CMSC 22610
Winter 2007

Implementation
of
Computer Languages

Project 3
February 6, 2007

MinML typechecker

(Revised 2007-02-14)

Due: February 21, 2007

1 Introduction

The third part of the project is to implement a typechecker for MinML. 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 from Part 2.

The bulk of this document is a formal specification of the typing rules for MinML. The type system for MinML is essentially a stripped down version of the SML type system and supports polymorphism with Hindley-Milner type inference. For a discussion of how to implement Hindley-Milner type inference, see Handout 5.

2 Syntactic restrictions

There are a number of syntactic restrictions that your typechecker should enforce. These restrictions could be specified as part of the typing rules below, but it is easier to specify them separately.

1. The type parameters in a type or datatype definition must have distinct names.
2. The constructors in a datatype definition must have distinct names.
3. The variables in a pattern must have distinct names.
4. The functions in a recursive binding must have distinct names.
5. Integer literals must be in the range $-2^{30}..2^{30}-1$ (*i.e.*, representable as a 31-bit 2's-complement integer).

3 Core syntax

The typing rules are given for a core of the concrete grammar, which is given in Figure 1. This grammar omits specification of parenthization, associativity, and precedence. It also treats infix operators as applications.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{exp} \\
& | & \mathit{topdcl}\ \texttt{;}\ \mathit{prog} \\
\\
\mathit{topdcl} & ::= & \mathit{tydcl} \\
& | & \mathit{valdcl} \\
\\
\mathit{tydcl} & ::= & \texttt{type}\ (\mathrm{tyvar}_1, \ldots, \mathrm{tyvar}_k)\ \mathrm{tid}\ \texttt{=}\ \mathit{ty} \\
& | & \texttt{datatype}\ (\mathrm{tyvar}_1, \ldots, \mathrm{tyvar}_k)\ \mathrm{tid}\ \texttt{=}\ \mathit{condcl}_1\ \texttt{|}\ \cdots\ \texttt{|}\ \mathit{condcl}_n \\
\\
\mathit{ty} & ::= & \mathrm{tyvar} \\
& | & (\mathit{ty}_1, \ldots, \mathit{ty}_k)\ \mathrm{tid} \\
& | & \mathit{ty}_1\texttt{->}\mathit{ty}_2 \\
& | & \mathit{ty}_1\texttt{*}\cdots\texttt{*}\mathit{ty}_n \\
\\
\mathit{condcl} & ::= & \mathrm{conid} \\
& | & \mathrm{conid}\ \texttt{of}\ \mathit{ty} \\
\\
\mathit{valdcl} & ::= & \texttt{val}\ \mathit{pat}\ \texttt{=}\ \mathit{exp} \\
& | & \texttt{fun}\ \mathit{fb}_1\ \texttt{and}\ \cdots\ \texttt{and}\ \mathit{fb}_n \\
\\
\mathit{fb} & ::= & \mathrm{vid}\ \mathit{pat}\ \texttt{=}\ \mathit{exp} \\
\\
\mathit{exp} & ::= & \texttt{let}\ \mathit{valdcl}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \\
& | & \texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match} \\
& | & \texttt{if}\ \mathit{exp}_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \\
& | & \mathit{exp}_1\ \mathit{exp}_2 \\
& | & \mathit{exp}_1\ \texttt{=}\ \mathit{exp}_2 \\
& | & \texttt{(}\ \mathit{exp}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exp}_n\ \texttt{)} \\
& | & \mathit{exp}_1\texttt{;}\ \mathit{exp}_2 \\
& | & \mathrm{vid} \\
& | & \mathrm{lit} \\
\\
\mathit{match} & ::= & \mathit{match}_1\ \texttt{|}\ \mathit{match}_2 \\
& | & \mathit{pat}\ \texttt{=>}\ \mathit{exp} \\
\\
\mathit{pat} & ::= & \mathrm{conid}\ \mathit{pat} \\
& | & \texttt{(}\ \mathit{pat}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{pat}_n\ \texttt{)} \\
& | & \mathrm{vid} \\
& | & \mathrm{conid} \\
& | & \mathrm{lit}
\end{array}
$$

Figure 1: Core MinML syntax

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{type variable} \\
 & | & (\tau_1, \ldots, \tau_k)\, T^{(k)} & \text{type constructor} \\
 & | & \tau_1 \rightarrow \tau_2 & \text{function type} \\
 & | & \tau_1 \times \cdots \times \tau_n & \text{tuple type} \\
 & & & \\
\sigma & ::= & \forall \alpha_1, \ldots, \alpha_n.\tau & \text{type scheme}
\end{array}
$$

Figure 2: MinML types

4 MinML types

In the MinML typing rules, we distinguish between *syntactic* types as they appear in the program text and *semantic* types that are inferred for expressions and patterns. MinML's semantic types are formed from type variables, type constructors (including nullary type constructors such as **`int`** and **`bool`**), the function-type constructor, and the tuple-type constructor. The abstract syntax of types is given in Figure 2. Note that a k-arity type constructor has its arity as a superscript.

Since data constructors and variables can be polymorphic, we also define *type schemes*. We say that a type τ is an *instance* of a scheme $\sigma = \forall \alpha_1, \ldots, \alpha_n.\tau'$ if there exists a substitution S from type variables to types with domain $\{\alpha_1, \ldots, \alpha_n\}$, such that $\tau = S(\tau')$. We write $\tau \prec \sigma$ when τ is an instance of σ.

5 Identifiers and environments

The typechecking rules of MinML use a number of environments to track binding information. These environments are

$$
\begin{array}{rcll}
\text{TE} & \in & \text{ID} \rightarrow \text{TYCON} \cup (\text{TYVAR}^* \times \text{TY}) & \text{type-name environment} \\
\text{TVE} & \in & \text{ID} \rightarrow \text{TYVAR} & \text{bound type variables} \\
\text{VE} & \in & \text{ID} \rightarrow (\text{IDSTATUS} \times \text{TYSCHEME}) & \text{value-identifier environment}
\end{array}
$$

where TYVAR is the set of type variables (α), ID is the set of syntactic identifiers, TYCON is the set of type constructors ($T^{(k)}$), and TYSCHEME is the set of type schemes (σ). Value identifiers can be constants (*i.e.*, nullary data constructors), data-constructor functions, or variables. We use

$$s \in \text{IDSTATUS} = \{\mathbf{c}, \mathbf{d}, \mathbf{v}\}$$

to distinguish between these, where $\mathbf{c}$ denotes a constant, $\mathbf{d}$ denotes a data-constructor function, and $\mathbf{v}$ denotes a variable.

We define the extension of a finite map (environment) E by another environment E' as

$$(E \pm E')(a) = \begin{cases} E'(a) & \textit{when } a \in \mathrm{dom}(E') \\ E(a) & \textit{when } a \notin \mathrm{dom}(E') \end{cases}$$

$\mathrm{TE}, \mathrm{VE} \vdash \mathit{prog} \Rightarrow \tau$	typechecking a program
$\mathrm{TE}, \mathrm{VE} \vdash \mathit{topdcl} \Rightarrow \mathrm{TE}', \mathrm{VE}'$	typechecking a top-level declaration
$\mathrm{TE}, \mathrm{VE} \vdash \mathit{tydcl} \Rightarrow \mathrm{TE}', \mathrm{VE}'$	typechecking a type declaration
$\mathrm{TE}, \mathrm{TVE} \vdash \mathit{ty} \Rightarrow \tau$	typechecking a type
$\mathrm{TE}, \mathrm{TVE}, (\vec{\alpha}, \tau) \vdash \mathit{condcl} \Rightarrow \mathrm{VE}$	typechecking a data-constructor definitions
$\mathrm{VE} \vdash \mathit{valdcl} \Rightarrow \mathrm{VE}'$	typechecking a value declaration
$\mathrm{VE} \vdash \mathit{fb} \Rightarrow \mathrm{VE}'$	typechecking a function binding
$\mathrm{VE} \vdash \mathit{exp} \Rightarrow \tau$	typechecking a expression
$\mathrm{VE} \vdash \mathit{match} \Rightarrow (\tau, \tau')$	typechecking a match rule
$\mathrm{VE} \vdash \mathit{pat} \Rightarrow (\mathrm{VE}, \tau)$	typechecking a pattern

Figure 3: MinML judgment forms

6 Typing rules

The typing rules (or judgments) for MinML provide both a specification for static correctness of MinML programs. The general form of a rule is

$$\mathit{Context} \vdash \mathit{Term} \Rightarrow \mathit{Type}$$

which can be read as "Term has Type in $\mathit{Context}$." The context usually consists of one or more environments, but may have other information, while the "Type" can be one or more types and or environments. Figure 3 summarizes the judgement forms that we use for typing MinML.

6.1 Programs

The first rule for programs just threads the environment from left to right.

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{topdcl} \Rightarrow \mathrm{VE}', \mathrm{TE}' \quad \mathrm{TE}', \mathrm{VE}' \vdash \mathit{prog} \Rightarrow \tau}{\mathrm{TE}', \mathrm{VE}' \vdash \mathit{topdcl}\texttt{;}\ \mathit{prog} \Rightarrow \tau}$$

When a program is just an expression, its type is that of the expression.

$$\frac{\mathrm{VE} \vdash \mathit{exp} \Rightarrow \tau}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{exp} \Rightarrow \tau}$$

6.2 Top-level declarations

Checking top-level declarations requires appealing to the appropriate declaration judgment form.

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{tydcl} \Rightarrow \mathrm{TE}', \mathrm{VE}'}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{tydcl} \Rightarrow \mathrm{TE}', \mathrm{VE}'}$$

$$\frac{\mathrm{VE} \vdash \mathit{valdcl} \Rightarrow \mathrm{VE}'}{\mathrm{TEVE} \vdash \mathit{valdcl} \Rightarrow \mathrm{TE}', \mathrm{VE}'}$$

6.3 Type declarations

$$\frac{\begin{array}{c}\mathrm{TVE} = \{\mathrm{tyvar}_i \mapsto \alpha_i \mid 1 \le i \le k \text{ and } \alpha_i \text{ are fresh}\} \\ \mathrm{TE}, \mathrm{TVE} \vdash ty \Rightarrow \tau \qquad \mathrm{TE}' = \mathrm{TE} \pm \{\mathrm{tid} \mapsto (\langle \alpha_1, \ldots, \alpha_k \rangle, \tau)\}\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{type}\ (\mathrm{tyvar}_1, \ldots, \mathrm{tyvar}_k)\,\mathrm{tid} = ty \Rightarrow \mathrm{TE}', \mathrm{VE}}$$

The rule for datatype definitions is somewhat complicated. We check each of the constructor declarations in a context that includes the type parameters and result type; these checks yield constructor environments that are merged to produce the final constructor environment.

$$\frac{\begin{array}{c}\mathrm{TVE} = \{\mathrm{tyvar}_i \mapsto \alpha_i \mid 1 \le i \le k \text{ and } \alpha_i \text{ are fresh}\} \\ \mathrm{D} = (\langle \alpha_1, \ldots, \alpha_k \rangle, (\alpha_1, \ldots, \alpha_k)\,\mathrm{tid}^{(k)}) \\ \mathrm{TE}, \mathrm{TVE}, \mathrm{D} \vdash \mathit{condcl}_1 \Rightarrow \mathrm{VE}_1 \quad \cdots \quad \mathrm{TE}, \mathrm{TVE}, \mathrm{D} \vdash \mathit{condcl}_n \Rightarrow \mathrm{VE}_n \\ \mathrm{TE}' = \mathrm{TE} \pm \{\mathrm{tid} \mapsto (\alpha_1, \ldots, \alpha_k)\,\mathrm{tid}^{(k)}\} \quad \mathrm{VE}' = \mathrm{VE} \pm \mathrm{VE}_1 \pm \cdots \pm \mathrm{VE}_n\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{datatype}\ (\mathrm{tyvar}_1, \ldots, \mathrm{tyvar}_k)\,\mathrm{tid} = \mathit{condcl}_1\ \texttt{|}\ \cdots\ \texttt{|}\ \mathit{condcl}_n \Rightarrow \mathrm{TE}', \mathrm{VE}'}$$

6.4 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

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

which should be read as: in the environments TE, TVE, the type expression *Type* is well-formed and translates to the abstract type τ.

Typechecking a type variable replaces it with its definition.

$$\frac{\mathrm{tyvar} \in \mathrm{dom}(\mathrm{TVE}) \quad \mathrm{TVE}(\mathrm{tyvar}) = \alpha}{\mathrm{TE}, \mathrm{TVE} \vdash \mathrm{tyvar} \Rightarrow \alpha}$$

There are two rules for type-constructor application, depending on whether the type ID names a type definition or a datatype (or abstract type). For type definitions, we substitute the type arguments for the type parameters to produces a new type:

$$\frac{\begin{array}{c}\mathrm{tid} \in \mathrm{dom}(\mathrm{TE}) \quad \mathrm{TE}(\mathrm{tid}) = (\langle \alpha_1, \ldots, \alpha_k \rangle, \tau) \\ \mathrm{TE}, \mathrm{TVE} \vdash ty_1 \Rightarrow \tau_1 \quad \cdots \quad \mathrm{TE}, \mathrm{TVE} \vdash ty_k \Rightarrow \tau_k\end{array}}{\mathrm{TE}, \mathrm{TVE} \vdash (ty_1, \ldots, ty_k)\,\mathrm{tid} \Rightarrow [\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k]\tau}$$

For datatypes and abstract types, we check the arguments and mapping the type ID to the type constructor.

$$\frac{\mathrm{tid} \in \mathrm{dom}(\mathrm{TE}) \quad \mathrm{TE}(\mathrm{tid}) = T^{(k)} \quad \mathrm{TE}, \mathrm{TVE} \vdash ty_1 \Rightarrow \tau_1 \quad \cdots \quad \mathrm{TE}, \mathrm{TVE} \vdash ty_k \Rightarrow \tau_k}{\mathrm{TE}, \mathrm{TVE} \vdash (ty_1, \ldots, ty_k)\,\mathrm{tid} \Rightarrow (\tau_1, \ldots, \tau_k)\,T^{(k)}}$$

Type checking a function type requires checking the two sides of the arrow.

$$\frac{\mathrm{TE}, \mathrm{TVE} \vdash ty_1 \Rightarrow \tau_1 \quad \mathrm{TE}, \mathrm{TVE} \vdash ty_2 \Rightarrow \tau_2}{\mathrm{TE}, \mathrm{TVE} \vdash ty_1\texttt{->}ty_2 \Rightarrow \tau_1 \to \tau_2}$$

Type checking a tuple type requires checking the component types.

$$\frac{\mathrm{TE}, \mathrm{TVE} \vdash ty_1 \Rightarrow \tau_1 \quad \cdots \quad \mathrm{TE}, \mathrm{TVE} \vdash ty_n \Rightarrow \tau_n}{\mathrm{TE}, \mathrm{TVE} \vdash ty_1\texttt{*}\cdots\texttt{*}ty_n \Rightarrow \tau_1 \times \cdots \times \tau_n}$$

6.5 Data-constructor definitions

The typing rules for a nullary data-constructor is

$$\frac{\sigma = \forall\alpha_1,\ \ldots,\ \alpha_k.\tau}{\mathrm{TE}, \mathrm{TVE}, (\langle\alpha_1,\ \ldots,\ \alpha_k\rangle, \tau) \vdash \mathrm{conid} \Rightarrow \{\mathrm{conid} \mapsto (\mathbf{c}, \sigma)\}}$$

and the rule for a data-constructor function is

$$\frac{\mathrm{TE}, \mathrm{TVE} \vdash \mathit{ty} \Rightarrow \tau' \qquad \sigma = \forall\alpha_1,\ \ldots,\ \alpha_k.\tau' \to \tau}{\mathrm{TE}, \mathrm{TVE}, (\langle\alpha_1,\ \ldots,\ \alpha_k\rangle, \tau) \vdash \mathrm{conid}\ \mathtt{of}\ \mathit{ty} \Rightarrow \{\mathrm{conid} \mapsto (\mathbf{d}, \sigma)\}}$$

6.6 Value declarations

For a value binding, we check the pattern, which yields a variable environment and a type, and we check the r.h.s. expression using the original environment. If the types match, we extend the value environment with the bindings from the pattern. Note that, unlike in the SML type system, we do not close over the type of the l.h.s.; in MinML, only data-type and function definitions introduce polymorphism.

$$\frac{\mathrm{VE} \vdash \mathit{pat} \Rightarrow (\mathrm{VE}', \tau) \qquad \mathrm{VE} \vdash \mathit{exp} \Rightarrow \tau}{\mathrm{VE} \vdash \mathtt{val}\ \mathit{pat}\ \mathtt{=}\ \mathit{exp} \Rightarrow \mathrm{VE} \pm \mathrm{VE}'}$$

Function bindings are tricky for two reasons: they are mutually recursive and we are allowed to generalize their types. We use the auxiliary function NameOf to extract the function name from a function binding.

$$\frac{\begin{array}{c}\mathrm{VE}' = \mathrm{VE} \pm \{f_i \mapsto (\mathbf{v}, \tau_{f_i}) \mid f_i = \mathrm{NameOf}(\mathit{fb}_i)\} \\ \mathrm{VE}' \vdash \mathit{fb}_i \Rightarrow \tau_{f_i} \quad \text{for } 1 \le i \le n \\ \mathrm{VE}'' = \mathrm{VE} \pm \{f_i \mapsto (\mathbf{v}, \sigma_{f_i}) \mid \sigma_{f_i} = \mathrm{Clos}_{\mathrm{VE}}(\tau_{f_i})\}\end{array}}{\mathrm{VE} \vdash \mathtt{fun}\ \mathit{fb}_1\ \mathtt{and}\ \cdots\ \mathtt{and}\ \mathit{fb}_n \Rightarrow \mathrm{VE}''}$$

6.7 Function bindings

Typechecking a function binding requires checking the parameter pattern and then using its bindings to check the function body.

$$\frac{\mathrm{VE} \vdash \mathit{pat} \Rightarrow (\mathrm{VE}', \tau) \qquad \mathrm{VE} \pm \mathrm{VE}' \vdash \mathit{exp} \Rightarrow \tau'}{\mathrm{VE} \vdash \mathrm{vid}\ \mathit{pat}\ \mathtt{=}\ \mathit{exp} \Rightarrow \tau \to \tau'}$$

6.8 Expressions

Checking a **`let`** expression requires checking the value declaration and then using the enriched environment to check the expression.

$$\frac{\mathrm{VE} \vdash \mathit{valdcl} \Rightarrow \mathrm{VE}' \qquad \mathrm{VE}' \vdash \mathit{exp} \Rightarrow \tau}{\mathrm{VE} \vdash \mathtt{let}\ \mathit{valdcl}\ \mathtt{in}\ \mathit{exp}\ \mathtt{end} \Rightarrow \tau}$$

Checking a case requires checking the type of the argument against the match.

$$\frac{\mathrm{VE} \vdash \mathit{exp} \Rightarrow \tau \qquad \mathrm{VE} \vdash \mathit{match} \Rightarrow (\tau, \tau')}{\mathrm{VE} \vdash \mathtt{case}\ \mathit{exp}\ \mathtt{of}\ \mathit{match} \Rightarrow \tau'}$$

The condition of an **`if`** expression must have boolean type and the types of the arms must agree.

$$\frac{\mathrm{VE} \vdash exp_1 \Rightarrow \mathbf{bool}^{(0)} \qquad \mathrm{VE} \vdash exp_2 \Rightarrow \tau \qquad \mathrm{VE} \vdash exp_3 \Rightarrow \tau}{\mathrm{VE} \vdash \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \Rightarrow \tau}$$

Function application requires checking the argument against the function's type.

$$\frac{\mathrm{VE} \vdash exp_1 \Rightarrow \tau \rightarrow \tau' \qquad \mathrm{VE} \vdash exp_2 \Rightarrow \tau}{\mathrm{VE} \vdash exp_1\ exp_2 \Rightarrow \tau'}$$

Typechecking the equality operator requires a special rule, because equality is an *ad hoc* polymorphic operator.

$$\frac{\mathrm{VE} \vdash exp_1 \Rightarrow \tau \qquad \mathrm{VE} \vdash exp_2 \Rightarrow \tau \qquad \tau \in \{\mathbf{bool}^{(0)}, \mathbf{int}^{(0)}, \mathbf{string}^{(0)}\}}{\mathrm{VE} \vdash exp_1\ \texttt{=}\ exp_2 \Rightarrow \mathbf{bool}^{(0)}}$$

The type of an empty tuple expression is $\mathbf{unit}^{(0)}$.

$$\frac{}{\mathrm{VE} \vdash \texttt{()} \Rightarrow \mathbf{unit}^{(0)}}$$

The type of a tuple expression is the tuple of the types of its sub-expressions.

$$\frac{\mathrm{VE} \vdash exp_i \Rightarrow \tau_i \quad \text{for } 1 \leq i \leq n}{\mathrm{VE} \vdash \texttt{(}\ exp_1\texttt{,}\ \ldots\texttt{,}\ exp_n\ \texttt{)} \Rightarrow \tau_1 \times \cdots \times \tau_n}$$

Expression sequencing ignores the type of the l.h.s. expression

$$\frac{\mathrm{VE} \vdash exp_1 \Rightarrow \tau_1 \qquad \mathrm{VE} \vdash exp_2 \Rightarrow \tau_2}{\mathrm{VE} \vdash exp_1\texttt{;}\ exp_2 \Rightarrow \tau_2}$$

The type of a value identifier is determined by its binding in the value environment. Note that this rule covers constants, data constructors, and variables.

$$\frac{\mathrm{vid} \in \mathrm{dom}(\mathrm{VE}) \qquad \mathrm{VE}(\mathrm{vid}) = (s, \sigma) \qquad \tau \prec \sigma}{\mathrm{VE} \vdash \mathrm{vid} \Rightarrow \tau}$$

We use the auxiliary function TypeOf to map literals to their types (*i.e.*, $\mathbf{int}^{(0)}$ and $\mathbf{string}^{(0)}$).

$$\frac{\mathrm{TypeOf}(\mathrm{lit}) = \tau}{\mathrm{VE} \vdash \mathrm{lit} \Rightarrow \tau}$$

6.9 Match rules

All of the matches in a case must have the same argument and result type, which is reflected in the rule for sequencing matches.

$$\frac{\mathrm{VE} \vdash match_1 \Rightarrow (\tau, \tau') \qquad \mathrm{VE} \vdash match_2 \Rightarrow (\tau, \tau')}{\mathrm{VE} \vdash match_1\ \texttt{|}\ match_2 \Rightarrow (\tau, \tau')}$$

Checking an match rule requires checking the r.h.s. expression in an environment enriched by the bindings from the l.h.s. pattern.

$$\frac{\mathrm{VE} \vdash pat \Rightarrow (\mathrm{VE}', \tau) \qquad \mathrm{VE} \pm \mathrm{VE}' \vdash exp \Rightarrow \tau'}{\mathrm{VE} \vdash pat\ \texttt{=>}\ exp \Rightarrow (\tau, \tau')}$$

6.10 Patterns

Typechecking a pattern yields a new environment, which assigns types to the variables bound in the pattern, and the type of values matched by the pattern.

Constructor application requires matching the argument pattern against the constructor's type.

$$\frac{\text{conid} \in \text{dom}(\text{VE}) \qquad \text{VE}(\text{conid}) = (\mathbf{d}, \sigma) \qquad \tau \to \tau' \prec \sigma \qquad \text{VE} \vdash \mathit{pat} \Rightarrow (\text{VE}', \tau)}{\text{VE} \vdash \text{conid}\ \mathit{pat} \Rightarrow (\text{VE}', \tau')}$$

The type of an empty tuple pattern is $\mathbf{unit}^{(0)}$.

$$\frac{}{\text{VE} \vdash \texttt{(}\,\texttt{)} \Rightarrow (\{\}, \mathbf{unit}^{(0)})}$$

Non-empty tuple patterns require merging the bindings from each sub-pattern.

$$\frac{\text{VE} \vdash \mathit{pat}_i \Rightarrow (\text{VE}_i, \tau_i) \quad \text{for } 1 \le i \le n \qquad \text{VE}' = \text{VE}_1 \pm \cdots \pm \text{VE}_n}{\text{VE} \vdash \texttt{(}\,\mathit{pat}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{pat}_n\,\texttt{)} \Rightarrow (\text{VE}', \tau_1 \times \cdots \times \tau_n)}$$

Value identifiers introduce new bindings.

$$\frac{\mathit{vid} \in \text{dom}(\text{VE}) \text{ and } \text{VE}(\mathit{vid}) = (s, \cdot) \Rightarrow s = \mathbf{v} \qquad \text{VE}' = \{\text{vid} \mapsto (\mathbf{v}, \tau)\}}{\text{VE} \vdash \text{vid} \Rightarrow (\text{VE}', \tau)}$$

Nullary constructors must be verified.

$$\frac{\text{conid} \in \text{dom}(\text{VE}) \qquad \text{VE}(\text{conid}) = (\mathbf{c}, \sigma)}{\text{VE} \vdash \text{conid} \Rightarrow (\{\}, \tau)}$$

Literals are checked using the same TypeOf function as for expressions.

$$\frac{\text{TypeOf}(\text{lit}) = \tau}{\text{VE} \vdash \text{lit} \Rightarrow (\{\}, \tau)}$$

7 Predefined types and operators

Your typechecker will typecheck programs in the context of an *initial basis* TE_0, VE_0. This basis is defined as follows:

$$\text{TE}_0 \;=\; \left\{ \begin{array}{rcl} \texttt{bool} & \mapsto & \mathbf{bool}^{(0)} \\ \texttt{int} & \mapsto & \mathbf{int}^{(0)} \\ \texttt{list} & \mapsto & \mathbf{list}^{(1)} \\ \texttt{string} & \mapsto & \mathbf{string}^{(0)} \\ \texttt{unit} & \mapsto & \mathbf{unit}^{(0)} \end{array} \right\}$$

The initial value environment defines the types of the operator symbols and some additional functions.

$$\mathrm{VE}_0 = \left\{ \begin{array}{rcl} \texttt{false} & \mapsto & (\mathbf{c}, \mathbf{bool}^{(0)}) \\ \texttt{true} & \mapsto & (\mathbf{c}, \mathbf{bool}^{(0)} \\ \texttt{nil} & \mapsto & (\mathbf{c}, \forall\alpha.\alpha\,\mathbf{list}^{(1)}) \\ \texttt{::} & \mapsto & (\mathbf{d}, \forall\alpha.(\alpha \times \alpha\,\mathbf{list}^{(1)}) \to \alpha\,\mathbf{list}^{(1)}) \\ \texttt{<=} & \mapsto & (\mathbf{v}, (\mathbf{int}^{(0)} \times \mathbf{int}^{(0)}) \to \mathbf{bool}^{(0)}) \\ \texttt{<} & \mapsto & (\mathbf{v}, (\mathbf{int}^{(0)} \times \mathbf{int}^{(0)}) \to \mathbf{bool}^{(0)}) \\ \texttt{@} & \mapsto & (\mathbf{v}, \forall\alpha.(\alpha\,\mathbf{list}^{(1)} \times \alpha\,\mathbf{list}^{(1)}) \to \alpha\,\mathbf{list}^{(1)}) \\ \texttt{+} & \mapsto & (\mathbf{v}, (\mathbf{int}^{(0)} \times \mathbf{int}^{(0)}) \to \mathbf{int}^{(0)}) \\ \texttt{-} & \mapsto & (\mathbf{v}, (\mathbf{int}^{(0)} \times \mathbf{int}^{(0)}) \to \mathbf{int}^{(0)}) \\ \texttt{*} & \mapsto & (\mathbf{v}, (\mathbf{int}^{(0)} \times \mathbf{int}^{(0)}) \to \mathbf{int}^{(0)}) \\ \texttt{div} & \mapsto & (\mathbf{v}, (\mathbf{int}^{(0)} \times \mathbf{int}^{(0)}) \to \mathbf{int}^{(0)}) \\ \texttt{mod} & \mapsto & (\mathbf{v}, (\mathbf{int}^{(0)} \times \mathbf{int}^{(0)}) \to \mathbf{int}^{(0)}) \\ \texttt{~} & \mapsto & (\mathbf{v}, \mathbf{int}^{(0)} \to \mathbf{int}^{(0)}) \\ \texttt{fail} & \mapsto & (\mathbf{v}, \forall\alpha.\mathbf{string}^{(0)} \to \alpha) \\ \texttt{print} & \mapsto & (\mathbf{v}, \mathbf{string}^{(0)} \to \mathbf{unit}^{(0)}) \end{array} \right\}$$

8 Derived forms

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

The parsing syntax allows multiple value bindings in a **let** expression. These can be translated into nested lets by repeated application of the following rule:

$$\begin{array}{rcl} \texttt{let}\ valdcl_1\ valdcl_2\ \texttt{in}\ exp\ \texttt{end} & = & \texttt{let}\ valdcl_1\ \texttt{in} \\ & & \quad \texttt{let}\ valdcl_2\ \texttt{in}\ exp\ \texttt{end} \\ & & \texttt{end} \end{array}$$

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

$$\begin{array}{rcl} exp_1\ \texttt{andalso}\ exp_2 & = & \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else false} \\ exp_1\ \texttt{orelse}\ exp_2 & = & \texttt{if}\ exp_1\ \texttt{then true else}\ exp_2 \end{array}$$

9 Requirements

9.1 Errors

Your typechecker should implement the above type system and report reasonable error messages. Errors that you should catch include violations of the syntactic restrictions in Section **??**, unbound identifiers, and any type errors.

9.2 Submission

We will set up a SVN repository for each group on the gforge server. This repository will be seeded with CVS modules for each of the projects. For this project, the SVN 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 10pm on Wednesday February 21st from the repositories, so make sure that you have committed your final version before then.

10 Document history

Feb. 14 Added `unit` type to initial basis and rules for empty tuple expressions and patterns. Also fixed typos in a couple of pattern rules.

Feb. 8 Added syntax and typing rule for equality.

Feb. 7 Merged value and data-constructor environments and added IDSTATUS. Fixed minor typos and added desugaring rule for **`let`** expressions.

Feb. 6 Original version.