# CS 441 Programming Languages
# Final

Prof. David Walker

This final should be the individual work of each student in the class. Do not talk to anyone other than myself (David Walker) or Rob Dockins about the questions on this final. If neither I nor Rob are available while you are taking the exam and you have a question, make a reasonable assumption, write down that assumption clearly and continue working. Talking to anyone else about this exam while you are taking it constitutes a violation of Princeton's code of academic integrity. You may consult your lecture notes, any of the course textbooks, any of the course web pages, slides, assignments, mailing list posts, etc. Do not search for the answers on the general Web (not that you'd find any).

You will work on the exam during a single, continuous 24-hour period occurring between Thursday Jan 8th and Tuesday Jan 14th at 4:30pm, 2009. You must complete the exam in the 24-hour period of time. At the top of the exam, write down the time you download the exam and the time you hand it in. Please also sign your name and write: "This exam represents my own work in accordance with University regulations." Hand in exams by slipping them under Rob's office door or by submitting them electronically via email to rdockins@cs.princeton.edu. The time you spend walking to Rob's office to hand in the exam does not count as exam time so you can feel free to do the exam at home, stop when the five hours are up, write your completion time at the top of the exam and walk it over at your leisure.

Reminders:

- Read questions carefully and completely before beginning your response. Part of the test is whether or not you are able to read and understand the questions, including the formal inference rules.
- Points will also be deducted for proofs that are unclear or poorly structured.
- Always use the exact syntax of expressions, types, judgments, etc. that you are given in a question, or clearly define the abbreviations that you're using. When in doubt, avoid abbreviations. Definitely do not just start using some new, informal notation without defining it – graders will not be able to figure out your intent.
- When writing out your proofs, use plenty of space (either electronically or on paper) to make them easy to read. One of the best ways is to format each line with one true statement (judgment) on the left and the justification for that statement on the right (in terms of earlier statements and inference rules, etc.).
- You might find some of the questions quite difficult – try not to get stressed out. Simply move on to trying to solve a different question. Don't spend all your time just trying to answer one or two questions. Partial credit will be given where appropriate when you give partial answers.

The exam is out of 35. Good luck!

## Short Answers

**Q. 1** [1 point] Explain in a sentence or two what "unification" is.

**Q. 2** [2 points] Explain in a paragraph, with help of an example, roughly how the "stack inspection" algorithm works.

**Q. 3** [1 point] Give two different stimulae that can cause a transient hardware fault.

**Q. 4** [1 point] When proving type safety for the simply-typed lambda calculus, one uses an Exchange Lemma. In an English sentence or two, explain what the Exchange Lemma says. (You can give the mathematical statement of the lemma, but I also what you to explain in a sentence what the math means. What is the "essence" of the lemma?)

**Q. 5** [2 points] Explain the Curry-Howard Isomorphism in a sentence or two. Enrich your answer by discussing the concepts involved in the context of the function $\lambda x{:}\tau.(x, x)$, which takes an argument and returns a pair. Please refer to the notion of a pair and the notion of a function in your answer.

**Q. 6** [2 points] Consider the following SML signature:

```
sig
  type key
  type map
  val compare : (key * key) -> bool
  val empty : map
  val insert : key -> int -> map
  val fold : map -> 'a -> (('a * int) -> 'a) -> 'a
end
```

Represent this signature using only pair, unit, sum, void, universal polymorphic, existential polymorphic, recursive, function, int and bool types (from the lambda calculus).

**Q. 7** [1 point] Consider the following SML datatype:

```
datatype crazy =
  Nuts of bool
| Bolts of int * crazy
```

Represent this datatype using only pair, unit, sum, void, universal polymorphic, existential polymorphic, recursive, function, int and bool types (from the lambda calculus).

**Q. 8** [1 point] Assuming you have done the encoding of the type `crazy` from Q. 7 above, give a lambda calculus *expression* (using unit, pairs, sum expressions, etc.) that corresponds to:

```
Bolts (3, Nuts true)
```

The expression you give should have the type that you used in Q. 7 to encode the type `crazy`.

# Type Inference

**Q. 9** [3 points] Consider each of the following unannotated expressions. If the type inference algorithm discussed in class can find a type for each expression, give the type. The type you give should be the most general type possible. It can include type variables, function types, int and bool types. If the type inference algorithm fails, briefly explain why it fails.

(a) `fun f (x) = if x then 1 else f x`

(b) `fun g (y) = (fun f (x) = if g x then 1 else g (f x))`

(c) `fun g (y) = (fun f (x) = if y x then x 1 else f x)`

**Q. 10** [1 point] Give an example of an expression for which type inference does not succeed because of the "occurs check." Give a brief English explanation of your example.

# A Language of Multi-Sets

The following questions are about a very, very simple typed language with integers and a primitive notion of multi-sets. A multi-set is exactly like a set, except a multi-set may contain repeated identical elements in it. For example, $\{1, 5, 1, 1, 3\}$ is a multi-set with three occurrences of "1", one occurrence of "5" and one occurrence of "3". We write $\{\ \}$ for the empty multiset.

The language itself allows you to create a set by simply writing down a set value as $\{1, 2, 3, 4\}$. You may also use the set union operator ($e_1\ \cup\ e_2$) to create a bigger set from smaller ones. A let expression allows you to bind variables to sets or their elements. Here is an example of a larger program:

```
let x = 1 in
let y = 2 in
let z = {3} in
({x} U {y}) U z
```

Here is the syntax of the language:

| | | | |
|---|---|---|---|
| *types* | $\tau$ | ::= | $\texttt{int} \mid \tau\ \texttt{set}$ |
| *typing contexts* | $\Gamma$ | ::= | $\cdot \mid \Gamma, x{:}\tau$ |
| *numbers* | $n$ | ::= | $0 \mid 1 \mid 2 \mid ...$ |
| *set values* | $s$ | ::= | $\{v_1, \ldots, v_k\}$ |
| *values* | $v$ | ::= | $n \mid s$ |
| *expressions* | $e$ | ::= | $x \mid v \mid \{e\} \mid e_1\ \cup\ e_2 \mid \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$ |

The operational semantics of the language is defined as follows:

$$\frac{e_1 \longrightarrow e_1'}{\{e_1\} \longrightarrow \{e_1'\}}\ \text{(OS-single)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ \cup\ e_2 \longrightarrow e_1'\ \cup\ e_2}\ \text{(OS-U1)} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ \cup\ e_2 \longrightarrow v_1\ \cup\ e_2'}\ \text{(OS-U2)}$$

$$\frac{}{\{v_1, \ldots, v_j\}\ \cup\ \{v_{j+1}, \ldots, v_k\} \longrightarrow \{v_1, \ldots, v_j, v_{j+1}, \ldots, v_k\}}\ \text{(OS-U3)}$$

$$\frac{e_1 \longrightarrow e_1'}{\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x\ =\ e_1'\ \texttt{in}\ e_2}\ \text{(OS-let1)} \qquad \frac{}{\texttt{let}\ x\ =\ v\ \texttt{in}\ e_2 \longrightarrow e_2[v/x]}\ \text{(OS-let2)}$$

And here are the typing rules:

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ (T-var)}$$

$$\frac{}{\Gamma \vdash n : \texttt{int}} \text{ (T-int)} \qquad \frac{\text{for } i : 1 \ldots k, \Gamma \vdash v_i : \tau}{\Gamma \vdash \{v_1, \ldots, v_k\} : \tau \texttt{ set}} \text{ (T-setval)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \tau \texttt{ set}} \text{ (T-single)}$$

$$\frac{\Gamma \vdash e_1 : \tau \texttt{ set} \qquad \Gamma \vdash e_2 : \tau \texttt{ set}}{\Gamma \vdash e_1 \ \cup \ e_2 : \tau \texttt{ set}} \text{ (T-union)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \ = \ e_1 \texttt{ in } e_2 : \tau_2} \text{ (T-let)}$$

**Q. 11** [2 points] Write down (but do not show any steps of the proof of) the statement of the *Canonical Forms Lemma*, as you would need it to prove a type safety theorem for this language.

**Q. 12** [1 point] Write down (but do not show any steps of the proof of) the statement of the (Value) *Substitution Lemma*, as you would need it to prove a type safety theorem for this language.

**Q. 13** [3 points] Assuming you have proven the lemmas you wrote down in Q. 10 and Q. 11, prove the Type Preservation lemma. In other words, prove:

**Type Preservation**: If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$.

You must do your proof by induction on the derivation of $e \longrightarrow e'$. In particular, you must do the cases of the proof involving operational rule (OS-U2) and operational rule (OS-let2) (you may omit the other cases). As usual in this class, you should do a well-structured, 2-column proof with valid judgements on the left and explanations on the right. Here is a proof outline you can start with:

case:

$$\frac{e_2 \longrightarrow e_2'}{v_1 \ \cup \ e_2 \longrightarrow v_1 \ \cup \ e_2'} \text{ (OS-U2)}$$

(1) $\cdot \vdash v_1 \ \cup \ e_2 : \tau$ (By Assumption)

$\vdots$

case:

$$\frac{}{\texttt{let } x \ = \ v \texttt{ in } e_2 \longrightarrow e_2[v/x]} \text{ (OS-let2)}$$

(1) $\cdot \vdash \texttt{let } x \ = \ v \texttt{ in } e_2 : \tau_2$ (By Assumption)

$\vdots$

**Q. 14** [3 points] Assuming you have proven the lemmas you wrote down in Q. 10 and Q. 11, prove the Progress lemma. In other words, prove:

**Progress**: If $\cdot \vdash e : \tau$ then either (a) $e$ is a value, or (b) $e \longrightarrow e'$, for some expression $e'$.

You must do your proof by induction on the derivation of $\cdot \vdash e : \tau$. In particular, you must do the cases of the proof involving typing rule (T-single) and typing rule (T-union) (you may omit the other cases). (Unlike in the previous question, I'm not giving you a proof outline to start from, but nevertheless use good form in the structure of your proof.)

**Q. 15** [2 points] Suppose the operational rules for our language are modified so that this rule:

$$\frac{e_1 \longrightarrow e_1'}{\{e_1\} \longrightarrow \{e_1'\}} \text{ (OS-single)}$$

is removed and replaced by this rule:

$$\frac{e_1 \longrightarrow e_1'}{\{e_1\} \longrightarrow e_1'} \text{ (OS-single2)}$$

Now for each statement below write "yes" if you agree with the statement. Write "no" if you disagree with the statement. Also write a brief explanation (a sentence or two or a counter-example) to explain why you said what you said.

(a) [1 point] The progress lemma is not true in the new type system.

(b) [1 point] The preservation lemma is not true in the new type system.

**Q. 16** [8 points] In a file named `ms.sml`, implement the syntax and the typing rules for the language. In order to implement the syntax, create your own ML datatype. Assuming your datatype is named `exp`, to implement the typing rules, you should write a function `typecheck` with the following type:

```
typecheck :  exp -> bool
```

Your function should return true if the expression has a type and false otherwise. Your function should not throw any exceptions. (You may use exceptions internally if you would like. However, they should not escape the scope of the function.)

In addition to implementing the function `typecheck`, provide a variety of test cases that show that your function operates correctly. (Alternatively, you could implement a random expression generator like the ones Rob implemented in the homeworks, but that is not necessary.) Your code should be well-commented so a grader can understand what it is doing. In particular, give the grader clear instructions at the top of the file for using your testing code to verify your implementation. In this question, you may reuse any code you find useful on the course website or any code you developed while doing your assignments. Even if you are handing in a paper copy of the exam, be sure to email rob your answer to this question.

# One Last Collection of Inductive Definitions!

Consider the following familiar syntactic definitions of lists and natural numbers:

$$\begin{array}{lllll} \textit{numbers} & n & ::= & Z \mid S\ n \\ \textit{integer lists} & l & ::= & \texttt{nil} \mid n :: l \end{array}$$

Assume that $Z$ represents the natural number zero and $S\ n$ represents $k+1$ if $n$ represents the number $k$. Also assume that `nil` represents the empty list and $n :: l$ represents a list with $n$ on the front followed by the list $l$. Consider the following inductive definitions, which define the relations `zowy` $l\ n$ and `zoink` $l_1\ l_2$.

$$\frac{}{\texttt{zowy nil } Z} \qquad \frac{\texttt{zowy } l\ n_1}{\texttt{zowy } (n :: l)\ (S\ n_1)}$$

$$\frac{}{\texttt{zoink nil nil}} \qquad \frac{\texttt{zoink } l_1\ l_2}{\texttt{zoink } (n_1 :: l_1)\ ((S\ (S\ n_1)) :: l_2)}$$

**Q. 17** [1 point] In English, what do the relations `zoink` and `zowy` do?

**Q. 18** [3 points] Prove that if `zowy` $l_1$ $n_1$ and `zoink` $l_1$ $l_2$ then `zowy` $l_2$ $n_1$.