CMPSCI 630 Systems **Fall 2014**

Lecture 2

Lecturer: Emery Berger *Scribe: Theodore Sudol, Amee Trivedi*

2.1 Advice on Reviews

Reviews from other semesters are available on HotCRP. This can help guide in writing your own reviews, along with pointing out facets of the paper that you may have missed. The summaries should be high-level syntheses, not simple summations. An understanding of the contributions of and concepts within the paper should be shown.

Keep in mind that, for many CS papers, a problem is solved with a novel solution, but the former is forgotten while the latter lives on.

A useful guideline is George Heilmeier's "catechism" or criteria for research projects. The relevant questions for our reviews include:

- What are you trying to do? Use no jargon.
- How is it done today? What are the limitations of current practice?
- What's new in your approach? Why do you think it will be successful?
- Who cares?
- What are the risks and payoffs?
- How long will it take? What are the milestones?

2.2 Compilation and Optimization

Remember that optimization should not change the semantics of a program. If you have some program P that can be optimized into program P', then $\forall x : P(x) = P'(x)$.

However, this is not always straightforward. Many programming languages exhibit undefined behavior, in which a particular syntactic construct has an unknown effect. For example, dereferencing an uninitiated pointer is undefined, and the compiler may take advantage of this and do whatever it wants. Typically "whatever it wants" is something reasonable, but it is not guaranteed to be useful or consistent.

The following statements in C may print anything, as it is undefined behavior:

```
int *p;
printf("%d\n", *p);
```

There are several classic optimizations:

- Strength Reduction: replace an expensive operation with a cheaper one that is semantically identical. Example: Division may be replaced by bit shifts. $8/2 \equiv 8 >> 1$

- Dead Value Elimination: a dead value is a value that does not affect the result of a function; these can be eliminated without changing the function. Dead values are discovered using dependency analysis, which creates an acyclic directed graph of all the statements in the program; any statements outside of the graph can be eliminated. For example:

```
inf fun(int v)              Graph: Return
    int x = 2;                       |
    int y = v + 1                    y
    return y                        / \
                                   v   1
```

  The variable x may be safely eliminated because it is not in the graph.

- Constant Propagation: the values of known constants are substituted into expressions at compile time. These substitutions can enable further optimizations. For example, a function f(x) that returns $x + 1$ can be replaced with $x + 1$ if the value of x is known.

- Function Inlining: small function calls are replaced with their body, which eliminates the overhead (stack allocation, jumps, post-function cleanup, etc.) of the function call.

- Register Allocation: controlling which values are assigned to registers can reduce load and store calls. This optimization can be reduced to a graph coloring problem; the variables in the program are the vertexes, which are connected by which variables need to be used at what times.

- Instruction Selection: different instructions may have the same result, but they may not take the same amount of time. For example, zeroing register `AX` may be accomplished with `MOVE 0,AX`, but `XOR AX,AX` does the same thing more efficiently.

- Tail Call Optimization: if the last statement of a function is a call to itself, the call can essentially be replaced with a GOTO to the beginning of the function. A new stack frame does not need to be allocated, so memory is preserved.

2.3 LISP

Lisp (LISt Processing) has many descendents, such as Scheme, Clojure and Racket. It was originally intended for "Classic" AI, which had the goal of creating intelligence by programming a logical system and describing states and the transformations upon them. (This does not affect their expressive or computational power, as they are Turing complete, as most programming languages are.)

Fortran used simple types, like ints and floats, with no way to express structures. Lisp, on the other hand, focuses on expressiveness and structure with few types. The language is based around mathematics. Managing memory (allocation, deallocation, etc.) is not particularly mathematical, so Lisp doesn't do it either. Instead, automatic memory management handles memory transactions. (This is in contrast to languages with explicit or manual memory management.)

Example Lisp function:

```
(defun time ()
    (prog (output(timeval))
          (time)))
```

This function outputs the current time (`timeval`) and then recurses. Because Lisp does not have tail call optimization, this program will eventually run out of stack space. Each call to `time` (and `timeval`) will create another stack frame, which will never be cleaned up because the function does not return.

2.4 Tangents

2.4.1 History of the Term 'Computer'

"Computer" originally referred to a person who performed calculations. For example, computers were used to decrypt transmissions during World War II. Many early developments in computer science were for "automatic computing" – calculations performed mechanically or electronically.

2.4.2 Brief Overview of Typing

There are three kinds of typing: untyped, static and dynamic. A type describes and confines the values assignable to a variable of that type. Untyped variables are essentially containers that can hold anything. With static typing, you assign types to boxes, and those boxes can only values of that type. With dynamic typing, values have types but boxes do not.

2.4.3 JavaScript and R

JavaScript is essentially the untyped lambda calculus. Which is impressive for a language designed in only 10 days.

R is a dynamically-typed language designed for statistical computing. It has many type coercions that can be confusing. For example: `[3,4,7] + 1` ≡ `[3,5,7] + 1` ≡ `[3,5,7] + [1,1,1] = [4,6.8]` However, `[3,5,7,9,11] + [2,4]` ≡ `[3,5,7,9,11] + [2,4,2,4,2]`. This is allegedly useful for statistical computations.