CS 565, Jens Palsberg | Dec 17, 1998

Final Exam + Qualifying Exam

Submit your solutions on separate paper (don't submit this handout). Write your name and id number on the top of the first page you submit. I will stable your pages together. This exam is intended to take you 2 hours. Budget your time wisely. There are 4 questions on the exam. Each question is worth 5 points, for a total of 20 points maximum. Write the solutions for different questions on different sheets of paper. Make clear what problem each solution is for. I have spare paper, just ask for it.

To those who are taking the qualifying exam: to pass the exam you must get an excellent score.

1. [Simple Types] Consider the following λ-term:

$$F = ((\lambda x.\lambda y.\lambda z.(x + (y\ z)))\ 5)$$

I have used + in infix notation, and I have used the standard convention that the body of a λ-abstraction extends "as far as possible." Write down a constraint system for F which captures typability with simple types. Decide if F has a simple type or not by doing unification of the constraints. In case F has a simple type, then show its principal type.

2. [Fast Flow Analysis] Consider the following λ-term:

$$G = ((\lambda^1 f.\lambda^2 x.f x)(\lambda^3 y.y))\ (\lambda^4 a.a)$$

Show the graph for G that is used by the Heintze/McAllester $O(n^2)$ time flow analysis algorithm. What is the label set for G produced by the algorithm?

3. [Flow Analysis] Let us define a family of Java programs $P_1, \ldots, P_n, \ldots$, where P_n consists of (see below):

- the interface `I` and the classes `S` and `T`, and
- $2n$ classes named `F1`,...,Fn, `B1`,...,Bn, where, for example, `B2` is obtained from `B1` by renaming all occurrences of `B1` to `B2`, and renaming all occurrences of `F1` to `F2`.

Show in reasonable detail how the $O(n^3)$ time flow analysis algorithm proceeds on P_n. What is the running time of the algorithm on P_n?

```
interface I {
  I id(I a);
}

class S implements I {
  public I id(I a) { return a; }
}

class T implements I {
  public I id(I a) { return a; }
}

class F1 implements I {
  public I id(I a) { return a; }
}

class B1 implements I {
  I x;
  I y;
  public I id(I a) {
    x = new B1().id(new S().id(new F1()));
    y = new T().id(new B1()).id(new F1());
    return a;
  }
}
```

4. [**Interpreter**] Here is the grammar for a subset of Scheme extended with expressions of the forms `throw` and `(try` e_1 `catch` e_2 `)`. The meaning of these expressions is essentially like in Java, except that `throw` takes no argument, so there is only one type of exception, and `catch` catches *all* exceptions. Write in Scheme an interpreter for this language.

Program	::=	*Expression*
Expression	::=	*IntegerLiteral*
	|	*Identifier*
	|	*ProcedureExp*
	|	*Application*
	|	*ThrowExp*
	|	*CatchExp*
IntegerLiteral	::=	*INTEGER_LITERAL*
Identifier	::=	*IDENTIFIER*
ProcedureExp	::=	`(` `lambda` `(` *Identifier* `)` *Expression* `)`
Application	::=	`(` *Expression Expression* `)`
ThrowExp	::=	`throw`
CatchExp	::=	`(` `try` *Expression* `catch` *Expression* `)`

Use the following definitions to represent nodes in syntax trees.

```
(define-record Program (Expression))
(define-record IntegerLiteral (Token))
(define-record Identifier (Token))
(define-record ProcedureExp (Token1 Token2 Token3
   Identifier Token4 Expression Token5))
(define-record Application (Token1 Expression1 Expression2
   Token2))
(define-record ThrowExp (Token))
(define-record CatchExp (Token1 Token2 Expression1
   Token3 Expression2 Token4))
```