

CS 536

Final Exam

Saturday, May 17, 2008

7:45 AM— 9:45 AM

1325 Computer Science

Instructions

Answer any *four* questions. (If you answer more, only the first four will count.) Each question is worth 25 points. Please try to make your answers neat and coherent. Remember, if we can't read it, it's wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. Assume that we add a conditional expression of the form

$(\text{Expr}_1 ? \text{Expr}_2 : \text{Expr}_3)$

to CSX.

Expr_1 is an expression that returns a `bool`. If Expr_1 is true, expression Expr_2 is evaluated, and its value is the value of the conditional expression. If Expr_1 is false, expression Expr_3 is evaluated, and its value is the value of the conditional expression.

Outline the changes that would be needed in a CSX type-checker and code generator to handle conditional expressions. Illustrate your answer using the following example:

```
a = (i != 0 ? j/i : 0);
```

2. (a) (20 points)

In CSX we type check a call to method M by first type checking M 's declaration. Then the actual parameters in the call to M are type checked. Finally, the number, type and kind of parameters found in the call are compared with the number, type and kind of parameters specified in M 's declaration.

Consider the following alternative. In a declaration of a method P , *no types* are given to P 's parameters; they are simply given names. For example,

```
int P(a,b,c) { ... }
```

When a call to P is found, the parameters in the call are type checked (as usual) and then these types are used as *definitions* of the types of P 's parameters.

What changes are needed in your type checking of method declarations and calls to implement this change?

- (b) (5 points)

A potential difficulty in the approach suggested in part (a) is that different calls to P may differ in the types used for a particular parameter. How would you handle a non-unique type for a parameter in different calls to P ?

3. Assume we add a new operator to CSX, the *delay* operator. The expression `delay expr` returns a *suspended* execution of `expr` (essentially a function of zero arguments whose body is `return expr`). The operation `force suspension` forces the given suspension (created by a `delay` operation) to be evaluated, returning the value of the suspended expression. For example,
- ```
s=delay delete_file(f);
```
- creates a suspended call to `delete_file` with parameter `f`. The `delete_file` operation is not done yet, it is just “set up.” At some later time the suspension created by the `delay` may be activated by using the `force` operation. Then the `delete_file` operation is performed. For example,
- ```
r=force s;
```
- forces the suspended `delete_file(f)` operation to run, returning its result to `r`.

Explain how you would implement `delay` and `force` in CSX. How can you be sure that all the variables visible when the `delay` was executed are still accessible when the `force` is executed?

4. Assume we are translating a function or procedure call in CSX. Explain how we could determine the exact number of stack locations that will be needed to evaluate and store (on the stack) the parameters for the call. Illustrate your technique on the following call:

```
p(a+g(1,2), x-(y-(p+1)));
```

5. A common compiler optimization is *inlining*. A call to a method `M` is inlined when the call is replaced with the body of `M`, with formal parameters initialized to the actual parameters used in the call.

What are the advantages of inlining? What are the disadvantages of inlining? Under what circumstances should inlining *not* be performed?

Given a CSX program in AST form, how would you inline a particular call of some method `M`?

6. (a) (10 points)

Code for an if statement is generated on the assumption that the value of the control expression will not be known until run-time. But in some cases the value of the control expression *is* known at compile-time. The simplest such case is when the control expression is just the boolean literal `true` or `false`.

What changes would you make in your CSX code generator for if statements to handle the special case of a control expression that is either the literal `true` or the literal `false`?

(b) (5 points)

The special case handled in part (a) is uncommon. More common is the case in which the control expression is an identifier declared to be a boolean constant with a literal initializer. For example,

```
const debug = true;
...
if (debug) ...
```

What changes are needed to your solution to part (a) to include the case of identifiers declared to be boolean constants?

(c) (10 points)

It may occur that the control expression of an if statement contains boolean operators (&&, ||, !) whose operands are all boolean literals or boolean constants with literal initializers. For example,

```
const debug1 = true;
const debug2 = false;
...
if (debug1 || debug2) ...
```

What changes are needed to your solution to part (b) to include the case of boolean operators whose operands are all boolean literals or boolean constants?