CS 498 - Program optimization by specialization

- Definition of “specialization”
- Two examples
  - Pattern-matching
  - Interpretation vs. compilation
But first...

Instructor: Sam Kamin

Research interests: programming languages, functional programming, generative programming, educational technology

Class meeting times: TR 9:30-10:45, 1129 SC

Final: None

Course web page: loome.cs.uiuc.edu/CS498F10/

Assignments: Programming; class presentations; term project
Specialization

- *Specialization* = customizing a program based on information not known at initial program-writing time.

- Specialization is an implementation of the $S_{m,n}$ theorem from recursion theory: If there is a program $P$ computing function $f(x, y)$, then for any given $x_0$, there is a program $P_0$ computing function $f_0(y)$, where $f_0(y) = f(x_0, y)$.

- In the proof of the theorem, $P_0$ works by just storing $x_0$ and then invoking the original $P$ on $x_0$ and $y$. However, it may be possible to take advantage of knowing $x_0$ to generate a version of $P_0$ that is more efficient than just invoking $P$. 
Specialization (cont.)

• Specialization involves *program generation* — the specialized program is *generated* — produced by a program, not written by hand — when the new data become available.

• We will mostly be interested in *run-time program generation* (RTPG), because the data for specialization normally is available only at run time. (However, the existing infrastructure for RTPG is very weak.)

• Specialization is an optimization technique distinct from, and orthogonal to, other available techniques: algorithmic/data structure improvements; low-level optimizations; parallel processing — or, rather, it is the technique of using those other techniques after coming into possession of new information.
What this course is about

• Specialization — especially, run-time specialization — is rarely used in practice. Numerous technical problems need to be overcome. This class is part of a research program to explore the potential of this technique, and to study ways to overcome the technical hurdles.

• In the first two classes, we will discuss, via examples, the potential benefits of the technique and some of the difficulties of using it.

• In class 3, I will give a broader overview of this area, and of the course.
Two examples

- Pattern-matching: `match(pat, text)` finds an occurrence of pattern `pat` in text file `text`. It may be advantageous to specialize the matching function for `pat`.

- Interpretation vs. compilation: `exec(prog, input)` interprets execution of program `prog` on given input. It may be advantageous to specialize the exec function for `prog`.
Pattern-matching

We use a simple notion of pattern-matching, where we attempt to find an occurrence of a specific word in a file.

```c
while( fgets (buffer, 2048, fp) != NULL) {
    lineno++;
    linelen = strlen(buffer);
    colnum = 0;
    while (linelen-colnum >= patlen) {
        if (match(argv[1], patlen, buffer, colnum)) {
            printf("Matched at line %d, char %d",
                   lineno, colnum);
            fclose(fp);
            return 0;
        }
        colnum++;
    }
}
```
Pattern-matching (cont.)

Use simple, obvious pattern-matching method:

```c
int match(const char *pat, int patlen,
          char *buff, int buffpos) {
    int i = 0;
    while (i < patlen) {
        if (pat[i] != buff[buffpos+i])
            return 0;
        i++;
    }
    return 1;
}
```
Pattern-matching (cont.)

Match is called repeatedly with the same pattern. Could specialize the match function on that pattern. For example, if pattern is "abcd", match would be:

```c
int match (char* buff, int i) {
    if ('a' != buff[i+0]) return 0;
    if ('b' != buff[i+1]) return 0;
    if ('c' != buff[i+2]) return 0;
    if ('d' != buff[i+3]) return 0;
    return 1;
}
```

Main loop is the same, except that call to match is altered to reflect fewer arguments.
Pattern-matching (cont.)

- Specialized version is much more efficient:

- On one run, with pattern `abcd` and a 1.4 MB input file, got these timing results (measured with Unix `time` command; in seconds):

<table>
<thead>
<tr>
<th></th>
<th>Generic</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>0.167</td>
<td>0.036</td>
</tr>
<tr>
<td>user</td>
<td>0.093</td>
<td>0.046</td>
</tr>
<tr>
<td>sys</td>
<td>0.03</td>
<td>0.015</td>
</tr>
</tbody>
</table>
Here is a program generator for pattern-matchers, using special quote/antiquote syntax:

```c
char* genmatchbody(const char *pat) {
    int j = 0; char *c = $<>$;
    while (pat[j] != '\0') {
        c = $<'(c)
        if ('(C(pat[j])) != buff[i+'(I(j))])
            return 0; >$
        j++;
    }
    return c;
}

char* genmatch(const char *pat) {
    return $< int match (char* buff, int i) {
        '(genmatchbody(pat))
        return 1; } >$
}
```
Pattern-matching (cont.)

```c
char *I(int j) {
    char *r = (char*)malloc(12);
    sprintf(r, "%d", j);
    return r;
}

char *C(char c) {
    char *r = (char*)malloc(4);
    r[0] = '\';
    r[1] = c;
    r[2] = '\';
    r[3] = '\0';
    return r;
}
```

(Note that this code is included only to complete the example. We will be using a different quotation processor for doing examples in this class.)
Language interpretation and compilation

• A well-known example of specialization is language compilation: A compiler can be viewed as a specialization of an interpreter on a specific program.

• A great deal of work has gone into specializing interpreters automatically; this is thought to be useful because interpreters are easier to write than compilers.

• For this example, I wrote an interpreter for a very simple language in OCaml, and hand-specialized it on a program.
Interpreter, part 1

define type stmt =
    | Asgn of string * exp
    | Seq of stmt list
    | While of exp * stmt

and exp =
    | Int of int
    | Var of string
    | Plus of exp * exp
    | Minus of exp * exp

define type state = (string * (int ref)) list;

let rec lookup x sigma =
    if sigma = [] then []
    else if x = fst(hd sigma)
        then sigma
        else lookup x (tl sigma);

let fetch x sigma =
    let r = lookup x sigma
    in if r = [] then 0 else !(snd (hd r));;

let assign' x i sigma =
    let r = lookup x sigma
    in if r = [] then false
        else ((snd (hd r) := i); true);

let assign x i sigma =
    let found = assign' x i sigma
    in if found then sigma else (x, ref i) :: sigma;
let rec exec s sigma = match s with
    (Asgn (x,e)) -> assign x (eval e sigma) sigma
    | (While (e,s')) -> if eval e sigma = 0
                  then sigma
                  else exec s (exec s' sigma)
    | (Seq ss) -> execlist ss sigma

and execlist ss sigma = if ss = [] then sigma
                        else execlist (tl ss) (exec (hd ss) sigma)

and eval e sigma = match e with
    (Int i) -> i
    | (Var x) -> fetch x sigma
    | (Plus(e1, e2)) -> (eval e1 sigma) + (eval e2 sigma)
    | (Minus(e1, e2)) -> (eval e1 sigma) - (eval e2 sigma);;
Running the interpreter

- Our interpreter runs programs in “abstract syntax.” Suppose we want to run this program:

\[
\begin{align*}
&z = x; \quad s = 0; \\
&\text{while } (z \neq 0) \{ \\
&\quad s = s+y; \quad z = z-1; \\
&\}\end{align*}
\]

- It would be rendered in abstract syntax as:

```plaintext
let s1 = Seq [Asgn("z", Var "x");
Asgn("s", Int 0);
While (Var "z",
 Seq [Asgn("s", Plus(Var "s", Var "y"));
Asgn("z", Minus(Var "z", Int 1))]);
```

August 24, 2010
Running the interpreter

- We provide inputs to the program by creating an initial state, e.g.
  \texttt{exec s1 (assign "x" 3 (assign "y" 4 [])}).

- Wrapping this in a timing function, and choosing values large enough to take a measurable amount of time:

  \begin{verbatim}
  let time f x =
      let t = Sys.time() in
      let result = f x in
      Printf.printf "Took %fs\n" (Sys.time() -. t);
      result;;
  
  time (exec s1) (assign "x" 1000000 (assign "y" 4 []));;
  \end{verbatim}

  we find this takes 4.75 seconds.
Specializing the interpreter

- We have a classic specialization situation: The program \((\text{exec})\) has two inputs. One of them (the program, \(s_1\)) remains the same for the entire execution, and the other (\(\sigma\)) changes continually. Specializing \(\text{exec}\) on \(s_1\) should lead to faster execution.

- Here we specialize by hand, in a couple of steps. The idea is to create a function \(\text{spec}\) which combines \(\text{exec}\) and \(s_1\), so that the result is a function just of the state.
Specializing the interpreter

First, create a specialized function for exec applied to the while loop of s1:

```ocaml
let s2 = Seq [Asgn("s", Int 0); Asgn("z", Var "x")];;
let s3 = Seq [Asgn("s", Plus(Var "s", Var "y"));
             Asgn("z", Minus(Var "z", Int 1))];;

let rec execwhile sigma =
  if (fetch "z" sigma = 0) then sigma
  else execwhile (exec s3 sigma);;
let spec2 sigma = execwhile (exec s2 sigma)
time spec2 (assign "x" 1000000 (assign "y" 4 []));;
```

This doesn’t help much: runs in 4.61 seconds.
We complete the specialization by “unfolding” calls to exec and eval, as well as fetch and assign, yielding:

```ocaml
let rec execwhile4 sigma =
    let sigma' =
        (snd (hd sigma) :=
            (!snd (hd sigma)) + !snd (hd (tl (tl (tl sigma)))));
        sigma)
    in let sigma'' =
        ((snd (hd (tl sigma'))) := (!snd (hd (tl sigma'))) - 1);
        sigma')
    in execwhile4 sigma'';
let spec5 sigma = execwhile4 (assign "s" 0
    (assign "z" (eval (Var "x") sigma) sigma));
time spec5 (assign "x" 1000000 (assign "y" 4 []));
```

This yields a significant speed-up: 0.75 seconds.
Specializing the interpreter (cont.)

The savings by “compiling,” generally calculated as an order of magnitude (6.3 here) is called the “cost of interpretation.” In effect, it is what we are always trying to eliminate when specializing.
Automatic specialization

• In these two examples, we can easily imagine the specialization being done automatically. A program that does automatic specialization is called a *partial evaluator*.

• These two examples are within the power of existing partial evaluators. (In particular, a great deal of work has gone into studying partial evaluation of interpreters.)

• In this class, we will not use a partial evaluator, because we will need to control exactly what the specialized code looks like. (Existing partial evaluators *can* produce the specializations above, but it is difficult to be sure that they *will*.)

• Later in the semester, we will discuss partial evaluation.
Auto-tuning

- When we attempt to optimize programs by specialization, we run into the problem that a specialized program that runs well on one machine will not necessarily run well on another.

- A particular factor that distinguishes one machine from another is its memory system. We can fairly well predict when one program will have a lower overall instruction count than another, and this is generally fairly stable from one machine to another, but memory systems vary widely and can have a very strong impact on performance.

- This is not actually a problem specific to specialization; normal programs and optimizing compilers deal with the same problem.
Auto-tuning (cont.)

• Furthermore, not only are different, equivalent programs optimal for different machines, it is very difficult to predict performance, even for a single, fixed machine.

• For this reason, some library designers have written code libraries that tune themselves at the time they are installed on a target machine. That is, they run some tests at install time, which then determine what is the best thing to do when the program executes. This is called auto-tuning, or empirical optimization.

• When using program generation, the variation among machines may be even more of a problem, because generated programs are usually larger than hand-written programs, and therefore lead to more memory performance problems.
Next class

- The question we will discuss in this class is: Can we always — or almost always — produce faster programs by specialization, in those cases where specialization is applicable?

- Next class: Examples where specialization is applicable, but simple approaches do not yield speed-ups.