CMSC 22610 **Winter 2005**

Implementation of Computer Languages

Project 3 **February 8, 2005**

MLR typechecker

Due: February 25, 2005

1 Introduction

The third part of the project is to implement a typechecker for MLR. The typechecker is responsible for checking that a given program is *statically correct*. The typechecker takes a parse tree (as produced by your parser) as input and produces a *typed abstract syntax tree* (AST). The AST includes information about the types and binding sites of variables. We will provide a sample scanner and parser, but you may also use your solution to Part 2.

The bulk of this document is a formal specification of the typing rules for MLR. To keep the formal notation simpler, we give the rules for the *functional* subset of MLR. In Section 6, we informally explain how these rules should be extended for mutable record fields.

MLR supports a simple form of record subtyping in that a function that takes a record argument can be given a record with additional fields. For example, the function

```
fun xCoord (r : {x : int}) : int = r.x
```

can be applied to any record value that has an integer `x` field.

2 MLR types

MLR types are either *base types*, such as **`int`** and **`bool`**, function types, list types, record types, or index types. The source language allows type declarations, but we replace these with their right-hand sides during typechecking.[1] The abstract syntax of types is given in Figure 1. We use ι to denote base types, $\tau_1 \rightarrow \tau_2$ to denote a function type, and $\tau\,\mathbf{list}$ for lists. A record type is denoted by $\mathbf{\{}l : \tau_l{}^{l \in \mathcal{L}}\mathbf{\}}$, where $\mathcal{L}$ is the finite set of labels in the record. Note that we omit mutable fields from this definition. We do this to keep the notation simple, but issues related to extending the system to imperative types are discussed in Section 6 below.

3 Typed AST

The abstract syntax of MLR programs is simplified from the concrete syntax specified for Project 2. A program is represented as a list of function declarations (type declarations are folded into the type

[1] In practice, one might note the occurrences of these type names so as to produce better error messages.

$$\begin{array}{rcl}
\tau & ::= & \iota \\
 & | & \tau_1 \rightarrow \tau_2 \\
 & | & \tau\ \textbf{list} \\
 & | & \boldsymbol{\{} l : {\tau_l}^{l \in \mathcal{L}} \boldsymbol{\}}
\end{array}$$

Figure 1: MLR types

$$\begin{array}{rcl}
p & ::= & d \\
 & | & d\ p \\
 & & \\
d & ::= & \textbf{fun}\, f\ (x_1 : \tau_1) \cdots (x_n : \tau_n)\ :\ \tau = e \\
 & & \\
e & ::= & \textbf{let}\ x : \tau = e_1\ \textbf{in}\ e_2 \\
 & | & \textbf{fun}\, f\ (x_1 : \tau_1) \cdots (x_n : \tau_n)\ :\ \tau = e_1\ \textbf{in}\ e_2 \\
 & | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
 & | & (e_1\ e_2) \\
 & | & p(e_1, e_2) \\
 & | & p(e) \\
 & | & \textbf{nil}_\tau \\
 & | & e.l \\
 & | & \boldsymbol{\{} l = {e_l}^{l \in \mathcal{L}} \boldsymbol{\}} \\
 & | & e_1;\ \ldots;\ e_n \\
 & | & x \\
 & | & b
\end{array}$$

Figure 2: MLR abstract syntax

environment).

4 Typing rules

The typing rules for MLR provide both a specification for static correctness of MLR programs, as well as defining a translation from the parse tree to the AST representation.

4.1 Environments

The MLR typing rules are defined with respect to type and value environments. These environments are finite maps from type and value variables to types. We use TE to denote a type environment and VE to denote a value environment. We define the extension of an environment E by a binding of a to τ as

$$E \pm \{a \mapsto \tau\}(b) = \begin{cases} \tau & \textit{when } a = b \\ E(b) & \textit{when } a \neq b \end{cases}$$

4.2 Types

The typing rules for types check types for well-formedness and translate the concrete syntax of types into the abstract syntax. The judgment form is

$$\mathrm{TE} \vdash \mathit{Type} \Rightarrow \tau$$

which should be read as: in the type environment TE, the type expression *Type* is well-formed and translates to the abstract type τ. Typechecking a type identifier replaces it with its definition.

$$\frac{\mathrm{tid} \in \mathrm{dom}(\mathrm{TE}) \quad \mathrm{TE}(\mathrm{tid}) = \iota}{\mathrm{TE} \vdash \mathrm{tid} \Rightarrow \iota}$$

$$\frac{\mathrm{TE} \vdash \mathit{Type}_1 \Rightarrow \tau_1 \quad \mathrm{TE} \vdash \mathit{Type}_2 \Rightarrow \tau_2}{\mathrm{TE} \vdash \mathit{Type}_1 \,\texttt{->}\, \mathit{Type}_2 \Rightarrow \tau_1 \to \tau_2}$$

$$\frac{\mathrm{TE} \vdash \mathit{Type} \Rightarrow \tau}{\mathrm{TE} \vdash \mathit{Type}\ \texttt{list} \Rightarrow \tau\ \mathbf{list}}$$

Record types must have disjoint fields (*i.e.,* the same label cannot appear twice).

$$\frac{l_1\ \ldots, l_n \text{ are disjoint} \quad \mathrm{TE} \vdash \mathit{Type}_i \Rightarrow \tau_{l_i} \text{ for } 1 \le i \le n}{\mathrm{TE} \vdash \texttt{\{}l_1 : \mathit{Type}_1, \ldots, l_n : \mathit{Type}_n\texttt{\}} \Rightarrow \texttt{\{}l : \tau_l{}^{l \in \{l_1 \ldots, l_n\}}\texttt{\}}}$$

We call the empty record type (`{ }`) the “unit” type.

4.3 Subtyping

MLR supports simple record subtyping. This system is specified in the following rules. The first rule states that any type is a subtype of itself.

$$\frac{}{\tau \mathrel{\texttt{<:}} \tau}$$

A list type is a subtype of another list type, if the elements types are in the subtyping relation:

$$\frac{\sigma \mathrel{\texttt{<:}} \tau}{\sigma\ \mathbf{list} \mathrel{\texttt{<:}} \tau\ \mathbf{list}}$$

Function types have *contra-variant* subtyping for the argument type and *co-variant* subtyping in the result position:

$$\frac{\sigma_2 \mathrel{\texttt{<:}} \sigma_1 \quad \tau_1 \mathrel{\texttt{<:}} \tau_2}{\sigma_1 \to \tau_1 \mathrel{\texttt{<:}} \sigma_2 \to \tau_2}$$

Record types obey what is known as *width* subtyping, which means that one record type is a subtype of another if it has all the same fields (with the same types) and possibly others.

$$\frac{\mathcal{L}' \subseteq \mathcal{L}}{\texttt{\{}l : \tau_l{}^{l \in \mathcal{L}}\texttt{\}} \mathrel{\texttt{<:}} \texttt{\{}l : \tau_l{}^{l \in \mathcal{L}'}\texttt{\}}}$$

4.4 Expression typing

The expression typing judgment has the form

$$\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp} \Rightarrow e : \tau$$

which states that in the environments TE and VE, the expression *Exp* translates to the AST term e and has type τ. The rule for **`let`** bindings extends the value environment with the new binding in the body of the **`let`**.

$$\frac{\mathrm{TE} \vdash \mathit{Type} \Rightarrow \tau \quad \mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_1 \Rightarrow e_1 : \tau_1 \quad \tau_1 \texttt{ <: } \tau \quad \mathrm{TE}, \mathrm{VE}\pm\{x \mapsto \tau\} \vdash \mathit{Exp}_2 \Rightarrow e_2 : \tau_2}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{let}\ x\ :\ \mathit{Type}\ =\ \mathit{Exp}_1\ \texttt{in}\ \mathit{Exp}_2 \Rightarrow \mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

Because functions may be recursive, we extend the value environment with their type when checking their body.

$$\frac{\begin{array}{c}\mathrm{TE} \vdash \mathit{Type}_1 \Rightarrow \sigma_1 \quad \cdots \quad \mathrm{TE} \vdash \mathit{Type}_n \Rightarrow \sigma_n \quad \mathrm{TE} \vdash \mathit{Type} \Rightarrow \tau \\ \mathrm{VE}' = \mathrm{VE}\pm\{f \mapsto \sigma_1 \to \cdots \to \sigma_n \to \tau\} \\ \mathrm{TE}, \mathrm{VE}'\pm\{x_i \mapsto \sigma_i \mid 1 \le i \le n\} \vdash \mathit{Exp}_1 \Rightarrow e_1 : \tau_1 \quad \tau_1 \texttt{ <: } \tau \\ \mathrm{TE}, \mathrm{VE}' \vdash \mathit{Exp}_2 \Rightarrow e_2 : \tau_2\end{array}}{\begin{array}{c}\mathrm{TE}, \mathrm{VE} \vdash \texttt{fun}\ f\ (x_1\ :\ \mathit{Type}_1) \cdots (x_n\ :\ \mathit{Type}_n)\ :\ \mathit{Type}\ \texttt{=}\ \mathit{Exp}_1\ \texttt{in}\ \mathit{Exp}_2 \\ \Rightarrow \mathbf{fun}\, f\ (x_1 : \sigma_1)\ \cdots\ (x_n : \sigma_n)\ :\ \tau = e_1\ \mathbf{in}\ e_2 : \tau_2\end{array}}$$

The rule for conditionals requires that both arms have the same type.

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_1 \Rightarrow e_1 : \mathbf{bool} \quad \mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_2 \Rightarrow e_2 : \tau \quad \mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_3 \Rightarrow e_3 : \tau}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{if}\ \mathit{Exp}_1\ \texttt{then}\ \mathit{Exp}_2\ \texttt{else}\ \mathit{Exp}_3 \Rightarrow \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}$$

Function application allows a function to be applied to a subtype of its argument type.

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_1 \Rightarrow e_1 : \sigma \to \tau \quad \mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_2 \Rightarrow e_2 : \sigma' \quad \sigma' \texttt{ <: } \sigma}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_1\ \mathit{Exp}_2 \Rightarrow (e_1\ e_2) : \tau}$$

Field selection requires that the expression have a field with the given label.

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp} \Rightarrow e : \sigma \quad \sigma \texttt{ <: } \texttt{\{}l : \tau\texttt{\}}}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}\,\texttt{.}\,l \Rightarrow e.l : \tau}$$

Record-value construction requires that the fields be disjoint.

$$\frac{l_1\ \ldots, l_n\ \textit{are disjoint} \quad \mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_i \Rightarrow e_{l_i} : \tau_{l_i}\ \textit{for}\ 1 \le i \le n}{\mathrm{TE}, \mathrm{VE} \vdash\Rightarrow \texttt{\{}l = e_l{}^{l\in\{l_1 \ldots, l_n\}}\texttt{\}} : \texttt{\{}l : \tau_l{}^{l\in\{l_1 \ldots, l_n\}}\texttt{\}}}$$

The equality operator requires that its arguments have the same type:

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_1 \Rightarrow e : \tau \quad \mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_2 \Rightarrow e : \tau \quad \tau\ \textit{admits equality}}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_1\ \texttt{==}\ \mathit{Exp}_2 \Rightarrow \texttt{==}_\tau(e_1, e_2) : \mathbf{bool}}$$

The list-cons operator is annotated with the list-element type in the abstract syntax.

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_1 \Rightarrow e : \tau \quad \mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_2 \Rightarrow e : \tau\ \mathbf{list}}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Exp}_1\ \texttt{::}\ \mathit{Exp}_2 \Rightarrow \texttt{::}_\tau(e_1, e_2) : \tau\ \mathbf{list}}$$

The other binary operators are translated with help of an auxiliary TypeOf function that gives their signature:

$$\frac{\mathrm{TE},\mathrm{VE} \vdash \mathit{Exp}_1 \Rightarrow e : \tau_1 \quad \mathrm{TE},\mathrm{VE} \vdash \mathit{Exp}_2 \Rightarrow e : \tau_2 \quad \mathrm{TypeOf}(\mathrm{bop}) = (\tau_1 \times \tau_2) \to \tau}{\mathrm{TE},\mathrm{VE} \vdash \mathit{Exp}_1 \ \mathrm{bop} \ \mathit{Exp}_2 \Rightarrow \mathrm{bop}(e_1, e_2) : \tau}$$

The empty list is *polymorphic*, so it can have any type. We annotate the AST with the item type.

$$\frac{}{\mathrm{TE},\mathrm{VE} \vdash \texttt{[ ]} \Rightarrow \mathbf{nil}_\tau : \tau\,\mathbf{list}}$$

Expression sequencing requires that the l.h.s. of a "**;**" have unit type.

$$\frac{\mathrm{TE},\mathrm{VE} \vdash \mathit{Exp}_i \Rightarrow e_i : \tau_i \ \mathit{for}\ 1 \leq i \leq n \quad \tau_i = \texttt{\{ \}} \ \mathit{for}\ 1 \leq i < n}{\mathrm{TE},\mathrm{VE} \vdash \mathit{Exp}_1 \texttt{;} \ldots \texttt{;} \mathit{Exp}_n \Rightarrow e_1; \ldots; e_n : \tau_n}$$

The type of a variable is determined by its binding in the value environment.

$$\frac{x \in \mathrm{dom}(\mathrm{VE}) \quad \mathrm{VE}(x) = \tau}{\mathrm{TE},\mathrm{VE} \vdash x \Rightarrow x : \tau}$$

We use the auxiliary function TypeOf to map literals to their types (*i.e.*, **bool**, **int**, and **string**).

$$\frac{\mathrm{TypeOf}(b) = \tau}{\mathrm{TE},\mathrm{VE} \vdash b \Rightarrow b : \tau}$$

One important property of this type system is that it does not support *subsumption* at any point; only at type constraints, function applications, and record field selections. Thus, even though $\texttt{\{}x : \mathbf{int}\texttt{\}} \texttt{<:} \texttt{\{ \}}$, the following code is not type correct:

```
(x = 1; 2)
```

since the first expression does not have unit type.

4.5 Operators

The TypeOf function on operators is defined as follows:

$$\begin{array}{rcl}
\mathrm{TypeOf}(\texttt{<=}) & = & (\mathbf{int} \times \mathbf{int}) \to \mathbf{bool} \\
\mathrm{TypeOf}(\texttt{<}) & = & (\mathbf{int} \times \mathbf{int}) \to \mathbf{bool} \\
\mathrm{TypeOf}(\texttt{+}) & = & (\mathbf{int} \times \mathbf{int}) \to \mathbf{int} \\
\mathrm{TypeOf}(\texttt{-}) & = & (\mathbf{int} \times \mathbf{int}) \to \mathbf{int} \\
\mathrm{TypeOf}(\texttt{*}) & = & (\mathbf{int} \times \mathbf{int}) \to \mathbf{int} \\
\mathrm{TypeOf}(\texttt{/}) & = & (\mathbf{int} \times \mathbf{int}) \to \mathbf{int} \\
\mathrm{TypeOf}(\texttt{\%}) & = & (\mathbf{int} \times \mathbf{int}) \to \mathbf{int} \\
\mathrm{TypeOf}(\mathit{unary}\ \texttt{-}) & = & \mathbf{int} \to \mathbf{int} \\
\mathrm{TypeOf}(\texttt{not}) & = & \mathbf{bool} \to \mathbf{bool} \\
\mathrm{TypeOf}(\texttt{isnull}) & = & \tau\,\mathbf{list} \to \mathbf{bool} \\
\mathrm{TypeOf}(\texttt{hd}) & = & \tau\,\mathbf{list} \to \tau \\
\mathrm{TypeOf}(\texttt{tl}) & = & \tau\,\mathbf{list} \to \tau\,\mathbf{list}
\end{array}$$

Note that the list operators are really a family of operators and should be typechecked in a way that is similar to the rule for **::**.

4.6 Declaration typing

An MLR program consists of a sequence of type and function declarations. We require that the last declaration be a function declaration defining the `main` function. Typechecking these declarations has the result of adding new bindings to the type and value environments. In the case of a function declaration, there is also a resulting AST representation. The judgment forms for declarations and programs are

$$\mathrm{TE}, \mathrm{VE} \vdash \mathit{Prog} \Rightarrow p$$
$$\mathrm{TE}, \mathrm{VE} \vdash \mathit{Dcl} \Rightarrow d : \mathrm{TE}', \mathrm{VE}'$$

Type declarations extend the type environment, but do not yield a abstract syntax term.

$$\frac{\mathrm{TE} \vdash \mathit{Type} \Rightarrow \tau}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{type}\ t\ \texttt{=}\ \mathit{Type} \Rightarrow - : \mathrm{TE}\pm\{t \mapsto \tau\}, \mathrm{VE}}$$

Function declarations extend the value environment.

$$\frac{\begin{array}{c}\mathrm{TE} \vdash \mathit{Type}_1 \Rightarrow \sigma_1 \quad \cdots \quad \mathrm{TE} \vdash \mathit{Type}_n \Rightarrow \sigma_n \quad \mathrm{TE} \vdash \mathit{Type} \Rightarrow \tau \\ \mathrm{VE}' = \mathrm{VE}\pm\{f \mapsto \sigma_1 \to \cdots \to \sigma_n \to \tau\} \\ \mathrm{TE}, \mathrm{VE}'\pm\{x_i \mapsto \sigma_i \mid 1 \leq i \leq n\} \vdash \mathit{Exp} \Rightarrow e : \tau' \quad \tau' \texttt{<:}\ \tau\end{array}}{\begin{array}{c}\mathrm{TE}, \mathrm{VE} \vdash \texttt{fun}\ f\ (x_1 : \mathit{Type}_1) \cdots (x_n : \mathit{Type}_n) : \mathit{Type}\ \texttt{=}\ \mathit{Exp} \\ \Rightarrow \mathbf{fun}\, f\ (x_1 : \sigma_1)\ \cdots\ (x_n : \sigma_n)\ :\ \tau = e : \mathrm{TE}, \mathrm{VE}'\end{array}}$$

The typechecking of programs just threads the environments from left to right. We adopt the convention that if $d = -$ (*i.e.*, the Dcl is a type declaration), then $d\ p = p$.

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Dcl} \Rightarrow d : \mathrm{VE}', \mathrm{TE}' \quad \mathrm{TE}', \mathrm{VE}' \vdash \mathit{Prog} \Rightarrow p}{\mathrm{TE}', \mathrm{VE}' \vdash \mathit{Dcl}\ \mathit{Prog} \Rightarrow d\ p}$$

The last declaration should be the `main` function:

$$\frac{\begin{array}{c}\mathrm{TE}, \mathrm{VE} \vdash \mathit{Dcl} \Rightarrow d : \mathrm{VE}', \mathrm{TE}' \\ d = \mathbf{fun}\, \texttt{main}\, (x : \mathbf{string\, list})\ :\ \mathbf{int} = e\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{Dcl} \Rightarrow d}$$

5 Derived forms

Some forms in the concrete syntax are defined in terms of a simple translation. This section describes these translations.

5.1 Functional record update

The **`with`** operator supports functional record update. An expression of the form

$$\mathit{Exp}\ \texttt{with}\ \texttt{\{}l'\texttt{=}\mathit{Exp}_{l'}{}^{l' \in \mathcal{L}_2}\texttt{\}}$$

where the type of Exp is $\texttt{\{}l : \tau_l{}^{l \in \mathcal{L}_1}\texttt{\}}$, is translated to

$$\begin{array}{l}\texttt{let}\ x\ :\ \texttt{\{}l : \tau_l{}^{l \in \mathcal{L}}\texttt{\}}\ =\ \mathit{Exp}\ \texttt{in} \\ \quad \texttt{\{}l\texttt{=}x\texttt{.}l^{l \in \mathcal{L}_1 \setminus \mathcal{L}_2},\ l'\texttt{=}\mathit{Exp}_{l'}{}^{l' \in \mathcal{L}_2}\texttt{\}}\end{array}$$

5.2 Conditional expressions

The operators **`andalso`** and **`orelse`** are translated to **`if`** expressions as follows:

$$\begin{aligned} \mathit{Exp}_1 \ \texttt{andalso}\ \mathit{Exp}_2 &= \texttt{if}\ \mathit{Exp}_1\ \texttt{then}\ \mathit{Exp}_2\ \texttt{else false} \\ \mathit{Exp}_1 \ \texttt{orelse}\ \mathit{Exp}_2 &= \texttt{if}\ \mathit{Exp}_1\ \texttt{then true else}\ \mathit{Exp}_2 \end{aligned}$$

5.3 List expressions

The form $\texttt{[}\mathit{Exp}_1, \ldots, \mathit{Exp}_n\texttt{]}$ is equivalent to

$$\mathit{Exp}_1 \ \texttt{::}\ \cdots\ \texttt{::}\ \mathit{Exp}_n\ \texttt{::}\ \texttt{[]}$$

6 Imperative features

Handling the imperative features of MLR requires adding mutability flags to field types and the requirement that the mutability flags of the supertype be matched by the subtype. The typing rule for field assignment is

$$\frac{\mathrm{TE},\mathrm{VE} \vdash \mathit{Exp_1} \Rightarrow e_1 : \tau_1 \quad \mathrm{TE},\mathrm{VE} \vdash \mathit{Exp_1} \Rightarrow e_1 : \tau_2 \quad \tau_1 \ \texttt{<:}\ \texttt{\{}l!\tau_2\texttt{\}}}{\mathrm{TE},\mathrm{VE} \vdash \mathit{Exp}_1\texttt{.}l \ \texttt{:=}\ \mathit{Exp}_2 \Rightarrow e_1.l := e_2 : \texttt{\{ \}}}$$

7 Requirements

7.1 Errors

Your typechecker should implement the above type system and report reasonable error messages. Errors that you should catch include (but are not limited to), use of undeclared variables (both type and value), non-disjoint labels in record types/expressions, and type mismatches on function applications, field selections, field assignments, *etc.*

7.2 Submission

We will set up a CVS repository for each student on the Computer Science server. This repository will be seeded with CVS modules for each of the projects. For this project, the CVS module is named `project-3` and contains the sample scanner and parser implementation. We will also provide the implementation of the AST representation. You should use this repository to hold the source for your project. We will collect the projects at 12 midnight on Friday February 25th from the repositories, so make sure that you have committed your final version before then.

8 Document history

Feb. 8 Original version.