that is, the register allocator understands that, for example, `%eax` constitutes the lower 32 bits of `%rax`.

Note that all *argument registers* and the *result register* are *caller-save*. This is justified by the fact that we often compute a value for the purposes of passing it into a function, but we do not require that value afterwards. Of course, the result register has to be caller-save, since it will be defined by the called function before it returns.

We refer to argument registers more abstractly as $arg_1, arg_2, \ldots, arg_6$ and ler_7 and ler_8 for the other two caller-save registers (even if they are not used for passing arguments to a function). We refer to the result register `%rax` as res_0.

Now if a temp t (except for d) is live after a function call, we have to add a edge connecting t with any of the fixed registers noted above, since the value of those registers are not preserved across a function call.

The other fixed used of argument registers is of course at the beginning of a function. Again, we should be careful to generate code that keeps the live ranges of functions short. We can accomplish this by moving the argument registers into temps. Under some heuristics in register allocation and coalescing, these moves

can sometimes be eliminated. A function $f(x, y, z)$ might then start with

$$\begin{aligned} f: & \\ & x \leftarrow arg_1 \\ & y \leftarrow arg_2 \\ & z \leftarrow arg_3 \end{aligned}$$

One more note: if it is possible that the function f is a function accepting a variable number of arguments, some additional considerations apply. For example, the low 8 bits of `%rax`, called `%al` hold the number of floating point arguments passed to the function. One therefore sometimes sees `xorl %eax, %eax` before a function call to define zero variable arguments.

7 Callee-Save Registers

The typical calling sequence above takes care of treating caller-save registers correctly. But what about callee-save registers, namely `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14` and `%r15`? In compiling a function we are required that the generated code preserves all the callee-save registers. We generically refer to these registers as lee_i where $9 \leq i \leq 14$.

The standard approach is to save those that are needed onto the stack in the function prologue and restore them from the stack in the function epilogue, just before returning. Of course, saving and restoring them all is safe, but may be overkill for small functions that do not require many registers.

Remember that callee-save registers are essentially live throughout the body of a function, since their value at the return instruction matters. This violates our general rule to keep the live ranges of precolored registers short—in fact, they are maximal!

One simple way to deal with this is by listing them last among the registers to be assigned by register allocation. If we need more than the available number of caller-save registers, we assign callee-save registers before we resort to spilling, but make sure the save them at the beginning of a function and restore them at the end. This is generally more efficient than the usual register spilling since such temps still live in a register throughout the function execution. We use this technique in the example in Section 8.

Another solution is to let register allocation together with register coalescing do the job for us. We can move the contents of all the callee-save registers into temps at the beginning of a function and then move them back at the end. If it turns out these temps are spilled, then they will be saved onto the stack. If not, they may be moved from one register to another and then back at the end. However, this only works well with the right heuristics for assigning registers or using

register coalescing.[2] Register coalescing consults the interference graph to check if we can assign the same register for variable-to-variable moves. Another optimization that can eliminate register-to-register moves is copy propagation, covered in a later lecture. However, copy propogation requires care because it might extend the live range of variables, possibly undoing the care we applied to keep precolored registers contained.

With this technique, the general shape of the code for a function f before register allocation would be

$$
\begin{array}{l}
f: \\
\quad t_1 \leftarrow lee_9 \\
\quad t_2 \leftarrow lee_{10} \\
\quad \ldots \\
\quad \textit{function body} \\
\quad \ldots \\
\quad lee_{10} \leftarrow t_2 \\
\quad lee_9 \leftarrow t_1 \\
\quad \mathsf{ret}
\end{array}
$$

One complication with this approach is that we need to be sure to spill the full 64-bit registers, while registers holding 32-bit integer values might be saved and restored (or directly used as operands) using only 32 bits. Looking ahead, we see that we will need both 32 bit and 64 bit registers and spill slots in the next lab, so we might decide to introduce this complication now. Or we can still treat callee-save registers specially and switch over to a more uniform treatment in the next lab.

With either of the techniques for using callee-save registers, the one additional rule (R_8') is not enough. We should also note that all *callee-save* registers should be considered *live* at the return instruction.

$$
\frac{\begin{array}{l} l : \mathsf{return}\ s \\ \mathsf{callee\text{-}save}(r) \end{array}}{\mathsf{use}(l, r)}\ J_2'
$$

We already know, by prior rule, that s itself is live at l. The rule new rule J_2' correctly flags all callee-save registers as live throughout the function body, unless they are assigned somewhere. The code pattern above achieves exactly that, cutting their live ranges down to a minimum.

8 An Extended Example

We use the recursive version of the power function as an example to illustrate register allocation in the presence of function calls. The C0 source is on the left; the abstract assembly on the right.

[2] One technique for register coalescing is briefly described in Section 8 of Lecture 3.

```
int pow(int b, int e)
//@requires e >= 0;
{
  if (e == 0)
    return 1;
  else
    return b * pow(b, e-1);
}
```

```
pow(b,e):
  if (e == 0) goto done
  t0 <- e - 1
  t1 <- pow(b, t0)
  t2 <- b * t1
  return t2
done:
  return 1
```

First, we convert it to SSA form. Looking at the right, we see it is already in static single assignment form! Looking on the left, we see a purely functional program. Since purely functional programs do not perform assignment, they must already be in SSA form!

Next, we perform liveness analysis. We proceed backward through the program to compute the following information.

program	live-in
pow(b, e) :	
if $(e == 0)$ goto done	b, e
$t_0 \leftarrow e - 1$	b, e
$t_1 \leftarrow$ pow(b, t_0)	b, t_0
$t_2 \leftarrow b * t_1$	b, t_1
return t_2	t_2
done :	
return 1	

Next, we move to a slightly lower-lever representation, making the precolored registers explicit with the code pattern in Section 6.

program	live-in
pow :	
$b \leftarrow arg_1$	arg_1, arg_2
$e \leftarrow arg_2$	b, arg_2
if $(e == 0)$ goto done	b, e
$t_0 \leftarrow e - 1$	b, e
$arg_2 \leftarrow t_0$	b, t_0
$arg_1 \leftarrow b$	b, arg_2
call pow	b, arg_1, arg_2
$t_1 \leftarrow res_0$	b, res_0
$t_2 \leftarrow b * t_1$	b, t_1
$res_0 \leftarrow t_2$	t_2
return	res_0
done :	
$res_0 \leftarrow 1$	
return	res_0

We have not made any callee-save registers explicit yet, in the hope we will not need them. After all, there are only two variables and three temps in the program, but we have eight caller-save registers.

Next, we build the interference graph. For each line l and each temp t defined at l, we create an edge between t and any variable live in the successor. The only exception is a move $t \leftarrow s$, where we don't create and edge between t and s because they could be consistently be assigned to the same register. We find that only b interferes with other temps and precolored registers:

temp	interfering with
b	$res_0, arg_1, arg_2, t_0, t_1$
e	b
t_0	b
t_1	b
t_2	

Implicitly all precolored registers interfere with each other.

However, we forgot one important piece of information[3], namely that the call instruction must be interpreted as *defining all caller-save registers*. Since b remains alive through the function call, it can therefore not be assigned to a caller-save register, based on the code that we have.

We proceed by admitting that we need one caller-save register lee_9 and save and restore it at the beginning and end of the function. We use the push and pop

[3] which we only recalled at the last minute in lecture, too

instructions for the save and restore operations.

program	live-in
pow :	
push lee_9	arg_1, arg_2, lee_9
$b \leftarrow arg_1$	arg_1, arg_2
$e \leftarrow arg_2$	b, arg_2
if $(e == 0)$ goto done	b, e
$t_0 \leftarrow e - 1$	b, e
$arg_2 \leftarrow t_0$	b, t_0
$arg_1 \leftarrow b$	b, arg_2
call pow	b, arg_1, arg_2
$t_1 \leftarrow res_0$	b, res_0
$t_2 \leftarrow b * t_1$	b, t_1
$res_0 \leftarrow t_2$	t_2
pop lee_9	res_0
return	res_0, lee_9
done :	
$res_0 \leftarrow 1$	
pop lee_9	res_0
return	res_0, lee_9

While the callee-save $lee_{10}, \ldots, lee_{14}$ are still (implicitly) live through this function, after the rewrite lee_9 no longer is. Therefore, it no longer interferes with any temps.

We can construct a *simplicial elimination ordering*, from the interference graph, such as:

$$b, e, t_0, t_1, t_2$$

We order the colors (machine registers) as

$$res_0, arg_1, \ldots, arg_6, ler_7, ler_8, lee_9$$

with the idea that caller-save registers come first (including argument registers which we will likely need anyway), followed by the only callee-save register we are currently permitted to use. If we needed more, we would first have to spill and restore them.

From this we construct the assignment

$$\begin{array}{lcl} b & \mapsto & lee_9 \\ e & \mapsto & res_0 \\ t_0 & \mapsto & res_0 \\ t_1 & \mapsto & res_0 \\ t_2 & \mapsto & res_0 \end{array}$$

Applying the substitutions:

```
pow :
   push lee9
   lee9 ← arg1
   res0 ← arg2
   if (res0 == 0) goto done
   res0 ← res0 − 1
   arg2 ← res0
   arg1 ← lee9                (redundant)
   call pow
   res0 ← res0                (redundant)
   res0 ← lee9 ∗ res0
   res0 ← res0                (redundant)
   pop lee9
   return
done :
   res0 ← 1
   pop lee9
   return
```

There are now some redundant instructions that can be eliminated. The self-moves are obvious, and one line becomes a self-move after copy propagation. One would also typically have just one epilog for the function (which restores the callee-save registers and the stack pointer, which is not visible here). Making these last changes, we obtain:

```
pow :
   push lee9
   lee9 ← arg1
   res0 ← arg2
   if (res0 == 0) goto done
   res0 ← res0 − 1
   arg2 ← res0
   call pow
   res0 ← lee9 ∗ res0
   goto epilogue
done :
   res0 ← 1
epilogue :
   pop lee9
   return
```

Using GNU assembler format for x86-64:

```
pow:
        pushq   %rbx

        movl    %edi, %ebx
        movl    %esi, %eax
        testl   %eax, %eax
        je L1
        subl    $1, %eax
        movl    %eax, %esi
        call    pow
        imull   %ebx, %eax
        goto L2
L1:
        movl $1, %eax
L2:
        popq %rbx
        ret
```

References

[MHJM09] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V application binary interface, AMD64 architecture processor supplement. Available at http://refspecs.linuxfoundation.org/elf/x86-64-abi-0.99.pdf, May 2009. Draft 0.99.

Lecture Notes on Static Semantics

15-411: Compiler Design
Frank Pfenning

Lecture 12
October 3, 2013

1 Introduction

After lexing and parsing, a compiler will usually apply *elaboration* to translate the parse tree to a high-level intermediate form often called *abstract syntax*. Then we verify that the abstract syntax satisfies the requirements of the *static semantics*. Sometimes, there is some ambiguity whether a given condition should be enforced by the grammar, by elaboration, or while checking the static semantics. We will not be concerned with details of attribution, but how to describe and then implement various static semantic conditions. The principal properties to verify for C0 and the sublanguages discussed in this course are:

- *Initialization:* variables must be defined before they are used.
- *Proper returns:* functions that return a value must have an explicit return statement on every control flow path starting at the beginning of the function.
- *Types:* the program must be well-typed.

Type checking is frequently discussed in the literature, so we use initialization as our running example and discuss typing at the end, in Section 9.

2 Abstract Syntax

We will use a slightly restricted form of the abstract syntax in Lecture 10 on IR trees, with the addition of variable declaration with their scope. This fragment exhibits all the relevant features for the purposes of the present lecture.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & n \mid x \mid e_1 \oplus e_2 \mid e_1 \texttt{ \&\& } e_2 \\
\text{Statements} & s & ::= & \mathsf{assign}(x, e) \mid \mathsf{if}(e, s_1, s_2) \mid \mathsf{while}(e, s) \\
& & & \mid \mathsf{return}(e) \mid \mathsf{nop} \mid \mathsf{seq}(s_1, s_2) \mid \mathsf{decl}(x, \tau, s)
\end{array}
$$

3 Definition and Use

Initialization guarantees that every variable is defined before it is used. The natural way to specify this in two parts: when is a variable is defined, and when it is used. An error is signaled if we cannot show that every variable in the program is defined before it is used. As usual, this property is an *approximation* of what actual behaviors can be exhibited at runtime.

First, we define when a variable is used in an expression, written as $\mathsf{use}(e, x)$. This is entirely straightforward, since we have a clear separation of expressions and statements in our language.

$$\text{no rule for } \mathsf{use}(n, x) \qquad \frac{}{\mathsf{use}(x, x)} \qquad \text{no rule for } \mathsf{use}(y, x), y \neq x$$

$$\frac{\mathsf{use}(e_1, x)}{\mathsf{use}(e_1 \oplus e_2, x)} \qquad \frac{\mathsf{use}(e_2, x)}{\mathsf{use}(e_1 \oplus e_2, x)}$$

$$\frac{\mathsf{use}(e_1, x)}{\mathsf{use}(e_1 \texttt{ \&\& } e_2, x)} \qquad \frac{\mathsf{use}(e_2, x)}{\mathsf{use}(e_1 \texttt{ \&\& } e_2, x)}$$

We see already here that $\mathsf{use}(e, x)$ is a so-called *may*-property: x *may* be used during the evaluation of x, but it is not guaranteed to be actually used. For example, the expression $y > 0$ `&&` $x > 0$ may or may not actually use x. The expression false `&&` $x > 0$ will actually never use x, and yet we flag it as possibly being used.

This is appropriate: we would like to raise an error if there is a possibility that an unitialized variable may be used. Because determining this in general is undecidable, we need to approximate it. Our approximation essentially says that any variable occurring in an expression may be used. The rule above express this more formally.

For a language to be usable, it is important that the rules governing the static semantics are easy to understand for the programmer and have some internal coherence. While it might make sense to allow false `&&` $x > 0$ in particular, what is the general rule? Designing programming languages and their static semantics is difficult and requires a good balance of formal understanding of the properties of programming languages and programmer's intuition.

Once we have defined use for expressions, we should consider statements. Does an assignment $x = e$ use x? Our prior experience with liveness analysis for register allocation on abstract machine could would say: *only if it is used in* e. We stay consistent with this intuition and terminology and write $\mathsf{live}(s, x)$ for the judgment that x is live in s. This means the value of x is relevant to the execution of s.

Before we specify liveness, we should specify when a variable is *defined*. This is because, for example, the variable x is not live before the statement $x = 3$, because

its current value does not matter for this statement, or any subsequent statement. We write $\mathsf{def}(s, x)$ is the execution of statement s will define s. This is an example of a *must*-property: we want to be sure that whenever s executes (and completes normally, without returning from the current function or raising an exception of some form), the x has been defined.

$$\frac{}{\mathsf{def}(\mathsf{assign}(x, e), x)} \qquad \text{no rule for } \mathsf{def}(\mathsf{assign}(y, e), x), y \neq x$$

$$\frac{\mathsf{def}(s_1, x) \quad \mathsf{def}(s_2, x)}{\mathsf{def}(\mathsf{if}(e, s_1, s_2), x)} \qquad \text{no rule for } \mathsf{def}(\mathsf{while}(e, s), x)$$

The last two rules clearly illustrate that $\mathsf{def}(s, x)$ is a *must*-property: A conditional only defines a variable if is it defined along both branches, and a while loop does not define any variable (since the body may never be executed).

$$\text{no rule for } \mathsf{def}(\mathsf{nop}, x) \qquad \frac{\mathsf{def}(s_1, x)}{\mathsf{def}(\mathsf{seq}(s_1, s_2), x)} \qquad \frac{\mathsf{def}(s_2, x)}{\mathsf{def}(\mathsf{seq}(s_1, s_2), x)}$$

$$\frac{\mathsf{def}(s, x) \quad y \neq x}{\mathsf{def}(\mathsf{decl}(y, \tau, s), x)}$$

The side condition on the last rule apply because s is the scope of y. If we have already checked variable scoping, then in the particular case of C0, y could not be equal to x because that would have led to an error earlier. However, even in this case it may be less error-prone to simply check the condition even if it might be redundant.

A strange case arises for return statement. Since a return statement never completes normally, any subsequent statements are unreachable. It is therefore permissible to claim that *all* variables currently in scope have been defined. We capture this by simply stating that $\mathsf{return}(e)$ defines any variable.

$$\frac{}{\mathsf{def}(\mathsf{return}(e), x)}$$

4 Liveness

We now lift the $\mathsf{use}(e, x)$ property to statements, written as $\mathsf{live}(s, x)$ (x is live in s). Liveness is again a *may*-property.

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{assign}(y, e), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)}$$

We observe that liveness is indeed a *may*-property, since a variable is live in a conditional if is used in the condition or live in one or more of the branches. Similarly, if a variable is live in the body of a loop, it is live before because the loop body *may* be executed.

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{while}(e, s), x)} \qquad \frac{\mathsf{live}(s, x)}{\mathsf{live}(\mathsf{while}(e, s), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{return}(e), x)} \qquad \text{no rule for } \mathsf{live}(\mathsf{nop}, x) \qquad \frac{\mathsf{live}(x, s) \quad y \neq x}{\mathsf{live}(\mathsf{decl}(y, \tau, s), x)}$$

In some way the most interesting case is a sequence of statements, $\mathsf{seq}(s_1, s_2)$. If a variable is live in s_2 it is only live in the composition if it is not defined in s_1!

$$\frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)} \qquad \frac{\neg\mathsf{def}(s_1, x) \quad \mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)}$$

5 Initialization

Given liveness, we can now say when proper initialization is violated: If a variable is live at the site of its declaration. That means that its value would be used somewhere before it is defined. Assume we have a program p and we write "s in p" if s is a statement appearing in p. The the following rule captures the general condition.

$$\frac{\mathsf{decl}(x, \tau, s) \text{ in } p \quad \mathsf{live}(s, x)}{\mathsf{error}}$$

Unlike the previous rules in the lecture, this one should be read from the premises to the conclusion. In this way it is similar to our rules for liveness from Lecture 4.

This brings out an important distinction when we try to convert the specification rules into an implementation. We have to decide if the rules should be read

from the premises to the conclusion, or from the conclusion to the premises. Sometimes, the same property can be specified in different directions. For example, we can define a predicate init which verifies that all variables are properly initialized and which works from the conclusion to the premises with the following schema.

$$\frac{}{\mathsf{init}(\mathsf{nop})} \qquad \frac{\mathsf{init}(s_1) \quad \mathsf{init}(s_2)}{\mathsf{init}(\mathsf{seq}(s_1, s_2))}$$

$$\frac{\mathsf{init}(s) \quad \neg\mathsf{live}(s, x)}{\mathsf{init}(\mathsf{decl}(x, \tau, s))} \qquad \text{(other rules omitted)}$$

The omitted rules just verify each substatement so that all declarations in the program are checked in the end.

6 From Judgments to Functions

We now focus on the special case that the inference rules are to be read bottom-up. Starting with the judgments we ultimately want to verify, consider $\mathsf{init}(s)$. When we start this, s is known and we are trying to determine if there is a deduction of $\mathsf{init}(s)$ given the rules we have put down. If there is such a deduction, we succeed. If not, we issue an error message. We can model this as a function returning a boolean, or a function returning no interesting value but raising an exception in case there the property is violated.

$$\mathsf{init} : \mathsf{stm} \to \mathsf{bool}$$

Now each of the rules becomes a case in the function definition.

$$\begin{array}{lcl} \mathsf{init}(\mathsf{nop}) & = & \top \\ \mathsf{init}(\mathsf{seq}(s_1, s_2)) & = & \mathsf{init}(s_1) \wedge \mathsf{init}(s_2) \\ \mathsf{init}(\mathsf{decl}(x, \tau, s)) & = & \mathsf{init}(s) \wedge \neg\mathsf{live}(s, x) \\ \ldots & & \end{array}$$

Here we assume a boolean constant $\top$ (for *true*) and boolean operators conjunction $A \wedge B$ and negation $\neg A$ in the functional language; later we might use disjunction $A \vee B$ and falsehood $\bot$. When we call $\mathsf{live}(s, x)$ we assume that it is a similar function.

$$\mathsf{live} : \mathsf{stm} \times \mathsf{var} \to \mathsf{bool}$$

This function is now a transcription of the rules for the live judgment. In this process we sometimes have to combine multiple rules into a single case of the function definition (as, for example, for $\mathsf{seq}(s_1, s_2)$.

$$
\begin{array}{lcl}
\mathsf{live}(\mathsf{nop}, x) & = & \bot \\
\mathsf{live}(\mathsf{seq}(s_1, s_2), x) & = & \mathsf{live}(s_1, x) \vee (\neg \mathsf{def}(s_1, x) \wedge \mathsf{live}(s_2, x)) \\
\mathsf{live}(\mathsf{decl}(y, \tau, s), x) & = & y \neq x \wedge \mathsf{live}(x, s) \\
\ldots
\end{array}
$$

We still have to write functions for predicates $\mathsf{def}(s, x)$ and $\mathsf{use}(e, x)$, but these are a straightforward exercise now.

$$
\begin{array}{lcl}
\mathsf{def} & : & \mathsf{stm} \times \mathsf{var} \to \mathsf{bool} \\
\mathsf{use} & : & \mathsf{exp} \times \mathsf{var} \to \mathsf{bool}
\end{array}
$$

The whole translation was relatively straightforward, primarily because the rules were well-designed, and because we always had enough information to just write a boolean function.

7 Maintaining Set of Variables

What we have done above is a perfectly adequate implementation of initialization checking. But we might also try to rewrite it in order limit the number of traversals of the statements. For example, in

$$
\mathsf{live}(\mathsf{seq}(s_1, s_2), x) \quad = \quad \mathsf{live}(s_1, x) \vee (\neg \mathsf{def}(s_1, x) \wedge \mathsf{live}(s_2, x))
$$

we may traverse s_1 twice: once to check if x is live in s_1, and once to see if x is defined in s_1. In general, we might traverse statements multiple times, namely for each variable declaration in whose scope it lies. This in itself is not a performance bug, but let's see how one might change it.

One way this can often be done is to notice that for any statement s, there could be multiple variables x such that $\mathsf{live}(s, x)$ or $\mathsf{def}(s, x)$ holds. We can try to combine these into a *set*. We denote a set of variables with δ and define the following two judgments:

- $\mathsf{init}(\delta, s, \delta')$: assuming all the variables in δ are defined when s is reached, then after its execution all the variables in δ' will be defined.
- $\mathsf{use}(\delta, e)$: e will only use variables defined in δ.

As a common convention, we isolate assumptions on the left-hand side of a turnstile symbols are write these:

- $\delta \vdash s \Rightarrow \delta'$ for $\mathsf{init}(\delta, s, \delta')$.
- $\delta \vdash e$ for $\mathsf{use}(\delta, e)$.

From the previous rules we develop the following:

$$\frac{}{\delta \vdash \mathsf{nop} \Rightarrow \delta} \qquad \frac{\delta \vdash s_1 \Rightarrow \delta_1 \quad \delta_1 \vdash s_2 \Rightarrow \delta_2}{\delta \vdash \mathsf{seq}(s_1, s_2) \Rightarrow \delta_2}$$

$$\frac{\delta \vdash e}{\delta \vdash \mathsf{assign}(x, e) \Rightarrow \delta \cup \{x\}} \qquad \frac{\delta \vdash e \quad \delta \vdash s_1 \Rightarrow \delta_1 \quad \delta \vdash s_2 \Rightarrow \delta_2}{\delta \vdash \mathsf{if}(e, s_1, s_2) \Rightarrow \delta_1 \cap \delta_2}$$

$$\frac{\delta \vdash e \quad \delta \vdash s \Rightarrow \delta'}{\delta \vdash \mathsf{while}(e, s) \Rightarrow \delta} \qquad \frac{\delta \vdash s \Rightarrow \delta'}{\delta \vdash \mathsf{decl}(y, \tau, s) \Rightarrow \delta' - \{y\}}$$

$$\frac{\delta \vdash e}{\delta \vdash \mathsf{return}(e) \Rightarrow \{x \mid x \text{ in scope}\}}$$

It is worth reading these rules carefully to make sure you understand them. The last one is somewhat problematic, since we don't have enough information to know which rules declarations we are in the scope of. We should generalize our judgment to $\Gamma \ ; \ \delta \vdash s \longrightarrow \delta'$, where Γ is the context containing all variables currently in scope. Usually, we have

$$\Gamma ::= \cdot \mid \Gamma, x{:}\tau$$

Then the last rule might become

$$\frac{\delta \vdash e}{\Gamma \ ; \ \delta \vdash \mathsf{return}(e) \Rightarrow \{x \mid x \in \mathsf{dom}(\Gamma)\}}$$

and we would systematically add Γ to all other judgments. We again leave this as an exercise.

In these judgments we have traded the complexity of traversing statements multiple times with the complexity of maintaining variables sets.

8 Modes of Judgments

If we consider the judgment $\delta \vdash e$ there is nothing new to consider: we would translate this to a function

$$\mathsf{use} : \mathsf{set\ var} \times \mathsf{exp} \to \mathsf{bool}$$

On the other hand, it does not work to translate $\delta \vdash s \Rightarrow \delta'$ as

$$\mathsf{init} : \mathsf{set\ var} \times \mathsf{stm} \times \mathsf{set\ var} \to \mathsf{bool}$$

This is because, in general, we do not know δ' before we start out. We need to *compute* it as part of building the deduction! So we need to implement

$$\mathsf{init} : \mathsf{set\,var} \times \mathsf{stm} \to \mathsf{bool} \times \mathsf{set\,var}$$

In order to handle $\mathsf{return}(e)$, we probably should also pass in a second set of declared variables or a context. We could also avoid returning a boolean by just returning an optional set of defined variables, or raise an exception in case we disover a variable that is used but not defined.

Examining the rules shows that we will need to be able to add variables to and remove variables from sets, as well as compute intersections. Otherwise, the code should be relatively straightforward.

Before we actually start this coding, we should go over the inference rules to make sure we always have enough information to compute the output δ' given the inputs δ and s. This is the purpose of *mode checking*. Let's go over one example:

$$\frac{\delta \vdash s_1 \Rightarrow \delta_1 \quad \delta_1 \vdash s_2 \Rightarrow \delta_2}{\delta \vdash \mathsf{seq}(s_1, s_2) \Rightarrow \delta_2}$$

Initially, we know the input δ and $s = \mathsf{seq}(s_1, s_2)$. This means we also know s_1 and s_2. We cannot yet compute δ_2, since the required input δ_1 in the second premise is unknown. But we can compute δ_1 from the first premise since we know δ and s_1 and this point. This gives us δ_1 and we can now compute δ_2 from the second premise and return it in the conclusion.

$$\mathsf{init}(\delta, \mathsf{seq}(s_1, s_2)) \quad = \quad \mathbf{let}\ \delta_1 = \mathsf{init}(\delta, s_1)\ \mathbf{in}\ \mathsf{init}(\delta_1, s_2)$$

9 Typing Judgments

Arguably the most important judgment on programs is whether they are well-typed. We have already introduced the *context* (or *type environment*) Γ that assigns types to variables. The typing judgment for expressions

$$\Gamma \vdash e : \tau$$

verifies that the expression e is well-typed with type τ, assuming the variables are typed as prescribed by Γ. Most of the rules are straightforward; we show a couple.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1\ \texttt{\&\&}\ e_2 : \mathsf{bool}}$$

Typing for statements is slightly more complex. Statements are executed for their effects, but statements in the body of a functions also ultimately return a value. We write

$$\Gamma \vdash s : [\tau]$$

to express that s is well-typed in context Γ. If s returns (using a $\mathsf{return}(e)$ statement), then e must be of type τ. We use this to check that no matter how a function returns, the returned value is always of the correct type.

$$\frac{\Gamma(x) = \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash \mathsf{assign}(x, e) : [\tau]} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{if}(e, s_1, s_2) : [\tau]}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s : [\tau]}{\Gamma \vdash \mathsf{while(e, s)} : [\tau]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{return}(e) : [\tau]}$$

$$\frac{}{\Gamma \vdash \mathsf{nop} : [\tau]} \qquad \frac{\Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{seq}(s_1, s_2) : [\tau]}$$

$$\frac{\Gamma, x{:}\tau' \vdash s : [\tau]}{\Gamma \vdash \mathsf{decl}(x, \tau', s) : [\tau]}$$

In the last rule for declarations, we might prohibit shadowing of variables by requiring that $x \notin \mathsf{dom}(\Gamma)$. Alternatively, we could stipulate that the rightmost occurrence of x in Γ is the one considered when calculating $\Gamma(x)$. It is also possible that we already know that no conflict can occur, since shadowing may have been ruled out during elaboration already.

10 Modes for Typing

When implementing type-checking, we need to decide on a *mode* for the judgment. Clearly, we want the context Γ and the expression e or statement s to be known, but what about the type?

We first look at expression typing, $\Gamma \vdash e : \tau$. Can we always know τ? Perhaps in our small language fragment from this lecture, but not in L3. For example, if we check an expression $e_1 == e_2 : \mathsf{bool}$, we may know the type boo but we do not know the types of e_1 or e_2 (they could be bool or int). Similarly, if we have an expression used as a statement, we do not know the type of expression. Therefore, we should implement a function that takes the context Γ and e as input and *synthesizes* a type τ such that $\Gamma \vdash e : \tau$ (if such a type exists, and fails otherwise). The resulting type τ can be then be compared to a given type if that is known. Of course, you should go through the rules and verify that one can indeed always synthesize a type.

For the typing of statements $\Gamma \vdash s : [\tau]$ the situation is slightly different. Because τ is the return type of the function in which s occurs, we will know τ instead of having to synthesize is. We say we *check* the statement s against the return type τ.

Therefore, if we assume that functions raise an exception if an expression or statement is not well-typed, we might have functions such as

$$\begin{array}{lcl} \mathsf{syn\_exp} & : & \mathsf{ctx} \times \mathsf{exp} \to \mathsf{tp} \\ \mathsf{chk\_stm} & : & \mathsf{ctx} \times \mathsf{stm} \times \mathsf{tp} \to \mathsf{unit} \end{array}$$

For convenience, we might also write a function

$$\mathsf{chk\_exp} \quad : \quad \mathsf{ctx} \times \mathsf{exp} \times \mathsf{tp} \to \mathsf{unit}$$

where $\mathsf{chk\_exp}(e, \tau)$ would simply synthesize a type τ' for e and compare it to τ.

Questions

1. Write out the rules for proper returns: along each control flow path starting at the beginning of a function, there must be a return statement. Clearly, this is a *must*-property.
2. Write some cases in the functions for type-checking.

Lecture Notes on Dynamic Semantics

15-411: Compiler Design
Frank Pfenning

Lecture 13
October 8, 2013

1 Introduction

In the previous lecture we have specified the *static semantics* of a small imperative language. In this lecture we proceed to discuss its *dynamic semantics*, that is, how programs execute. The relationship between the dynamic semantics for a language and what a compiler actually implements is usually much less direct than for the static semantics. That's because a compiler doesn't actually run the program. Instead, it translates it from some source language to a target language, and then the program in the target language is actually executed.

In our context, the purpose of writing down the dynamic semantics is therefore primarily to precisely specify how programs are supposed to execute. Just the exercise of writing this down formally should help us think about the special cases and make sure our implementation is correct.

Another important purpose is to verify properties of the language itself in a formal (mathematical) way. Much of the theory of programming languages is concerned with just that and therefore requires an operational semantics. A third purpose is to actually *prove* that a compiler is correct. That requires at least two operational specifications: one for the source language and one for the target language. To date, this still requires a major research effort (and is, in any case, out of the scope of this course).

2 Evaluating Expressions

When trying to specify the operational semantics of a programming language, there are a bewildering array of choices regarding the *style* of presentation. Some choices are natural semantics, structural operational semantics, abstract machines,

substructural operational semantics, and many more. Having developed *substructural operational semantics* (SSOS) myself, I have a natural bias towards that style of specification. It has the great virtue that in many cases one can extend the language with new constructs without having to rewrite the rules already in place. However, it requires some machinery, namely *substructural logic*, which is a little more extensive than what I would like to introduce in this course. So instead I am using *natural semantics*, despite some of its shortcomings.

In *natural semantics*, which is a form of so-called *big-step operational semantics*, we relate an expression e directly to its value v. So the basic judgment might be written $\mathsf{eval}(e, v)$. While accurate, this can be a bit lengthy, so we write $e \downarrow v$ instead. Here, e are expression in our (elaborated) abstract syntax, and v are 32 bit integers, interpreted in two's complement representation.

We begin with straightforward rule. The rules of natural semantics are intended to be read bottom-up, from the conclusion to the premise.

$$\frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2 \quad v = v_1 + v_2 \ (\mathsf{mod}\, 2^{32})}{e_1 \mathbin{+} e_2 \downarrow v}$$

We read this as follows:

> *To evaluate e_1 + e_2 we evaluate e_1 to some value v_1, then e_2 to some value v_2 and return the sum $v_1 + v_2$ in arithmetic modulo 2^{32}.*

We can continue along this line, be we get stuck for variables. Where do their values come from? We need to add an *environment* η that maps variables to their values. We write

$$\eta ::= \cdot \mid \eta, x \mapsto v$$

and $\eta[x \mapsto v]$ for either adding $x \mapsto v$ to η or overwriting the current value of x by v (if $\eta(x)$ is already defined). The rule above now carries along η, and the case of a variable looks it up.

$$\frac{\eta \vdash e_1 \downarrow v_1 \quad \eta \vdash e_2 \downarrow v_2 \quad v = v_1 + v_2 \ (\mathsf{mod}\, 2^{32})}{\eta \vdash e_1 \mathbin{+} e_2 \downarrow v} \qquad \frac{\eta(x) = v}{\eta \vdash x \downarrow v}$$

The next problem is posed by operations that may raise an exception. We write $\eta \vdash e \uparrow \mathit{exn}$ if e raises the exception exn. We use a only a few predefined exceptions, and the language provides no way to handle such exceptions, greatly simplifying its semantics. We obtain four rules to specify the behavior of division. We write

$\mathsf{trunc}(x)$ for truncation of x towards 0.

$$\frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2 \quad \begin{array}{c} -2^{31} \le v_1/v_2 < 2^{31} \\ v = \mathsf{trunc}(v_1/v_2) \end{array}}{e_1 \mathbin{/} e_2 \downarrow v} \qquad \frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2 \quad \begin{array}{c} v_2 = 0 \\ \text{or } (v_1 = -2^{31} \text{ and } v_2 = -1) \end{array}}{e_1 \mathbin{/} e_2 \uparrow \mathsf{arith}}$$

$$\frac{e_1 \uparrow exn}{e_1 \mathbin{/} e_2 \uparrow exn} \qquad \frac{e_1 \downarrow v_1 \quad e_2 \uparrow exn}{e_1 \mathbin{/} e_2 \uparrow exn}$$

The last two rules follow from the general convention that we perform left-to-right evaluation of subexpressions. This leads to an unfortunate proliferation of rules. We follow the convention of annotating a rule with $^{(+LR)}$ to indicate that we have omitted additional versions of the rule which can be obtained by replacing premises that converge ($\downarrow v$) by premises that raise an exception ($\uparrow exn$), from left to right, and propagating the exception in the conclusion.

The remaining kind of expressions are fairly straightforward, but we have to remember that some boolean operators shirtcircuit evaluation. We also fix the interpretation of true as 0 and false as 1.

$$\frac{}{\eta \vdash \mathsf{false} \downarrow 0} \qquad \frac{}{\eta \vdash \mathsf{true} \downarrow 1}$$

$$\frac{\eta \vdash e_1 \downarrow 0}{\eta \vdash e_1 \mathbin{\&\&} e_2 \downarrow 0} \qquad \frac{\eta \vdash e_1 \downarrow 1 \quad \eta \vdash e_2 \downarrow v_2}{\eta \vdash e_1 \mathbin{\&\&} e_2 \downarrow v_2}\ {}^{(+LR)}$$

The $^{(+LR)}$ annotation on the second rule means that the following two rules are implied.

$$\frac{\eta \vdash e_1 \uparrow exn}{\eta \vdash e_1 \mathbin{\&\&} e_2 \uparrow exn} \qquad \frac{\eta \vdash e_1 \downarrow 1 \quad \eta \vdash e_2 \uparrow exn}{\eta \vdash e_1 \mathbin{\&\&} e_2 \uparrow exn}$$

3 Relating Static and Dynamic Semantics

The judgments in the static and dynamic semantics are designed to be closely related. We will not prove any of these relationships, but they might help us consider the correctness and completeness of our rules. Here are some of the relationships for expressions.

$$\begin{array}{ccccc} \Gamma & \vdash & e & : & \tau \\ : & & & & : \\ \eta & \vdash & e & \downarrow & v \end{array}$$

This picture expresses that the environment η should match the context Γ and that, furthermore, v should have type τ. We say that η *matches* Γ ($\eta : \Gamma$) if for every

declaration $\Gamma, x{:}\tau$ we have a definition $\eta(x) = v$ with $v : \tau$. The typing for values here is a bit degenerate, but it should stipulate, for example, that only $0 : \mathsf{bool}$ and $1 : \mathsf{bool}$. Note that values are typed without an environment because they are just 32 bit words and cannot contain variables.

The above relationship does not quite hold in our semantics, because not all variables in Γ may have been initialized. But we will have checked statically that

$$\delta \vdash e$$

where $\delta \subseteq \mathsf{dom}(\Gamma)$. So we can refine the above by restricting Γ to the defined variables in δ.

$$\begin{array}{ccccc} \delta & \vdash & e & & \\ \Gamma|_\delta & \vdash & e & : & \tau \\ : & & & & : \\ \eta & \vdash & e & \downarrow & v \end{array}$$

Going back to the earlier rules, we can see the significance of these relationships. For example, we see that in the rules for variables, we can never encounter an uninitialized variable. In the rules for logical conjunction, we see that the two cases for the value of e_1 in e_1 `&&` e_2, namely 0 and 1, capture all possibilities because the value v_1 such that $\eta \vdash e_1 \downarrow v_1$ must be of type `bool`.

4 Executing Statements

Executing statements in L3, the fragment of C0 we have considered so far, can either complete normally, return from the current function with a `return` statement, or raise an exception.

- $\eta \vdash s \to \eta'$: executing s in environment e completes normally with environment η'.
- $\eta \vdash s \downarrow [v]$: executing s in environment η does not complete normally, but instead returns value v.
- $\eta \vdash s \uparrow exn$: executing s in environment η raises exception exn.

We start with some simple cases:

$$\frac{}{\eta \vdash \mathsf{nop} \to \eta} \qquad \frac{\eta \vdash s_1 \to \eta_1 \quad \eta_1 \vdash s_2 \longrightarrow \eta_2}{\eta \vdash \mathsf{seq}(s_1, s_2) \to \eta_2}\ (+LR)$$

The second rule highlights that sequences of statements are executed left to right. We extend our convention regarding the propogation of exception, where any exception by s_1 is propagated, and an exception in s_2 is propagated if s_1 completes normally. So the annotation $^{(+LR)}$ implies the following two rules:

$$\frac{\eta \vdash s_1 \uparrow exn}{\eta \vdash \mathsf{seq}(s_1, s_2) \uparrow exn} \qquad \frac{\eta \vdash s_1 \to \eta_1 \quad \eta_1 \vdash s_2 \uparrow exn}{\eta \vdash \mathsf{seq}(s_1, s_2) \uparrow exn}$$

Because s_1 or s_2 may also execute a return statement, we also need the following additional rules:

$$\frac{\eta \vdash s_1 \downarrow [v]}{\eta \vdash \mathsf{seq}(s_1, s_2) \downarrow [v]} \qquad \frac{\eta \vdash s_1 \to \eta_1 \quad \eta_1 \vdash s_2 \downarrow [v]}{\eta \vdash \mathsf{seq}(s_1, s_2) \downarrow [v]}$$

How do these judgments line up with our static semantics?

$$\begin{array}{ccccc} \Gamma|_\delta & \vdash & s & : & [\tau] \\ \delta & \vdash & s & \Rightarrow & \delta' \\ : & & & & : \\ \eta & \vdash & s & \to & \eta' \end{array}$$

The diagram is trying to express that if η matches $\Gamma|_\delta$ then η' matches $\Gamma|_{\delta'}$. In other words, if s completes normally then it will define exactly those variables that the static semantics claimed it must, namely those in δ'. Moreover, all the values have the right type.

For return values we have a related diagram:

$$\begin{array}{ccccc} \delta & \vdash & s & \Rightarrow & \delta' \\ \Gamma|_\delta & \vdash & s & : & [\tau] \\ : & & & & : \\ \eta & \vdash & s & \downarrow & [v] \end{array}$$

That is, if s returns a value v, then that must have the type τ. The rule on the right clearly should satisfy this.

$$\frac{\eta \vdash e \downarrow v}{\eta \vdash \mathsf{return}(e) \downarrow [v]} \qquad \frac{\eta \vdash e \uparrow exn}{\eta \vdash \mathsf{return}(e) \uparrow exn}$$

Assignment straightforwardly updates the environment, propagating exceptions.

$$\frac{\eta \vdash e \downarrow v}{\eta \vdash \mathsf{assign}(x, e) \to \eta[x \mapsto v]}\ {}^{(+LR)}$$

For conditionals, we evaluate only the relevant branch.

$$\frac{\eta \vdash e \downarrow 1 \quad \eta \vdash s_1 \to \eta'}{\eta \vdash \mathsf{if}(e, s_1, s_2) \to \eta'}\ (+LR) \qquad \frac{\eta \vdash e \downarrow 1 \quad \eta \vdash s_1 \downarrow [v]}{\eta \vdash \mathsf{if}(e, s_1, s_2) \downarrow [v]}$$

$$\frac{\eta \vdash e \downarrow 0 \quad \eta \vdash s_2 \to \eta'}{\eta \vdash \mathsf{if}(e, s_1, s_2) \to \eta'}\ (+LR) \qquad \frac{\eta \vdash e \downarrow 0 \quad \eta \vdash s_2 \downarrow [v]}{\eta \vdash \mathsf{if}(e, s_1, s_2) \downarrow [v]}$$

Loops are somewhat more interesting. If the loop guard is false, we exit the loop. If it is true, we execute the loop body once (obtaining a new environment η') and then repeat *in the new environment* η'.

$$\frac{\eta \vdash e \downarrow 0}{\eta \vdash \mathsf{while}(e, s) \to \eta}\ (+LR) \qquad \frac{\eta \vdash e \downarrow 1 \quad \eta \vdash s \to \eta' \quad \eta' \vdash \mathsf{while}(e, s) \to \eta''}{\eta \vdash \mathsf{while}(e, s) \to \eta''}\ (+LR)$$

We omit the additional, obvious rules for dealing with possible returns.

Loops bring up the question of nontermination. Natural semantics is not particularly well-suited to reflect on nontermination. If, say, an expression e does not terminate in environment η, we cannot find any value v such that $\eta \vdash e \downarrow v$ can be proved, nor is there an exception exn such that $\eta \vdash e \uparrow exn$. But nontermination is a bit stronger than that, because, intuitively, we can always continue with our proof construction but never complete it. In the case of the while loop, the third premise of the second rule would again apply the same rule, with again the same while loop in the third premise, and so on without ever completing.

Declarations are not particularly difficult; we just have to be careful to track the scopes of variables correctly during elaboration so that there are no surprises during execution.

$$\frac{\eta \vdash s \to \eta'}{\eta \vdash \mathsf{decl}(x, \tau, s) \to \eta' \setminus [x \mapsto \_]}\ (+LR)$$

Here we just remove whatever definition might have been given to x during the execution of s.

5 Function Calls

Finally, for this lecture, we come to another connection between statements and expressions: *function calls*. We stack th premises on top of each other so the rule

doesn't become too wide.

$$\frac{\begin{array}{l}\eta \vdash e_1 \downarrow v_1 \\ \cdots \\ \eta \vdash e_n \downarrow v_n \\ f(x_1, \ldots, x_n) = s \\ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \vdash s \downarrow [v]\end{array}}{\eta \vdash f(e_1, \ldots, e_n) \downarrow v} \; (+LR)$$

Here, η is entirely ignored in the body of f (called s), because s only has access to the function parameters $x_1, \ldots, x_n$. If all goes well, we know that s must raise an exception or return a value, it cannot complete normally. That's because we have checked in the static semantics that there is a return statement along each control flow path through s. We can provide an additional version of this rule in case we have a function not returning a value (void), or we can elaborate void into return of a distinguished value, say, 0.

Recall that according to our convention, the function call raises an exception if e_1 does, or if e_1 returns a value and e_2 raises an exception, etc. Finally, any exception from s is passed on.

Lecture Notes on Mutable Store

15-411: Compiler Design
Frank Pfenning

Lecture 14
October 10, 2013

1 Introduction

In this lecture we extend our language with the ability to allocate data structures on the so-called *heap*. Addresses of heap elements serve as pointers which can be dereferenced to read stored values, or used as destinations for write operations. Similarly, arrays are stored on the heap[1] and via appropriate address calculations.

Adding mutable store requires yet again a significant change in the structure of the rules of the dynamic semantics. By contrast, the static semantics is relatively easy to extend.

2 Pointers

We extend our language of types with $\tau*$, where τ is a type.

$$\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \alpha \mid \tau*$$

In the language of expressions, we can allocate a cell on the heap that can hold a value of type τ, we have a distinguished null pointer, and we can dereference a pointer to obtain the stored value.

$$e ::= \ldots \mid \mathsf{alloc}(\tau) \mid *e \mid \mathsf{null}$$

They have the following typing rules:

$$\frac{}{\Gamma \vdash \mathsf{alloc}(\tau) : \tau*} \qquad \frac{\Gamma \vdash e : \tau*}{\Gamma \vdash *e : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \tau*}$$

At first glance they might be harmless, but the third rule should raise a red flag: we previously claimed in our mode analysis of typing, that given Γ and e we can synthesize the type of e (if it exists). However, in the rule for null that's not the case.

[1] C0 does not have stack-allocated arrays

3 Detail: Typing $*\mathsf{null}$

We cannot synthesize a *definite* type for null. Unfortunately, we also cannot, in general, know what type to check an expression against. So we'll synthesize an indefinite type, let's call it $any*$, the type of a pointer to data of potentially any type.

Now we have to walk through all the constructs in the language to see whether we can resolve $any*$, assuming it can only arise for null. Let's consider *pointer equality* first, that is, an expression p == q where p and q are pointers. If p and q both have definite type $\tau*$, we just treat it as well-typed. If one has type $\tau*$ and the other $\tau'*$ for $\tau \neq \tau'$, we reject the comparison as ill-typed. If one is definite $\tau*$ and the other indefinite, we allow the comparison, because the indefinite type has only one value (null) which can be compared to a pointer of any definite type. If both are indefinite, we would be comparing null with null, which is also fine.

One way to capture this is to have a so-called *type subsumption rule* that allows a "silent" transition:

$$\frac{\Gamma \vdash e : any*}{\Gamma \vdash e : \tau*}$$

Then three rules suffice for our overloaded equality:[2]

$$\frac{\Gamma \vdash e_1 : \tau* \quad \Gamma \vdash e_2 : \tau*}{\Gamma \vdash e_1 \texttt{ == } e_2 : \mathsf{bool}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \texttt{ == } e_2 : \mathsf{bool}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \texttt{ == } e_2 : \mathsf{bool}}$$

A difficulty arises with the dereferencing operator: $*\mathsf{null}$ would have *any* type, which means it could essentially appear anywhere. Of course, when run, it will always yield an exception, since dereferencing the null pointer is disallowed. We therefore rewrite our earlier rule to disallow dereferencing values of indefinite type.

$$\frac{\Gamma \vdash e : \tau* \quad \Gamma \not\vdash e : any*}{\Gamma \vdash *e : \tau}$$

In particular $*\mathsf{null}$ is disallowed, and so is $*(b \texttt{ ? null : null})$ and variants thereof, because the conditional still has indefinite type $any*$. Of course, indefinite types are not part of the source language and only used internally during type checking.

4 Dynamic Semantics for Pointers

A value of type $\tau*$ is just an address where a value of type τ is stored, or the special address 0. Allocation returns an unused address, and dereferencing the pointer retrieves the stored value. But where is the store? We currently only carry an

[2] Actually, in this language fragment just one would suffice, since elements of all types can be compared for equality.

environment η that maps variables to their values. We now also carry a *heap* H that maps addresses to stored values.

Evaluation of expressions *may change the heap*, because it may call a function that changes its state. So expression evaluation now looks like:[3]

$$H \,;\, \eta \vdash e \downarrow v \,;\, H'$$

Here the semicolon ';' is just a separator between the heap and the environment on the left and the value and the new heap on the right. We read it as

> *Given heap H and environment η, evaluation of e returns value v in the new heap state H'.*

When we raise an exception, we do not need to carry a new heap H' since C0 has no mechanism for catching an exception.

$$H \,;\, \eta \vdash e \uparrow \mathit{exn}$$

All the prior rules now carry thread through the heap, always following left-to-right evaluation order. For example,

$$\frac{H \,;\, \eta \vdash e_1 \downarrow v_1 \,;\, H' \quad H' \,;\, \eta \vdash e_2 \downarrow v_2 \,;\, H'' \quad v = v_1 + v_2 \ (\mathsf{mod}\ 2^{32})}{H \,;\, \eta \vdash e_1 \mathbin{+} e_2 \downarrow v \,;\, H''} \ (+\uparrow)$$

The new rules for pointers should be not particularly surprising. We write a for addresses, in our architecture a 64-bit word. Allocation returns a fresh address a and maps it to an appropriate default value in a new heap.

$$\frac{[a, a + |\tau|) \cap \mathsf{dom}(H) = \{\}, a \neq 0}{H \,;\, \eta \vdash \mathsf{alloc}(\tau) \downarrow a \,;\, H[a \mapsto \mathsf{default}(\tau)]} \ (+\uparrow)$$

Freshly allocated cells are initialized with a default value for the type τ. In the implementation, this is arranged to always be 0 (in whatever word length required by the size of τ). For booleans this means `false`, for integers 0 and for pointers `null` in the source language.

For the implementation of this rule, we need to know the sizes of each type. This is, of course, highly dependent on the processor architecture and conventions. For this course, we compile to x86-64, in which case we have:

$$\begin{array}{lcl} |\mathsf{int}| & = & 4 \\ |\mathsf{bool}| & = & 4 \\ |\tau*| & = & 8 \\ |\tau[]| & = & 8 \end{array}$$

[3] This is a slight departure from lecture, where we tried to combine the heap H and the environment η into a single memory M. However, certain operations like function calls are difficult to describe in that formulation.

Dereferencing a pointer just retrieves from the address, assuming it is not 0. If it is 0, we raise the memory exception mem, which is actually signal `SIGSEGV` (11) on our architecture.

$$\frac{H\ ;\ \eta \vdash e \downarrow a\ ;\ H' \quad a \neq 0, H(a) = v}{H\ ;\ \eta \vdash *e \downarrow v\ ;\ H'}\ (+\uparrow) \qquad \frac{H\ ;\ \eta \vdash e \downarrow 0\ ;\ H'}{H\ ;\ \eta \vdash *e \uparrow \mathsf{mem}}$$

The null pointer of course just evaluates to 0.

$$\frac{}{H\ ;\ \eta \vdash \mathsf{null} \downarrow 0}$$

On our architecture, an attempt to access the memory location with address 0 will raise the appropriate memory exception for us.

This leaves us with a puzzle: how do we *write* to memory? In C0 (and C) this is accomplished via assignments where the left-hand side dictates the destination of the write operation. These are sometimes called *l-values*, where *l* stands for *left-hand side*.

5 Writing to Heap Destinations

We define *destinations* (or *l-values*)

$$d ::= x \mid *d$$

Adding arrays and structs will add more kinds of destinations. Every kind of destination is also a valid expression, so we can just destinations as expressions.

$$\frac{\Gamma \vdash d : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{assign}(d, e)}$$

In the operational semantics we now distinguish variables from other destinations, since variables are on the stack (or in registers), while destinations $*d$ are on the heap.

For this we need to fix the new judgments describing how statements are executed. In analogy with the judgments for evaluating expressions, we have

$H\ ;\ \eta \vdash s \rightarrow \eta'\ ;\ H'$	s completes normally with new env. η' and heap H'
$H\ ;\ \eta \vdash s \downarrow [v]\ ;\ H'$	s invokes return with value v
$H\ ;\ \eta \vdash s \uparrow exn$	s raises exception exn

For assignment, we obtain:

$$\frac{H\ ;\ \eta \vdash e \downarrow v\ ;\ H'}{H\ ;\ \eta \vdash \mathsf{assign}(x, e) \rightarrow \eta[x \mapsto v]\ ;\ H'}\ (+\uparrow)$$

$$\frac{H\ ;\ \eta \vdash d \downarrow a\ ;\ H' \quad H'\ ;\ \eta \vdash e \downarrow v\ ;\ H'' \quad a \neq 0}{H\ ;\ \eta \vdash \mathsf{assign}(*d, e) \rightarrow \eta\ ;\ H'[a \mapsto v]}\ (+\uparrow)$$

$$\frac{H\ ;\ \eta \vdash d \downarrow a\ ;\ H' \quad H'\ ;\ \eta \vdash e \downarrow v\ ;\ H'' \quad a = 0}{H\ ;\ \eta \vdash \mathsf{assign}(*d, e) \uparrow \mathsf{mem}}$$

Detail: Evaluating Assignments

Based on the rules above, what should happen in the following code fragments.

```
int* p = NULL;
*p = 1/0;
```

First we define p to be 0. Then we evaluate the assignment from left to right. This means we first evaluate p to 0. Second we evaluate $1/0$. This will raise an arith exception, which is therefore the outcome of the execution.

```
int** p = NULL;
**p = 1/0;
```

First we define p to be 0. Then we evaluate the assignment from left to right. This means we first evaluation $*p$. Since the value of p is 0 this raises a mem exception, which is therefore the outcome of the execution

6 Arrays

Arrays are in many ways similar to pointers, but there are no null arrays. We'll discuss default arrays below. For now, though, this is a simplification since the typing rules are more straightforward.

$$\begin{array}{lcl} \tau & ::= & \ldots \mid \tau[] \\ e & ::= & \ldots \mid \mathsf{alloc\_array}(\tau, e) \mid e_1[e_2] \\ d & ::= & \ldots \mid d[e] \end{array}$$

$$\frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash \mathsf{alloc\_array}(\tau, e) : \tau[]} \qquad \frac{\Gamma \vdash e_1 : \tau[] \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1[e_2] : \tau}$$

The dynamic semantics for allocation obtains a fresh segment of memory and initializes all n elements of the array with the default value of type τ.

$$\frac{\begin{array}{l} H\ ;\ \eta \vdash e \downarrow n\ ;\ H' \\ n \geq 0 \\ [a, a + n|\tau|) \cap \mathsf{dom}(H') = \{\,\}, a \neq 0 \\ H'' = H'[a + 0|\tau| \mapsto \mathsf{default}(\tau), \ldots, a + (n-1)|\tau| \mapsto \mathsf{default}(\tau)] \end{array}}{H\ ;\ \eta \vdash \mathsf{alloc\_array}(\tau, e) \downarrow a\ ;\ H''} \ (+\uparrow)$$

Array access evaluates from left to right and then computes the correct memory address for the value.

$$\frac{\begin{array}{l} H\ ;\ \eta \vdash e_1 \downarrow a\ ;\ H' \\ H'\ ;\ \eta \vdash e_2 \downarrow i\ ;\ H'' \\ a \neq 0, 0 \leq i < \mathsf{length}(a), e_1 : \tau[] \\ v = H''(a + i|\tau|) \end{array}}{H\ ;\ \eta \vdash e_1[e_2] \downarrow v\ ;\ H''} \ (+\uparrow)$$

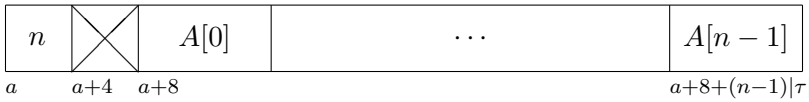
There are two significant complications here: where do we obtain the length of the array stored at address a, and where do we get the type τ?

The second is actually easier: when we compile an array access, we will know the type of e_1. It must be of the form $\tau[]$ for some τ. Then we calculate its size *at compile time* and generate code to multiply it by the index i.

Finding the length of the array is actually harder, since it not known at compile time. This is because array allocation has the form $\mathsf{alloc\_array}(e)$ where e is an arbitrary expression that should evaluate to the number of elements in the array to allocate.

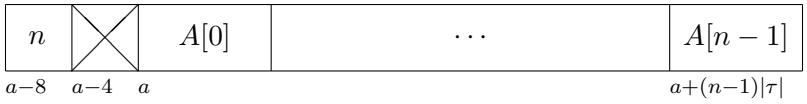
Detail: Storing the Array Length

One possibility is to allocate a few additional bytes to store the length of the array. This could be layed out as follows, where a is the address of the array A with elements of type τ.



Alternatively, we could lay it out with the address a pointing to the first array element. This simplifies the address arithmetic, and would also allow passing this pointer directly to C (which would not care about the length information to the

left).



The reason we locate the length n at $a - 8$ and not $a - 4$ is so that a itself will be aligned at 0 modulo 8, if the whole memory block as returned from calloc is aligned that way.

Under this second regime, the code pattern for $e_1[e_2]$ with $e_1 : \tau[]$ and $|\tau| = k$ could be like this:

cogen(e_1, a)	(a new)
cogen(e_2, i)	(i new)
$a_1 \leftarrow a - 8$	
$t_2 \leftarrow M[a_1]$	
if $(i < 0)$ goto error	
if $(i \geq t_2)$ goto error	
$a_3 \leftarrow i * \$k$	
$a_4 \leftarrow a + a_3$	
$t_5 \leftarrow M[a_4]$	

Here, a, a_1, a_3, a_4 would be 64 bit temps, t_2 would be 32 bits, and t_5 would be k bytes. We have written $\$k$ to indicate that this is an immediate operand (that is, a compile-time constant). Some compound memory operands can be used on x86-64 to avoid some intermediate computation such as a_1 or a_4. Also, we can exploit properties of two's complement arithmetic and combine the two comparisons into a single unsigned comparison of i and t_2.

Of course, there are still limits to interoperability with C: if C passes an array to a C0 program, we somehow need to find out its length and marshal it somewhere else so we can add the length information. Alternatively, we can compile the code in *unsafe* mode where array bounds are not checked, which is just what C does.

Executing assignments with the new destinations is quite similar to reading.

$$\frac{\begin{array}{l} H\ ;\ \eta \vdash d_1 \downarrow a\ ;\ H' \\ H'\ ;\ \eta \vdash e_2 \downarrow i\ ;\ H'' \\ a \neq 0, 0 \leq i < \mathsf{length}(a), d_1 : \tau[] \\ a' = a + i|\tau| \\ H''\ ;\ \eta \vdash e_3 \downarrow v\ ;\ H''' \end{array}}{H\ ;\ \eta \vdash d_1[e_2] = e_3 \rightarrow \eta\ ;\ H'''[a' \mapsto v]}\ (+\uparrow)$$

If the bounds check is not satisfied, a memory exception is raised.

$$\frac{\begin{array}{l}H\ ;\ \eta \vdash d_1 \downarrow a\ ;\ H' \\ H'\ ;\ \eta \vdash e_2 \downarrow i\ ;\ H'' \\ a = 0 \text{ or } i < 0 \text{ or } i \geq \mathsf{length}(a)\end{array}}{H\ ;\ \eta \vdash d_1[e_2] = e_3 \uparrow \mathsf{mem}}$$

7 Detail: Default Values of Array Type

Each type has a default value. For integers it is 0, for booleans 0 (which represents false), and for pointers it is 0 (which represents null). The default for arrays is also 0, which represents an array of size 0. We can never legally access any element of this default array, since the condition that the index must be in bounds can never be satisfied. Nevertheless, arrays can be compared for equality and disequality (which is a comparison of their address), so zero-sized arrays are not entirely useless. In particular, $\mathsf{alloc\_array}(0)$ must return a fresh zero-sized array that's different from all other arrays already allocated, and also different from the default array of size 0.

The fact that $a = 0$ is a valid array address creates an issue when we try to access $M[a - 8]$ to obtain its size. We could rely on the operating system to raise a mem exception, although that may not be reliably so. To be sure, we should check whether a is 0 before doing address calculation. Of course, if we are in *unsafe* mode when bounds-checking is turned off (which we will implement in Lab 5), then this is not necessary.

8 Detail: Compound Assignment Operators

Previously, we could expand `x += e` to `x = x + e`. However, with the addition of arrays, this has become problematic. The difficulty is `d1[e2] += e3`. After syntactic expansion we obtain `d1[e2] = d1[e2] + e3` in which both d_1 and e_2 would be evaluated twice. Since evaluation of expressions and destinations now can have an effect, that effect would be unexpectedly repeated.

Instead we have to more-or-less repeat the rules for assignment. For example:

$$\frac{\begin{array}{l}H\ ;\ \eta \vdash d_1 \downarrow a\ ;\ H' \\ H'\ ;\ \eta \vdash e_2 \downarrow i\ ;\ H'' \\ a \neq 0, 0 \leq i < \mathsf{length}(a), d_1 : \tau[] \\ a' = a + i|\tau| \\ H''\ ;\ \eta \vdash e_3 \downarrow v\ ;\ H''' \\ v' = H'''[a'] \oplus v\end{array}}{H\ ;\ \eta \vdash d_1[e_2] \oplus= e_3 \to \eta\ ;\ H'''[a' \mapsto v']}\ (+\uparrow)$$

This should be modified in a systematic way to handle out-of-bounds array access and a possible effect of the binary operation as may happen for division, modulus, or shift operation.

Lecture Notes on Structs

15-411: Compiler Design
Frank Pfenning

Lecture 15
October 15, 2013

1 Introduction

Pointers allow access to data stored in the heap. Arrays allow us to aggregate data of the same type. Structs provides means to aggregate data of different types. This creates few additional challenges in the C0 language definition and also in its implementation (and, of course, the language fragment L4 used in this course).

2 Struct Declarations and Definitions

C0 (and L4) support a subset of the struct-related constructs in C. Structs may be *declared* with

$$\mathsf{struct}\ s;$$

or they can be *defined* by specifying the fields $f_1, \ldots, f_n$ of the struct with their types

$$\mathsf{struct}\ s\ \{\tau_1\ f_1; \ldots \tau_n\ f_n;\};$$

We will elaborate this into a form where, for typing purposes, we know $s.f_i : \tau_i$. For compilation purposes we also compute $\mathsf{offset}(s, f_i)$; see remarks later in this lecture.

Because structs might require an arbitrary amount of memory, we stipulate that they can never be held in variables, but must be allocated on the heap. To specify this concisely we distinguish *small types* from *large types*. Values of small type fit in registers, while values of large type must be on the heap. In L4, we have

- small types int, bool, $\tau*$, $\tau[]$, and
- large types struct s

We have the following significant restrictions on types:

- Local variables, function parameters, and return values must have small type.
- Left- and right-hand sides of assignments must have small type.
- Conditional expressions must have small type.
- Equality and disequality must compare expressions of small type.
- Expressions used as statements must have small type.

There are some scoping requirements imposed on structs, but they are surprisingly lenient. The reason is that undefined structs provide a very weak form of polymorphism. For example, we can pass values of type struct s $*$ as pointers without needing to know how struct s is defined, as long as we don't attempt to access its fields. The following static semantic rules apply:

1. Field names occupy their own name space, so they cannot clash with variable, function, or type names (but they must be distinct from keywords). The same field names can be used in different struct definitions.
2. In a given struct definition, all field names must be distinct.
3. A struct may be defined at most once.
4. Types struct s that have not (yet) been defined may be referenced as long as their size is irrelevant. The size of a struct is *relevant* in expressions alloc(struct s), alloc_array(struct s, e), and in struct definitions when serving as the type of a field.
5. An occurrence of struct s in a context where its size is irrelevant serves as an *implicit declaration* of the type struct s. In effect this means that explicit struct declarations are optional (but encouraged as good style).

3 Expressions and Typing

The extension of the language of expressions and destinations is surprisingly economical.

$$\begin{array}{rcl} e & ::= & \ldots \mid e.f \\ d & ::= & \ldots \mid d.f \end{array}$$

We also define (typically during elaboration):

$$e{\rightarrow}f \equiv (*e).f$$

which can also be used as a destination in the form $d{\rightarrow}f$.

$$\frac{\Gamma \vdash e : \mathsf{struct}\ s \quad s.f : \tau}{\Gamma \vdash e.f : \tau}$$

For this rule to apply, struct s must have been defined. It is not sufficient for it to have just been declared, because we could not determine the type of field f.

Because destinations are also expressions, no additional typing rules are needed for destinations. But recall from the restrictions in Section 2 that prior rules are severely restricted by allowing only small types.

4 Dynamic Semantics

As might be suspected, the dynamic semantics for structs is more difficult. This is because we write programs as if structs would fit into variables; in reality we are mostly manipulating their addresses. For example, under the definition

```
struct point {
  int x;
  int y;
};
```

and after

```
struct point* p = alloc(struct point);
```

the expression $(*p).y$ should evaluate to 0. But what is the value of $*p$? We cannot just dereference the p, since that just holds the *address* of the beginning of the struct stored in memory. Instead we have to use this address itself and then compute the offset of the field y (4, under the x86-64 ABI we are using), counting from the beginning of the struct, add that to p, and then retrieve the integer stored in that position.

So, in this context (when the expression has a large type), we evaluate $*e$, essentially just taking the value of e but not dereferencing it. This is quite similar to what we have to do when $*d$ appears as an l-value on the left-hand side of an assignment. To unify these, we introduce a new judgment

$$H\ ;\ \eta \vdash e \Downarrow a\ ;\ H' \quad e \textit{ denotes address } a$$

which we only really need if e has large type. We use this to evaluate $e.f$ and also $d.f$ below.

$$\frac{\begin{array}{l} e : \mathsf{struct}\ s \\ \mathsf{small}(s.f) \\ H\ ;\ \eta \vdash e \Downarrow a\ ;\ H' \\ a \neq 0 \end{array}}{H\ ;\ \eta \vdash e.f \downarrow H'(a + \mathsf{offset}(s, f))\ ;\ H'}\ (+\uparrow)$$

The judgment itself is defined by only three rules, because these are the only possibilities for expressions of large type: a field dereference, a pointer dereference, or an array access.

$$\frac{\begin{array}{l} e : \mathsf{struct}\ s \\ \mathsf{large}(s.f) \\ H\ ;\ \eta \vdash e \Downarrow a\ ;\ H' \\ a \neq 0 \end{array}}{H\ ;\ \eta \vdash e.f \Downarrow a + \mathsf{offset}(s, f)\ ;\ H'}\ (+\uparrow)$$

$$\frac{H\ ;\ \eta \vdash e \downarrow a\ ;\ H'}{H\ ;\ \eta \vdash *e \Downarrow a\ ;\ H'}\ (+\uparrow) \qquad \frac{\begin{array}{l} H\ ;\ \eta \vdash e_1 \downarrow a\ ;\ H' \\ H'\ ;\ \eta \vdash e_2 \downarrow i \\ a \neq 0, 0 \leq i < \mathsf{length}(a) \\ e_1 : \tau[] \end{array}}{H\ ;\ \eta \vdash e_1[e_2] \Downarrow a + i|\tau|\ ;\ H'}\ (+\uparrow)$$

Similarly, on the left-hand side of an assignment, we get

$$\frac{\begin{array}{l} d : \mathsf{struct}\ s \\ H\ ;\ \eta \vdash d.f \Downarrow a\ ;\ H' \\ H'\ ;\ \eta \vdash e \downarrow v\ ;\ H'' \end{array}}{H\ ;\ \eta \vdash \mathsf{assign}(d.f, e) \rightarrow \eta'; H''[a \mapsto v]}\ (+\uparrow)$$

We could revisit the earlier rules for assignment as well, perhaps simplify them.

When structs are allocated in memory, all the fields are initialized with their default values. As mentioned in the previous lecture, this just means filling the memory with 0, which is what the C library function `calloc` does.

5 Dealing with Different Data Sizes

In L2 and L3 we only had integers and booleans, but in L4 we have data of different sizes. For small types, we have the following table:

L4 type		size in bytes	C type
$\lvert\mathsf{int}\rvert$	$=$	4	`int`
$\lvert\mathsf{bool}\rvert$	$=$	4	`int`
$\lvert\tau*\rvert$	$=$	8	`t *`
$\lvert\tau[]\rvert$	$=$	8	`t *`
$\lvert\mathsf{struct}\ s\rvert$	$=$	$\mathsf{size}(s)$	`struct s`

Note that we have decided to represent L3 booleans as integers in C, rather than as members of the type `bool` (defined as an alias to `_Bool`). This is because booleans

in C, according to the x86-64 ABI, have width 1 byte and do not need to be aligned.[1] Actually, the introduction of type `bool` to C seems relatively recent, so just using type `int` to represent truth values is not inconsistent with the C philosophy. In full C0 we decided on representing C0 booleans as C booleans, since we also have characters of width 1 byte and therefore cannot avoid dealing with data of size 1.

The size of a struct type is computed by laying out the structs in memory from left to right, inserting padding to make sure that each field is properly aligned. Each integer and boolean must be aligned at 0 modulo 4, each pointer or array reference must be aligned at 0 modulo 8, and each enclosed struct must be aligned according to its most stringent field requirement. Furthermore, we add padding at the end so that the whole struct has a size which is 0 modulo its most stringent field requirement. This is so arrays can be laid out simply by knowing the size of its type. The C library function `calloc` should always return a pointer that is 0 modulo 8 and therefore appropriate for any struct we might want to allocate.

6 Detail: Register Sizes

Dealing with data of different sizes will likely require maintaining additional information in your compiler so you can pick the right load/store and register movement instructions (`movl` vs. `movq`), the right comparisons (`cmpl` vs. `cmpq`), reserve the appropriate amount of stack space, allocate the appropriate amount of heap space, and do correct address calculations.

The good news is that in L3 and L4, registers only need to hold 4 byte or 8 byte values. Still, it is very easy to introduce bugs when you do not explicitly mediate changes in data size. For example, for the intermediate form we recommend disallowing instructions of the form

$$d^{64} \leftarrow s^{32}$$

where s and d are registers of the indicated sizes, but writing one of

$$\begin{array}{lcl} d^{64} & \leftarrow & \mathsf{zeroextend}\ s^{32} \\ d^{64} & \leftarrow & \mathsf{signextend}\ s^{32} \end{array}$$

and similarly for truncations in the other directions. This should ensure that you do not accidentally apply incorrect transformations, like copy propagation, if the destination and source of a "move" have different sizes.

On the x86-64 architecture, both move and arithmetic instructions that target a 32-bit register have the peculiar effect of zero-extending the value into the whole 64-bit register. For example,

[1] This created some significant complications in writing the compiler for L3 that we wanted to avoid.

```
MOVL %EAX, %EAX
```

has an effect: it replaces bits 32–63 of `%RAX` by 0. Similarly,

```
XORL %EAX, %EAX
```

will set all 64 bits of `%RAX` to 0, not just the lowest 32. For more on this we recommend Bryant and O'Hallaron's note on x86-64 Machine-Level Programming, in particular the information on Page 9.

Lecture Notes on Basic Optimizations

15-411: Compiler Design
Frank Pfenning

Lecture 16
Oct 17, 2013

1 Introduction

The opportunities for optimizations[1] in compiler-generated code are plentiful. Generally speaking, they arise more from the tensions between the high-level source language and the lower-level target language, rather than any intrinsic inefficiencies in the source. One common source, sometimes estimated to constitute as much as 70% of optimization opportunities, is address arithmetic and is therefore tied to structs and arrays.

In this lecture we discuss basic optimizations that apply pervasively during the compilation process. In the next two lectures we will discuss specifically optimizations of loops. Another class of optimizations is concerned with functions calls, like tail-call optimization and inlining. You have the opportunity to consider these in Assignment 3.

2 Dead Code Elimination

Optimizations have two components: (1) a condition under which they are application and the (2) code transformation itself. The applicability condition can come in various forms, but often requires a dataflow analysis.

As a warm-up exercise, we reconsider dead code elimination from Section 4 of Lecture 5. We defined there a predicate $\mathsf{needed}(l, x)$ which is defined via a backward dataflow analysis. Instructions of the forms

$$\begin{array}{lcl} l & : & x \leftarrow s_1 \oplus s_2 \\ l & : & x \leftarrow s \end{array}$$

[1] Very little in a compiler is actually optimal, so "optimizations" should be interpreted as "efficiency improvements".

with a pure (effect-free) right-hand side are considered dead code if x is not needed in the successor $l + 1$ of l.

We can describe the transformation by stating how the given lines are affected, and which new lines should perhaps be added. We use the notation

$$\frac{P_1 \dots P_m}{L_1 \dots L_n}$$

which replaces $P_1 \dots P_m$ with $L_1 \dots L_n$. Any condition of applicability is also listed among the premises. This is an example of *linear inference*, where the process of inference consumes the premises and adds the conclusions.[2] In this notation, dead code elimination could be described as

$$\frac{l : x \leftarrow s_1 \oplus s_2 \quad \mathsf{succ}(l, l') \quad \neg\mathsf{needed}(l', x)}{l : \mathsf{nop}}$$

$$\frac{l : x \leftarrow s \quad \mathsf{succ}(l, l') \quad \neg\mathsf{needed}(l', x)}{l : \mathsf{nop}}$$

where $\oplus$ is an effect-free operation. We replace the instruction with a nop instead of just deleting it so that, for example, jumps to line l will continue to remain value. At a later stage of optimization, spurious no-ops can be deleted from the code.

3 Constant Propagation

Another straightforward optimization is *constant propagation*. If we have definition $l : x \leftarrow c$ for a constant c, we might want to replace an occurrence of x by c in the hope that we may be able to eliminate the assignment (and x) altogether. Moreover, we may be able to apply other optimizations where we have substituted c for x, such as constant folding.

The tricky question is when this is a correct optimization. For example, in the code

$$\begin{array}{lcl} l & : & x \leftarrow c \\ & & \dots \\ k & : & y \leftarrow x + 1 \end{array}$$

it depends on what happens in the lines between l and k. Jumps may target lines in between, or there may be another assignment to x so that x no longer has the value c when execution reaches k.

[2] Linear inference gives rise to a relatively new kind of logic called *linear logic* which we do not discuss further in the class. Some materials on linear inference and linear logic can be found at http://www.cs.cmu.edu/~fp/courses/15816-s12/

Rather than give a general solution to this question, we greatly simplify our lives by assuming that the code has been transformed in static single-assignment (SSA) form. In that form, any variable is defined exactly one so a reference to x must be the correct one. We use the notation $l' : instr(x)$ for an instruction that *uses* x (that is, uses(l', x)) and $instr(c)$ for the result of replacing x by c.

$$\frac{l : x \leftarrow c \qquad l' : instr(x)}{l : x \leftarrow c \qquad l' : instr(c)}$$

Note that we need to repeat $l : x \leftarrow c$ in the conclusion since the premise is consumed when the inference rule is applied.

4 Copy Propagation

Copy propagation is very similar to constant propagation, except that one variable is defined in terms of another.

$$\frac{l : x \leftarrow y \qquad l' : instr(x)}{l : x \leftarrow y \qquad l' : instr(y)}$$

Again, we should ask if this is sound, assuming the program is in SSA form. We know there is exactly one definition of y that is available at line l. Since x is available at line l', y must also be available there so the replacement is sound.

5 Termination

When applying code transformations, we should always consider if the transformations terminate. Clearly, each step of dead code elimination reduces the number of assignments in the code. We can therefore apply it arbitrarily until we reach *quiescence*, that is, neither of the dead code elimination rules is applicable any more. Quiescence is the linear inference counterpart to saturation for ordinary inference, as we have discussed in prior lectures. Saturation means that any inference we might apply only has conclusions that are already known. Quiescence means that we can no longer apply any linear inference.

A single application of constant propagation reduces the number of variable occurrence in the program and must therefore reach quiescence. It also does not increase the number of definitions in the code, and can therefore be mixed freely with dead code elimination.

It is more difficult to see whether copy propagation will always terminate, since the number of variable occurrences stays the same, as does the number of variable

definitions. In fact, in a code pattern

$$\begin{array}{lcl} l & : & x \leftarrow y \\ k & : & w \leftarrow x \\ m & : & \mathit{instr}(w) \\ m' & : & \mathit{instr}(x) \end{array}$$

we could for decrease the number of occurrence of x by copy propagation from line l and then increase it again by copy propagation from line k. However, if we consider a string partial order $x > y$ among variables if the definition of x uses y (transitively closed), then copy progation reduces the occurrence of a variable by a strictly smaller one. This order is well-founded since in SSA we cannot have a cycle among the definitions. If x is defined in terms of y, then y could not be defined in terms of x since it the single definition of y must come before x in the control flow graph.

6 Constant Folding

Constant folding evaluates a constant epxression at compile time. In the three-address form, this is simply:

$$\frac{l : x \leftarrow c_1 \odot c_2 \quad c_1 \odot c_2 = c}{l : x \leftarrow c}$$

where $\oplus$ doubles as a syntactic binary operation and its arithmetic counterpart. We need to make sure that $c_1 \odot c_2$ is defined in this case (and should not raise an exception at runtime). There is really not other precondition to this transformations.

7 Common Subexpression Elimination

It is natural to try to apply a transformation similar to copy or constant propagation to of the form

$$\begin{array}{lcl} l & : & x \leftarrow s_1 \oplus s_2 \\ & & \cdots \\ k & : & \mathit{instr}(x) \end{array}$$

where we replace x by $s_1 \oplus s_2$. However, this will not work most of the time, because the result may not even be a valid instruction (for example, if $\mathit{instr}(x) = (y \leftarrow x \oplus 1)$. Moreover, the program becomes bigger, plus we are computing an expression more than once instead of just once, so this is likely to make the code slower rather than faster.

However, we can consider the *opposite*: In a situation

$$\begin{array}{lcl} l & : & x \leftarrow s_1 \oplus s_2 \\ & & \cdots \\ k & : & y \leftarrow s_1 \oplus s_2 \end{array}$$

we can replace the second computation of $s_1 \oplus s_2$ by a reference to x (under some conditions), saving a reduction computation. This is called *common subexpression elimination* (CSE).

For this to be correct we need to know that x will have the right value when execution reaches line k. Because we are in SSA form, the right-hand sides will always have the same meaning if they are syntactically identical. But will x be available at k?

What we would like to know is that every control flow path from the beginning of the code (that is, the beginning of the function we are compiling) to line k goes through line l. Then we can be sure that x has the right value when we reach k. This is the definition of the *dominance* relation between lines of code. We write $l \geq k$ if l dominates k and $l > k$ if it l strictly dominated k. We see how to define it in the next section; once it is defined we use it as follows:

$$\begin{array}{l} l : x \leftarrow s_1 \oplus s_2 \\ k : y \leftarrow s_1 \oplus s_2 \\ l > k \\ \hline l : x \leftarrow s_1 \oplus s_2 \\ k : y \leftarrow x \end{array}$$

8 Dominance

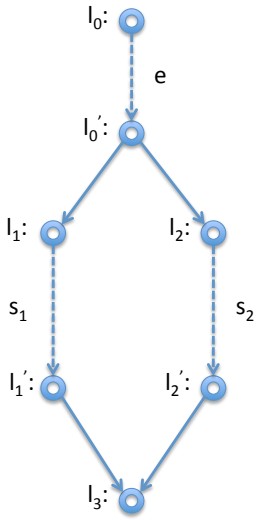
On general control flow graphs, dominance is an interesting relation and there are several algorithms for computing this relationship. We can cast it as a form of *forward data-flow analysis*. One of the approaches exploits the simplicity of our language to directly generate the dominance relationship as part of code generation. We briefly discuss this here. The drawback is that if your code generation is slightly different or more efficient, or if your transformation change the essential structure of the control flow graph, then you need to update the relationship. A simple and fast algorithm that works particularly well in our simple language is described by Cooper et al. [CHK06] which is empirically faster than the traditional Lengauer-Tarjan algorithm [LT79] (which is asymptotically faster). In this lecture, we consider just the basic cases.

For *straight-line code* the predecessor if each line is its immediate dominator, and any preceding line is a dominator.

For conditionals, consider

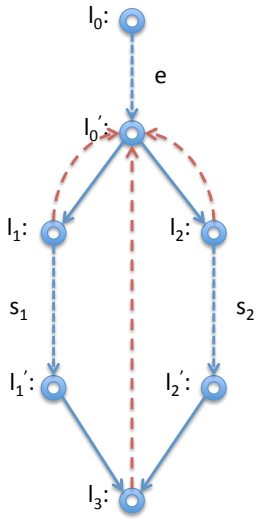
$$\mathsf{if}(e, s_1, s_2)$$

We translate this to the following code, $\check{e}$ or $\check{s}$ is the code for e and s, respectively and $\hat{e}$ is the temp through which we can refer to the result of evaluating e.

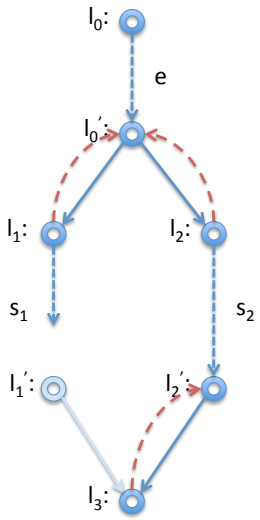


l_0 : $\check{e}$
l'_0 : if $(\hat{e}$!= $0)$ goto l_1 ; goto l_2
l_1 : $\check{s}_1$; l'_1 : goto l_3
l_2 : $\check{s}_2$; l'_2 : goto l_3
l_3 :

On the right is the corresponding control-flow graph. Now the immediate dominator of l_1 should be l'_0 and the immediate dominator of l_2 should also be l'_0. Now for l_3 we don't know if we arrive from l'_1 or from l'_2. Therefore, neither of these nodes will dominate l_3. Instead, the immediate dominator is l'_0, the last node we can be sure to be traversed before we arrive at l'_3. Indicating immediate dominators with dashed read lines, we show the result below.

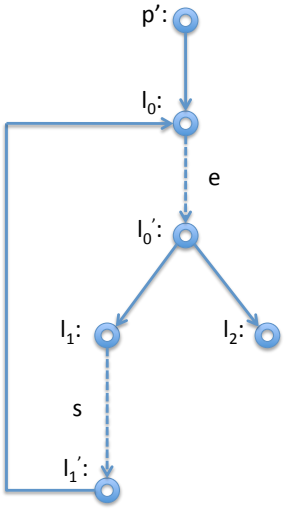


However, if it turns out, say, l_1' is not reachable, then the dominator relationship looks different. This is the case, for example, if s_1 in this example is a return statement or is known to raise an error. Then we have instead:



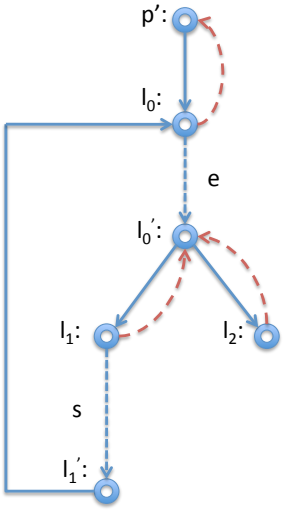
In this case, l_1' : goto l_3 is *unreachable* code and can be optimized away. Of course, the case where l_2' is unreachable is symmetric.

For loops, it is pretty easy to see that the beginning of the loop dominates all the statements in the loop. Again, considering the straightforward compilation of a while loop with the control flow graph on the right.



l_0 : $\check{e}$
l_0' : if $(\hat{e} == 0)$ goto l_2 ; goto l_1
l_1 : $\check{s}$
l_1' : goto l_0
l_2 :

Interesting here is mainly that the node p' just before the loop header l_0 is indeed the immediate dominator of l_0, even l_0 has l_1' as another predecessor. The definition makes this obvious: when we enter the loop we have to come through p' node, on subsequent iterations we come from l_1'. So we cannot be guaranteed to come through l_1', but we are guaranteed to come through p' on our way to l_0.



9 Implementing Common Subexpression Elimination

To implement common subexpression elimination we traverse the program, looking for definitions $l : x \leftarrow s_1 \odot s_2$. If $s_1 \odot s_2$ is already in the table, defining variable y at k, we replace l with $l : x \leftarrow y$ if k dominates l. Otherwise, we add the expression, line, and variable to the hash table.

Dominance can usually be checked quite quickly if we maintain a dominator tree, where each line has a pointer to its immediate dominator. We just follow these pointers until we either reach k (and so $k > l$) or the root of the control-flow graph (in which case k does not dominate l).

10 Strength Reduction

Strength reduction in general replaces and expensive operation with a simpler one. Sometimes it can also eliminate an operation altogether, based on the laws of modular, two's complement arithmetic. Recall that we have the usual laws of arithmetic modulo 2^{32} for addition, subtraction, multiplication, but that comparisons are more

difficult to transform[3]

Common simplifications (and some symmetric counterparts):

$$\begin{aligned} a + 0 &= a \\ a - 0 &= a \\ a * 0 &= 0 \\ a * 1 &= a \end{aligned}$$

but one can easily think of others involving further arithmetic of bit-level operations. One that may be interesting for optimization of array accesses is the distributive law:

$$a * (b + c) = a * b + a * c$$

where a could be the size of an array element and $(b + c)$ could be an index calculation.

11 A Simple Example

Let's consider the rather innocuous C0 code fragment

```
A[i] = A[i] + 1
```

Assuming we perform no null or array bound checking, and a holds the address of the array, we would obtain something like the following. The semantics of C0 require left-to-right evaluation, so we first obtain the address of $A[i]$ in t_1 (lines $l_0 - -l_1$), then we evaluate the right-hand-side (lines $l_2 - -l_5$), and then we write to the memory at address t_3 (line l_6). The number 4 is the size of $|\mathsf{int}|$, which is the type of the array elements.

$$\begin{array}{lcll} l_0 & : & t_0 \leftarrow \mathbf{4 * i} & \text{\# cse} \\ l_1 & : & t_1 \leftarrow a + t_0 & \\ l_2 & : & t_2 \leftarrow \mathbf{4 * i} & \text{\# cse} \\ l_3 & : & t_3 \leftarrow a + t_2 & \\ l_4 & : & t_4 \leftarrow M[t_3] & \\ l_5 & : & t_5 \leftarrow t_4 + 1 & \\ l_6 & : & M[t_1] \leftarrow t_5 & \end{array}$$

[3]For example, $x + 1 > x$ is false in general, because x could be the maximal integer, $2^{31} - 1$.

We notice that l_0 and l_2 both compute $4 * i$ so we obtain the code on the left. This is now subject to copy propagation from l_2 to l_3 to obtain the code on the right.

$$
\begin{array}{lcll}
l_0 & : & t_0 & \leftarrow 4 * i \\
l_1 & : & t_1 & \leftarrow a + t_0 \\
l_2 & : & \mathbf{t_2} & \leftarrow \mathbf{t_0} \\
l_3 & : & t_3 & \leftarrow a + \mathbf{t_2} \\
l_4 & : & t_4 & \leftarrow M[t_3] \\
l_5 & : & t_5 & \leftarrow t_4 + 1 \\
l_6 & : & M[t_1] & \leftarrow t_5
\end{array}
\qquad
\begin{array}{lcll}
l_0 & : & t_0 & \leftarrow 4 * i \\
l_1 & : & t_1 & \leftarrow \mathbf{a + t_0} \quad \text{\# cse} \\
l_2 & : & t_2 & \leftarrow t_0 \\
l_3 & : & t_3 & \leftarrow \mathbf{a + t_0} \quad \text{\# cse} \\
l_4 & : & t_4 & \leftarrow M[t_3] \\
l_5 & : & t_5 & \leftarrow t_4 + 1 \\
l_6 & : & M[t_1] & \leftarrow t_5
\end{array}
$$

The code on the right yields another opportunity for common subexpressions elimination for lines l_1 and l_3. The result is pictured on the left, followed again by copy propagation on the right.

$$
\begin{array}{lcll}
l_0 & : & t_0 & \leftarrow 4 * i \\
l_1 & : & t_1 & \leftarrow a + t_0 \\
l_2 & : & t_2 & \leftarrow t_0 \\
l_3 & : & \mathbf{t_3} & \leftarrow \mathbf{t_1} \\
l_4 & : & t_4 & \leftarrow M[\mathbf{t_3}] \\
l_5 & : & t_5 & \leftarrow t_4 + 1 \\
l_6 & : & M[t_1] & \leftarrow t_5
\end{array}
\qquad
\begin{array}{lcll}
l_0 & : & t_0 & \leftarrow 4 * i \\
l_1 & : & t_1 & \leftarrow a + t_0 \\
l_2 & : & \mathbf{t_2} & \leftarrow t_0 \quad \text{\# dead} \\
l_3 & : & \mathbf{t_3} & \leftarrow t_1 \quad \text{\# dead} \\
l_4 & : & t_4 & \leftarrow M[\mathbf{t_1}] \\
l_5 & : & t_5 & \leftarrow t_4 + 1 \\
l_6 & : & M[t_1] & \leftarrow t_5
\end{array}
$$

A pass of dead code elimination yields the code in which the address of $A[i]$ is computed only once.

$$
\begin{array}{lcll}
l_0 & : & t_0 & \leftarrow 4 * i \\
l_1 & : & t_1 & \leftarrow a + t_0 \\
l_2 & : & \mathsf{nop} & \\
l_3 & : & \mathsf{nop} & \\
l_4 & : & t_4 & \leftarrow M[t_1] \\
l_5 & : & t_5 & \leftarrow t_4 + 1 \\
l_6 & : & M[t_1] & \leftarrow t_5
\end{array}
$$

This example illustrates the *cascading* of optimizations: initially, we only had two common subexpressions, but after some optimizations more were uncovered. Techniques such as global value numbering help to avoid multiple passes over code by combining several iterations into one. Neededness analysis is another example where multiple lines are declared dead code at once, rather than in sequence with new analysis in between.

References

[CHK06] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. Technical Report TR-06-33870, Department of Computer Science, Rice University, 2006.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):115–120, July 1979.

Lecture Notes on Loop Optimizations

15-411: Compiler Design
Frank Pfenning

Lecture 17
October 22, 2013

1 Introduction

Optimizing loops is particularly important in compilation, since loops (and in particular the inner loops) account for much of the executions times of many programs. Since tail-recursive functions are usually also turned into loops, the importance of loop optimizations is further magnified. In this lecture we will discuss two main ones: hoisting loop-invariant computation out of a loop, and optimizations based on induction variables.

2 What Is a Loop?

Before we discuss loop optimizations, we should discuss what we identify as a loop. In our source language, this is rather straightforward, since loops are formed with while or for, where it is convenient to just elaborate a for loop into its corresponding while form.

The key to a loop is a back edge in the control-flow graph from a node l to a node h that dominates l. We call h the *header node* of the loop. The loop itself then consists of the nodes on a path from h to l. It is convenient to organize the code so that a loop can be identified with its header node. We then write $\mathsf{loop}(h, l)$ if line l is in the loop with header h.

When loops are nested, we generally optimize the inner loops before the outer loops. For one, inner loops are likely to be executed more often. For another, it could move computation to an outer loop from which it is hoisted further when the outer loop is optimized and so on.

3 Hoisting Loop-Invariant Computation

An (pure) expression is *loop invariant* if its value does not change throughout the loop. We can then define the predicate $\mathsf{inv}(h, p)$, where p is a pure expression, as follows:

$$\frac{c \text{ constant}}{\mathsf{inv}(h, c)} \qquad \frac{\mathsf{def}(l, x) \quad \neg\mathsf{loop}(h, l)}{\mathsf{inv}(h, x)} \qquad \frac{\mathsf{inv}(h, s_1) \quad \mathsf{inv}(h, s_2)}{\mathsf{inv}(h, s_1 \oplus s_2)}$$

Since we are concerned only with programs in SSA form, it is easy to see that variables are loop invariant if they are not parameters of the header label. However, the definition above does not quite capture this for definitions $t \leftarrow p$ where p is loop-invariant but t is not part of the label parameters. So we add a second propagation rule.

$$\frac{l : t \leftarrow p \quad \mathsf{inv}(h, p) \quad \mathsf{loop}(h, l)}{\mathsf{inv}(h, t)}$$

Note that we do not consider memory references of function calls to be loop invariant, although under some additional conditions they may be hoisted as well.

In order to hoist loop invariant computations out of a loop we should have a *loop preheader* in the control-flow graph, which immediately dominates the loop header. When then move all the loop invariant computations to the preheader, in order.

Some care must be taken with this optimization. For example, when the loop body is never executed the code could become significantly slower. Another problem if we have conditionals in the body of the loop: values computed only on one branch or the other will be loop invariant, but depending on the boolean condition one or the other may never be executed.

In some cases, when the loop guard is inexpensive and effect-free but the loop-invariant code is expensive, we might consider duplicating the test so that instead of

$$\mathsf{seq}(pre, \mathsf{while}(e, s))$$

we generate code for

$$\mathsf{seq}(\mathsf{if}(e, \mathsf{seq}(pre, \mathsf{while}(e, s)), \mathsf{nop}))$$

where pre is the hoisted computation in the loop pre-header.

A typical example of hoisting loop invariant computation would be a loop to initialize all elements of a two-dimensional array:

```
for (int i = 0; i < width * height; i++)
  A[i] = 1;
```

We show the relevant part of the abstract assembly on the left. In the right is the result of hoisting the multiplication, enabled because both $width$ and $height$ are loop invariant and therefore their product is.

```
    i0 ← 0                          i0 ← 0
                                    t ← width * height
    goto loop(i0)                   goto loop(i0)
loop(i1) :                      loop(i1) :
    t ← width * height
    if (i1 ≥ t) goto exit           if (i1 ≥ t) goto exit
    ...                             ...
    i2 ← i1 + 1                     i2 ← i1 + 1
    goto loop(i2)                   goto loop(i2)
exit :                          exit :
```

4 Induction Variables

Hoisting loop invariant computation is significant; optimizing computation which changes by a constant amount each time around the loop is probably even more important. We call such variables *basic induction variables*. The opportunity for optimization arises from *derived induction variables*, that is, variables that are computed from basic induction variables.

As an example we will use a function check if a given array is sorted in ascending order.

```
bool is_sorted(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
{
  for (int i = 0; i < n-1; i++)
    //@loop_invariant 0 <= i && i <= n-1;
    if (A[i] > A[i+1]) return false;
  return true;
}
```

Below is a possible compiled SSA version of this code, assuming that we do not

perform array bounds checks (or have eliminated them).

```
is_sorted(A, n) :
    i_0 ← 0
    goto loop(i_0)
loop(i_1) :
    t_0 ← n − 1
    if (i_1 ≥ t_0) goto rtrue
    t_1 ← 4 ∗ i_1
    t_2 ← A + t_1
    t_3 ← M[t_2]
    t_4 ← i_1 + 1
    t_5 ← 4 ∗ t_4
    t_6 ← A + t_5
    t_7 ← M[t_6]
    if (t_3 > t_7) goto rfalse
    i_2 ← i_1 + 1
    goto loop(i_2)
rtrue :
    return 1
rfalse :
    return 0
```

Here, i_1 is the basic induction variable, and $t_1 = 4 * i_1$ and $t_4 = i_1 + 1$ are the derived induction variables. In general, we consider a variable a derived induction variable if its has the form $a * i + b$, where a and b are loop invariant.

Let's consider t_4 first. We see that common subexpression elimination applies. However, we would like to preserve the basic induction variable i_1 and its version

i_2, so we apply code motion and then eliminate the second occurrence of $i_1 + 1$.

```
is_sorted(A, n) :                 is_sorted(A, n) :
    i_0 ← 0                           i_0 ← 0
    goto loop(i_0)                    goto loop(i_0)
loop(i_1) :                       loop(i_1) :
    t_0 ← n − 1                       t_0 ← n − 1
    if (i_1 ≥ t_0) goto rtrue         if (i_1 ≥ t_0) goto rtrue
    t_1 ← 4 ∗ i_1                     t_1 ← 4 ∗ i_1
    t_2 ← A + t_1                     t_2 ← A + t_1
    t_3 ← M[t_2]                      t_3 ← M[t_2]
                                      i_2 ← i_1 + 1
    t_4 ← i_1 + 1                     t_4 ← i_2
    t_5 ← 4 ∗ t_4                     t_5 ← 4 ∗ t_4
    t_6 ← A + t_5                     t_6 ← A + t_5
    t_7 ← M[t_6]                      t_7 ← M[t_6]
    if (t_3 > t_7) goto rfalse        if (t_3 > t_7) goto rfalse
    i_2 ← i_1 + 1
    goto loop(i_2)                    goto loop(i_2)
```

Next we look at the derived induction variable $t_1 \leftarrow 4 * i_1$. The idea is to see how we can calculate t_1 at a subsequent iteration from t_1 at a prior iteration. In order to achieve this effect, we add a new induction variable to represent $4 * i_1$. We call this j and add it to our loop variables in SSA form.

```
is_sorted(A, n) :
    i_0 ← 0
    j_0 ← 4 ∗ i_0                 @ensures j_0 = 4 ∗ i_0
    goto loop(i_0, j_0)
loop(i_1, j_1) :                  @requires j_1 = 4 ∗ i_1
    t_0 ← n − 1
    if (i_1 ≥ t_0) goto rtrue
    t_1 ← j_1                     @assert j_1 = 4 ∗ i_1
    t_2 ← A + t_1
    t_3 ← M[t_2]
    i_2 ← i_1 + 1
    j_2 ← 4 ∗ i_2                 @ensures j_2 = 4 ∗ i_2
    t_4 ← i_2
    t_5 ← 4 ∗ t_4
    t_6 ← A + t_5
    t_7 ← M[t_6]
    if (t_3 > t_7) goto rfalse
    goto loop(i_2, j_2)
```

Crucial here is the invariant that $j_1 = 4 * i_1$ when label $\mathsf{loop}(i_1, j_1)$ is reached. Now we calculate

$$j_2 = 4 * i_2 = 4 * (i_1 + 1) = 4 * i_1 + 4 = j_1 + 4$$

so we can express j_2 in terms of j_1 without multiplication. This is an example of *strength reduction* since addition is faster than multiplication. Recall that all the laws we used are valid for modular arithmetic. Similarly:

$$j_0 = 4 * i_0 = 0$$

since $i_0 = 0$, which is an example of constant propagation followed by constant folding.

```
is_sorted(A, n) :
    i_0 ← 0
    j_0 ← 0                    @ensures j_0 = 4 * i_0
    goto loop(i_0, j_0)
loop(i_1, j_1) :               @requires j_1 = 4 * i_1
    t_0 ← n − 1
    if (i_1 ≥ t_0) goto rtrue
    t_1 ← j_1                  @assert j_1 = 4 * i_1
    t_2 ← A + t_1
    t_3 ← M[t_2]
    i_2 ← i_1 + 1
    j_2 ← j_1 + 4              @ensures j_2 = 4 * i_2
    t_4 ← i_2
    t_5 ← 4 * t_4
    t_6 ← A + t_5
    t_7 ← M[t_6]
    if (t_3 > t_7) goto rfalse
    goto loop(i_2, j_2)
```

With some copy propagation, and noticing that $n-1$ is loop invariant, we next get:

```
is_sorted(A, n) :
    i_0 ← 0
    j_0 ← 0                      @ensures j_0 = 4 * i_0
    t_0 ← n − 1
    goto loop(i_0, j_0)
loop(i_1, j_1) :                 @requires j_1 = 4 * i_1
    if (i_1 ≥ t_0) goto rtrue
    t_2 ← A + j_1
    t_3 ← M[t_2]
    i_2 ← i_1 + 1
    j_2 ← j_1 + 4                @ensures j_2 = 4 * i_2
    t_5 ← 4 * i_2
    t_6 ← A + t_5
    t_7 ← M[t_6]
    if (t_3 > t_7) goto rfalse
    goto loop(i_2, j_2)
```

With common subexpression elimination (noting the additional assertions we are aware of), we can replace $4 * i_2$ by j_2. We combine this with copy propagation.

```
is_sorted(A, n) :
    i_0 ← 0
    j_0 ← 0                      @ensures j_0 = 4 * i_0
    t_0 ← n − 1
    goto loop(i_0, j_0)
loop(i_1, j_1) :                 @requires j_1 = 4 * i_1
    if (i_1 ≥ t_0) goto rtrue
    t_2 ← A + j_1
    t_3 ← M[t_2]
    i_2 ← i_1 + 1
    j_2 ← j_1 + 4                @ensures j_2 = 4 * i_2
    t_6 ← A + j_2
    t_7 ← M[t_6]
    if (t_3 > t_7) goto rfalse
    goto loop(i_2, j_2)
```

We observe another derived induction variable, namely $t_2 = A + j_1$. We give this a new name ($k_1 = A + j_1$) and introduce it into our function. Again we just calculate:

$k_2 = A + j_2 = A + j_1 + 4 = k_1 + 4$ and $k_0 = A + j_0 = A$.

```
is_sorted(A, n) :
    i_0 ← 0
    j_0 ← 0                    @ensures j_0 = 4 * i_0
    k_0 ← A + j_0              @ensures k_0 = A + j_0
    t_0 ← n − 1
    goto loop(i_0, j_0, k_0)
loop(i_1, j_1, k_1) :          @requires j_1 = 4 * i_1 ∧ k_1 = A + j_1
    if (i_1 ≥ t_0) goto rtrue
    t_2 ← k_1
    t_3 ← M[t_2]
    i_2 ← i_1 + 1
    j_2 ← j_1 + 4              @ensures j_2 = 4 * i_2
    k_2 ← k_1 + 4              @ensures k_2 = A + j_2
    t_6 ← A + j_2
    t_7 ← M[t_6]
    if (t_3 > t_7) goto rfalse
    goto loop(i_2, j_2, k_2)
```

After more round of constant propagtion, common subexpression elimination, and dead code elimination we get:

```
is_sorted(A, n) :
    i_0 ← 0
    j_0 ← 0                    @ensures j_0 = 4 * i_0
    k_0 ← A                    @ensures k_0 = A + j_0
    t_0 ← n − 1
    goto loop(i_0, j_0, k_0)
loop(i_1, j_1, k_1) :          @requires j_1 = 4 * i_1 ∧ k_1 = A + j_1
    if (i_1 ≥ t_0) goto rtrue
    t_3 ← M[k_1]
    i_2 ← i_1 + 1
    j_2 ← j_1 + 4              @ensures j_2 = 4 * i_2
    k_2 ← k_1 + 4              @ensures k_2 = A + j_2
    t_7 ← M[k_2]
    if (t_3 > t_7) goto rfalse
    goto loop(i_2, j_2, k_2)
```

With neededness analysis we can say that j_0, j_1, and j_2 are no longer needed and

can be eliminated.

```
is_sorted(A, n) :
    i_0 ← 0
    k_0 ← A                      @ensures k_0 = A + 4 * i_0
    t_0 ← n − 1
    goto loop(i_0, k_0)
loop(i_1, k_1) :                 @requires k_1 = A + 4 * i_1
    if (i_1 ≥ t_0) goto rtrue
    t_3 ← M[k_1]
    i_2 ← i_1 + 1
    k_2 ← k_1 + 4                @ensures k_2 = A + 4 * i_2
    t_7 ← M[k_2]
    if (t_3 > t_7) goto rfalse
    goto loop(i_2, k_2)
```

Unfortunately, i_1 is still needed, since it governs a conditional jump. In order to eliminate that we would have to observe that

$$i_1 \geq t_0 \text{ iff } A + 4 * i_1 \geq A + 4 * t_0$$

This holds the the addition here is a on 64 bit quantities where the second operand is 32 bits, so no overflow can occur. If we exploit this we obtain:

```
is_sorted(A, n) :
    i_0 ← 0
    k_0 ← A                      @ensures k_0 = A + 4 * i_0
    t_0 ← n − 1
    goto loop(i_0, k_0)
loop(i_1, k_1) :                 @requires k_1 = A + 4 * i_1
    if (k_1 ≥ A + 4 * t_0) goto rtrue
    t_3 ← M[k_1]
    i_2 ← i_1 + 1
    k_2 ← k_1 + 4                @ensures k_2 = A + 4 * i_2
    t_7 ← M[k_2]
    if (t_3 > t_7) goto rfalse
    goto loop(i_2, k_2)
```

Now i_0, i_1, and i_2 are no longer needed and can be eliminated. Moreover, $A + 4 * t_0$

is loop invariant and can be hoisted.

```
is_sorted(A, n) :
    k_0 ← A
    t_0 ← n − 1
    t_8 ← 4 ∗ t_0
    t_9 ← A + t_8
    goto loop(k_0)
loop(k_1) :
    if (k_1 ≥ t_9) goto rtrue
    t_3 ← M[k_1]
    k_2 ← k_1 + 4
    t_7 ← M[k_2]
    if (t_3 > t_7) goto rfalse
    goto loop(k_2)
rtrue :
    return 1
rfalse :
    return 0
```

It was suggested that we can avoid two memory accesses per iteration by unrolling the loop once. This make sense, but this opimization is beyond the scope of this lecture.

We have carried out the optimizations here on concrete programs and values, but it is straightforward to generalize them to arbitrary induction variables x that are updated with $x_2 \leftarrow x_1 \pm c$ for a constant c, and derived variables that arise from constant multiplication with or addition to a basic induction variable.

Lecture Notes on Memory Optimizations

15-411: Compiler Design
Frank Pfenning

Lecture 19
October 29, 2013

1 Introduction

Even on modern architecures with hierarchical memory caches, memory access, on average, is still significantly more expensive than register access or even most arithmetic operations. Therefore, memory optimizations play a significant role in generating fast code. As we will see, whether certain memory optimizations are possible or not depends on properties of the whole language. For example, whether or not we can obtain pointers to the middle of heap-allocated objects will be a crucial question to answer.

2 A Simple Example

We will use a simple running example to illustrate memory optimization and their conditions of applicability. In this example, $\mathsf{mult}(A, p, q)$ will multiply matrix A with vector p and return the result in vector q.

```
struct point {
  int x;
  int y;
};
typedef struct point pt;

void mult(int[] A, pt* p, pt* q) {
  q->x = A[0] * p->x + A[1] * p->y;
  q->y = A[2] * p->x + A[3] * p->y;
  return;
}
```

Below is the translation into abstract assembly, with the small twist that we have allowed memory reference to be of the form $M[base + offset]$. The memory optimization question we investigate is whether some load instructions $t \leftarrow M[s]$ can be avoided because the corresponding value is already held in a temp.

```
mult(A, p, q) :
    t0 ← M[A + 0]
    t1 ← M[p + 0]
    t2 ← t0 + t1
    t3 ← M[A + 4]
    t4 ← M[p + 4]
    t5 ← t3 * t4
    t6 ← t2 + t5
    M[q + 0] ← t6
    t8 ← M[A + 8]
    t9 ← M[p + 0]      # redundant load?
    t10 ← t8 + t9
    t11 ← M[A + 12]
    t12 ← M[p + 4]     # redundant load?
    t13 ← t11 * t12
    t14 ← t10 + t13
    M[q + 4] ← t14
    return
```

We see that the source refers to `p->x` and `p->y` twice, and those are reflected in the two, potentially redundant loads above. Before you read on, consider if we could replace the lines with $t_9 \leftarrow t_1$ and $t_{12} \leftarrow t_4$. We can do that if we can be assured that memory at the addresses $p + 0$ and $p + 4$, respectively, has not changed since the previous load instructions.

It turns out that in C0 the second load is definitely redundant, but the first one may not be.

The first load is not redundant because when this function is called, the pointers p and q might be the same (they might *aliased*). When this is the case, the store to $M[q+0]$ will likely change the value stored at $M[p+0]$, leading to a different answer than expected for the second line.

On the other hand, this cannot happen for the first line, because $M[q+0]$ could never be the same as $M[p+4]$ since one accesses the x field and the other the y field of a struct.

Of course, the answer is mostly likely wrong when $p = q$. One could either rewrite the code, or require that $p \neq q$ in the precondition to the function.

In C, the question is more delicate because the use of the address-of (`&`) operator could obtain pointers to the middle of objects. For example, the argument `int[] A` would be `int* A` in C, and such a pointer might have been obtained with `&q->x`.

3 Using the Results of Alias Analysis

In C0, the types of pointers are a powerful basis of alias analysis. The way alias analysis is usually phrased is as a *may-alias* analysis, because we try to infer which pointers in a program may alias. Then we know for optimization purposes that if two pointers are not in the *may-alias* relationship that they must be different. Writing to one address cannot change the value stored at the other.

Let's consider how we might use the results of alias analysis, embodied in a predicate $\mathsf{may\text{-}alias}(a, b)$ for two addresses a and b. We assume we have a load instruction

$$l : t \leftarrow M[a]$$

and we want to infer if this is *available* at some other line $l' : t' \leftarrow M[a]$ so we could replace it with $l' : t' \leftarrow t$. Our optimization rule (in the notation of linear inference from Lecture 16):

$$\frac{\begin{array}{l} l : t \leftarrow M[a] \\ l' : t' \leftarrow M[a] \\ l > l' \\ \mathsf{avail}(l, l') \end{array}}{\begin{array}{l} l : t \leftarrow M[a] \\ l' : t' \leftarrow t \end{array}}$$

The fact that l dominates l' is sufficient here in SSA form to guarantee that the meaning of t and a remains unchanged. reaches is supposed to check that $M[a]$ also remains unchanged.

Reaching analysis for memory references is a simple forward dataflow analysis. If we have a node with two or more incoming control flow edges, it must be avail-

able along all of them. For the purposes of traversing loops we assume availability, essentially trying to find a counterexample in the loop. To express this concisely, our analysis rules propagate *unavailability* of a definition $l : t \leftarrow M[a]$ an other instructions l' that are dominated by l.

For unavailability, $\mathsf{unavail}(l, l')$, we have the seeding rule on the left and the general propagation rule on the right. Because we are in SSA, we know in the seeding rule that $l > l''$ where l'' is the (unique) successor of l''.

$$\frac{\begin{array}{l} l : t \leftarrow M[a] \\ l > l' \\ l' : M[b] \leftarrow s \\ \mathsf{may\text{-}alias}(a, b) \\ \mathsf{succ}(l', l'') \end{array}}{\mathsf{unavail}(l, l'')} \qquad \frac{\begin{array}{l} \mathsf{unavail}(l, l') \\ \mathsf{succ}(l', l'') \\ l > l'' \end{array}}{\mathsf{unavail}(l, l'')}$$

The rule on the right includes the cases of jumps or conditional jumps. This ensures that in a node with multiple predecessors, if a value is unavailable in just one of them, in will be unavailable at the node. From this we can deduce which memory values are still available, namely those that are not unavailable (restriction attention to those that are dominated by the load—otherwise the question is not asked).

$$\frac{\begin{array}{l} l : t \leftarrow M[a] \\ l > l' \\ \neg\mathsf{unavail}(l, l') \end{array}}{\mathsf{avail}(l, l')}$$

4 Type-Based Alias Analysis

The simplest form of alias analysis is based on the type and offset of the address. We call this an *alias class*, with the idea that pointers in different alias classes cannot alias. The basic predicate here is $\mathsf{class}(a, \tau, \mathit{offset})$ which expresses that a is an address derived from a source of type τ and offset offset from the start of the memory of type τ.

Then the *may-alias* relation is defined by

$$\frac{\mathsf{class}(a, \tau, k) \quad \mathsf{class}(b, \tau, k)}{\mathsf{may\text{-}alias}(a, b)}$$

There is a couple of special cases we do not treat explicitly. For example, the location of the array length (which is stored in safe mode at least) may be at offset -8. But such a location can never be written to (array lengths never change, once

allocated), so a load of the array length is available at all locations dominated by the load.

The seed of the class relation comes from the compiler, that annotates an address with this information. In our example,

$$\begin{array}{l}\mathsf{mult}(A, p, q):\\ \quad t_0 \leftarrow M[A+0]\\ \quad t_1 \leftarrow M[p+0]\\ \quad t_2 \leftarrow t_0 + t_1\\ \quad t_3 \leftarrow M[A+4]\\ \quad \ldots\end{array}$$

the compiler would generate

$$\begin{array}{l}\mathsf{class}(A, \mathsf{int}[\,], 0)\\ \mathsf{class}(p, \mathsf{struct\ point}*, 0)\\ \mathsf{class}(q, \mathsf{struct\ point}*, 0)\end{array}$$

We now propagate the information through a forward dataflow analysis. For example:

$$\frac{l : b \leftarrow a \quad \mathsf{class}(a, \tau, k)}{\mathsf{class}(b, \tau, k)} \qquad \frac{l : b \leftarrow a + \$n \quad \mathsf{class}(a, \tau, k)}{\mathsf{class}(b, \tau, k+n)}$$

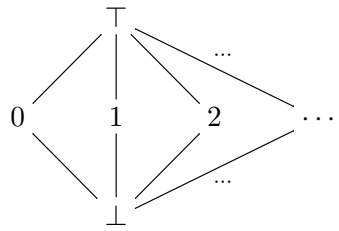
In the second case we have written $\$n$ to emphasize the second summand is a constant n. Unfortunately, if it is a variable, we cannot precisely calculate the offset. This may happen with arrays, but not with pointers, including pointers to structs. So we need to generalize the third argument to class to be either a variable or $\top$, which indicates any value may be possible. We then have, for example

$$\frac{l : b \leftarrow a + t \quad \mathsf{class}(a, \tau, k)}{\mathsf{class}(b, \tau, \top)}$$

Now $\top$ behaves like an information sink. For example, $\top + k = k + \top = \top$. Since in SSA form a is defined only once, we should not have to change our mind about the class assigned to a variable. However, at parameterized jump targets (which is equivalent to Φ-functions), we need to "disjoin" the information so that if the argument is known to be k at one predecessor but unknown at $\top$ at another predecessor, the result should be $\top$.

Because of loops, we then need to generalize further and introduce $\bot$ which means that we believe (for now) that the variable is never used. Because of the seeding by the compiler, this will mostly happen for loop variables. The values are

arranged in a *lattice*



where at the bottom we have more information, at the top the least. The $\sqcup$ operation between lattice elements finds the least upper bounds of its two arguments. For example, $0 \sqcup 4 = \top$ and $\bot \sqcup 2 = 2$. We use it in SSA form to combine information about offsets. Written with Φ-functions, we would have

$$\frac{\begin{array}{l}\mathsf{class}(a_0, \tau, k_0) \\ a_0 \leftarrow \Phi(a_1, \ldots, a_n) \\ \mathsf{class}(a_i, \tau, k_i) \quad (1 \leq k \leq n)\end{array}}{\mathsf{class}(a_0, \tau, k_0 \sqcup k_1 \sqcup \cdots \sqcup k_n)}$$

Because of loops we might perform this calculation multiple times until we have reached a fixed point. In this case the fixed point is least upper bound of all the offset classes we compute, which is a little different than the saturated data base we considered before.

This is an example of *abstract interpretation*, which may be a subject of a future lecture. One can obtain a more precise alias analysis if one refines the *abstract domain*, which is lattice shown above.

5 Allocation-Based Alias Analysis

Another technique to infer that pointers may not alias is based on their allocation point. In brief, if two pointers are allocated with different calls to alloc or alloc_array, then they cannot be aliased. Because allocation may happen in a different function than we are currently compiling (and hopefully optimizing), this is an example of an *interprocedural analysis*.

Lecture Notes on Decompilation

15411: Compiler Design
Maxime Serrano

Lecture 20
October 31, 2013

1 Introduction

In this lecture, we consider the problem of doing compilation "backwards" - that is, transforming from a compiled binary into a reasonable representation of its original source. Solving this problem will involve significant consideration of our standard dataflow analyses, as well as a discussion of good selection of internal representations of code.

While the motivation for the existence of compilers is fairly clear, the motivation for the existence of *decompilers* is less so. However, in the modern world there exist *many* legacy systems for which the original source code has been lost, which need bugs fixed in them or to be ported to a more modern architecture. Decompilers facilitate this process greatly. In addition, in malware analysis, generally source is not provided. It is therefore extremely useful to have some way to go from binary to a reasonable approximation of the original code.

For this lecture, we will focus on decompiling machine code, originally C0 code, that conforms to the C ABI, into a version of C0 with pointer arithmetic and `goto`. This comes *nowhere near* to being a treatment of decompilation of arbitrary binaries (and in fact the algorithms as described here will frequently fail to work on arbitrary binaries!), though more complex variants of the same ideas will continue to work.

2 Steps of Decompilation

Roughly, decompilation follows a few steps:

1. Disassembly - transformation from machine code to the assembly equivalent. There are a surprising number of pitfalls here.
2. Lifting and dataflow analysis - transforming the resulting assembly code into a higher-level internal representation, such as our three-operand assembly. One of the tricky parts here is recognizing distinct variables, and detaching variables from registers or addresses. We also recover expressions, function return values and arguments.
3. Control flow analysis - recovering control flow structure information, such as if and while statements, as well as their nesting level.
4. Type analysis - recovering types of variables, functions, and other pieces of data.

3 Disassembly

The first step of writing a good decompiler is writing a good disassembler. While the details of individual disassemblers can be extremely complex, the general idea is fairly simple. The mapping between assembly and machine code is in theory one-to-one, so a straight-line translation should be feasible.

However, disassemblers rapidly run into a problem: it is *very difficult* to reliably distinguish code from data.

In order to do so, generally disassemblers will take one of two strategies:

1. Disassemble the sections that are generally filled with code (`.plt`, `.text`, some others) and treat the rest of them as data. One tool that follows this strategy is `objdump`. While this works decently well on code produced by most modern compilers, there exist (or existed!) compilers that place data into these executable sections, causing the disassembler some confusion. Further, any confusingly-aligned instructions will also confuse these disassemblers.

2. Consider the starting address given by the binary's header, and recursively disassemble all code reachable from that address. This approach is frequently defeated by indirect jumps, though most of the disassemblers that use it have additional heuristics that allow them to deal with this. An example tool that follows this strategy is Hex-Ray's Interactive Disassembler.

While disassembly is a difficult problem with many pitfalls, it is not particularly interesting from an implementation perspective for us. Many program "obfuscators" have many steps that are targeted at fooling disassemblers, however, as without correct disassembly it is impossible to carry on the later steps.

4 Lifting and Dataflow Analysis

Given correct disassembly, another problem rears its head. As you may have noticed while writing your compilers, doing any form of reasonable analysis on x86_64 is an exercise in futility. The structure of most assembly language does not lend itself well to any kind of sophisticated analysis.

In order to deal with this, decompilers generally do something which closely resembles a backwards form of instruction selection. However, decompilers cannot just tile sequences of assembly instructions with sequences of abstract instructions, as different compilers may produce radically different assembly for the same sequence of abstract instructions.

Further, frequently a single abstract instruction can expand into a very long sequence of "real" instructions, many of which are optimized away by the compiler later on.

There are two primary approaches to dealing with this issue. The first is to simply translate our complex x86_64 into a simpler RISC instruction set. The tools produced by Zynamics frequently take this approach. The alternative is to translate into an exactly semantics-preserving, perhaps more complicated, instruction set, which has more cross-platform ways of performing analysis on it. This is the approach taken by CMU's BAP research project, as well as by the Hex-Rays decompiler.

The choice of the internal representation can be very important. For our purposes, we'll consider a modified version of the 3-operand IR that we've been using throughout the semester. We'll consider a version that is extended to allow instructions of the form `s <- e` where `e` is an expression.

We will summarize the translation from x86_64 to our IR by simply effectively doing instruction selection in reverse. The difficulty here is generally in the design of the IR, which we most likely do not have the time to discuss in detail. Some places to learn about IRs include the BAP website (`bap.ece.cmu.edu`) and the Zynamics paper "REIL: A platform-independent intermediate representation of disassembled code for static code analysis" by Thomas Dullien and Sebastian Porst.

Once we have obtained an IR, we would now like to eliminate as many details about the underlying machine as possible. This is generally done using a form of dataflow analysis, in order to recover variables, expressions and the straight-line statements.

Recall the dataflow analyses that have been presented in past lectures. Many of these analyses will be available to help us "refactor" the IR produced by our direct translation.

We will follow two preliminary analyses, both of which are predicated on liveness analysis:

1. Dead register elimination. This is necessary to efficiently deal with instructions such as `idiv`, as well as to notice `void` functions. It should be noted that unlike in your compilers, it is *sometimes* possible to eliminate instructions with additional state. For example, if `idiv %ecx` translates into:

```
t <- %edx:%eax
%eax <- t / %ecx
%edx <- t % %ecx
```

and `%eax` is not live in the successor, *it is permissible* to remove the second line of the result, since the third line will cause the division by 0 in the case that `%ecx` is zero.

Dead register elimination is done following effectively the same rules as dead code elimination from the homeworks, with some special cases like the above.

2. Dead flag elimination. Our translation makes direct use of the condition flags, and keeps track of which of them are defined and used at which time. We treat flags effectively as registers of their own. In this case, if a flag f is defined at a line l and is not live-in in $l + 1$, then we remove the definition of f from the line l. This will simplify our later analyses greatly, allowing us to collapse conditions more effectively.

3. Conditional collapsing. At this stage, we collapse sequences of the form comparison-cjump into a conditional jump on an expression. For example, after flag elimination, we collapse:

   ```
   zf <- cmp(%eax,0)
   jz label
   ```

   into

   ```
   jcond (%eax == 0) label
   ```

   In C0, generally every conditional will have this form. However, sufficiently clever optimizing compilers may be able to optimize some conditional chains more efficiently. A discussion of transforming more optimized conditions can be found in Cristina Cifuentes' thesis.

Having reached this point in the analysis, we would like to lose registers. Hence, we may simply replace each register with an appropriate temp, *taking care to keep argument and result registers pinned.* We then do the function-call-expansion step in reverse, replacing sequences of moves into argument registers followed by a call with a parametrized call. We note that in order to do so, we must first make a pass over all functions to determine how many arguments they take, in order to deal with the possibility of certain moves being optimized out.

At this stage, it is possible to effectively perform a slightly modified SSA analysis on the resulting code. Hence, for the future we will assume that this SSA analysis has been executed, and define our further analysis over SSA code. We may now perform an extended copy-propagation pass to collapse expressions.

This is sufficient to perform the next stages of the analysis. However, many decompilers apply much more sophisticated techniques to this stage. Cristina Cifuentes' thesis contains a description of many such algorithms.

5 Control Flow Analysis

Having reached this stage, we now have a reasonable control flow graph, with "real" variables in it. At this point, we *could* produce C code which is semantically equivalent to the original machine code. However, this is frequently undesirable. Few programs are written with as much abuse of the `goto` keyword as this approach would entail. Most control flow graphs are generated by *structured* programs, using `if`, `for` and `while`. It is then desirable for the decompiler to attempt to recover this original structure and arrive at a fair approximation of the original code.

This form of analysis relies largely on graph transformations. A primary element of this analysis relies on considering *dominator* nodes. Given a start node a, a node b is said to dominate a node c if every path from a to c in the graph passes through b. The *immediate dominator* of c is the node b such that for every node d, if d dominates c, then either $d = b$ or d dominates b.

5.1 Structuring Loops

We will consider three primary different classes of loops. While other loops may appear in decompiled code, analysis of these more complex loops is more difficult. Further reading can be found in the paper "A Structuring Algorithm for Decompilation" by Cristina Cifuentes. Our three primary classes are as follows:

1. *While loops*: the node at the start of the loop is a conditional, and the latching node is unconditional.
2. *Repeat loops*: the latching node is conditional.
3. *Endless loops*: both the latching and the start nodes are unconditional.

The *latching node* here is the node with the back-edge to the start node. We note that there are at most *one of these* per loop in our language, as `break` and `continue` do not exist.

In order to do so, we will consider *intervals* on a digraph. If h is a node in G, the interval $I(h)$ is the maximal subgraph in which h is the only entry node and in which all closed paths contain h. It is a theorem that there exists a set $\{h_1, ...h_k\}$ of header nodes such that the set $\{I(h_1), ...I(h_k)\}$ is a partition of the graph, and further there exists an algorithm to find this partition.

We then define the sequence of *derived graphs* of G as follows:

1. $G^1 = G$.
2. G^{n+1} is the graph formed by contracting every interval of G^n into a single node.

This procedure eventually reaches a fixed point, at which point the resulting graph is *irreducible*.

Note that for any interval $I(h)$, there exists a loop rooted at h if there is a back-edge to h from some node $z \in I(h)$. One way to find such a node is to simply perform DFS on the interval. Then, in order to find the nodes in the loop, we define h as being part of the loop and then proceed by noting that a node k is in the loop if and only if its immediate dominator is in the loop and h is reachable from k.

The algorithm for finding loops in the graph then proceeds as follows. Compute the derived graphs of G until you reach the fixed point, and find the loops in each derived graph. Note that if any node is found to be the latching node for *two* loops, one of these loops will need to be labeled with a goto instead. While there do exist algorithms that can recover more complex structures, this is not one of them.

5.2 Structuring Ifs

An *if* statement is a 2-way conditional branch with a common end node. The final end node is referred to as the *follow* node and is immediately dominated by the header node.

First, compute a post-ordering of the graph, and traverse it in that order. This guarantees that we will analyze inner nested ifs before outer ones.

We now find if statements as follows:

1. For every conditional node a, find the set of nodes immediately dominated by a.
2. Produce G' from G by reversing all the arrows. Filter out nodes from the set above that do not dominate a in G'.
3. Find the closest node to a in the resulting set, by considering the one with the highest post-order number.

The resulting node is the *follow* node of a.

We note that this algorithm does not do a particularly good job of dealing with boolean short-circuiting.

Any control flow that does not match the patterns above will be replaced with an if with a `goto`.

6 Type Analysis

Given control flow and some idea of which variables are which, it is frequently useful to be able to determine what the *types* of various variables are. While it may be correct to produce a result where every variable is of type `void *`, no one actually writes programs that way. Therefore, we would like to be able to assign variables and functions their types, as well as hopefully recover structure layout.

A compiler has significant advantages over a decompiler in this respect. The compiler knows which sections of a structure are padding, and which are actually useful; it also knows which things a function can take or accept. A compiler notices that the functions below are different, and so compiles them separately; a decompiler may not be able to notice that these functions accept different types without some more sophisticated analysis. In particular, on a 32-bit machine, these functions will produce *identical* assembly.

```
struct s1 { int a; };
int s1_get(struct s1 *s) { return s->a; }
struct s2 { struct s1 *a; };
struct s1 *s2_get(struct s2 *s) { return s->a; }
```

Given this problem, how does type analysis work?

In short, the answer is: this is an open problem. The TIE paper by CyLab claims to resolve many such cases, but is far from complete. The Hex-Rays decompiler fails to recognize structures altogether, and often defaults to `int` even when the variable is in fact a pointer.

We can model a simple type analysis as follows:

1. Multiplication, substraction, shifting, xor, binary and, binary or and division force their "parameters" to be integers.
2. Dereferencing forces its parameter to be a pointer.
3. The return values of standard library functions are maintained.
4. Any variable that is branched on is a boolean.
5. If two variables are added together and one is a pointer, the other is an integer.
6. If two variables are added together and one is an integer, the other is either a pointer or an integer.
7. If two variables are compared with `<`, `>`, `>=` or `<=`, they are both integers.
8. If two variables are compared with `==` or `!=`, they have the same type.
9. If something is returned from `main()`, it is an integer.
10. If the value of one variable is moved into another variable, they have the same type.
11. If the dereferenced value of a pointer has type τ, then the pointer has type τ^*.
12. The sum of a pointer of type τ^* and an integer is a pointer, but not necessarily of type τ^*.

We note that in order to get high-quality types, we will often need to perform analysis across function boundaries. We also note that this analysis is entirely unable to distinguish between structures and arrays. A more sophisticated type analysis is described in the TIE paper in the references section. There is plenty of research being done in this area, however!

7 Other Issues

Other issues that haven't been discussed here include doing things like automatically detecting vulnerabilities, detecting and possibly collapsing aliases, recovering scoping information, extracting inlined functions, or dealing with tail call optimizations. Many of these problems (and, in fact, many of the things discussed above!) do not have satisfactory solutions, and remain open research problems. For one, CMU's CyLab contains a group actively doing research on these topics. They recently (a few days ago!) released a paper containing a description of their solutions to many of these problems. Since they decompile arbitrary native code, rather than caring mostly about a specific language, they encounter some very interesting and difficult problems.

Decompilation as a whole is very much an open research topic, and there exist very few reasonable decompilers. One of the better-known ones is the Hex-Rays decompiler, and it is sadly entirely closed-source. As far as I know, there are *no* high-quality open-source decompilers for x86 or x86_64.

8 References

The material for this lecture was almost entirely gleaned from the following:

1. Cifuentes, Cristina. "A Structuring Algorithm for Decompilation." *XIX Conferencia Latinoamericana de Informática*, 2-6 August 1993, 267-276.

2. Cifuentes, Cristina. "Reverse Compilation Techniques." PhD thesis, Queensland University of Technology, 1994.

3. Dullien, Thomas, and Sebastian Porst. "REIL: A platform-independent intermediate representation of disassembled code for static code analysis." *CanSecWest*, 2009.

4. Lee, JongHyup, Thanassis Avgerinos, and David Brumley. "TIE: Principled Reverse Engineering of Types in Binary Programs." *NDSS Symposium*, 2011.

5. Schwartz, Edward J., JongHyup Lee, Maverick Woo, and David Brumley. "Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring." *Usenix*, 2013.

Lecture Notes on Garbage Collection

15-411: Compiler Design
Frank Pfenning

Lecture 21
November 5, 2013

These brief notes only contain a short overview, a few pointers to the literature with detailed descriptions, and a few remarks particularly relevant to C0.

1 Introduction

So far, in C0 we have had only primitives for *allocation* of memory on the heap. Memory was never freed. In C, the `free` function accomplishes this, but it is very error-prone. Memory may not be freed that is no longer needed (a *leak*), or, worse, memory may be free that will be referenced later in the computation. In type-safe languages this can avoided by using *garbage collection* that automically reclaims storage that can no longer be referenced. Since it is undecidable if memory might still be referenced, a garbage collector uses a conservative approximation, where different techniques may approximate in different ways.

There are three basic garbage collection techniques.

Reference Counting. Each heap object maintains an additional field containing the number of references to the object. The compiler must generate code that maintains this reference count correctly. When the count reaches 0, the object is deallocated, possibly triggering the reduction of other reference counts. Reference counts are hard to maintain, especially in the presence of optimizations. The other problem is that reference counting does not work well for circular data structures because reference counts in a cycle can remain positive even though the structure is unreachable. Nevertheless, reference counting appears to remain popular for scripting languages like Perl, PHP, or Python. Another use of reference counting is in part of an operating system where we know that no circularities can arise.

Mark-and-Sweep. A mark-and-sweep collector traverses the stack to find pointers into the heap and follows each of them, marking all reachable objects. It then

sweeps through memory, collecting unmarked objects into a free list while leaving marked objects in place. It is usually invoked if there is not enough space for a requested allocation. Because objects are never moved once allocated, a mark-and-sweep collector runs the risk of fragmented memory which can translate to poor performance. Another difficulty with a mark-and-sweep collector is that the cost of a collection is proportional to all available memory (which will be touched in the sweep phase).

Copying Collection. A copying collector also traverses the heap, starting from the so-called *root pointers* on the stack. Instead of marking objects it moves reachable objects from the heap to a new area called the *to-space*. When all reachable objects have been moved, the old heap (the *from-space*) and the to-space switch roles. The copying phase will compact memory, leading to good locality of reference. Moreover, the cost is only proportional to the *reachable* memory rather than all allocated memory. On the other hand, a copying collector typically needs to run with twice as much memory than a mark-and-sweep collector.

Mark-and-sweep and copying collectors are called *tracing* collectors, since they determine the reachable (or *live*) objects on the heap by following pointers. They tend to suffer from long pauses when a garbage collection is performed. Many variations and refinements have been proposed to overcome some of the difficulties and drawbacks in various forms of garbage collectors. A somewhat dated, but still excellent survey on garbage collection by Wilson[1] [Wil92].

For this course and the C0 language we recommend implementing a simple copying collector. Experience shows that it is less error-prone and easier to implement than a mark-and-sweep or copying collector. Since there is extensive and accessible literature on garbage collection, in the remainder of this note we focus on the compiler support that is necessary for a tracing collector. The issues for mark-and-sweep and copying collectors are very similar.

We explicit do not discuss many optimizations and refinements of the basic schemes. Some of these are common sense, others can be found in the literature.

2 Allocation

In a mark-and-sweep collector, allocation is handled via a free list, usually doubly linked. This is analogous to the data structure maintained by implementations of malloc/free in C. When allocation is called we traverse the free list until we find a block that is big enough for the requested payload plus header information. We return a pointer to this object (usually with the header to the left of the pointer) which should be aligned at 0 modulo 8. If a sufficiently large portion of the block

[1] Revised version at http://www.cs.cmu.edu/~fp/courses/15411-f13/misc/wilson94-gc.pdf

is unused, it is returned to the free list for further allocations. We run out of space if we cannot find any block that is big enough.

In a copying collector there is no free list. Instead we have a currently used half-space and a next pointer to the end of the currently used portion of the half-space. We return a pointer to its beginning (perhaps after adding a fixed offset to allow for a header) and advance the next pointer. Allocation in a copying collector is typically significantly faster than in a mark-and-sweep collector.

3 Finding Root Pointers

Assume that either `alloc` or `alloc_array` is called and we have run out of space. We now need to find the *root pointers* on the stack that point to the heap. This task is simplified in C0 since we have no pointers to the stack, unless the compiler optimizes to allocate some data on the stack. The compiler lays out each stack frame, so it knows where to find pointers and what their types are. Which pointers exactly are still live (and even where they are on the stack) may change during the computation of the function. We therefore best associate this information with each *return address*.

The information we need at a snapshot of the stack frame is the place where pointers are. This may be kept in a *pointer map* for the stack frame. A reference to the pointer map may be kept in a seperate data structure, or in the stack frame itself (for example, at the bottom or top of the stack frame).

Since have the traverse the stack it is convenient to keep base pointers for each frame which are pushed onto the stack just as in the x86 calling convention. While optional for the x86-64, it is quite useful for the garbage collector. So, once we have processed the pointers in the current frame, we find the return address to get the pointer map for the previous frame until we get to the first frame on the stack.

There are some subtleties regarding registers. Since the last call will always be to `alloc` or `alloc_array` we don't have to worry about registers except callee-save registers. The called function will not be able to tell if any of the callee-save registers contains a root pointer. A simple strategy to handle such callee-save registers is for the caller (who knows what they are) to simply push callee-save registers containing live root pointers onto the stack and add these locations to the pointer map. They do not need to be restored, due to the callee-save protocol. Caller-save and argument registers will already be on the stack (and in the pointer map) if they are live.

4 Derived Pointers

Some computations and optimizations will compute addresses of data in the middle of heap objects. It is important that we don't follow such pointers since, for

example, the header of an object will not be at a fixed offset from such a derived pointer. Instead, the compiler should arrange to keep mark the corresponding base pointer as *live* and keep it somewhere were it is recognized as a root pointer. A typical example of this is the computation of the address of an array element, which will be a derived pointer.

5 Traversing the Heap

When we traverse the heap, whether just marking or copying reachable objects, we need to identify pointers in the objects we reach. The traditional way to accomplish this is to keep a reference to a pointer map in the object header. Just as for a stack frame, every pointer in the heap object has an entry with its offset in the pointer map, with a special indicator for arrays. In addition the header should have a bit that can marked when visited (for mark-and-sweep) or marked when copied (for a copying collector). In the case of a copying collector, we need to make sure the object is big enough to hold the *forwarding pointer* when it is moved to the *to-space*. In the case of a mark-and-sweep collector the object has to be big enough to hold the next and prev pointers of the doubly-linked free list.

6 Tagless Garbage Collection

In a safe, statically typed language such as C0, pointer maps are not strictly necessary for heap objects as long as we can determine the types of the root pointers. After that, the type of every heap object we reach, and the types of the components, are determined entirely from the types and the computed offsets for structs.

As suggested by Goldberg[2] [Gol91] an efficient and convenient way to achieve this is for the compiler to generate a structure traversal function for each type that may be allocated on the heap. For each root pointer we then just need to know which garbage collection traversal function to call. This is something the compiler can know (since types are known at compile-time) and store at an appropriate place, either in the stack frame, the text segment, or allocate during an initialization phase in a global variable.

References

[Gol91] Benjamin Goldberg. Tag-free garbage collection fro strongly typed programming languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI'91, pages 165–176. ACM, June 1991.

[2] and a student during lecture

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM'92, pages 1–42, London, UK, 1992. Springer-Verlag.

Lecture Notes on Polymorphism

15-411: Compiler Design
Frank Pfenning

Lecture 24
November 14, 2013

1 Introduction

Polymorphism in programming languages refers to the possibility that a function or data structure can accommodate data of different types. There are two principal forms of polymorphism: *ad hoc polymorphism* and *parametric polymorphism*. Ad hoc polymorphism allows a function to compute differently, based on the type of the argument. Parametric polymorphism means that a function behaves uniformly across the various types [Rey74].

In C0, the equality == and disequality != operators are ad hoc polymorphic: they can be applied to small types (int, bool, $\tau*$, $\tau[]$, and also char, which we don't have in L4), and they behave differently at different types (32 bit vs 64 bit comparisons). A common example from other languages are arithmetic operators so that $e_1 + e_2$ could be addition of integers or floating point numbers or even concatenation of strings. Type checking should resolve the ambiguities and translate the expression to the correct internal form.

The language extension of void$*$ we discussed in Assignment 4 is a (somewhat borderline) example of parametric polymorphism, as long as we do not add a construct hastype(τ, e) or eqtype(e_1, e_2) into the language and as long as the execution does not raise a dynamic tag exception. It should therefore be considered somewhat borderline parametric, since implementations must treat it uniformly but a dynamic tag error depends on the run-time type of a polymorphic value.

Generally, whether polymorphism is parametric depends on all the details of the language definition. The importance of parametricity for data abstraction in language implementations cannot be overstated. Failure of parametricity often means failure of data abstraction: an implementation of a generic data structure cannot necessarily be replaced by another one (even if it is correct!) without breaking a client.

2 Parametric Polymorphism

The prototypical example of a parametric function is the identity function, $\lambda x.\, x : \alpha \to \alpha$. In C0, we might write this as

```
a id(a x) {
  return x;
}
```

which interprets the undefined type name a as a type variable whose scope is the current function. The projection function, which ignores its second argument, would be

```
a proj(a x, b y) {
  return x;
}
```

with both a and b as type variables. From this we extract an abstract form of definition

$$\begin{array}{lcl} id & : & \forall a.\, (a) \to a \\ proj & : & \forall a, b.\, (a, b) \to a \end{array}$$

When type-checking the body of a function, the free variables in the function definition are treated like new basic types. In particular, they are *not* subject to instantiation, since in the end the function has to work for *all* types. To account for this we allow a new form of declaration a : type in our typecontext Γ.

When type-checking the use of a polymorphic function, we can instantiated the type variables to other types. For example,

```
if (id(true)) return id(id(4));
```

should be well-typed. In order to formalize this we will need a *substitution* θ for the (quantified) type variables from the definition of a function, using concrete types and other type variables declared in the context. We write

$$\Gamma \vdash \theta : (a_1, \ldots, a_k)$$

if θ substitutes types that are well-formed in Γ for the type variables $a_1, \ldots, a_k$. Furthermore, we write $\theta(\tau)$ for the result of applying the substitution θ to the type τ.

Our typing rule then shapes up as follows:

$$\frac{\begin{array}{l} f : \forall a_1, \ldots, a_k.(\tau_1, \ldots, \tau_n) \to \tau \\ \Gamma \vdash \theta : (a_1, \ldots, a_k) \\ \Gamma \vdash e_1 : \theta(\tau_1) \\ \cdots \\ \Gamma \vdash e_n : \theta(\tau_n) \end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) : \theta(\tau)}$$

Note that there is a single substitution θ, so the type variables $a_1, \ldots, a_n$ must be instantiated consistently for all arguments and the result. For example:

$$\frac{\begin{array}{l} id : \forall a.\,(a) \to a \\ \cdot \vdash (\mathsf{int}/a) : (a) \\ \cdot \vdash 4 : \mathsf{int} \end{array}}{\cdot \vdash id(4) : \mathsf{int}}$$

where $4 : \mathsf{int}$ arises from $4 : (\mathsf{int}/a)(a)$.

3 Generic Data Structures

In a first-order imperative language, the main use of polymorphism is for generic data structures. For example, we may want to have a stack with elements of arbitrary type a.

```
struct list_node<a> {
  a data;
  struct list_node<a>* next;
};

typedef struct list_node<a> list<a>;
```

During compilation, we would like to create parametric code, which works the same independently of the type a. If we restrict type variables to be instantiated to *small types* then we can allocate 8 bytes for a polymorphic field of a struct, which should always be enough room. During allocation, the polymorphic field will be initialized with 0, which by design represents the default value of all types.

In other languages we may *box* polymorphic data (replace them by a reference to the actual data), or *monomorphise* the whole program and compile multiple versions of a function.

The type parameter of the structure is indicated inside the angle brackets. Functions manipulating the structure would be correspondingly polymorphic. For example:

```
list<a>* cons(list<a>* p, a elem) {
  list<a>* q = alloc(list<a>);
  q->data = elem;
  q->next = p;
  return q;
}
```

4 Pairs

We can easily define a product type, which would usually be written as $a * b$ in a functional language.

```
struct prod<a,b> {
  a fst;
  b snd;
};

typedef struct prod<a,b>* prod<a,b>;

a fst(prod<a,b> p) {
  return p->fst;
}

b snd(prod<a,b> p) {
  return p->snd;
}

prod<a,b> pair(a x, b y) {
  prod<a,b> p = alloc(struct prod<a,b>);
  p->fst = x;
  p->snd = y;
  return p;
}
```

5 Function Pointers

Polymorphism in data structures is severely handicapped unless we can store function pointers. For example, a hash table may be parameterized by a type key for keys and a type a for the elements stored in the table. We store in the header functions to hash a key value, to compare keys, and extracting a key from an element.

```
struct ht_header<key,a> {
  int size;                 /* size >= 0 */
  int capacity;             /* capacity > 0 */
  list<a*>*[] table;        /* \length(table) == capacity */
  int (*hash)(key k);       /* hash function */
  bool (*key_equal)(key k1, key k2); /* key comparison */
  key (*elem_key)(a elem);  /* extracting key from element */
};
```

```
typedef struct ht_header<key,a> ht<key,a>;

a* ht_lookup(ht<key,a> H, key k)
//@requires is_ht(H);
{
  int i = (*H->hash)(k);
  list<a*>* p = H->table[i];
  while (p != NULL) {
      //@assert p->data != NULL;
      if ((*H->key_equal)((*H->elem_key)(*p->data), k))
        return p->data;
      else
        p = p->next;
  }
  /* not in list */
  return NULL;
}
```

6 Interactions With Other Language Features

The interactions between parametric and ad hoc polymorphism are often tricky. In C0 with parametric polymorphism, the main issue arises with equality. If we have e_1 == e_2 where e_1 and e_2 are of type a? If a stands for a small type, this might be feasible, but there is still a difference between 32-bit and 64-bit comparisons. Alternative, we could simply rule this out. This would suggest itself in particular in C0 with a type string, which is not subject to equality testing.

A general approach to interactions between ad hoc and parametric polymorphism are *type classes* as they are used in Haskell. In lecture, students proposed some extensions of the above so that polymorphism can be limited to type classes. Since I did not take any pictures of the blackboard at the time, these extensions are lost to posterity unless someone sends me some suggestions.

7 Type Inference

Often associated with parametric polymorphism is the idea of *type inference*. For the polymorphic part of the language, this actually presents rather few problems, since the scope of type variables is naturally delineated by function definitions. However, in C0 there is a problem with field selection, $e.f$. Since fields are global and can freely be shared between different structs, it will be difficult to disambiguate uses of the field names f and therefore the type of e.

8 Type Conversions and Coherence

Ad hoc polymorphism is often associated with (implicit) conversions between types. For example, in an expression $3 + x$ where $x : \mathsf{float}$ we might *promote* the integer 3 to a floating point number, since the other summand is a floating point number. There is a complicated set of rules in the definition of C [KR88] regarding such conversions between types, including integral types of varying sizes, pointers, and other numeric types like float or double.

Inside a compiler, such promotions should be turned into explicit operators, for example $\mathsf{itof}(3) + x$, where itof converts an integer to its floating point representation.

The problem with such implicit conversion is that it can easily lead to errors. The more complicated the rules in the language definition, the more likely it is to lead to errors which are often hard to find. Particularly pernicious are error arising from truncation of wider types to narrower ones, since they can remain undetected for a long time on smaller inputs. A language satisfies *coherence* if various legal ways of inserting type conversions always leads to the same answer [Rey91]. In such language the meaning of expressions is less dependent on arcane details. Nevertheless, overloading and implicit conversions ought to be viewed with suspicion.

References

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Berlin, 1991. Springer-Verlag.

Lecture Notes on Session-Typed Concurrency

15-411: Compiler Design
Frank Pfenning

Lecture 25
November 19, 2013

1 Introduction

Generally speaking, it is difficult to add concurrency to a language and retain the same kind of strong guarantees that static typing in a language like C0 gives us. For a sequential language, type safety usually includes preservation (that program remains well-typed as it executes) and progress (there is always a well-defined action to take). In the presence of concurrency, we would like to add deadlock-freedom (which is an analogue to progress) and the absence of race conditions (to guarantee the result is well-defined).

In order achieve these properties, we work under the following conditions:

- Communication between processes is by message-passing rather then via shared memory. In a concrete implementation the message-passing might be accomplished using shared memory, but the computational model itself is at a higher level of abstraction.
- Processes communicate with each other along channels with just two end-points, one process on each end.
- A process offers a service along a unique channel and uses services along possibly many other channels. This allows us to identify a process with the channel along which it offers a service.

Under these assumptions we have designed a type system that guarantees preservation and progress, including the absence of deadlock and race conditions [CP10]. It uses the idea of *linear typing*, which is closely related to the concept of *linear inference* we used to specify program transformations. It is a particular instantiation of the idea of *session types* [Hon93] that prescribe interactions between processes along private channels. The settings for the prior work was process calculi [CPT13]

and functional languages [TCP13]; here we import some of the ideas in the (first-order) imperative setting. It should be emphasized that we have not proven anything about this extension of C0, so all the above properties may be false for this language instance.

2 Process Definitions

We have to extend our language of types by *session types* σ. We continue to use τ for ordinary value types. We will introduce various forms of session types incrementally and summarize them at the end.

Session type σ are used to prescribe communication behavior along *channels*, which are written as $\$c$. A channel declaration is therefore of the form $\sigma\ \$c$ in the concrete syntax and $\$c : \sigma$ in the typing judgments.

A process that *offers* service σ along channel $\$c$ and *uses* services $\sigma_1, \ldots, \sigma_n$ along channels $\$d_1, \ldots, \$d_n$ is spawned by invoking a process definition p with prototype

$$\sigma\ \$c\ p(\sigma_1\ \$d_1, \ldots, \sigma_n\ \$d_n);$$

A process can additionally take value arguments of *primitive types*, a feature we will exploit shortly. The body of a process definition contains the computation and communications to be performed by the process when spawned.

As a first example we consider a process producing a (potentially infinite) stream of integers. The protocol requires that the consumer request a new integer by sending the label 'next', to which the process responds with the next integers. In addition, the consumer can stop the process by sending the label 'stop'. This behavior is expressed by the type

```
choice natstream {
  int /\ choice natstream next;
  void stop;
};
```

The keyword `choice` indicate that the client chooses the operation to be performed by sending of the labels. *natstream* is the name for this particular choice. Syntactically, this is analogous to `struct s`, where s is the name of the struct.

The session type $\tau \wedge \sigma$ (here $\tau =$ int and $\sigma =$ choice *natstream* means that the process hand to send a value of type τ and then follow the session type σ. next is the label of the first alternative, stop the label of the second alternative. The session type void indicates that the process should terminate without further interactions.

Instead of writing out `choice natstream` we create `nats` as a synonym for it:

```
typedef choice natstream nats;
```

Next we would like to define a process that outputs a stream of integers according to the above protocol, starting at an integer n.

```
nats $c from(int n) {           /* $c : nats */
  switch ($c) {                 /* receive label along $c */
  case next:                    /* $c : int /\ nats */
    send($c, n);                /* $c : nats */
    $c = from(n+1);             /* tail call, continue in current process */
  case stop:
    close($c);                  /* $c : void */
  }
}
```

The construct

$$\mathsf{switch}\ (\$c)\ \{\mathsf{case}\ l_i : seq_i\}_i$$

waits to receive a label l_i along channel $\$c$ and then executes the corresponding case in the body of the switch statement. For this to be correct, the channel $\$c$ must be a choice among the labels l_i. In case the label next is received, we have the command $\mathsf{send}(\$c, n)$ which has the general form

$$\mathsf{send}(\$c, e);$$

which sends the value of e along channel $\$c$. For this to be correct, the channel $\$c$ must have type $\tau \wedge \sigma$, for some session type σ, and e must have type τ. Some restrictions may also be imposed on the type of e, so we do not have potentially complex marshaling and unmarshaling operations to be performed. The primitive types int, bool, char and string seem to be a reasonable choice.

What is curious here is that the switch statement requires $\$c$ to present a choice, while the subsequent send command requires $\$c$ to be a conjunction. This reflects the fact that the types of channels must change throughout the interactions of a process with its environment. If a channel $\$c$ has type choice $name\ \{\sigma_i : l_i\}_i$ then after receiving label l_i it will have type σ_i. Similarly, if a channel $\$c$ has type $\tau \wedge \sigma$, then after sending τ the channel will have type σ.

Next we come to

```
$c = from(n+1);
```

where the current process offers along $\$c$ and $\mathsf{from}(n+1)$ is a process invocation. This is the process analogue of a tail-call for a function, and means the current process continues by executing $\mathsf{from}(n+1)$. This is also the last statement along this branch of the switch statement, since this process invocation will never return. Notice that the value argument n to a process essentially functions as a variable local to the process.

In the stop branch of the switch, channel $\$c$ has type void. This means we have to close the channel, which is the last action the from process will take.

3 Typing

Because channels change their type during communcations, the typing judgment is a bit unusual. In addition to the context Γ that types value variables, we have a second context

$$\Delta = (c_1 : \sigma_1, \ldots, c_n : \sigma_n)$$

of channel typings. Since channels are distinguished by their position in the judgment, we drop the $-prefix for the names. The general form of the judgment then is

$$\Gamma \,;\, \Delta \vdash P :: (c : \sigma)$$

which says that P is a process expression that offers service of session type σ along channel c, using value variables in Γ and channels in Δ. As usual, we do not care about the order of declarations in Γ or Δ.

$$\frac{\{\Gamma \,;\, \Delta \vdash P_i :: c : \sigma_i\}_i}{\Gamma \,;\, \Delta \vdash \mathsf{switch}(c, \{l_i : P_i\}_i) :: c : \mathsf{choice}\{l_i : \sigma_i\}_i}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \,;\, \Delta \vdash P :: c : \sigma}{\Gamma \,;\, \Delta \vdash \mathsf{send}(c, e) \,;\, P :: c : \tau \wedge \sigma}$$

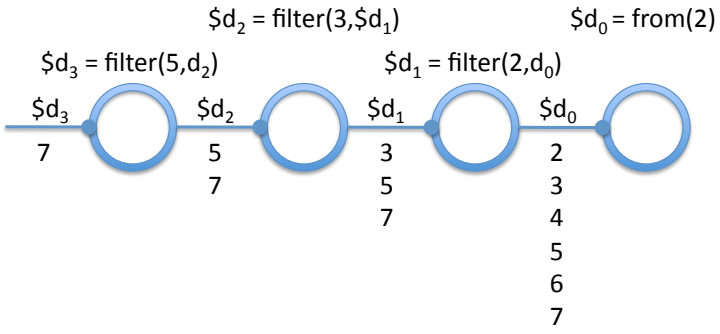
$$\frac{}{\Gamma \,;\, \cdot \vdash \mathsf{close}(c) :: c : \mathsf{void}}$$

$$\frac{p : (\sigma_1, \ldots, \sigma_n \vdash \sigma) \quad \Delta \sim (d_1{:}\sigma_1, \ldots, d_n{:}\sigma_n)}{\Gamma \,;\, \Delta \vdash c = p(d_1, \ldots, d_n) :: c : \sigma}$$

In the last rule, $\Delta \sim \Delta'$ means that Δ' is a permutation of Δ. We therefore have a precise account for all channels: they are used exactly once (even if that use might be in a recursive procedure invocation). For simplicity, we omitted value arguments, but they would just be expressions e_j of ordinary types τ_j, matching the process declaration. For example, when we close a channel c there may not be any unused channels left in the context. In a switch construct we check every branch in the same context Δ. This is correct, because exactly one of the branches will be chosen when the program is executed.

4 Stream Transducers

Our overall goal of this lecture will be to write a code for the prime sieve (or Sieve of Eratosthenes) that produces a stream of prime numbers. The sieve is implemented as a sequence of filters, each of which filters out all multiples of one particular prime. This is illustrated in the following picture.



Each circle represents a process, where the bullet indicates the channel along which it offers. Below each channel is the sequence of integers flowing from right to left, eliding the request labels next flowing from left to right.

We have already implemented the from process on the far right. Next we implement the filter processes. filter uses one process ($\$d$) and offers along another ($\$c$). It drops all multiples of p from $\$d$ and forwards the rest ot $\$c$. It begins by receiving a label along $\$c$, which must be one of next and stop.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    ...
  case stop:
    ...
  }
}
```

If it is next, we now need to request the next integer along $\$d$. This means we have to *send* the label next along $\$d$. We are using the channel $\$d$, and $\$d$ has type choice $natstream$, so this follows the protocol correctly. Sending a label l along a channel $\$d$ has the syntax $\$d.l$, analogous to the field selection in a struct.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    $d.next;
    ...
  case stop:
    ...
  }
}
```

After sending next, $\$d$ has evolved to type int $\wedge$ choice $natstream$. This means that the process providing $\$d$ will *send* an integer, and we have to receive it. The syntax is $x =$ recv($\$d$), where x must be declared if it hasn't already.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int k = recv($d);
    /* $d : nats |- $c : int /\ nats */
    ...
  case stop:
  }
}
```

We have indicated the types of $\$d$ and $\$c$ at this point during the computation. We now need to check whether p divides k. If so, we keep requesting integers from $\$d$ until we receive a value that is not a multiple of p. We then send the first such value along $\$c$ and recurse. We could accomplish this with two mututally recursive function, or with a loop. For illustration purposes we use a loop.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int k = recv($d);
    /* $d : nats |- $c : int /\ nats */
    while (k % p == 0) {
      $d.next;
      k = recv($d);
    }
    /* $d : nats |- $c : int /\ nats */
    send($c, k);
    $c = filter(p, $d);
  case stop:
    ...
  }
}
```

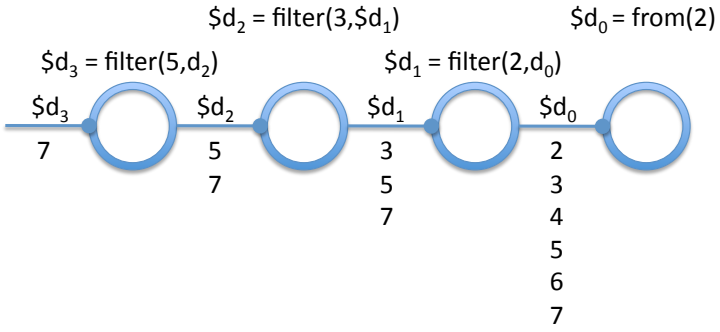
Because we don't know how often the loop will be traversed, the channel types must remain invariant throughout the loop. We have indicated those types in a comment before the loop. This will also be the type of the channels immediately after the loop.

In the case we receive the stop label, we cannot just close the channel $\$c$, because the channel $\$d$ would be left unaccounted for. Instead, we send it in turn a stop label and wait until it finishes. Of course, waiting wouldn't be strictly necessary if we trust it to complete, but for typing purposes we would like to consume that channel.

```
nats $c filter(int p, nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int k = recv($d);
    /* $d : nats |- $c : int /\ nats */
    while (k % p == 0) {
      $d.next;
      k = recv($d);
    }
    /* $d : nats |- $c : int /\ nats */
    send($c, k);
    $c = filter(p, $d);
  case stop:
    $d.stop;
    wait($d);
    close($c);
  }
}
```

5 Spawning New Processes

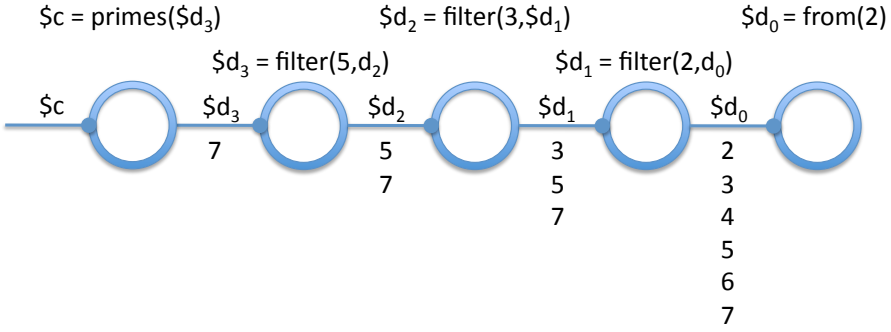
We have now written from and filter. Assume we would like to write the enclosing process primes which is supposed to produce the increasing sequence of prime numbers on channel $\$c$. Let's refer back to the diagram.



We want to write an outermost process

```
nats $c primes(nats $d);
```

which *uses* $\$d_3$ (in the diagram above) and produces along $\$c$ as in the extended diagram below.



If we are requested to produce a number (by receiving next along $\$c$), we ask $\$d$ for the next integer. Since this has already been filtered by all smaller primes, it should be a prime number and we can send it along $\$c$.

```
nats $c primes(nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int p = recv($d);
    send($c, p);
    ...
```

```
  case stop:
    ...
  }
}
```

Before we can recurse we need to spawn a new process, which filters the multiples of p. We do this simply by invoking the process definition and assigning a (new!) channel to the output. We then recurse, using this new channel. It's easy to see that there is exactly one filtering channel for each prime number that we send along $\$c$. In the case we are asked to stop, we just stop the process we use (which will cascade to the right).

```
nats $c primes(nats $d) {
  switch ($c) {
  case next:
    $d.next;
    int p = recv($d);
    send($c, p);
    nats $e = filter(p, $d);    /* spawn new process */
    $c = primes($e);
  case stop:
    $d.stop; wait($d);
    close($c);
  }
}
```

We could also reuse the channel name $\$d$, since $\$d$ is no longer in the context since it is used in the invocation of filter.

```
   $d = filter(p, $d);          /* spawn new process */
   $c = primes($d);
```

At the risk of blurring the line between process invocation and function call, we might also abbreviate these two lines to

```
   $c = primes(filter(p, $d));  /* spawn new process */
```

which would desugar to the first phrasing.

We can now create a process that just produces a stream of primes by creating the process on the far right (producing $2, 3, 4, 5, 6, \ldots$) and supplying it to primes.

```
nats $c prime_stream() {
  nats $d = from(2);
  $c = primes($d);
}
```

Here the tail call preserves the process (identified from the channel $\$c$) even though it is not a recursive call.

We can now embed this in a top-level *function* which creates a fresh stream of prime numbers and requests the first n before terminating the stream.

```
void print_primes(int n) {
  nats $c = prime_stream();
  /* $c : nats |- */
  for (int i = 0; i < n; i++) {
    $c.next;
    int p = recv($c);
    printint(p);
  }
  $c.stop; wait($c);
  return;
}
```

6 Additional Typing Rules

We show another few sample rules to supplement the ones shown earlier. The first is process invocation.

$$\frac{p : (\sigma_1, \ldots, \sigma_n \vdash \sigma') \quad \Delta \sim (d_1{:}\sigma_1, \ldots, d_n{:}\sigma_n) \quad \Delta', e{:}\sigma' \vdash P :: c : \sigma}{\Gamma \,;\, \Delta, \Delta' \vdash e = p(d_1, \ldots, d_n); P :: c : \sigma}$$

Implicit here is that e is different from c and all the channels declared in Δ'. If e is a variable previously declared, it must have been used (not in Δ') and its new type σ' should be consistent (which is something we do not track explicitly in these rules).

$$\frac{\Gamma \,;\, \Delta, d{:}\sigma_i \vdash P :: c : \sigma}{\Gamma \,;\, \Delta, d{:}\mathsf{choice}\{l_i : \sigma_i\}_i \vdash c.l_i \,;\, P :: c : \sigma}$$

$$\frac{\Gamma, x{:}\tau \,;\, \Delta, d{:}\sigma' \vdash P :: c : \sigma}{\Gamma \,;\, \Delta, d{:}\tau \wedge \sigma' \vdash x = \mathsf{recv}(d) \,;\, P :: c : \sigma}$$

$$\frac{\Gamma \,;\, \Delta \vdash P :: c : \sigma}{\Gamma \,;\, \Delta, d{:}\mathsf{void} \vdash \mathsf{wait}(d) \,;\, P :: c : \sigma}$$

We see that each type construct has a rule when it types the channel we are offering and when it types a channel we are using. One corresponds to a send, while the other corresponds to a receive, reflecting the complementary roles of the processes on the two ends of a channel.

7 Internal Choice and Forwarding

We refer to the choice construct we introduced so far as *external choice*. This is because if we offer $\mathsf{choice}\{l_i : \sigma_i\}_i$ along channel $\$c$, then the client can choose the label. We write

$$\mathsf{branch}\{l_i : \sigma_i\}_i$$

for the opposite *internal choice*, where the provider can choose the label and the consumer has to account for all possibilities.

An example of this is a simple implementaton of stacks, where each process holds an element of the stack. In the absence of further operations on the stack elements, this does not exhibit any concurrency, but it illustrates both internal choice and forwarding (explained later).

The operations on a stack are push, pop, and deallocation. When the client indicates he want to push an element onto the stack, we then have to *receive* an element along the same channel. For this we have the type

$$\tau \Rightarrow \sigma$$

(receive a value of type τ and then behave according to σ) which is symmetric to $\tau \wedge \sigma$ (send a value of type τ and then behave according to σ).

When the client would like to *pop* an element from the stack, we send one of two labels: none if there is no element on the stack, and some if there is one. In the latter case we follow it up with the element itself. This is an example of the internal choice we mentioned above. We define:

```
choice stack_node {
  int => choice stack_node push;
  branch opt_int pop;
  void dealloc;
};

/* Optional result from pop */
branch opt_int {
  int /\ choice stack some;
  choice stack_node none;
};

typedef choice stack_node stack;
```

We now represent the stack constructors empty and node simply as processes. We start with the empty stack. In case of a push, we call spawn a new empty process and recurse as a node with the new element. In case of a pop we indicate that the stack is empty and recurse. Deallocation just closes the channel and terminates the process.

```
stack $c empty();                /* empty stack */
stack $c node(int k, stack $d); /* stack with top element k */

stack $c empty() {
  switch ($c) {
  case push:
    int k = recv($c);
    stack $d = empty();
    $c = node(k, $d);            /* tail call: continue as nonempty */
  case pop:
    $c.none;                     /* no element available */
    $c = empty();                /* tail call: continue as empty */
  case dealloc:
    close($c);
  }
}
```

Second, the case of a nonempty stack with top element k. When we receive a push, we just spawn a new process and recursive. The interesting operation is that of pop. We first send the label some, indicating that the stack is non-empty, then we send the top element.

```
stack $c node(int k, stack $d) {
  switch ($c) {
  case push:
    int n = recv($c);
    stack $e = node(k, $d);      /* spawn new process */
    $c = node(n, $e);            /* new top of stack in current process */
  case pop:
    $c.some; send($c, k);        /* send current element */
    ...
 case dealloc:
    ...
  }
}
```

At this point, we would like to terminate the current process and hand off any interactions along $\$c$ to $\$d$. This is an example of *channel forwarding*. We write this as

```
$c = $d;
```

which identifies $\$c$ and $\$d$.

This could be implemented in several ways. Perhaps the cleanest way is for the current process to send $\$d$ to its client, essentially telling it "*communicate along* $\$d$

from now on instead of $\$c$". Then the process identified with $\$c$ can terminate since the channel $\$c$ is effectively no longer in use.

At a great cost of efficiency, we could also keep the process identified with $\$c$ alive, continuously forwarding messages in both directions until the channel is closed.

Deallocation is straightforward, leading to the following final process definition.

```
/* nonempty stack */
stack $c node(int k, stack $d) {
  switch ($c) {
  case push:
    int n = recv($c);
    stack $e = node(k, $d);     /* spawn new process */
    $c = node(n, $e);           /* new top of stack in current process */
  case pop:
    $c.some; send($c, k);       /* send current element */
    $c = $d;                    /* identify $c and $d; or: forward $d along $c */
  case dealloc:
    $d.dealloc; wait($d);
    close($c);
  }
}
```

We do not show any additional typing rules for the branch construct, since they are symmetric to the choice construct, just swapping offer and use. For forwarding, we just need to keep in mind that there is no continuation after forwarding, so the forwarded channel must be the only one in the context.

$$\frac{}{\Gamma\,;\, d{:}\sigma \vdash c = d :: c : \sigma}$$

8 Further Constructs

The type language in [TCP13] contains further important constructs. One allows the sending and receiving of channels along channels. Another allows *persistent channels* that do not change their type, but allow new instances of a persistent service to be spawned. We elide persistent channels entirely and just briefly show the rules for sending or receiving channels along channels, since they are used in the next example. We have type $\sigma_1 \otimes \sigma_2$ (concrete syntax `s1 ** s2`) for sending and $\sigma_1 \multimap \sigma_2$ (concrete syntax `s1 -o s2`) for receiving a channel.

$$\frac{\Gamma\ ;\ \Delta \vdash P :: c : \sigma}{\Gamma\ ;\ \Delta, d{:}\sigma' \vdash \mathsf{send}(c, d)\ ;\ P :: c : \sigma' \otimes \sigma}$$

$$\frac{\Gamma\ ;\ \Delta, d{:}\sigma' \vdash P :: d : \sigma}{\Gamma\ ;\ \Delta \vdash d = \mathsf{recv}(c)\ ;\ P :: c : \sigma' \multimap \sigma}$$

9 Further Example: Mergesort

Sending and receiving channels is exemplified in the mergesort program below.[1] The main complication in this implementation is setting up and tearing down the processes to, eventually, achieve an in-place sort. For the sake of brevity, we present it here without further comment.

```
/* Mergesort */
/* Henry DeYoung, transcribed from SILL by fp */

branch list {
  int /\ branch list cons;
  void nil;
};
typedef branch list list;

branch forest {
  int /\ (list ** branch forest) cons; /* int nl = num. of elems in list l */
  void nil;
};
typedef branch forest forest;

/* $c = nil() */
list $c nil() {
  $c.nil;
  close($c);
}

/* $c = cons(k, $d) */
list $c cons(int k, list $d) {
  $c.cons;
  send($c, k);
  $c = $d;
}

/* $c = merge($l, $r) offers sorted merge of $l and $r along $c */
```

[1] requested by a student in lecture

```
list $c merge(list $l, list $r) {
  switch ($l) {
  case cons:
    int x = recv($l);
    switch ($r) {
    case cons:
      int y = recv($r);
      if (x < y) {
        list $r2 = cons(y, $r); /* push y back onto r */
        list $d = merge($l, $r2); /* spawn new process */
        $c = cons(x, $d);
      } else {
        list $l2 = cons(x, $l); /* push x back onto l */
        list $d = merge($l2, $r); /* spawn new process */
        $c = cons(y, $d);
      }
    case nil:                   /* $r is empty */
      wait($r);
      $c = cons(x, $l);         /* push x back onto l */
    }
  case nil:                     /* $l is empty */
    wait($l);
    $c = $r;                    /* forward $r to $c */
  }
}

/* $f = fnil() */
forest $f fnil() {
  $f.nil;
  close($f);
}

/* $g = fcons(nl, $l, $f), adjoins list to forest */
/* invariant: nl = num of elements in $l */
forest $g fcons(int nl, $l list, $f forest) {
  $g.cons;
  send($g, nl);
  send($g, $l);                 /* send channel $l along $g */
  $g = $f;
}

/* $g = join(nl, $l, $f), adjoins list to forest */
/* rebalances if necessary */
/* invariant: nl = num of elements in $l */
forest $g join(int nl, list $l, forest $f) {
  switch ($f) {
  case cons:
    int nr = recv($f);
```

```
    list $r = recv($f);
    if (nl >= nr) {             /* $l is bigger than $r, first list in $f */
      list $m = merge($l, $r);  /* merge $l and $r */
      $g = join(nl+nr, $m, $f); /* adjoin merged list to forest */
    } else {
      forest $f1 = fcons(nr, $r, $f); /* push $r back onto $f */
      $g = fcons(nl, $l, $f1);        /* adjoin $l */
    }
  case nil:
    wait($f);
    $f1 = fnil();               /* recreate empty forest */
    $g = fcons(nl, $l, $f1);
  }
}

/* $m = compress($f), linearizes $f to obtain list $m */
list $m compress(forest $f) {
  switch ($f) {
  case cons:
    int nl = recv($f);
    list $l = recv($f);
    list $r = compress($f);     /* spawn new process */
    $m = merge($l,$r);
  case nil:
    wait($f);
    $m = nil();
  }
}

/* $g = load(A, n) */
/* \length(A) = n, load A[0..n) into forest $g */
/* A should be read-only here; perhaps this should
 * be inlined in sort instead */
forest $g load(int[] A, int n) {
  forest $f = fnil();
  for (int i = 0; i < n; i++) {
    list $l = nil();
    list $l1 = cons(A[i], $l);
    $f = join(1, $l1, $f);
  }
  $g = $f;
}

/* unload($f, A, n) */
/* \length(A) = n, load $f onto A[0..n) */
/* We can write A here, since unload is a function, not process */
void unload(forest $f, int[] A, int n) {
  list $m = compress($f);
```

```
  for (int i = 0, i < n; i++)
    switch ($m) {
    case cons:
      int k = recv($m);
      A[i] = k;
    /* case nil should be impossible */
    }
  switch ($m) {
  /* case cons should be impossible */
  case nil:
    wait($m);
  }
  return;
}

/* sort(A, n), sort A[0..n) in ascending order */
void sort(int[] A, int n) {
  forest $f = load(A, n);       /* create $f */
  unload($f, A, n);             /* consume $f */
  return;
}
```

References

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

[CPT13] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 2013. To appear. Special Issue on Behavioural Types.

[Hon93] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.

[TCP13] Bernardo Toninho, Luíis Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 350–369, Rome, Italy, March 2013. Springer LNCS 7792.

Lecture 26: Obfuscation

15411: Compiler Design
Robbie Harwood and Maxime Serrano

21 November 2013

1 Introduction

We have previously (lecture 20) considered the problem of doing compilation "backwards" (i.e., extracting a reasonable approximation of the original source from its compiled code). However, rather than analyzing our dataflow, we will be going one level beyond, and analyzing our analyses of dataflow.

The ethics of code obfuscation, DRM, and the like is a complicated topic, and we will not discuss it here. Regardless of one's ethical stance on the topic of obscurity, it is important as an proponent to understand the basic ideas behind implementing it, as an opponent to understand it well enough to defeat it, and as an intellectual to learn from it.

As alluded to above, if not the largest than certainly the most controversial application of code obfuscation is as a means to implement copy protection and Digital Rights Management Schemes (DRM). However, on occasion obfuscation can arise more organically. Hand-written assembly, despite being frowned upon as a software development process today, is still alarmingly prevalent within industry. Somewhat unfortunately, the less understandable snippets tend to be the most resistant to replacement, since writing code that performs the same function is difficult.

Finally, it is worth noting that these lecture notes are by no means the final word on obfuscation. Obfuscation is dependent on the workings of the tools used for reverse engineering, and so as they change, so to will the methods used for obfuscation. This lecture will in particular focus on defeating IDA Pro, `objdump`, and `gdb`.

2 Disassembly

The two disassembly tools, whlie they to some degree do a very similar job (converting machine code directly into assembly language) have very different philosophies about how one might do this. This is most likely due to the vastly higher degree of sophistication present in IDA, and to the effort that the authors of IDA make in order to defeat obfuscating compilers.

In a perfect world, disassembly is very similar. Much like an assembler can take every mnemonic to a single sequence of bytes, using a relatively simile one-to-one mapping, a disassembler ought to be able to read the sequence of bytes and decide which instructions map to those bytes. For example, `ret` maps to `C3`, so when a disassembler notices an instruction beginning with `C3`, it should produce a `ret` instruction.

There are, however, a few problems that appear. First of all, in modern executable file formats, the `.text` section that contains program code is also allowed to contain program data. While most program code is easy to disassemble, as long as no effort has been made to make it difficult, program *data* is very unlikely to

disassemble to valid code - much less code that makes any sense. Therefore, modern disassemblers must be able to differentiate code from data, even when the entire block is marked as code.

Here is where the differences between the tools mostly lie. IDA attempts to solve this problem; `objdump` assumes your data is in the various data sections, and that `.text` is entirely made up of code. It should be noted that while in general, IDA makes the correct choice, there are ways to trick it as well. In particular, since the `00 00` bytes form a valid instruction (albeit `add %al, (%eax)`), IDA assumes that it is data. One can use this to trivially trick `objdump` by placing data in the `.text` section.

Both tools assume that a single sequence of bytes will only be executed in exactly one way. This is a wonderful efficiency gain for `objdump` in particular: rather than needing to do recursive-descent disassembly and "pretend" to execute the program, `objdump` can just do a single, straight-line disassembly pass and be done with it.

However, this causes a very significant issue the moment programs stop obeying this restriction. In particular, consider the sequence of instructions:

```
push %rax
xor %rax, %rax
.byte 0x74
.byte 0x01
.byte 0x0f
pop %rax
mov $r, %rax
```

Given this code, `gcc` happily produces the following sequence of bytes:

```
50 48 31 c0 74 01 0f 58 48 c7 c0 03 00 00 00
```

We can then run `objdump` on these bytes, and notice that the following is produced:

```
Disassembly of section .text:

0000000000000000 <.text>
   0:   50                      push %rax
   1:   48 31 c0                xor %rax, %rax
   4:   74 01                   je 0x07
   6:   0f 58 48 c7             addps -0x39(%rax), %xmm1
   a:   c0 03 00                rolb $0x0, (%rbx)
        ...
```

What has happened here is this: the bytes `74 01` lead to a jump that skips the following byte (here, `0f`), which happens to be the first byte of all two-byte opcodes. The presence of the `0f`, however, causes `objdump` to decide that the bytes following the jump must be a mapping From an instruction with a two-byte opcode (in this case, `addps`, a SIMD instruction). As the misalignment constinues (the `48 c7` bytes are the first two bytes of our `mov` instruction from the original assembly, but have been "eaten up" by the SIMD instruction), we continue to produce nonsensical assembly afterwards.

Now, a human reading the assembly can notice by the first three instructions that something fishy is going on, but unless they can read machine code, they will still have to modify the `0f` byte into something more like `90` in order to read correct disassembly.

One might expect IDA, which does not perform straight-line disassembly but rather uses a recursive-descent algorithm, to not be fooled by this trick. HOwever, IDA's recursive descent follows branch-not-taken before it follows branch-taken, and it also assumes that each sequence of bytes can only be disassembled one way. Therefore, when it follows branch-taken (which would produce correct output!) it notices that the target of the jump is marked as disassembled already and hence does not bothered to disassemble it correctly.

It should be noted, however, that the processor has no issues executing this code. Since `xor %rax, %rax` will *always* cause the jump to be triggered, furthermore, it does not execute the potentially fatal SIMD instruction (which dereferences `-0x39(%rax)`, which is probably not a good place).

3 Anti-Analysis

The anti-analysis techniques described here are mostly targeted at IDA Pro, though one could probably also adapt them to target CMU's BAP system.[1].

IDA's analysis capabilities are formidable. With 32-bit code, in fact, they are often (though not always) able to generate C code that preserves the semantics of the original code that was compiled. While the 64-bit version of IDA does not yet have decompilation support, it does have a variety of other analysis tricks up its sleeve.

One thing in particular that is exceedingly useful to reverse engineer is IDA's capacity to recognize function boundaries. However, different compilers tend to produce vastly different function boundary code!

In 32-bit code, the differences are relatively small. In particular, many Microsoft compilers produce code that follows the `stdcall` calling convention rather than the more standard `cdecl` convention, which causes end-of-function code to be more complicated. Similarly, the `fastcall` convention is different depending on platform and compiler.

In 64-bit code, however, there is a significant difference between code produced by compilers that follow the Microsoft calling conventions and the code produced by compilers that follow the System V calling conventions (such as `gcc`, `clang`, and others).

In order to support all of these different conventions, as users of IDA expect it to, IDA uses a variety of heuristics to decide where functions begin and end. One can exploit the characteristics of these heuristics in order to cause IDA to believe that functions end before they do, or simply refuse to believe that they end at all.

Other analyses performed by IDA involve stack variables. In order to confuse IDA here, one can temporarily confuse the stack pointer. As IDA depends on symbolic execution of the code being analyzed, and expects the stack pointer to remain at "sensible" values, this causes the analysis steps to fail entirely. It is possible to produce 32-bit code that fails entirely to be recompiled by IDA, even if the disassembly step succeeds.

Confusing the stack pointer and IDA's function boundary detection can be done by faking a sequence of instructions that are similar to those done on return From a function, and transforming the "normal" return sequence.

For example, the fake sequence:

```
push %rcx
push %rbx
push %rdx
.byte 0xe8
.byte 0x00
.byte 0x00
.byte 0x00
.byte 0x00
pop %rdx
add $08, %rdx
push %rdx
```

[1] More information on BAP can be found at `http://bap.ece.cmu.edu/`

```
ret
.byte 0x0f
pop %rdx
pop %rbx
pop %rcx
```

These bytes `e8 00 00 00` decode to `call $+0`, or simply "`push %rip`" (were such a thing allowed). This confuses IDA's stack pointer, since it resets the "relative" value of `%rsp` (which is all it keeps track of) upon executing a `call` instruction. Following it up with a `pop` causes the sign of the stack pointer adjustment to be something IDA fails to understand.

Further, the presence of the `ret` instruction so soon after a `pop` instruction causes some versions of IDA to believe that this is the end of a function. Unfortunately (or fortunately, depending on your perspective), there does not seem to be such a sequence for all versions.

Afterward, one can modify the real return sequence very slightly. Rather than ending functions with a `ret` instruction, one could end them with, say:

```
pop %r15
pop %r14
pop %r13
pop %r12
pop %rbx
pop %r11
pop %r10
pop %rdi
jmp *%rdi
```

IDA attempts to decide what it believes the indirect jump is, and often labels the function a "chunk" rather than a full function and does not list it in the functions list.

To understand what this means, some understanding of IDA's internals is required. The way the function detection works appears to be as follows:

1. Notice a standard function-beginning sequence.
2. "Evaluate" the code from the start (except backwards edges) until every piece of control flow reaches a function end sequence.
3. If the pieces of the function are spread across memory, and don't end in return statements, label them as "chunks".

If one can cause IDA to recognize a given piece of code as a chunk that has no corresponding function, then it analyzes it and promptly tosses out its analysis.

4 Anti-Debugging

Often, when a reverse engineer cannot figure out the details of the program statically (either due to massive program scale, unreadable assembly, or any of a variety of other factors), he or she will attempt to do so dynamically, observing the program's behavior as it runs. Debuggers such as `gdb` are a critical part of this effort. Therefore, any full anti-reverse-engineering effort would need to have some anti-debugging component.

The primary method for anti-debugging is to abuse the fact that each program can only be debugged by one debugger. To use this, we inserted a `ptrace()` call into the program that causes the program's parent

to attach itself to it as a debugger. If the program already has a debugger attached, this call fails and the program segfaults.

This is perhaps the simplest anti-debugging mechanism, and can be countered by a simple modification of the binary, by very careful use of the debugger, or by interposing the `ptrace()` call. However, in combination with anti-disassembly it is generally enough to delay reverse engineers for a significant amount of time, especially since the `ptrace()` call can be well-hidden.

A more sophisticated trick is as follows:

1. The program begins, and before calling `main()`, calls a *constructor* function `c()`. The list of such functions is held in the `ctors` section, and can be user-controlled.
2. This constructor forks the program. One fork is marked as the child, and the other is marked as the parent.
3. The parent `ptrace()`s the child, and the child `ptrace()`s the parent.
4. Each inserts a critical structure into the memory of the other, or modifies the other's code, or alters the `main()` function to be a no-op, while inserting the address of the "real" main into the `dtors` section to be called at exit.
5. One of the two runs the program as "normal".

This trick (circular debugging) is very difficult to get around. It is, however, also very difficult to implement at the compiler level. In general, mitigating this anti-debugging technique involves writing a custom loader for the target program, and loading the structure into memory oneself. In general, understanding what the structure is entails a full static analysis of the original program, effectively defeating the entire purpose of debugging in the first place.

The last technique is trivial both in implementation and in mitigation: forced breakpoints. The `int 3` instruction, on most platforms, forces a breakpoint. Inserting `int 3` in various locations in the program would cause the reverse engineer to need to manually deal with vastly more breakpoints than desired (especially in tight loops) and hence hinder his or her debugging efforts. However, simply replacing all instances of that instruction with a `nop` instruction trivially defeats this technique, so it is mostly just a nuisance to any reverse engineer.

5 Other techniques

(This section was not reached in lecture since we did a demo.)

5.1 SSA

Obfuscation interacts quite well with SSA (and basic block analysis), since one can then use it to do IR-level hiding of information. To do this, one could add extra unnecessary control flow nodes, without losing *too* much in performance. Further, given the basic block information, one can then reorder blocks in the binary, making reading disassembler output an exercise in scrolling.

5.2 Packers

A technique that has become more prevalent in recent years is the use of packers. The general idea here is to keep a compressed or encrypted version of the binary on the disk ("packed"), as well as a small loader stub which can then decrypt/decompress ("unpack") the rest of the binary in memory. Advanced packers

can even unpack individual functions or basic blocks one at a time, and then re-pack them when they are no longer running. This completely defeats static analysis, as the bytes that make up the target program are quite simply not decipherable until they are unpacked.

Packing is generally defeated by dumping the contents of memory at runtime, though the specific implementation of this technique can make memory dumps much harder. Packing has perhaps the highest return of any anti-reverse-engineering technique, but is itself very time-consuming to write.

5.3 Calling conventions

Per-function calling conventions are the extreme of making human analysis difficult, as then the human would need to keep track of which function takes which convention. From an implementation perspective, however, it requires quite a lot of information be passed around at compile time and instruction selection time to make it workable, and so is less viable than some other techniques.

5.4 Program bugs

The use of bugs in `objdump` and IDA can make them entirely useless. While writing our obfuscating compiler, for instance, we successfully produced a binary that caused `objdump` to segfault upon attempted disassembly. (It's not exploitable as far as we could tell.) However, such transformations can be unreliable in implementation, since they require the construction of decidedly odd corner cases.

6 Want more?

The biggest problem with writing obfuscating compilers is debugging the compilers themselves. In effect, the author is attempting to shoot himself or herself in the foot by preventing debugging while demanding correctness, since determining why correctness was not maintained requires debugging. In effect, it requires the author to mitigate their own anti-reverse-engineering transformations in order to reverse-engineer their own program.

Finally, we should mention that if reversing and code obfuscation are topics you find interesting, the PPP Security Research Group and Hacking Team may be for you. For more information, visit our webiste `http://pwning.net`, sign up for our mailing list, or join our IRC channel `irc://freenode.net/#pwning`. Our faculty advisor is Professor David Brumley; more information on his projects can be found at his website `https://users.ece.cmu.edu/~dbrumley/`.