CMSC 22610
Winter 2011

Implementation
of
Computer Languages I

Project 3
February 13, 2011

MinML typechecker

Due: March 1, 2011

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 parameter type variables 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. There should be no *free* names (either type or value). That is, every use of a name (in var, con, tyv, tyc) should be in the scope of a declaration of that name.

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. *lit* ranges over integer and string constants (*literals*). Nullary type constructors are applied to null-tuples to form types.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{exp} \\
& | & \mathit{topdcl}\ \texttt{;}\ \mathit{prog} \\
\\
\mathit{topdcl} & ::= & \mathit{tydcl} \\
& | & \mathit{valdcl} \\
\\
\mathit{tydcl} & ::= & \texttt{type}\ \mathrm{tyc}\ \texttt{(}\mathrm{tyv}_1, \ldots, \mathrm{tyv}_k\texttt{)}\ \texttt{=}\ \mathit{ty} \\
& | & \texttt{datatype}\ \mathrm{tyc}\ \texttt{(}\mathrm{tyv}_1, \ldots, \mathrm{tyv}_k\texttt{)}\ \texttt{=}\ \mathit{condcl}_1\ \texttt{|}\ \cdots\ \texttt{|}\ \mathit{condcl}_n \\
\\
\mathit{ty} & ::= & \mathrm{tyv} \\
& | & \mathrm{tyc}\ \texttt{(}\mathit{ty}_1, \ldots, \mathit{ty}_k\texttt{)} \\
& | & \mathit{ty}_1\texttt{->}\mathit{ty}_2 \\
& | & \mathit{ty}_1\texttt{*}\cdots\texttt{*}\mathit{ty}_n \\
\\
\mathit{condcl} & ::= & \mathrm{con} \\
& | & \mathrm{con}\ \texttt{(}\mathit{ty}\texttt{)} \\
\\
\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{var}\ \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{end} \\
& | & \texttt{if}\ \mathit{exp}_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \\
& | & \texttt{fn}\ \mathit{pat}\ \texttt{=>}\ \mathit{exp} \\
& | & \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{var} \\
& | & \mathrm{con} \\
& | & \mathrm{lit} \\
\\
\mathit{match} & ::= & \mathit{match}_1\ \texttt{|}\ \mathit{match}_2 \\
& | & \mathit{pat}\ \texttt{=>}\ \mathit{exp} \\
\\
\mathit{pat} & ::= & \mathrm{con}\ \mathit{pat} \\
& | & \texttt{(}\ \mathit{pat}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{pat}_n\ \texttt{)} \\
& | & \mathrm{var} \\
& | & \mathrm{con} \\
& | & \mathrm{lit}
\end{array}
$$

Figure 1: Core MinML syntax

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{type variable} \\
 & | & T^k\ (\tau_1,\ \ldots,\ \tau_k) & \text{tycon application} \\
\sigma & ::= & \forall\alpha_1,\ \ldots,\ \alpha_n.\tau & \text{type scheme}
\end{array}
$$

Figure 2: MinML semantic 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 and bound to variables and data constructors in type environments.

MinML's semantic types are terms formed from type variables and type constructors (referred to as *tycons*). Where it is not obvious or understood, a superscript is used to specify the arity of a tycon. Tycons include nullary type constructors such as **`Int`** and **`Bool`**, a binary tycon **`Fun`** for function types, and a family of product (tuple) tycons $\{\mathbf{Prod}^n \mid n \geq 1\}$. Informally, we use the notation $\tau_1 \rightarrow \tau_2$ for $\mathbf{Fun}(\tau_1, \tau_2)$, and $\tau_1 \times \ldots \times \tau_n$ for $\mathbf{Prod}^n(\tau_1, ldots, \tau_n)$. The abstract syntax of (semantic) types is given in Figure 2.

Since data constructors and variables can be polymorphic, we also define *type schemes*, also known as polymorphic types or *polytypes*. 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 mapping 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. Type schemes appear only in the bindings of variables and constructors in type environments.

There are three sorts of tycons: primitives like **`Int`** and **`Fun`**, tycons defined by datatype declarations like **`List`**, and derived tycons introduced by **`type`** declarations. Primitive and datatype tycons are atomic, and each has its own unique identity (which is distinct from the *name* of the tycons – multiple datatype tycons can have the same name in the same program). Together these constitute the set AtomCon.

Derived tycons can be represented as type functions $\lambda(\alpha_1, \ldots, \alpha_n).\tau$. The abstract syntax of type functions and tycons is given by

$$
\begin{array}{rcll}
\text{TyFun} & \in & \text{TyVar}^* \times \text{Ty} & \text{type functions} \\
\text{TyCon} & \in & \text{AtomCon} \cup \text{TyFun} & \text{type constructors}
\end{array}
$$

Applications of derived tycons in TyFun can by eliminated by β-reduction, and we will perform this reduction in Rule (8) below to insure that the type expressions that occur during type checking are in *normal form*, where all tycons are in AtomCon.

5 Identifiers and environments

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

$$
\begin{array}{rcll}
\text{TE} & \in & \text{Tyc} \rightarrow \text{TyCon} & \text{tycon environment} \\
\text{TVE} & \in & \text{Tyv} \rightarrow \text{TyVar} & \text{bound type variables} \\
\text{VE} & \in & \text{Var} \rightarrow \text{TyScheme} & \text{value-identifier environment}
\end{array}
$$

$\mathrm{TE}, \mathrm{VE} \vdash \mathit{prog} \Rightarrow \tau$	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
$\mathrm{TE}, \mathrm{TVE} \vdash \mathit{ty} \Rightarrow \tau$	translating 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

where TYVAR is the set of type variables (α), TYC is the set of syntactic tycon identifiers (capitalized), TYCON is the set of type constructors (T^k), and TYSCHEME is the set of type schemes (σ).

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

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

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.

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{topdcl} \Rightarrow \mathrm{VE}', \mathrm{TE}' \quad \mathrm{TE}', \mathrm{VE}' \vdash \mathit{prog} \Rightarrow \tau}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{topdcl}\texttt{;}\ \mathit{prog} \Rightarrow \tau} \quad (1)$$

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

$$\frac{\mathrm{VE} \vdash \mathit{exp} \Rightarrow \tau}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{exp} \Rightarrow \tau} \quad (2)$$

6.2 Top-level declarations

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

$$\frac{\mathrm{TE}, \mathrm{VE} \vdash \mathit{tydcl} \Rightarrow \mathrm{TE}', \mathrm{VE}'}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{tydcl} \Rightarrow \mathrm{TE}', \mathrm{VE}'} \quad (3)$$

$$\frac{\mathrm{VE} \vdash \mathit{valdcl} \Rightarrow \mathrm{VE}'}{\mathrm{TE}, \mathrm{VE} \vdash \mathit{valdcl} \Rightarrow \mathrm{TE}', \mathrm{VE}'} \quad (4)$$

6.3 Type declarations

$$\frac{\begin{array}{c}\mathrm{TVE} = \{\mathrm{tyv}_i \mapsto \alpha_i \mid 1 \le i \le k \text{ and } \alpha_i \text{ are fresh}\} \\ \mathrm{TE}, \mathrm{TVE} \vdash \mathit{ty} \Rightarrow \tau \qquad \mathrm{TE}' = \mathrm{TE} \pm \{\mathrm{tyc} \mapsto (\langle \alpha_1, \ldots, \alpha_k \rangle, \tau)\}\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{type (}\mathrm{tyv}_1, \ldots, \mathrm{tyv}_k\texttt{)}\ \mathrm{tyc}\ \texttt{=}\ \mathit{ty} \Rightarrow \mathrm{TE}', \mathrm{VE}} \quad (5)$$

The rule for datatype definitions is somewhat complicated. We introduce a fresh atomic tycon T^k representing the new datatype, and bind it to the datatype name in an extended type environment TE'. We check each of the constructor declarations in a context that includes the type parameters variables and result type; these checks yield single constructor environments that are merged to produce the final constructor environment.

$$\frac{\begin{array}{c}\mathrm{TVE} = \{\mathrm{tyv}_i \mapsto \alpha_i \mid 1 \le i \le k \text{ and } \alpha_i \text{ fresh}\} \qquad \mathrm{T}^k \in \textsc{AtomCon}\ (\mathrm{T}^k \text{ fresh}) \\ \mathrm{TE}' = \mathrm{TE} \pm \{\mathrm{tyc} \mapsto \mathrm{T}^k\} \qquad \mathrm{D} = (\langle \alpha_1, \ldots, \alpha_k \rangle, \mathrm{T}^k) \\ \mathrm{TE}', \mathrm{TVE}, \mathrm{D} \vdash \mathit{condcl}_i \Rightarrow \mathrm{VE}_i \ \ (1 \le i \le n) \qquad \mathrm{VE}' = \mathrm{VE} \pm \mathrm{VE}_1 \pm \cdots \pm \mathrm{VE}_n\end{array}}{\mathrm{TE}, \mathrm{VE} \vdash \texttt{datatype}\ \mathrm{tyc}\ \texttt{(}\mathrm{tyv}_1, \ldots, \mathrm{tyv}_k\texttt{)}\ \texttt{=}\ \mathit{condcl}_1\ \texttt{|}\ \cdots\ \texttt{|}\ \mathit{condcl}_n \Rightarrow \mathrm{TE}', \mathrm{VE}'} \quad (6)$$

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.

$$\frac{\mathrm{TVE}(\mathrm{tyv}) = \alpha}{\mathrm{TE}, \mathrm{TVE} \vdash \mathrm{tyv} \Rightarrow \alpha} \quad (7)$$

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:

$$\frac{\begin{array}{c}\mathrm{TE}(\mathrm{tyc}) = \lambda(\alpha_1, \ldots, \alpha_k).\tau \\ \mathrm{TE}, \mathrm{TVE} \vdash \mathit{ty}_1 \Rightarrow \tau_1 \quad \cdots \quad \mathrm{TE}, \mathrm{TVE} \vdash \mathit{ty}_k \Rightarrow \tau_k\end{array}}{\mathrm{TE}, \mathrm{TVE} \vdash \mathrm{tyc}\ \texttt{(}\mathit{ty}_1, \ldots, \mathit{ty}_k\texttt{)} \Rightarrow [\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k]\tau} \quad (8)$$

For applicatons of datatypes and primitive tycons, we translate the arguments and map the tyc identifier to the associated type constructor.

$$\frac{\mathrm{TE}(\mathrm{tyc}) = T^k \quad \mathrm{TE}, \mathrm{TVE} \vdash ty_i \Rightarrow \tau_i \;\; (1 \leq i \leq k)}{\mathrm{TE}, \mathrm{TVE} \vdash \mathrm{tyc}\ \texttt{(}ty_1, \ldots, ty_k\texttt{)} \Rightarrow T^k\,(\tau_1, \ldots, \tau_k)} \quad (9)$$

Translating a function type requires translating the two sides of the arrow.

$$\frac{\mathrm{TE}, \mathrm{TVE} \vdash ty_1 \Rightarrow \tau_1 \quad \mathrm{TE}, \mathrm{TVE} \vdash ty_2 \Rightarrow \tau_2}{\mathrm{TE}, \mathrm{TVE} \vdash ty_1\texttt{->}ty_2 \Rightarrow \tau_1 \to \tau_2} \quad (10)$$

Translating a product type requires translating the component types.

$$\frac{\mathrm{TE}, \mathrm{TVE} \vdash ty_1 \Rightarrow \tau_1 \quad \cdots \quad \mathrm{TE}, \mathrm{TVE} \vdash ty_n \Rightarrow \tau_n}{\mathrm{TE}, \mathrm{TVE} \vdash ty_1\texttt{*}\cdots\texttt{*}ty_n \Rightarrow \tau_1 \times \cdots \times \tau_n} \quad (11)$$

6.5 Data-constructor definitions

The typing rules for a nullary data-constructor specification is

$$\frac{\sigma = \forall\alpha_1, \ldots, \alpha_k.\mathrm{T}^k(\alpha_1, \ldots, \alpha_k)}{\mathrm{TE}, \mathrm{TVE}, (\langle\alpha_1, \ldots, \alpha_k\rangle, \mathrm{T}^k) \vdash \mathrm{con} \Rightarrow \{\mathrm{con} \mapsto \sigma\}} \quad (12)$$

and the rule for a data-constructor function is

$$\frac{\mathrm{TE}, \mathrm{TVE} \vdash ty \Rightarrow \tau' \qquad \sigma = \forall\alpha_1, \ldots, \alpha_k.\tau' \to \mathrm{T}^k\,(\alpha_1, \ldots, \alpha_k)}{\mathrm{TE}, \mathrm{TVE}, (\langle\alpha_1, \ldots, \alpha_k\rangle, \mathrm{T}^k) \vdash \mathrm{con}(ty) \Rightarrow \{\mathrm{con} \mapsto \sigma\}} \quad (13)$$

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, but with its types generalized with respect to the original environment.

$$\frac{\mathrm{VE} \vdash pat \Rightarrow (\mathrm{VE}', \tau) \qquad \mathrm{VE} \vdash exp \Rightarrow \tau}{\mathrm{VE} \vdash \texttt{val}\ pat\ \texttt{=}\ exp \Rightarrow \mathrm{VE} \pm \mathrm{Clos}(\mathrm{VE}, \mathrm{VE}')} \quad (14)$$

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.

$$\frac{\begin{array}{c}\mathrm{VE}' = \mathrm{VE} \pm \{f_i \mapsto \tau_{f_i} \mid f_i = \mathrm{NameOf}(fb_i)\} \\ \mathrm{VE}' \vdash fb_i \Rightarrow \tau_{f_i} \quad \text{for } 1 \leq i \leq n \\ \mathrm{VE}'' = \mathrm{VE} \pm \{f_i \mapsto \sigma_{f_i} \mid \sigma_{f_i} = \mathrm{Clos}(\mathrm{VE}, \tau_{f_i})\}\end{array}}{\mathrm{VE} \vdash \texttt{fun}\ fb_1\ \texttt{and}\ \cdots\ \texttt{and}\ fb_n \Rightarrow \mathrm{VE}''} \quad (15)$$

In these rules, we use the Clos function. $\mathrm{Clos}(\mathrm{VE}, \tau)$ polymorphically quantifies any type variables that are free in τ, but not free in the bindings in VE. When applied to an environment, as in $\mathrm{Clos}(\mathrm{VE}, \mathrm{VE}')$, it similarly generalizes all the the types in the range of that the environment VE'.

6.7 Function bindings

Typechecking a function binding (*fb* = f *pat* **=** *exp*) requires checking the parameter pattern and then using its bindings to check the function body.

$$\frac{\mathrm{VE} \vdash pat \Rightarrow (\mathrm{VE}', \tau) \qquad \mathrm{VE} \pm \mathrm{VE}' \vdash exp \Rightarrow \tau'}{\mathrm{VE} \vdash \mathrm{var}\ pat \texttt{ = } exp \Rightarrow \tau \to \tau'} \quad (16)$$

6.8 Expressions

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

$$\frac{\mathrm{VE} \vdash valdcl \Rightarrow \mathrm{VE}' \qquad \mathrm{VE}' \vdash exp \Rightarrow \tau}{\mathrm{VE} \vdash \texttt{let}\ valdcl\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \quad (17)$$

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

$$\frac{\mathrm{VE} \vdash exp \Rightarrow \tau \qquad \mathrm{VE} \vdash match \Rightarrow (\tau, \tau')}{\mathrm{VE} \vdash \texttt{case}\ exp\ \texttt{of}\ match\ \texttt{end} \Rightarrow \tau'} \quad (18)$$

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

$$\frac{\mathrm{VE} \vdash exp_1 \Rightarrow \mathbf{Bool}^0 \qquad \mathrm{VE} \vdash exp_2 \Rightarrow \tau \qquad \mathrm{VE} \vdash exp_3 \Rightarrow \tau}{\mathrm{VE} \vdash \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \Rightarrow \tau} \quad (19)$$

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

$$\frac{\mathrm{VE} \vdash exp_1 \Rightarrow \tau \to \tau' \qquad \mathrm{VE} \vdash exp_2 \Rightarrow \tau}{\mathrm{VE} \vdash exp_1\ exp_2 \Rightarrow \tau'} \quad (20)$$

Typechecking the equality operator, which we use as a representative of the other relational operators, requires a special rule, because equality is an *ad hoc* polymorphic operator, also known as an overloaded operator. The operators **`==`** and **`<>`** are defined for Int, Bool, and Str, while the other relational operators are only defined for Int and Str.

$$\frac{\mathrm{VE} \vdash exp_1 \Rightarrow \tau \qquad \mathrm{VE} \vdash exp_2 \Rightarrow \tau \qquad \tau \in \{\mathbf{Bool}^0, \mathbf{Int}^0, \mathbf{Str}^0\}}{\mathrm{VE} \vdash exp_1 \texttt{ == } exp_2 \Rightarrow \mathbf{Bool}^0} \quad (21)$$

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

$$\frac{}{\mathrm{VE} \vdash \texttt{()} \Rightarrow \mathbf{Unit}^0} \quad (22)$$

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

$$\frac{\mathrm{VE} \vdash exp_i \Rightarrow \tau_i \quad \text{for } 1 \le i \le n}{\mathrm{VE} \vdash \texttt{(}\ exp_1\texttt{,}\ \ldots\texttt{,}\ exp_n\ \texttt{)} \Rightarrow \tau_1 \times \cdots \times \tau_n} \quad (23)$$

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

$$\frac{\mathrm{VE} \vdash exp_1 \Rightarrow \tau_1 \qquad \mathrm{VE} \vdash exp_2 \Rightarrow \tau_2}{\mathrm{VE} \vdash exp_1\texttt{;}\ exp_2 \Rightarrow \tau_2} \quad (24)$$

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.

$$\frac{\mathrm{VE}(\mathrm{var}) = \sigma \qquad \tau \prec \sigma}{\mathrm{VE} \vdash \mathrm{var} \Rightarrow \tau} \quad (25)$$

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

$$\frac{\mathrm{TypeOf}(\mathrm{lit}) = \tau}{\mathrm{VE} \vdash \mathrm{lit} \Rightarrow \tau} \quad (26)$$

Function expressions are typed by typing the parameter pattern and then typing the body expression in the base environment (VE) augmented with parameter variable bindings (VE').

$$\frac{\mathrm{VE} \vdash \mathit{pat} \Rightarrow (\mathrm{VE}', \tau) \qquad \mathrm{VE} \pm \mathrm{VE}' \vdash \mathit{exp} \Rightarrow \tau'}{\mathrm{VE} \vdash \texttt{fn}\ \mathit{pat}\ \texttt{=>}\ \mathit{exp} \Rightarrow \tau \to \tau'} \quad (27)$$

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.

$$\frac{\mathrm{VE} \vdash \mathit{match}_1 \Rightarrow (\tau, \tau') \qquad \mathrm{VE} \vdash \mathit{match}_2 \Rightarrow (\tau, \tau')}{\mathrm{VE} \vdash \mathit{match}_1\ \texttt{|}\ \mathit{match}_2 \Rightarrow (\tau, \tau')} \quad (28)$$

Checking an match rule requires checking the r.h.s. expression in an environment enriched by the bindings from the l.h.s. pattern (i.e. the same process as for function expressions).

$$\frac{\mathrm{VE} \vdash \mathit{pat} \Rightarrow (\mathrm{VE}', \tau) \qquad \mathrm{VE} \pm \mathrm{VE}' \vdash \mathit{exp} \Rightarrow \tau'}{\mathrm{VE} \vdash \mathit{pat}\ \texttt{=>}\ \mathit{exp} \Rightarrow (\tau, \tau')} \quad (29)$$

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.

$$\frac{\mathrm{VE}(\mathrm{con}) = \sigma \qquad \tau \to \tau' \prec \sigma \qquad \mathrm{VE} \vdash \mathit{pat} \Rightarrow (\mathrm{VE}', \tau)}{\mathrm{VE} \vdash \mathrm{con}\ \mathit{pat} \Rightarrow (\mathrm{VE}', \tau')} \quad (30)$$

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

$$\frac{\mathrm{VE} \vdash \mathit{pat}_i \Rightarrow (\mathrm{VE}_i, \tau_i)\ \ (1 \le i \le n) \qquad \mathrm{VE}' = \mathrm{VE}_1 \pm \cdots \pm \mathrm{VE}_n}{\mathrm{VE} \vdash \texttt{(}\ \mathit{pat}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{pat}_n\ \texttt{)} \Rightarrow (\mathrm{VE}', \tau_1 \times \cdots \times \tau_n)} \quad (31)$$

Value identifiers introduce new bindings. Not that this rule is nondeterministic, since the type assigned to the pattern variable can be any arbitrary τ.

$$\frac{}{\mathrm{VE} \vdash \mathrm{var} \Rightarrow (\{\mathrm{var} \mapsto \tau\}, \tau)} \quad (32)$$

Nullary constructors can be typed with any instance of the type assigned to them by the current environment.

$$\frac{\mathrm{VE}(\mathrm{con}) = \sigma \qquad \tau \prec \sigma}{\mathrm{VE} \vdash \mathrm{con} \Rightarrow (\{\}, \tau)} \quad (33)$$

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

$$\frac{\mathrm{TypeOf}(\mathrm{lit}) = \tau}{\mathrm{VE} \vdash \mathrm{lit} \Rightarrow (\{\}, \tau)} \quad (34)$$

7 Predefined types and operators

Your typechecker will typecheck programs in the context of an *initial basis* $\mathrm{TE}_0, \mathrm{VE}_0$. This basis is defined as follows:

$$\mathrm{TE}_0 = \left\{ \begin{array}{rcl} \texttt{Bool} & \mapsto & \mathbf{Bool}^0 \\ \texttt{Int} & \mapsto & \mathbf{Int}^0 \\ \texttt{list} & \mapsto & \mathbf{list}^{(1)} \\ \texttt{Str} & \mapsto & \mathbf{Str}^0 \\ \texttt{Unit} & \mapsto & \mathbf{Unit}^0 \end{array} \right\}$$

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

$$\mathrm{VE}_0 = \left\{ \begin{array}{rcl} \texttt{False} & \mapsto & \mathbf{Bool}^0 \\ \texttt{True} & \mapsto & \mathbf{Bool}^0 \\ \texttt{Unit} & \mapsto & \mathbf{Unit}^0 \\ \texttt{Nil} & \mapsto & \forall\alpha.\mathbf{List}^1(\alpha) \\ \texttt{Cons} & \mapsto & \forall\alpha.(\alpha \times \mathbf{List}^1(\alpha)) \to \mathbf{List}^1(\alpha) \\ \texttt{<=} & \mapsto & (\mathbf{Int}^0 \times \mathbf{Int}^0) \to \mathbf{Bool}^0 \\ \texttt{<} & \mapsto & (\mathbf{Int}^0 \times \mathbf{Int}^0) \to \mathbf{Bool}^0 \\ \texttt{@} & \mapsto & \forall\alpha.(\mathbf{List}^1(\alpha) \times \mathbf{List}^1(\alpha)) \to \mathbf{List}^1(\alpha) \\ \texttt{+} & \mapsto & (\mathbf{Int}^0 \times \mathbf{Int}^0) \to \mathbf{Int}^0 \\ \texttt{-} & \mapsto & (\mathbf{Int}^0 \times \mathbf{Int}^0) \to \mathbf{Int}^0 \\ \texttt{*} & \mapsto & (\mathbf{Int}^0 \times \mathbf{Int}^0) \to \mathbf{Int}^0 \\ \texttt{div} & \mapsto & (\mathbf{Int}^0 \times \mathbf{Int}^0) \to \mathbf{Int}^0 \\ \texttt{mod} & \mapsto & (\mathbf{Int}^0 \times \mathbf{Int}^0) \to \mathbf{Int}^0 \\ \texttt{~} & \mapsto & \mathbf{Int}^0 \to \mathbf{Int}^0 \\ \texttt{\^{}} & \mapsto & \mathbf{Str}^0 \times \mathbf{Str}^0 \to \mathbf{Str}^0 \\ \texttt{fail} & \mapsto & \forall\alpha.\mathbf{Str}^0 \to \alpha \\ \texttt{print} & \mapsto & \mathbf{Str}^0 \to \mathbf{Unit}^0 \end{array} \right\}$$

Note that we have omitted some of the relational operators for brevity here, but they should also be included in the initial environment. We have also added two special functions, `fail` and `print` that have side effects.

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 **&&** and **||** are translated to **if** expressions as follows:

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

The null-tuple **()** is translated to the constructor Unit of the predefined datatype Unit.

9 Discussion

These typing or elaboration rules are used to form *typing derivations* for expressions, patterns, declarations, and programs. The rules are nondeterministic, because at various points (Rules 15, 25, 30, 32, 33) we are allowed to pull a type (or an instantiation of a polytype) out of our hat. So a given expression may have many different typings that can be derived by these rules. For instance, (**fn** x **=>** x) has types an infinite number of typings, including $\mathbf{Int}^0 \to \mathbf{Int}^0$ and $\mathbf{Bool}^0 \to \mathbf{Bool}^0$.

The problem we face is to implement a type checking algorithm that chooses the *best* typiing, in some sense. Our notion of the best typing is a *principal* typing, meaning a typing that is the most general possible; every other typing can be obtained by specializing (or *instantiating*) the principal typing. Milner in 1978 described an algorithm for finding this principal typing, called Algorithm W. We will discuss Algorithm W in Handout 5.

10 Requirements

10.1 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 2, unbound identifiers, and any type errors.

10.2 Submission

We will set up a project3 directory in your phoenixforge repository, containing the reference lexer and parser implementations, and the syntax tree definitions in syntax.sml. You should use this repository to hold the source for your project. We will check out your project3 code at 11pm on Tuesday, March 1st, so make sure that you have committed your final version before then.

11 Document history