CS411 Prelim 1 Solutions

A. Demers

15 Mar 2001

Question 1: Consider a language of expressions over two types:

$$\tau \quad ::= \quad \text{int} \mid \text{real}$$

$n \in \mathbf{N}$ integers 1, 2, 3, ...
$r \in \mathbf{R}$ reals 3.1415926, $6.02E23$, ...

$$e \quad ::= \quad n \mid r \mid (e_1 \theta e_2)$$

Semantically there are two versions of each operator:

$$\theta_N \text{ is } N \times N \to N \qquad \text{and} \qquad \theta_R \text{ is } R \times R \to R$$

In source programs both versions of the operator are represented by the same symbol θ. No (implicit or explicit) type conversions or mixed-mode arithmetic are supported.

(a) Give a set of typing rules for this language.

(solution a) There are the natural axioms for int and real constants:

$$\frac{}{\vdash n : \text{int}} \qquad \frac{}{\vdash r : \text{real}}$$

and for each operator θ there are two rules that precisely reflect the types of the two versions of the operator θ_N and θ_R:

$$\frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash (e_1 \theta e_2) : \text{int}} \qquad \frac{\vdash e_1 : \text{real} \quad \vdash e_2 : \text{real}}{\vdash (e_1 \theta e_2) : \text{real}}$$

There are no "mixed mode" rules.

(b) Give examples of correctly typed int and real expressions, and an example of a type-incorrect expression.

(solution b) Of course the constants are correctly typed, so

$$\vdash 1 : \text{int} \qquad\qquad \vdash 3.1415926 : \text{real}$$

No mixed mode rules exist, so the expression

$$1 + 3.1415926$$

is type-incorrect.

(c) We now introduce the subtype relation int $\prec$ real, and add the simple type checking rule

$$\frac{\vdash e : \text{ int}}{\vdash e : \text{ real}}$$

to reflect the subtype relation. Using this rule, give an expression that can be typed as either int or real. Give another expression that can be typed only as real, but in at least two different ways.

(solution c) The subtype rule above explicitly allows any int expression to be "promoted" to real; thus even an integer constant expression will do:

$$\vdash 1 : \text{int} \qquad\qquad \vdash 1 : \text{real}$$

works fine as a solution to the first part. For a real expression that can be typed in two ways, consider (1.0+(1/3)). This expression necessarily must be given type real because the left operand 1.0 is a real constant. Thus, the right subexpression (1/3) must be promoted to real. This can be done in two ways:

$$\frac{\begin{array}{cc}\vdots & \\ \vdash 1 : \text{real} & \vdash (3) : \text{real}\end{array}}{\vdash 1/3 : \text{real}} \qquad \text{or} \qquad \frac{\begin{array}{cc}\vdots & \\ \vdash 1 : \text{int} & \vdash (3) : \text{int}\end{array}}{\begin{array}{c}\vdash (1/3) : \text{int} \\ \hline \vdash 1/3 : \text{real}\end{array}}$$

Note the first case promotes the operands of / to real before performing the division; the second case does int division and then promotes the int result to real.

(d) Assuming the evaluation rules always use whichever one of θ_N or θ_R is required by the type of the expression being evaluated, use your second example from part (c) to show the semantics is not *coherent* – that is, that the value of a type correct expression can depend on how the type annotation was derived.

(solution d) In the first typing of (1.0+(1/3)) above, the division (1/3) is performed in the real type, so the quotient is $0.333\ldots$ and the entire expression evaluates to $1.333\ldots$. In the second typing, the division is performed in the int type, so the int quotient is 0 whether rounding or truncation is used, and the entire expression evaluates to 1.0.

(e) Fix the problem identified in part (d). That is, give a revised set of type rules for this language that guarantee unique typing. (Your rules should guarantee unique typing, but you don't need to prove they do so. Also, your rules needn't (indeed, can't) produce *every* possible typing, as long as they assign *some* valid type to every legal expression.)

(solution e) Given that there are only two types in this language, this question sounds a lot harder than it is. Just remove the subtyping rule, and add mixed mode computation rules that promote expressions to real only when forced to – that is, only when at least one of the operands is forced to be real. This introduces two new rules for each operator:

$$\frac{\vdash e_1 : \text{int} \qquad \vdash e_2 : \text{real}}{\vdash (e_1 \theta e_2) : \text{real}} \qquad\qquad \frac{\vdash e_1 : \text{real} \qquad \vdash e_2 : \text{int}}{\vdash (e_1 \theta e_2) : \text{real}}$$

This reflects the choice that the quotient of two int expressions will be evaluated as an int (thus do truncation or rounding) rather than as a real. Since, as we showed in part d, different typings can produce different values, it is inescapable that we make some such choice. □

Question 2: Consider the following three integer expression languages L_{par}, L_{nd} and L_{orcl}. All three languages include the following expression constructors:

$$e \quad ::= \quad n \mid (e_1; e_2) \mid (e_1 \theta e_2) \mid (a \leftarrow e_1) \mid (a \uparrow)$$

with the interpretations we usually give to these constructors. Note in particular the presence of assignable variables (side-effects). Each language has one additional distinguishing constructor. These are:

$$L_{par}: \qquad e \quad ::= \quad \mathbf{par}(e_1, e_2)$$

The intended meaning is to evaluate e_1 and e_2 in parallel, returning the result value of e_2.

$$L_{nd}: \quad e \ ::= \ \mathbf{nd}(e_1, e_2)$$

The intended meaning is nondeterministically to choose to evaluate and return either e_1 or e_2 (but not both).

$$L_{orcl}: \quad e \ ::= \ \mathbf{orcl}(e_1, e_2)$$

The intended meaning is to evaluate whichever one of e_1 and e_2 will return the value 0 (but not both). It is as if an oracle tells you in advance what the result of an expression evaluation will be – hence the language name. If neither e_1 nor e_2 can evaluate to 0, the evaluation fails; if both e_1 and e_2 evaluate to 0, one expression is chosen arbitrarily.

Now, given these three languages,

(a) For which of the languages would LS (large step) evaluation rules be appropriate? For which would SS (small step) rules be appropriate? For which would either style work? Justify your answers briefly.

(solution a) We discussed the equivalent of L_{par} in lecture to motivate SS rules – since the language includes assignment, concurrent expression evaluations could interfere with one another through their effects on the store, and a small step semantics is required to expose intermediate states.

Rules for L_{nd} could be either SS or LS – the evaluation rules for $\mathbf{nd}(e_1, e_2)$ are essentially the same as those for **if** with the hypotheses about the condition value eliminated. This can be expressed with either SS or LS rules.

For L_{orcl}, note that choosing to evaluate the first subexpression of an **orcl** construct requires a hypothesis of the form

$$\langle e_1, \phi, \sigma \rangle \ \rightarrow \ \langle 0, \sigma' \rangle$$

giving the result of a complete evaluation of e_1. Such a hypothesis can be expressed in LS rules but not in SS rules.

(b) Show how to simulate a conditional expression

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$$

in L_{orcl}. You may use assignments and as many new variable names as you wish to implement your simulation, and you may use whatever arithmetic operators you need. As usual assume e_1 nonzero for **true**, zero for **false**. Hint: note the expression

$$\textbf{orcl}(\ 1/0,\ 0\ )$$

reduces to 0 in spite of the division by zero in e_1.

(solution b) We introduce two variables, c to hold the value of the condition part and a to hold the answer.

$$\begin{array}{l} c \leftarrow e_0\,; \\ \textbf{orcl}(\ ((a \leftarrow e_2)\,;(1 - c/c)),\ \ ((a \leftarrow e_3)\,;c)\ )\,; \\ a \end{array}$$

(c) Give either LS or SS evaluation rules for L_{orcl}.

(solution c) By part (a), these will be LS rules. Since there are no binding constructs, no environment is required. The store is as usual

$$\sigma = \{\ldots\ a_i \sim n_i\ \ldots\}$$

and judgements look like

$$\langle e, \sigma \rangle\ \rightarrow\ \langle n, \sigma' \rangle$$

Rules for the common constructors are completely vanilla:

$$\frac{}{\langle n, \sigma \rangle\ \rightarrow\ \langle n, \sigma \rangle}$$

$$\frac{\begin{array}{l}\langle e_1, \sigma \rangle\ \rightarrow\ \langle n_1, \sigma'' \rangle \\ \langle e_2, \sigma'' \rangle\ \rightarrow\ \langle n_2, \sigma' \rangle\end{array}}{\langle (e_1; e_2), \sigma \rangle\ \rightarrow\ \langle n_2, \sigma' \rangle}$$

$$\frac{\langle e_1, \sigma\rangle \;\rightarrow\; \langle n_1, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \;\rightarrow\; \langle n_2, \sigma'\rangle}{\langle (e_1 \theta e_2), \sigma\rangle \;\rightarrow\; \langle n, \sigma'\rangle} \quad \text{where } n = (n_1 \theta n_2)$$

$$\frac{\langle e_1, \sigma\rangle \;\rightarrow\; \langle n_1, \sigma'\rangle}{\langle (a \leftarrow e_1), \sigma\rangle \;\rightarrow\; \langle n_1, \sigma' \oplus \{a \sim n_1\}\rangle}$$

$$\frac{}{\langle (a \uparrow), \sigma\rangle \;\rightarrow\; \langle n, \sigma\rangle} \quad \text{where } (a \sim n) \in \sigma$$

This leaves us with the interesting two rules, the ones for **orcl**:

$$\frac{\langle e_1, \sigma\rangle \;\rightarrow\; \langle 0, \sigma'\rangle}{\langle \mathbf{orcl}(e_1, e_2), \sigma\rangle \;\rightarrow\; \langle 0, \sigma'\rangle} \qquad \frac{\langle e_2, \sigma\rangle \;\rightarrow\; \langle 0, \sigma'\rangle}{\langle \mathbf{orcl}(e_1, e_2), \sigma\rangle \;\rightarrow\; \langle 0, \sigma'\rangle}$$

The amusing thing to note – large step rule(s) are expressive enough to make a choice based on whether that choice will succeed, that is, they can choose which of e_1 or e_2 to evaluate based on what the outcome of the evaluation *will be*. □

Question 3: Here is yet another little language

$$\begin{aligned} e \;::=\;& n \mid (e_1; e_2) \\ & \mid (\mathbf{catch} \Rightarrow e_1 \;\mathbf{in}\; e_2) \mid \mathbf{throw} \\ & \mid (\mathbf{let}\; x \sim e_1 \;\mathbf{in}\; e_2) \mid x \end{aligned}$$

This language does not have assignable variables. It has **let** blocks just like we've been using all along. The **catch** block is similar to the one discussed in the first programming problem, but with e_1 and e_2 reordered to be similar to the **let** block. The meaning of

$$\mathbf{catch} \Rightarrow e_1 \;\mathbf{in}\; e_2$$

is to produce the value of e_2 unless e_2 executes a **throw**, in which case it produces the value of e_1. Thus

$$\begin{aligned} &\mathbf{catch} \Rightarrow 1 \;\mathbf{in}\; 2 \quad \rightarrow \quad 2 \\ &\mathbf{catch} \Rightarrow 1 \;\mathbf{in}\; (\mathbf{throw}; 3) \quad \rightarrow \quad 1 \end{aligned}$$

These simple examples leave many unanswered questions.

We have discussed eager and lazy evaluation strategies for e_1 in a **let** block. There is a similar concept of eager or lazy evaluation of e_1 in a **catch** block. In fact, all four possibilities make sense. So we define four different interpretations for our language:

L_{EE} = eager **let**, eager **catch**.

L_{EL} = eager **let**, lazy **catch**.

L_{LE} = lazy **let**, eager **catch**.

L_{LL} = lazy **let**, lazy **catch**.

(a) Show (by giving short example programs) that all six possible pairs of languages are different; that is show

$$L_{EE} \neq L_{EL} \qquad L_{EL} \neq L_{LL}$$

$$L_{EE} \neq L_{LE} \qquad L_{LE} \neq L_{LL}$$

$$L_{EE} \neq L_{LL} \qquad L_{LE} \neq L_{EL}$$

by giving example programs that evaluate differently. Assume static scope is used for **let** blocks if you wish (you don't need this assumption – examples exist that are independent of the scope rule used). It is possible to answer this question with only two simple example programs, or even a single (more complicated) one.

(solution a) Here is a solution using two expressions. Consider

$$E_1 = (\textbf{catch} \Rightarrow 1 \textbf{ in } (\textbf{catch} \Rightarrow \textbf{ throw in } 2\ ))$$

This expression reduces to 1 if **catch** block evaluation is eager and 0 if **catch** block evaluation is lazy, independent of how **let** blocks are evaluated. Thus,

$$L_{EE}, L_{LE} : E_1 \to 1 \qquad L_{EL}, L_{LL} : E_1 \to 2$$

so E_1 shows

$$L_{EE} \neq L_{EL} \qquad L_{LE} \neq L_{LL}$$

$$L_{EE} \neq L_{LL} \qquad L_{LE} \neq L_{EL}$$

For the remaining two distinctions, consider

$$E_2 = (\textbf{catch} \Rightarrow 1 \textbf{ in } (\textbf{let } x \sim \textbf{ throw in } 2\ ))$$

Note in the **catch** phrase the expression e_1 (=1) is a closed value; thus the behavior of E_2 does not depend on whether **catch** phrase evaluation is eager or lazy. Thus, Thus,

$$L_{EE}, L_{EL} : E_2 \to 1 \qquad L_{LE}, L_{LL} : E_1 \to 2$$

so E_2 shows

$$L_{EE} \neq L_{LE} \qquad L_{EL} \neq L_{LL}$$

$$L_{EE} \neq L_{LL} \qquad L_{EL} \neq L_{LE}$$

which includes the two distinctions that were not covered by E_1.

(b) Give a set of LS lazy dynamic scope evaluation rules defining L_{LL}. Show that your rules correctly reduce

$$\textbf{let } x \sim \textbf{throw in } (\textbf{catch} \Rightarrow 1 \textbf{ in } x\ )$$

to 1 (not **throw**).

(solution b) These will be LS rules with an environment but no store. The environment will be

$$\phi \ = \ \{\ldots\ a_i \sim e_i\ \ldots\}$$

and judgements will take the form

$$\langle e, \phi \rangle \ \to \ v$$

where the values are

$$v \in (\mathbf{N} \cup \{\textbf{throw}\})$$

The rules are

$$\frac{}{\langle n, \phi \rangle \ \to \ n}$$

$$\frac{\langle e_1, \phi \rangle \ \to \ n_1 \quad \langle e_2, \phi \rangle \ \to \ n_2}{\langle (e_1; e_2), \phi \rangle \ \to \ n_2}$$

$$\frac{\langle e_2, \phi\rangle \;\rightarrow\; n_2}{\langle (\textbf{catch} \Rightarrow e_1 \textbf{ in } e_2), \phi\rangle \;\rightarrow\; n_2}$$

$$\frac{\langle e_2, \phi\rangle \;\rightarrow\; \textbf{throw} \quad \langle e_1, \phi\rangle \;\rightarrow\; v}{\langle (\textbf{catch} \Rightarrow e_1 \textbf{ in } e_2), \phi\rangle \;\rightarrow\; v}$$

$$\frac{}{\langle \textbf{throw}, \phi\rangle \;\rightarrow\; \textbf{throw}}$$

$$\frac{\langle e_2, (\phi \oplus \{x \sim e_1\})\rangle \;\rightarrow\; v}{\langle (\textbf{let } x \sim e_1 \textbf{ in } e_2), \phi\rangle \;\rightarrow\; v}$$

$$\frac{\langle e, \phi\rangle \;\rightarrow\; v}{\langle x, \phi\rangle \;\rightarrow\; v} \quad \text{where } (x \sim e) \in \phi$$

These rules do indeed reduce the expression in question to 1. The important path in the derivation tree is

$$\langle \textbf{let } x \sim \textbf{throw in } (\textbf{catch} \Rightarrow 1 \textbf{ in } x\ ), \{\}\rangle \quad \rightarrow \quad 1$$

generates the hypothesis

$$\langle (\textbf{catch} \Rightarrow 1 \textbf{ in } x\ ), \{\} \oplus \{x \sim \textbf{throw}\}\rangle \quad \rightarrow \quad 1$$

which generates the two hypotheses

$$\langle 1, \phi\rangle \;\rightarrow\; 1 \qquad \text{and} \qquad \langle x, \{\} \oplus \{x \sim \textbf{throw}\}\rangle \;\rightarrow\; \textbf{throw}$$

and it is easy to fill in the details.

(c) (A thought question) We have discussed static versus dynamic scope for **let** blocks. Does the same concept exist for **catch** blocks? What would it mean to say that **catch** blocks use static (or dynamic) scope? You (almost certainly) won't be able to write down evaluation rules for this, but you might give some examples and argue for what they should return under a static or dynamic scope interpretation.

(solution c) A statically scoped interpretation for **throw** would use the nearest enclosing **catch** around the definition rather than the execution of the **throw** – this makes sense only for a **throw** that occurs inside the bound expression of a lazily evaluated **let**. An expression to consider is

$$\begin{array}{l}\textbf{catch} \Rightarrow 1 \textbf{ in} \\ \quad \textbf{let } x \sim \textbf{throw in} \\ \quad\quad ((\textbf{catch} \Rightarrow 2 \textbf{ in } x); 3)\end{array}$$

The rules of part (b) do lazy evaluation of **let** bindings and dynamic scope for **catch**. Under these rules, the evaluation of x does a **throw**. This is caught by the innermost **catch** phrase, which produces 2. Execution continues in the ";" composition, evaluating 3 and producing that as the final result.

Under static **catch** scoping rules, a **throw** from inside x would be caught by the outermost **catch** phrase. Evaluation of 3 would be skipped entirely, and the final result would be 1. This is perfectly sensible behavior, and very difficult to capture with the style of semantic rules we have been using. □