CMSC 22610 | **Implementation of** | **Project 3**
Winter 2009 | **Computer Languages - I** | **February 12, 2009**

LangF Type Checker

Due: February 27, 2009

1 Introduction

The third project is to implement a type checker for LangF, which checks whether or not a parse tree is *statically correct* and produces a *typed abstract syntax tree* (AST). The abstract syntax tree includes information about the binding sites of identifiers and about the types of variables and expressions. The project seed code will provide hand-written and ML-ULex based scanners (but you may also use your hand-written scanner from Projects 1 and 2), ML-Antlr and ML-Yacc based parsers (but you may also use your parser specification from Project 2), and modules for implementing the abstract-syntax-tree representation. The bulk of this document is a formal specification of the typing rules for LangF.[1] The type system for LangF is essentially an enrichment of the *System F* type system.

2 LangF Syntactic Restrictions

There are a number of syntactic restrictions that should be enforced by the type checker. Although these restrictions could be specified as part of the typing rules below, it is easier to specify them separately.

- The type variable identifiers in the type parameters of a **`type`** declaration must be distinct.
- The type variable identifiers in the type parameters of each *DataDecl* of a **`datatype`** declaration must be distinct.
- The type constructor identifiers in a **`datatype`** declaration must be distinct.
- The data constructor identifiers in a **`datatype`** declaration must be distinct. (On the other hand, a **`datatype`** declaration may introduce the same constructor name for both a type constructor and a data constructor. For example, **`datatype`** `Unit = Unit`.)
- The variable identifiers denoting function variable identifiers in a **`fun`** declaration must be distinct.
- The variable identifiers in a pattern must be distinct.
- Integer literals must be in the range $-2^{30} \ldots 2^{30} - 1$ (*i.e.*, representable as a 31-bit 2's-complement integer).
- **Extra credit:** The patterns in a **`case`** expression must be *irredundant* and *exhaustive*. Consider the match rules Pat_1 **`=>`** Exp_1 **`|`** $\cdots$ **`|`** Pat_1 **`=>`** Exp_n. For the patterns to be irredundant, each Pat_j must match some value (of the right type) that is not matched by Pat_i for any $i < j$. For the patterns to be exhaustive, every value (of the right type) must be matched by some Pat_i.

[1] Remember, a *specification* is a description of a property (yes/no question; true/false statement). It does not define (though it may suggest) an *implementation* for deciding whether or not the property holds. A significant component of this project is to develop the skills needed to produce an impementation from a specification.

$$
\begin{array}{rcll}
\tau \in \textsc{Type} & ::= & \forall\alpha \rightarrow \tau_r & \text{type function, } \alpha \in \textsc{TyVar} \\
& | & \tau_a \rightarrow \tau_r & \text{function} \\
& | & \theta^{(k)}(\tau_1, \ldots, \tau_k) & \text{type constructor, } \theta \in \textsc{TyCon} \\
& | & \alpha & \text{type variable}
\end{array}
$$

Figure 1: LangF semantic types

3 LangF Types

In the LangF typing rules, we distinguish between *syntactic types* as they appear in the program text (or parse-tree representation) and *semantic types* that are inferred for various syntactic forms. To understand why we make this distinction, consider the following LangF program:

```
datatype T = A {Integer} | B
val x : T = A {1}
datatype T = C {Integer} | D
val y : T = B
; 0
```

There is a type error at line 4 in the declaration **`val`** `y : T = B`, because the type of the data constructor expression `B` is the type constructor corresponding to the **`datatype`** declaration at line 1, but the type constraint `T` is the type constructor corresponding to the **`datatype`** declaration at line 3. The **`datatype`** declaration at line 3 *shadows* the **`datatype`** declaration at line 1. However, in the parse-tree representation, all instances of `T` correspond to the same type constructor name (that is, as values of the `ParseTree.TyVarName.t` type, they all carry the same `Atom.atom` in the `node` field).

The abstract syntax of LangF semantic types is given in Figure 1 (and represented by the `AbsSynTree.Type.t` datatype in the project seed code). A semantic type is formed from type variables, type constructors (including nullary and abstract type constructors like $\mathbf{bool}^{(0)}$ and $\mathbf{integer}^{(0)}$), function types, and type-function types. (Note that a k-arity type constructor records its arity as a superscript.)

Although this syntax mirrors that of the syntactic types, there are some crucial differences. First, there will be distinct type variables (α) and type constructors (θ) for each binding occurence of a type variable name and type constructor name in the parse-tree representation. Hence, type checking the **`datatype`** declaration at line 1 will introduce one type constructor, say $\mathbf{t}_1^{(0)}$, and type checking the **`datatype`** declaration at line 3 will introduce a different type constructor, say $\mathbf{t}_2^{(0)}$, which are not equal. Second, we will consider semantic types equal upto renaming of type-function type variables. That is, we will consider the semantic types $\forall\alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \mathbf{bool}^{(0)}$ and $\forall\beta \rightarrow \beta \rightarrow \beta \rightarrow \mathbf{bool}^{(0)}$ to be equal, whereas the parse trees corresponding to `['a] -> 'a -> 'a -> bool` and `['b] -> 'b -> 'b -> bool` are not equal, because they use different type variable names.

4 Identifiers and Environments

The typing rules for LangF use a number of different environments to track binding information. There is a separate environment for each kind of identifier in the parse-tree repesentation:

$$
\begin{array}{rcll}
\text{TVE} & \in & \textsc{TyVarEnv} = \textit{TyVarId} \rightharpoonup \textsc{TyVar} & \text{type-variable environment} \\
\text{TCE} & \in & \textsc{TyConEnv} = \textit{TyConId} \rightharpoonup \textsc{TyCon} \cup (\textsc{TyVar}^* \times \textsc{Type}) & \text{type-constructor environment} \\
\text{DCE} & \in & \textsc{DaConEnv} = \textit{DaConId} \rightharpoonup \textsc{TyVar}^* \times \textsc{Type}^* \times \textsc{TyCon} & \text{data-constructor environment} \\
\text{VE} & \in & \textsc{VarEnv} = \textit{VarId} \rightharpoonup \textsc{Type} & \text{variable environment}
\end{array}
$$

where *TyVarId* is the set of syntactic type-variable identifiers (`'a` in the LangF concrete syntax, *tyvarid* in the LangF grammar, or `ParseTree.TyVarName.t` in the project seed code), *TyConId* is the set of syn-

$$
\begin{array}{rl}
\mathrm{E} \vdash \ \textit{Type} \ \Rightarrow \tau & \text{type checking a type} \\
\mathrm{E} \vdash \ \textit{Param} \ \Rightarrow \mathrm{E}'; \mathcal{T} & \text{type checking a parameter} \\
\mathrm{E}; \tau \vdash \ \textit{SimplePat} \ \Rightarrow \mathrm{E}' & \text{type checking a simple pattern} \\
\mathrm{E}; \tau \vdash \ \textit{Pat} \ \Rightarrow \mathrm{E}' & \text{type checking a pattern} \\
\mathrm{E} \vdash \ \textit{Exp} \ \Rightarrow \tau & \text{type checking an expression} \\
\mathrm{E}; \tau \vdash \ \textit{MatchRule} \ \Rightarrow \tau' & \text{type checking a match rule} \\
\mathrm{E} \vdash \ \textit{Decl} \ \Rightarrow \mathrm{E}' & \text{type checking a declaration} \\
\mathrm{E}; \langle \alpha_1, \ldots, \alpha_n \rangle; \theta^{(n)} \vdash \ \textit{DaConDecl} \ \Rightarrow \mathrm{E}' & \text{type checking a data constructor declaration} \\
\vdash \ \textit{Prog} \ \Rightarrow \checkmark & \text{type checking a program}
\end{array}
$$

Figure 2: LangF judgement forms

tactic type-constructor identifiers (`T`, *tyconid*, or `ParseTree.TyConName.t`), *DaConId* is the set of syntactic data-constructor identifiers (`A`, *daconid*, or `ParseTree.DaConName.t`), *VarId* is the set of syntactic variable identifiers (`x`, *varid*, or `ParseTree.VarName.t`), TYPE is the set of semantic types (τ in Figure 1 or `AbsSynTree.Type.t` in the project seed code), TYVAR is the set of semantic type variables (α or `AbsSynTree.TyVar.t`), and TYCON is the set of semantic type constructors (θ or `AbsSynTree.TyCon.t`).

We define the extension of an environment E by another environment E′ as follows:

$$
(\mathrm{E} \oplus \mathrm{E}')(\mathrm{x}) \quad = \quad \begin{cases} \mathrm{E}'(\mathrm{x}) & \text{if } \mathrm{x} \in \mathrm{dom}(\mathrm{E}') \\ \mathrm{E}(\mathrm{x}) & \text{if } \mathrm{x} \notin \mathrm{dom}(\mathrm{E}') \end{cases}
$$

Since each of the environments has a different domain, it is convenient to consider a combined environment:

$$
\mathrm{E} \ \in \ \textsc{Env} = \textsc{TyVarEnv} \times \textsc{TyConEnv} \times \textsc{DaConEnv} \times \textsc{VarEnv} \quad \text{combined environment}
$$

For combined environments $\mathrm{E} = \langle \mathrm{TVE}, \mathrm{TCE}, \mathrm{DCE}, \mathrm{VE} \rangle$ and $\mathrm{E}' = \langle \mathrm{TVE}', \mathrm{TCE}', \mathrm{DCE}', \mathrm{VE}' \rangle$, we define lookup by lookup in the environment appropriate to the identifier:

$$
\mathrm{E}(\mathrm{x}) \quad = \quad \begin{cases} \mathrm{TVE}(\mathrm{x}) & \text{if } \mathrm{x} \in \textit{TyVarId} \\ \mathrm{TCE}(\mathrm{x}) & \text{if } \mathrm{x} \in \textit{TyConId} \\ \mathrm{DCE}(\mathrm{x}) & \text{if } \mathrm{x} \in \textit{DaConId} \\ \mathrm{VE}(\mathrm{x}) & \text{if } \mathrm{x} \in \textit{VarId} \end{cases}
$$

and define extension by extension of each environment:

$$
\mathrm{E} \oplus \mathrm{E}' \quad = \quad \langle \mathrm{TVE} \oplus \mathrm{TVE}', \mathrm{TCE} \oplus \mathrm{TCE}', \mathrm{DCE} \oplus \mathrm{DCE}', \mathrm{VE} \oplus \mathrm{VE}' \rangle
$$

5 LangF Typing Rules

The typing rules for LangF provide a specification for the static correctness of LangF programs. The general form of a judgement, as used in the LangF typing rules, is

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

which can be read as "in *Context*, *Term* has *Descr*." The context is usually an environment, but may include other information, while the description is usually a semantic type and/or an (extended) environment. The judgement forms used in the typing rules for LangF are given in Figure 2.

The typing rules for LangF are *syntax directed*, which means that there is a typing rule for each (major) syntactic form in the parse-tree representation of LangF programs. In the following, a typing rule is labelled by the SML data constructor in the `ParseTree` module that corresponds to the syntactic form handled by the typing rule.

5.1 Types

The typing rules for types check for well-formedness and translate the syntactic types to semantic types. The typing rules for types use a judgement of the form

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

which can be read as "in the environment E, the syntactic type *Type* is well-formed and translates to the semantic type τ."

Type checking a type-function type requires introducing a new semantic type variable (α) for the syntactic type variable identifier (*tyvarid*) when checking the result type.

$$\frac{\mathrm{E} \oplus \{\mathit{tyvarid} \mapsto \alpha \mid \alpha \text{ fresh}\} \vdash \ \mathit{Type}_r \ \Rightarrow \tau_r}{\mathrm{E} \vdash \ \texttt{[}\ \mathit{tyvarid}\ \texttt{]}\ \texttt{->}\ \mathit{Type}_r \ \Rightarrow \forall\alpha \to \tau_r}\ \text{T\_TyFn}$$

Type checking a function type requires checking the argument type and the result type.

$$\frac{\mathrm{E} \vdash \ \mathit{Type}_a \ \Rightarrow \tau_a \qquad \mathrm{E} \vdash \ \mathit{Type}_r \ \Rightarrow \tau_r}{\mathrm{E} \vdash \ \mathit{Type}_a\ \texttt{->}\ \mathit{Type}_r \ \Rightarrow \tau_a \to \tau_r}\ \text{T\_Fn}$$

There are two rules for type checking a type-constructor application, depending on whether the type constructor identifier corresponds to a **`type`** definition or a **`datatype`** definition (or an abstract type). For **`type`** definitions, we check the actual (syntactic) type arguments and then substitute the actual (semantic) type arguments for the formal type parameters to produce a new (semantic) type.

$$\frac{\begin{array}{c}\mathit{tyconid} \in \mathrm{dom}(\mathrm{E}) \qquad \mathrm{E}(\mathit{tyconid}) = (\langle\alpha_1, \ldots, \alpha_n\rangle, \tau) \\ \mathrm{E} \vdash \ \mathit{Type}_1 \ \Rightarrow \tau_1 \qquad \cdots \qquad \mathrm{E} \vdash \ \mathit{Type}_n \ \Rightarrow \tau_n\end{array}}{\mathrm{E} \vdash \ \mathit{tyconid}\ \texttt{[}\ \mathit{Type}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Type}_n\ \texttt{]} \ \Rightarrow \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}\ \text{T\_TyCon(}\texttt{type}\text{)}$$

For **`datatype`** definitions (or abstract types), we check the type arguments and then construct a new (semantic) type.

$$\frac{\begin{array}{c}\mathit{tyconid} \in \mathrm{dom}(\mathrm{E}) \qquad \mathrm{E}(\mathit{tyconid}) = \theta^{(n)} \\ \mathrm{E} \vdash \ \mathit{Type}_1 \ \Rightarrow \tau_1 \qquad \cdots \qquad \mathrm{E} \vdash \ \mathit{Type}_n \ \Rightarrow \tau_n\end{array}}{\mathrm{E} \vdash \ \mathit{tyconid}\ \texttt{[}\ \mathit{Type}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Type}_n\ \texttt{]} \ \Rightarrow \theta^{(n)}(\tau_1, \ldots, \tau_n)}\ \text{T\_TyCon(}\texttt{datatype}\text{)}$$

Type checking a type variable identifier returns its semantic type variable, as recorded in the environment.

$$\frac{\mathit{tyvarid} \in \mathrm{dom}(\mathrm{E}) \qquad \mathrm{E}(\mathit{tyvarid}) = \alpha}{\mathrm{E} \vdash \ \mathit{tyvarid} \ \Rightarrow \alpha}\ \text{T\_TyVar}$$

5.2 Parameters

Type checking a parameter returns a new environment, which includes a binding for the type variable identifier or variable identifier in the parameter, and a meta-function that constructs a (semantic) type when given the (semantic) type of the expression over which the parameter abstracts. Note that the output environment only includes bindings for the identifiers appearing in the parameter; the output environment is not an extension of the input environment.

Type checking a variable parameter requires type checking the declared type and returns an environment with a single binding and a meta-function that creates a (semantic) function type.

$$\frac{\mathrm{E} \vdash \ \mathit{Type} \ \Rightarrow \tau \qquad \mathrm{E}' = \{\mathit{varid} \mapsto \tau\} \qquad \mathcal{T}'(\mathrm{x}) = \tau \to \mathrm{x}}{\mathrm{E} \vdash \ \texttt{(}\ \mathit{varid}\ \texttt{:}\ \mathit{Type}\ \texttt{)} \ \Rightarrow \mathrm{E}'; \mathcal{T}'}\ \text{P\_VarName}$$

Type checking a type variable parameter requires introducing a new semantic type variable (α) and returns an environment with a single binding and a meta-function that creates a (semantic) type-function type.

$$\frac{\mathrm{E}' = \{\mathit{tyvarid} \mapsto \alpha \mid \alpha \text{ fresh}\} \qquad \mathcal{T}'(\mathrm{x}) = \forall\alpha \to \mathrm{x}}{\mathrm{E} \vdash \ \texttt{[}\ \mathit{tyvarid}\ \texttt{]} \ \Rightarrow \mathrm{E}'; \mathcal{T}'}\ \text{P\_TyVarName}$$

5.2.1 Multiple Parameters

Type checking a list of parameters requires type checking each paramter in turn. Since type variable identifiers in earlier parameters are bound in later parameters, each parameter is type checked in an environment extended with output environments of each earlier parameter. The final output environment is the extension of all the individual output environments, while the final meta-function is the composition of all the individual meta-functions.

$$\frac{\begin{array}{c} n \geq 0 \qquad \mathrm{E} \vdash \ \mathit{Param}_1 \ \Rightarrow \mathrm{E}_1; \mathcal{T}_1 \\ \cdots \qquad \mathrm{E} \oplus \mathrm{E}_1 \oplus \cdots \oplus \mathrm{E}_{n-1} \vdash \ \mathit{Param}_n \ \Rightarrow \mathrm{E}_n; \mathcal{T}_n \\ \mathrm{E}' = \mathrm{E}_1 \oplus \cdots \oplus \mathrm{E}_{n-1} \oplus \mathrm{E}_n \qquad \mathcal{T}' = \mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n \end{array}}{\mathrm{E} \vdash \ \mathit{Param}_1 \ \cdots \ \mathit{Param}_n \ \Rightarrow \mathrm{E}'; \mathcal{T}'} \ \mathit{Params}$$

5.3 Simple Patterns and Patterns

Type checking a pattern is similar to type checking a parameter, in that it returns a new environment, which includes a binding for any variable identifier in the pattern. Again, the output environment only includes bindings for the identifiers appearing in the pattern; the output environment is not an extension of the input environment. The typing rules for patterns use judgements of the form:

$$\mathrm{E}; \tau \vdash \ \mathit{SimplePat} \ \Rightarrow \mathrm{E}' \qquad\qquad \mathrm{E}; \tau \vdash \ \mathit{Pat} \ \Rightarrow \mathrm{E}'$$

where τ is the (semantic) type of values matched by the pattern.

5.3.1 Simple Patterns

Type checking a variable identifier simple pattern returns an environment with a single binding (of the variable to the input (semantic) type).

$$\frac{\mathrm{E}' = \{\mathit{varid} \mapsto \tau\}}{\mathrm{E}; \tau \vdash \ \mathit{varid} \ \Rightarrow \mathrm{E}'} \ \text{P\_VarName}$$

Type checking a wildcard simple pattern returns an empty environment.

$$\frac{\mathrm{E}' = \{\}}{\mathrm{E}; \tau \vdash \ \_ \ \Rightarrow \mathrm{E}'} \ \text{P\_Wild}$$

5.3.2 Patterns

Type checking a data constructor pattern requires checking a number of things. First, the (semantic) type of values being matched by the pattern must be a type-constructor type ($\theta^{(n)}(\tau'_1, \ldots, \tau'_n)$). Second, the data constructor identifier is looked up in the environment, returning its semantic type constructor ($\theta^{(n)}$, which must match the type of values being matched by the pattern), the formal types of its arguments ($\langle \tau_1, \ldots, \tau_m \rangle$), and the formal type variables ($\langle \alpha_1, \ldots, \alpha_n \rangle$) over which the type constructor and the formal types of its arguments is abstracted. Third, each of the actual type arguments is type checked ($\mathrm{E} \vdash \ \mathit{Type}_i \ \Rightarrow \tau'_i$). Fourth, each of the constituent simple patterns is type checked with an input type equal to the substitution of the actual type arguments for the formal type variables in the formal type of the corresponding data constructor argument ($\mathrm{E}; \tau_i[\alpha_1/\tau'_1, \ldots, \alpha_n/\tau'_n] \vdash \ \mathit{SimplePat}_i \ \Rightarrow \mathrm{E}_i$). The final output environment is the extension of all the individual output environments of the constituent simple patterns.

$$\frac{\begin{array}{c} \tau = \theta^{(n)}(\tau'_1, \ldots, \tau'_n) \qquad \mathit{daconid} \in \mathrm{dom}(\mathrm{E}) \qquad \mathrm{E}(\mathit{daconid}) = (\langle \alpha_1, \ldots, \alpha_n \rangle, \langle \tau_1, \ldots, \tau_m \rangle, \theta^{(n)}) \\ \mathrm{E} \vdash \ \mathit{Type}_1 \ \Rightarrow \tau'_1 \qquad \cdots \qquad \mathrm{E} \vdash \ \mathit{Type}_n \ \Rightarrow \tau'_n \\ \mathrm{E}; \tau_1[\alpha_1/\tau'_1, \ldots, \alpha_n/\tau'_n] \vdash \ \mathit{SimplePat}_1 \ \Rightarrow \mathrm{E}_1 \\ \cdots \qquad \mathrm{E}; \tau_m[\alpha_1/\tau'_1, \ldots, \alpha_n/\tau'_n] \vdash \ \mathit{SimplePat}_m \ \Rightarrow \mathrm{E}_m \\ \mathrm{E}' = \mathrm{E}_1 \oplus \cdots \oplus \mathrm{E}_m \end{array}}{\mathrm{E}; \tau \vdash \ \mathit{daconid} \ \texttt{[} \ \mathit{Type}_1 \ \texttt{,} \ \ldots \ \texttt{,} \ \mathit{Type}_n \ \texttt{]} \ \texttt{\{} \ \mathit{SimplePat}_1 \ \texttt{,} \ \ldots \ \texttt{,} \ \mathit{SimplePat}_m \ \texttt{\}} \ \Rightarrow \mathrm{E}'} \ \text{P\_DaCon}$$

Note that in the typing rule for type checking a simple pattern, the premise judgement corresponds to the rules given in Section 5.3.1, while the conclusion judgement corresponds to the rules given in this section.

$$\frac{\mathrm{E};\tau \vdash \ \mathit{SimplePat} \ \Rightarrow \mathrm{E}'}{\mathrm{E};\tau \vdash \ \mathit{SimplePat} \ \Rightarrow \mathrm{E}'}\ \text{P\_SIMPLEPAT}$$

5.4 Expressions

The typing rules for expressions use a judgement of the form

$$\mathrm{E} \vdash \ \mathit{Exp} \ \Rightarrow \tau$$

which can be read as "in the environment E, the expression *Exp* has the semantic type τ."

Type checking an anonymous function requires type checking the parameters and then type checking the function body in an environment extended with the bindings of the parameters. The anonymous function will have either a type-function type or a function type, as constructed by the meta-function from the type checking of the parameters.

$$\frac{\mathrm{E} \vdash \ \mathit{Params} \ \Rightarrow \mathrm{E}';\mathcal{T} \qquad \mathrm{E} \oplus \mathrm{E}' \vdash \ \mathit{Exp} \ \Rightarrow \tau}{\mathrm{E} \vdash \ \texttt{fn}\ \mathit{Params}\ \texttt{=>}\ \mathit{Exp} \ \Rightarrow \mathcal{T}(\tau)}\ \text{E\_FN}$$

Type checking an **`if`** expression requires checking that the condition expression has the boolean type and that the **`then`** expression and the **`else`** expression have the same type.

$$\frac{\mathrm{E} \vdash \ \mathit{Exp}_c \ \Rightarrow \mathbf{bool}^{(0)} \qquad \mathrm{E} \vdash \ \mathit{Exp}_t \ \Rightarrow \tau \qquad \mathrm{E} \vdash \ \mathit{Exp}_e \ \Rightarrow \tau}{\mathrm{E} \vdash \ \texttt{if}\ \mathit{Exp}_i\ \texttt{then}\ \mathit{Exp}_t\ \texttt{else}\ \mathit{Exp}_e \ \Rightarrow \tau}\ \text{E\_IF}$$

Type checking an **`orelse`** or **`andalso`** expression requires that both operands have the boolean type and expression itself has the boolean type.

$$\frac{\mathrm{E} \vdash \ \mathit{Exp}_l \ \Rightarrow \mathbf{bool}^{(0)} \qquad \mathrm{E} \vdash \ \mathit{Exp}_r \ \Rightarrow \mathbf{bool}^{(0)}}{\mathrm{E} \vdash \ \mathit{Exp}_l\ \texttt{orelse}\ \mathit{Exp}_r \ \Rightarrow \mathbf{bool}^{(0)}}\ \text{E\_ORELSE}$$

$$\frac{\mathrm{E} \vdash \ \mathit{Exp}_l \ \Rightarrow \mathbf{bool}^{(0)} \qquad \mathrm{E} \vdash \ \mathit{Exp}_r \ \Rightarrow \mathbf{bool}^{(0)}}{\mathrm{E} \vdash \ \mathit{Exp}_l\ \texttt{andalso}\ \mathit{Exp}_r \ \Rightarrow \mathbf{bool}^{(0)}}\ \text{E\_ANDALSO}$$

Type checking a constraint expression requires checking that the expression and the type constraint have the same (sematic) type.

$$\frac{\mathrm{E} \vdash \ \mathit{Exp} \ \Rightarrow \tau \qquad \mathrm{E} \vdash \ \mathit{Type} \ \Rightarrow \tau}{\mathrm{E} \vdash \ \mathit{Exp}\ \texttt{:}\ \mathit{Type} \ \Rightarrow \tau}\ \text{E\_CONSTRAINT}$$

Type checking the binary and unary operations requires checking that the operands have the types appropriate for the operation.

$$\frac{\mathit{op} \in \{\texttt{==}, \texttt{<>}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\} \qquad \mathrm{E} \vdash \ \mathit{Exp}_l \ \Rightarrow \mathbf{integer}^{(0)} \qquad \mathrm{E} \vdash \ \mathit{Exp}_r \ \Rightarrow \mathbf{integer}^{(0)}}{\mathrm{E} \vdash \ \mathit{Exp}_l\ \mathit{op}\ \mathit{Exp}_r \ \Rightarrow \mathbf{bool}^{(0)}}\ \text{E\_BINOP(CMPOP)}$$

$$\frac{\mathit{op} \in \{\texttt{\^{}}\} \qquad \mathrm{E} \vdash \ \mathit{Exp}_l \ \Rightarrow \mathbf{string}^{(0)} \qquad \mathrm{E} \vdash \ \mathit{Exp}_r \ \Rightarrow \mathbf{string}^{(0)}}{\mathrm{E} \vdash \ \mathit{Exp}_l\ \mathit{op}\ \mathit{Exp}_r \ \Rightarrow \mathbf{string}^{(0)}}\ \text{E\_BINOP(CONCATOP)}$$

$$\frac{\mathit{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\} \qquad \mathrm{E} \vdash \ \mathit{Exp}_l \ \Rightarrow \mathbf{integer}^{(0)} \qquad \mathrm{E} \vdash \ \mathit{Exp}_r \ \Rightarrow \mathbf{integer}^{(0)}}{\mathrm{E} \vdash \ \mathit{Exp}_l\ \mathit{op}\ \mathit{Exp}_r \ \Rightarrow \mathbf{integer}^{(0)}}\ \text{E\_BINOP(ARITHOP)}$$

$$\frac{\mathit{op} \in \{\texttt{\~{}}\} \qquad \mathrm{E} \vdash \ \mathit{Exp} \ \Rightarrow \mathbf{integer}^{(0)}}{\mathrm{E} \vdash \ \mathit{op}\ \mathit{Exp} \ \Rightarrow \mathbf{integer}^{(0)}}\ \text{E\_UNOP(ARITHOP)}$$

Type checking a data constructor expression requires checking a number of things. First, the data constructor identifier is looked up in the environment, returning its semantic type constructor ($\theta^{(n)}$, the formal types of its arguments ($\langle \tau_1, \ldots, \tau_m \rangle$), and the formal type variables ($\langle \alpha_1, \ldots, \alpha_n \rangle$) over which the type constructor and the formal types of its arguments is abstracted. Second, each of the actual type arguments is type checked ($\mathrm{E} \vdash \ \mathit{Type}_i \ \Rightarrow \tau_i'$). Third, each of the actual expression arguments is type checked, and must return a type equal to the substitution of the actual type arguments for the formal type variables in the formal type of the corresponding data constructor argument ($\mathrm{E}; \tau_i[\alpha_1/\tau_1', \ldots, \alpha_n/\tau_n'] \vdash \ \mathit{SimplePat}_i \ \Rightarrow \mathrm{E}_i$). The final output type is the type constructor applied to the actual type arguments.

$$\frac{\begin{array}{c}\mathit{daconid} \in \mathrm{dom}(\mathrm{E}) \qquad \mathrm{E}(\mathit{daconid}) = (\langle \alpha_1, \ldots, \alpha_n \rangle, \langle \tau_1, \ldots, \tau_m \rangle, \theta^{(n)}) \\ \mathrm{E} \vdash \ \mathit{Type}_1 \ \Rightarrow \tau_1' \qquad \cdots \qquad \mathrm{E} \vdash \ \mathit{Type}_n \ \Rightarrow \tau_n' \\ \mathrm{E} \vdash \ \mathit{Exp}_1 \ \Rightarrow \tau_1[\alpha_1/\tau_1', \ldots, \alpha_n/\tau_n'] \qquad \cdots \qquad \mathrm{E} \vdash \ \mathit{Exp}_m \ \Rightarrow \tau_m[\alpha_1/\tau_1', \ldots, \alpha_n/\tau_n']\end{array}}{\mathrm{E} \vdash \ \mathit{daconid}\ \texttt{[}\ \mathit{Type}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Type}_n\ \texttt{]}\ \texttt{\{}\ \mathit{Exp}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Exp}_m\ \texttt{\}} \ \Rightarrow \theta^{(n)}(\tau_1', \ldots, \tau_n')}\ \text{E\_DaCon}$$

There are two rules for type checking an application, depending on the kind of the apply argument. For an expression apply argument, we check that the function expression has a function type and that the argument expression has the argument type.

$$\frac{\mathrm{E} \vdash \ \mathit{Exp}_f \ \Rightarrow \tau_a \to \tau_r \qquad \mathrm{E} \vdash \ \mathit{Exp}_a \ \Rightarrow \tau_a}{\mathrm{E} \vdash \ \mathit{Exp}_f\ \mathit{Exp}_a \ \Rightarrow \tau_r}\ \text{E\_Apply(A\_Exp)}$$

For a type apply argument, we check that the function expression has a type-function type and that the argument type is well-formed. The type of the application expression is the subtitution of the (semantic) type argument for the abstracted type variable in the result type.

$$\frac{\mathrm{E} \vdash \ \mathit{Exp}_f \ \Rightarrow \forall \alpha \to \tau_r \qquad \mathrm{E} \vdash \ \mathit{Type}_a \ \Rightarrow \tau_a}{\mathrm{E} \vdash \ \mathit{Exp}_f\ \texttt{[}\ \mathit{Type}_a\ \texttt{]} \ \Rightarrow \tau_r[\tau_a/\alpha]}\ \text{E\_Apply(A\_Type)}$$

Type checking a variable identifier returns its semantic type, as recorded in the environment.

$$\frac{\mathit{varid} \in \mathrm{dom}(\mathrm{E}) \qquad \mathrm{E}(\mathit{varid}) = \tau}{\mathrm{E} \vdash \ \mathit{varid} \ \Rightarrow \tau}\ \text{E\_VarName}$$

Type checking an integer or string constant returns the obvious type.

$$\frac{}{\mathrm{E} \vdash \ \mathit{integer} \ \Rightarrow \mathbf{integer}^{(0)}}\ \text{E\_Integer} \qquad\qquad \frac{}{\mathrm{E} \vdash \ \mathit{string} \ \Rightarrow \mathbf{string}^{(0)}}\ \text{E\_String}$$

Type checking a sequence expression requires type checking each expression, but only returns the type of the final expression.

$$\frac{n \geq 0 \qquad \mathrm{E} \vdash \ \mathit{Exp}_1 \ \Rightarrow \tau_1 \qquad \cdots \qquad \mathrm{E} \vdash \ \mathit{Exp}_n \ \Rightarrow \tau_n}{\mathrm{E} \vdash \ \texttt{(}\ \mathit{Exp}_1\ \texttt{;}\ \cdots\ \texttt{;}\ \mathit{Exp}_n\ \texttt{)} \ \Rightarrow \tau_n}\ \text{E\_Seq}$$

Type checking a **`let`** expression requires checking the declarations and then type checking the body expression in an environment extended with the bindings of the declarations.

$$\frac{\begin{array}{c}\mathrm{E} \vdash \ \mathit{Decls} \ \Rightarrow \mathrm{E}' \\ n \geq 0 \qquad \mathrm{E} \oplus \mathrm{E}' \vdash \ \mathit{Exp}_1 \ \Rightarrow \tau_1 \qquad \cdots \qquad \mathrm{E} \oplus \mathrm{E}' \vdash \ \mathit{Exp}_n \ \Rightarrow \tau_n \qquad \Theta(\tau_n) \subseteq \Theta(\mathrm{E})\end{array}}{\mathrm{E} \vdash \ \texttt{let}\ \mathit{Decls}\ \texttt{in}\ \mathit{Exp}_1\ \texttt{;}\ \cdots\ \texttt{;}\ \mathit{Exp}_n\ \texttt{end} \ \Rightarrow \tau}\ \text{E\_Let}$$

The $\Theta(\tau_n) \subseteq \Theta(\mathrm{E})$ premise checks that every (semantic) type constructor in τ_n is available in the environment E. We define $\Theta(\tau)$ and $\Theta(\mathrm{E})$ as follows:

$$\begin{array}{rcl}
\Theta(\forall\alpha \to \tau_a) & = & \Theta(\tau_a) \\
\Theta(\tau_a \to \tau_r) & = & \Theta(\tau_a) \cup \Theta(\tau_r) \\
\Theta(\theta^{(k)}(\tau_1, \ldots, \tau_k) & = & \{\theta^{(k)}\} \cup \Theta(\tau_1) \cup \cdots \cup \Theta(\tau_k) \\
\Theta(\alpha) & = & \{\} \\
\\
\Theta(\langle \mathrm{TVE}, \mathrm{TCE}, \mathrm{DCE}, \mathrm{VE} \rangle) & = & \{\theta^{(k)} \mid \mathit{tyconid} \in \mathit{dom}(\mathrm{TCE}) \text{ and } \mathrm{TCE}(\mathit{tyconid}) = \theta^{(k)}\}
\end{array}$$

This prevents type constructors introduced by **`datatype`** declarations in *Decls* from appearing in the type of *Exp*. For example, the following LangF program does not type check:

```
(* Type constructor escapes the scope of its definition. *)
let
  datatype T = A {Integer} | B {String}
  fun f (x: Integer) : T = A {x}
in
  f
end
```

On the other hand, the following LangF program does type check:

```
let
  type T = Integer
  fun f (x: Integer) : T = Integer
in
  f
end
```

because the **`type`** declaration does not introduce a fresh (semantic) type constructor.

Type checking a **`case`** expression requires type checking the scruitinee and checking the match rules against the type of the scruitinee.

$$\frac{\mathrm{E} \vdash \ \mathit{Exp} \ \Rightarrow \tau \qquad \mathrm{E}; \tau \vdash \ \mathit{MatchRules} \ \Rightarrow \tau'}{\mathrm{E} \vdash \ \texttt{case}\ \mathit{Exp}\ \texttt{of}\ \mathit{MatchRules}\ \texttt{end} \ \Rightarrow \tau'}\ \text{E\_CASE}$$

5.5 MatchRules

Type checking a match rule requires checking the right-hand-side expression in the environment extended with the bindings from the left-hand-side pattern.

$$\frac{\mathrm{E}; \tau \vdash \ \mathit{Pat} \ \Rightarrow \mathrm{E}' \qquad \mathrm{E} \oplus \mathrm{E}' \vdash \ \mathit{Exp} \ \Rightarrow \tau'}{\mathrm{E}; \tau \vdash \ \mathit{Pat}\ \texttt{=>}\ \mathit{Exp} \ \Rightarrow \tau'}\ \text{MATCHRULE}$$

5.5.1 Multiple Match Rules

Type checking a list of match rules requires that every match rule has the same result type.

$$\frac{n \geq 0 \qquad \mathrm{E}; \tau \vdash \ \mathit{MatchRule}_1 \ \Rightarrow \tau \qquad \cdots \qquad \mathrm{E}; \tau \vdash \ \mathit{MatchRule}_n \ \Rightarrow \tau}{\mathrm{E}; \tau \vdash \ \mathit{MatchRule}_1\ \texttt{|}\ \cdots\ \texttt{|}\ \mathit{MatchRule}_n \ \Rightarrow \tau'}\ \mathit{MatchRules}$$

5.6 Declarations

The typing rules for declarations use a judgement of the form

$$\mathrm{E} \vdash \ \mathit{Decl} \ \Rightarrow \mathrm{E}'$$

which can be read as "in the environment E, the declaration *Decl* returns the environment E′." The output environment only includes bindings for the identifiers defined in the declaration; the output environment is not an extension of the input environment.

5.6.1 Type Declarations

Type checking a **`type`** declaration requires introducing new semantic type variables for the syntactic type variable identifiers when checking the right-hand-side type. The final environment is an environment with a binding for the (syntactic) type constructor identifier that maps to the (semantic) type variables (acting as formal type parameters) and the (semantic) type.

$$\frac{\begin{array}{c}\mathrm{E}_{tv} = \{\mathit{tyvarid}_i \mapsto \alpha_i \mid 1 \le i \le n \text{ and } \alpha_i \text{ fresh}\} \\ \mathrm{E} \oplus \mathrm{E}_{tv} \vdash \ \mathit{Type} \ \Rightarrow \tau \qquad \mathrm{E}' = \{\mathit{tyconid} \mapsto (\langle \alpha_1, \ldots, \alpha_n \rangle, \tau)\}\end{array}}{\mathrm{E} \vdash \ \texttt{type}\ \mathit{tyconid}\ \texttt{[}\ \mathit{tyvarid}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{tyvarid}_n\ \texttt{]}\ \texttt{=}\ \mathit{Type} \ \Rightarrow \mathrm{E}'}\ \text{D\_TYPE}$$

The typing rule for a **`datatype`** declaration is somewhat complicated. A new semantic type constructor (θ_i) is introduced for each syntactic type constructor identifier ($\mathit{tyconid}_i$). Furthermore, a new semantic type variable ($\alpha_{i,j}$) is introduced for each syntactic type variable identifier ($\mathit{tyvarid}_{i,j}$). Each of the data constructor declarations is type checked in an environment that includes *all* of the type constructors (E_{tc}) and *only* the appropriate type variables (E_{tv_i}). Note that type checking a data constructor declaration takes the type variables and the type constructor as part of the context; it returns an environment of bindings for the data constructor identifiers defined by the data constructor declaration. The final output environment includes all of the type constructors and all of the data constructors (but none of the type variables).

$$\frac{\begin{array}{c}n \ge 0 \qquad \mathrm{E}_{tc} = \{\mathit{tyconid}_i \mapsto \theta^{(m_i)} \mid 1 \le i \le n \text{ and } \theta_i \text{ fresh}\} \\ \mathrm{E}_{tv_1} = \{\mathit{tyvarid}_{i,1} \mapsto \alpha_{i,1} \mid 1 \le i \le m_1 \text{ and } \alpha_{i,1} \text{ fresh}\} \\ \mathrm{E} \oplus \mathrm{E}_{tc} \oplus \mathrm{E}_{tv_1}; (\langle \alpha_{1,1}, \ldots, \alpha_{1,m_1} \rangle, \theta^{(m_1)}) \vdash \ \mathit{DaConDecls}_1 \ \Rightarrow \mathrm{E}_{DC_1} \\ \cdots \qquad \mathrm{E}_{tv_n} = \{\mathit{tyvarid}_{i,n} \mapsto \alpha_{i,n} \mid 1 \le i \le m_n \text{ and } \alpha_{i,n} \text{ fresh}\} \\ \mathrm{E} \oplus \mathrm{E}_{tc} \oplus \mathrm{E}_{tv_n}; (\langle \alpha_{n,1}, \ldots, \alpha_{n,m_n} \rangle, \theta^{(m_n)}) \vdash \ \mathit{DaConDecls}_n \ \Rightarrow \mathrm{E}_{DC_n} \\ \mathrm{E}' = \mathrm{E}_{tc} \oplus \mathrm{E}_{DC_1} \oplus \cdots \oplus \mathrm{E}_{DC_n}\end{array}}{\begin{array}{l}\mathrm{E} \vdash \ \texttt{datatype}\ \mathit{tyconid}_1\ \texttt{[}\ \mathit{tyvarid}_{1,1}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{tyvarid}_{1,m_1}\ \texttt{]}\ \texttt{=}\ \mathit{DaConDecls}_1 \\ \qquad\qquad \cdots \\ \qquad\qquad \texttt{and}\ \mathit{tyconid}_n\ \texttt{[}\ \mathit{tyvarid}_{n,1}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{tyvarid}_{n,m_n}\ \texttt{]}\ \texttt{=}\ \mathit{DaConDecls}_n \ \Rightarrow \mathrm{E}'\end{array}}\ \text{D\_DATATYPE}$$

5.6.2 Data Constructor Declarations

Type checking a single data constructor declaration requires type checking each of its declared argument types. The result environment binds the data constructor identifier to its type constructor ($\theta^{(n)}$), the formal types of its arguments ($\langle \tau_1, \ldots, \tau_m \rangle$), and the formal type variables ($\langle \alpha_1, \ldots, \alpha_n \rangle$) over which the type constructor and the formal types of its arguments is abstracted. The type constructor and the formal type variables are provided by the context.

$$\frac{\begin{array}{c}\mathrm{E} \vdash \ \mathit{Type}_1 \ \Rightarrow \tau_1 \qquad \cdots \qquad \mathrm{E} \vdash \ \mathit{Type}_m \ \Rightarrow \tau_m \\ \mathrm{E}' = \{\mathit{daconid} \mapsto (\langle \alpha_1, \ldots, \alpha_n \rangle, \langle \tau_1, \ldots, \tau_m \rangle, \theta^{(n)})\}\end{array}}{\mathrm{E}; \langle \alpha_1, \ldots, \alpha_n \rangle; \theta^{(n)} \vdash \ \mathit{daconid}\ \texttt{\{}\ \mathit{Type}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Type}_m\ \texttt{\}} \ \Rightarrow \mathrm{E}'}\ \mathit{DaConDecl}$$

Type checking a list of data constructor declarations requires type checking each data constructor declaration. The final output environment is the extension of all the individual output environments.

$$\frac{\begin{array}{c} m \geq 0 \qquad \mathrm{E}; \langle \alpha_1, \ldots, \alpha_n \rangle; \theta^{(n)} \vdash \ \mathit{DaConDecl}_1 \ \Rightarrow \mathrm{E}_{DC_1} \\ \cdots \qquad \mathrm{E}; \langle \alpha_1, \ldots, \alpha_n \rangle; \theta^{(n)} \vdash \ \mathit{DaConDecl}_m \ \Rightarrow \mathrm{E}_{DC_m} \\ \mathrm{E}' = \mathrm{E}_{DC_1} \oplus \cdots \oplus \mathrm{E}_{DC_m} \end{array}}{\mathrm{E}; \langle \alpha_1, \ldots, \alpha_n \rangle; \theta^{(n)} \vdash \ \mathit{DaConDecl}_1 \ \texttt{|} \ \cdots \ \texttt{|} \ \mathit{DaConDecls}_m \ \Rightarrow \mathrm{E}'} \ \mathit{DaConDecls}$$

5.6.3 Value Declarations

Type checking a **`val`** declaration requires type checking the expression and type checking the pattern, which returns the bindings to be returned by the declaration. If there is a type assertion, then it must return the same (semantic) type as the expression.

$$\frac{\mathrm{E} \vdash \ \mathit{Exp} \ \Rightarrow \tau \qquad (\mathrm{E} \vdash \ \mathit{Type} \ \Rightarrow \tau)^? \qquad \mathrm{E}; \tau \vdash \ \mathit{SimplePat} \ \Rightarrow \mathrm{E}'}{\mathrm{E} \vdash \ \texttt{val}\ \mathit{SimplePat}\ (\texttt{:}\ \mathit{Type})^?\ \texttt{=}\ \mathit{Exp} \ \Rightarrow \mathrm{E}'} \ \text{D\_VAL}$$

Type checking a **`fun`** declaration requires constructing an environment that assigns a type to each of the function variable identifiers in the declaration. First, each of the parameter lists is type checked ($\mathrm{E} \vdash \ \mathit{Params}_i \ \Rightarrow \mathrm{E}_{P_i}$), returning an environment and a meta-function. The environment is used to type check the return type ($\mathrm{E} \oplus \mathrm{E}_{P_i} \vdash \ \mathit{Type}_i \ \Rightarrow \tau_i$) and the meta-function is used to construct the type of the function ($\mathcal{T}(\tau_i)$). Each of the function bodies is type checked in an environment extended with the appropriate parameters and the function variable identifiers ($\mathrm{E} \oplus \mathrm{E}' \oplus \mathrm{E}_{P_i} \vdash \ \mathit{Exp}_i \ \Rightarrow \tau_i$); note that each function body must have its declared return type.

$$\frac{\begin{array}{c} n \geq 0 \qquad \mathrm{E} \vdash \ \mathit{Params}_1 \ \Rightarrow \mathrm{E}_{P_1}; \mathcal{T}_1 \qquad \mathrm{E} \oplus \mathrm{E}_{P_1} \vdash \ \mathit{Type}_1 \ \Rightarrow \tau_1 \\ \cdots \qquad \mathrm{E} \vdash \ \mathit{Params}_n \ \Rightarrow \mathrm{E}_{P_n}; \mathcal{T}_n \qquad \mathrm{E} \oplus \mathrm{E}_{P_n} \vdash \ \mathit{Type}_n \ \Rightarrow \tau_n \\ \mathrm{E}' = \{\mathit{varid}_i \mapsto \mathcal{T}_i(\tau_i) \mid 1 \leq i \leq n\} \\ \mathrm{E} \oplus \mathrm{E}' \oplus \mathrm{E}_{P_1} \vdash \ \mathit{Exp}_1 \ \Rightarrow \tau_1 \qquad \cdots \qquad \mathrm{E} \oplus \mathrm{E}' \oplus \mathrm{E}_{P_n} \vdash \ \mathit{Exp}_n \ \Rightarrow \tau_n \end{array}}{\begin{array}{l} \texttt{fun}\ \mathit{varid}_1\ \mathit{Params}_1\ \texttt{:}\ \mathit{Type}_1\ \texttt{=}\ \mathit{Exp}_1 \\ \mathrm{E} \vdash \ \cdots \qquad \Rightarrow \mathrm{E}' \\ \texttt{and}\ \mathit{varid}_n\ \mathit{Params}_n\ \texttt{:}\ \mathit{Type}_n\ \texttt{=}\ \mathit{Exp}_n \end{array}} \ \text{D\_FUN}$$

5.6.4 Multiple Declarations

Type checking a list of declarations requires type checking each declaration in turn. Since identifiers in earlier declarations are bound in later declarations, each declaration is type checked in an environment extended with output environments of each earlier declaration. The final output environment is the extension of all the individual output environments.

$$\frac{\begin{array}{c} \mathrm{E} \vdash \ \mathit{Decl}_1 \ \Rightarrow \mathrm{E}_1 \qquad \cdots \qquad \mathrm{E} \oplus \mathrm{E}_1 \oplus \cdots \oplus \mathrm{E}_{n-1} \vdash \ \mathit{Decl}_n \ \Rightarrow \mathrm{E}_n \\ \mathrm{E}' = \mathrm{E}_1 \oplus \cdots \oplus \mathrm{E}_{n-1} \oplus \mathrm{E}_n \end{array}}{\mathrm{E} \vdash \ \mathit{Decl}_1 \ \cdots \ \mathit{Decl}_n \ \Rightarrow \mathrm{E}'} \ \mathit{Decls}$$

5.7 Programs and the Initial Environment

The typing rule for programs use a judgement of the form

$$\vdash \ \mathit{Prog} \ \Rightarrow \checkmark$$

which can be read as "the program *Prog* is statically correct."

Type checking a program is similar to type checking a **`let`** expression; it requires type checking the declarations and then checking the expression in an environment extended with the bindings of the declarations.

$$\frac{\mathrm{E}_0 \vdash \ \mathit{Decls} \ \Rightarrow \mathrm{E}' \qquad \mathrm{E}_0 \oplus \mathrm{E}' \vdash \ \mathit{Exp} \ \Rightarrow \tau}{\vdash \ \mathit{Decls} \ \texttt{;} \ \mathit{Exp} \ \Rightarrow \checkmark} \ \textsc{Prog}$$

The declarations and expression are type checked in the context of an *initial environment* E_0 that provides predefined type constructors, data constructors, and variables. This initial environment is defined as follows:

$$\mathrm{E}_0 = \langle \mathrm{TVE}_0, \mathrm{TCE}_0, \mathrm{DCE}_0, \mathrm{VE}_0 \rangle$$

$$\mathrm{TVE}_0 = \{\}$$

$$\mathrm{TCE}_0 = \left\{ \begin{array}{rcl} \texttt{Bool} & \mapsto & \mathbf{bool}^{(0)} \\ \texttt{Integer} & \mapsto & \mathbf{integer}^{(0)} \\ \texttt{String} & \mapsto & \mathbf{string}^{(0)} \\ \texttt{Unit} & \mapsto & \mathbf{unit}^{(0)} \end{array} \right\}$$

$$\mathrm{DCE}_0 = \left\{ \begin{array}{rcl} \texttt{False} & \mapsto & (\langle\rangle, \langle\rangle, \mathbf{bool}^{(0)}) \\ \texttt{True} & \mapsto & (\langle\rangle, \langle\rangle, \mathbf{bool}^{(0)}) \\ \texttt{Unit} & \mapsto & (\langle\rangle, \langle\rangle, \mathbf{unit}^{(0)}) \end{array} \right\}$$

$$\mathrm{VE}_0 = \left\{ \begin{array}{rcl} \texttt{argc} & \mapsto & \mathbf{unit}^{(0)} \rightarrow \mathbf{integer}^{(0)} \\ \texttt{arg} & \mapsto & \mathbf{integer}^{(0)} \rightarrow \mathbf{string}^{(0)} \\ \texttt{fail} & \mapsto & \forall\alpha \rightarrow \mathbf{string}^{(0)} \rightarrow \alpha \\ \texttt{print} & \mapsto & \mathbf{string}^{(0)} \rightarrow \mathbf{unit}^{(0)} \\ \texttt{size} & \mapsto & \mathbf{string}^{(0)} \rightarrow \mathbf{integer}^{(0)} \\ \texttt{sub} & \mapsto & \mathbf{string}^{(0)} \rightarrow \mathbf{integer}^{(0)} \rightarrow \mathbf{integer}^{(0)} \end{array} \right\}$$

6 Conversion to Abstract Syntax Tree

In addition to checking that a program is statically correct, a type checker must produce a program representation that can be used by the rest of the compiler for optimization and code generation. In the LangF compiler, the type checker will produce a typed abstract syntax tree (the `structure AbsSynTree : ABS_SYN_TREE` module in the project seed code). The typed abstract syntax tree (AST) representation is very close to the parse tree (PT) representation, but with some crucial differences:

- The AST forms include no source location information.
- Type variables, type constructors, data constructors, and variables in the AST representation are implemented by structures with the `ID` signature (`langfc-src/common/id.sig`). As discussed in Section 3, semantic type variables, type constructors, data constructors, and variables are used to distinguish different binding occurrences, which otherwise have the same syntactic type variable names, type constructor names, data constructor names, or variable names. There are operations in the `ID` signature that can be used to create fresh identifiers, not equal to any other identifier previously created.
- Variables bound (and wildcards) in simple patterns include their type.
- There is no type constraint expression form; instead, every expression form is annotated with its type.

- There is no **`type`** declaration form; all type abbreviations will have been expanded during type checking and conversion to the abstract syntax tree.
- The **`val`** declaration form has no type constraint; the simple pattern includes the type.

We have already introduced one form in the abstract syntax tree representation: the semantic types from Section 3. Similarly, we have already introduced one judgement for translating a parse tree representation form into an abstract syntax tree representation form: the judgement for type checking types from Section 5.1:

$\mathrm{E} \vdash$ *Type* $\Rightarrow \tau$ in the environment E, the parse tree type *Type* is well-formed and translates to the abstract syntax tree type τ.

We can imagine other judgements that combine type checking with translation to the abstract syntax tree:

$\mathrm{E} \vdash$ *Exp* $\Rightarrow \tau; e$ in the environment E, the parse tree expression *Exp* has the abstract syntax tree type τ and translates to the abstract syntax tree expression e.

$\mathrm{E} \vdash$ *Decl* $\Rightarrow \mathrm{E}'; d$ in the environment E, the parse tree declaration *Decl* returns the environment E' and translates to the abstract syntax tree declaration d.

$\vdash$ *Prog* $\Rightarrow p$ the parse tree program *Prog* is statically correct and translates to the abstract syntax tree program p.

The inference rules for these judgements will be very similar to those given in Section 5, except they will construct an appropriate output abstract syntax tree form.

7 Requirements

You should implement a type checker for LangF that enforces the type system from Section 5 and produces a typed abstract syntax tree. Your implementation should include (at least) the following modules:

```
structure Environment : ENVIRONMENT
structure TypeChecker : TYPE_CHECKER
```

The `ENVIRONMENT` signature in the project seed code is as follows:

```
signature ENVIRONMENT =
sig
   (* The combined environment is composed of a type variable
    * environment, a type constructor environment, a data constructor
    * environment, and a variable environment.  Each individual
    * environment has its own domain and co-domain.
    *
    * For a simple binding analysis, we can take the co-domain to be
    * 'unit', since we only need to know whether or not the identifier
    * is in the environment.
    *
    * For type checking (without producing an abstract syntax tree),
    * you will require co-domains similar to those in the project
    * description.
    *
    * For type checking and producing an abstract syntax tree,
    * you will require additional components in the co-domains.
    *)
   structure TyVarEnv :
      sig
         type dom = ParseTree.TyVarName.t
         type cod = unit
      end
   structure TyConEnv :
      sig
         type dom = ParseTree.TyConName.t
         type cod = unit
      end
   structure DaConEnv :
      sig
         type dom = ParseTree.DaConName.t
         type cod = unit
      end
   structure VarEnv :
      sig
         type dom = ParseTree.VarName.t
         type cod = unit
      end

   type t
```

```
    (* The empty environment {}. *)
    val empty : t

    (* Create an environment with a single TyVar entry. *)
    val singletonTyVar : TyVarEnv.dom * TyVarEnv.cod -> t
    (* Lookup a TyVar in the environment. *)
    val lookupTyVar : t * TyVarEnv.dom -> TyVarEnv.cod option

    (* Create an environment with a single TyCon entry. *)
    val singletonTyCon : TyConEnv.dom * TyConEnv.cod -> t
    (* Lookup a TyCon in the environment. *)
    val lookupTyCon : t * TyConEnv.dom -> TyConEnv.cod option

    (* Create an environment with a single DaCon entry. *)
    val singletonDaCon : DaConEnv.dom * DaConEnv.cod -> t
    (* Lookup a DaCon in the environment. *)
    val lookupDaCon : t * DaConEnv.dom -> DaConEnv.cod option

    (* Create an environment with a single Var entry. *)
    val singletonVar : VarEnv.dom * VarEnv.cod -> t
    (* Lookup a Var in the environment. *)
    val lookupVar : t * VarEnv.dom -> VarEnv.cod option

    (* Implements E1 (+) E2. *)
    val extend : t * t -> t

    (* Implements \Theta(E). *)
    val tycons : t -> AbsSynTree.TyCon.Set.set

    (* The initial environment E_0. *)
    val initial: t
end
```

You will need to extend the `ENVIRONMENT` signature (and the `Environment` structure) with new co-domain types as required by your type-checker implementation.

The `TYPE_CHECKER` signature is as follows:

```
signature TYPE_CHECKER =
sig
    val typeCheck : ErrorStream.t *
                    ParseTree.Prog.t ->
                    AbsSynTree.Prog.t option
end
```

The **structure** `ParseTree : PARSE_TREE` and **structure** `AbsSynTree : ABS_SYN_TREE` modules are provided in the seed code; the `PARSE_TREE` signature implementation is at `langfc-src/parse-tree/parse-tree.sig`; the `ParseTree` structure implementation is at `langfc-src/parse-tree/parse-tree.sml`; the `ABS_SYN_TREE` signature implementation is at `langfc-src/abs-syn-tree/abs-syn-tree.sig`; and the `AbsSynTree` structure implementation is at `langfc-src/abs-syn-tree/abs-syn-tree.sml`.

7.1 Errors

To support error reporting, the `TypeChecker.typeCheck` function takes an argument of the type `ErrorStream.t`. The `ErrorStream: ERROR_STREAM` module is provided in the seed code and provides a common error reporting utility in the LangF compiler. (The module was used implicitly in Projects 1 and 2.) The `ERROR_STREAM` signature implementation is at `langfc-src/common/error-stream.sig`; the `ErrorStream` structure implementation is at `langfc-src/common/error-stream.sml`.

Your type checker should report reasonable error messages. You should report violations of the syntactic restrictions of Section 2, unbound identifiers, and type errors. There is a lot of room for creativity and style in reporting errors. For a program with multiple type errors, you are required to report at least one error message, but need not report more than one error message.

8 GForge and Submission

Sources for Project 3 have been (or will shortly be) committed to your repository in the `project3` sub-directory. You will need to *update* your local copy, by running the command:

```
svn update
```

from the *`cnetid`*`-proj` directory.

We will collect projects from the SVN repositories at 10pm on Friday, February 27; make sure that you have committed your final version before then. To do so, run the command:

```
svn commit
```

from the *`cnetid`*`-proj` directory.

9 Hints

- Start early!
- Study the interfaces. You will need to be familiar with the types and operations in the `PARSE_TREE`, `ABS_SYN_TREE`, and `ID` signatures.
- Think about how to represent the environment(s). In the parse tree representation, each structure implementing a kind of identifier provides **`structure`** `Set: ORD_SET`, **`structure`** `Map: ORD_MAP`, and **`structure`** `Tbl: MONO_HASH_TABLE` modules for finite maps, sets, and hash tables over that kind of identifier; you can view the `ORD_SET`, `ORD_MAP`, and `MONO_HASH_TABLE` signatures via links in the HTML version of this document.
- Think about the structure of the type-checker implementation. Each judgement can be implemented as a function; just as the parse tree datatypes for expressions, match rules, and declarations are mutually recursive, the functions for judgements that type check expressions, match rules, and declarations will be mutually recursive. Each function for a judgement will have a case for each typing rule with that judgement as the conclusion.
- Work in stages. First implement a simple binding checker that only checks that the program has no unbound variables (but does not check types and does not produce an abstract syntax tree). This binding checker will establish the basic structure of the implementation. Next, implement a type checker that checks for unbound variables and checks types (but does not produce an abstract syntax tree). This type checker will require extending the simple binding checker, but will very closely match the typing rules from Section 5. Next, implement a full type checker that checks for unbound variables, checks types, and produces an abstract

syntax tree. This full type checker will require additional information to be carried in the environment and to be returned by each type checking function. Finally, extend the full type checker to additionally check the syntactic restrictions of Section 2.

- Work on error reporting last. Detecting errors and producing good error messages can be difficult; it is more important for your type checker to work on good programs than for it to "work" on bad programs. Again, work in stages. First implement a type checker that stops after detecting the first error. Then implement a type checker that continues after detecting an error.

- To complete the assignment, you should only need to make changes to the *cnetid*`-proj/project3/langfc-src/type-checker/environment.sig`, *cnetid*`-proj/project3/langfc-src/type-checker/environment.sml`, and *cnetid*`-proj/project3/langfc-src/type-checker/type-checker.sml` files.

- Executing the compiler (from the *cnetid*`-proj/project3` directory) with the command

  `./bin/langfc -Ckeep-type-check=true` *file*`.lgf`

  will produce a *file*`.type-check.ast` file that contains the abstract syntax tree returned by the type checker. Use this control and its output to check that your type checker is working as expected. The `tests/type-checker` directory includes a number of tests (of increasing complexity); for each `test`*NN*`.lgf` file, if the test has type errors, there is a `test`*NN*`.err` file containing sample error messages to be reported by the type checker. If the test has no type errors, then there is no output file; rather, the abstract parse tree returned by the type checker should convert to the core intermediate representation and type check in the core intermediate representation without errors.

- As in past projects, you are not required to match the sample error messages exactly. In particular, you need not have particularly "pretty" error messages. The sample error messages (and sample solution) have gone to some length to produce good error messages, with types written using the identifiers in the parse tree representation. You will probably find it much easier to print types using the identifiers in the abstract syntax tree representation (using the functions `Layout.toString` and `AST.Type.layout`), which will print indentifiers with a uniqueifying suffix. For example, in `test64.lgf`, the sample error message is:

```
test46.lgf:5.0-test46.lgf:6.0 Error:
Constraint and expression of 'val' disagree.
  constraint: T
  expression: ?T?
  in: val y : T = B
```

  where the constraint type is printed using the tycon identifier `T` from the parse tree representation, and the expression type is also printed using the tycon identifier `T` from the parse tree representation, but printed as `?T?` to indicate that this tycon is shadowed. It will be much easier to produce an error message like:

```
test46.lgf:5.0-test46.lgf:6.0 Error:
Constraint and expression of 'val' disagree.
  constraint: T__019
  expression: T__015
  in: val y : T = B
```

  where the constraint type and the expression type are printed using the tycon identifiers from the abstract syntax tree representation (with their uniqueifying suffixes). The fact that one tycon is shadowed by the other (in the parse tree representation) is indicated by the different type constructor indentifiers in the abstract syntax tree representation.

Document history

February 24, 2009

Section 5.4: $\Theta(\langle \mathrm{TVE}, \mathrm{TCE}, \mathrm{DCE}, \mathrm{VE} \rangle) = cod(\mathrm{TCE}) \Rightarrow$
$\Theta(\langle \mathrm{TVE}, \mathrm{TCE}, \mathrm{DCE}, \mathrm{VE} \rangle) = \{\theta^{(k)} \mid \textit{tyconid} \in dom(\mathrm{TCE}) \text{ and } \mathrm{TCE}(\textit{tyconid}) = \theta^{(k)}\}$

February 23, 2009

Section 9: Added discussion of unit tests and error messages.

February 16, 2009

Section 3: the **`datatype`** declaration at line 1 will introduce a different type constructor $\Rightarrow$
the **`datatype`** declaration at line 3 will introduce a different type constructor

February 13, 2009

- Made redundancy and exhaustiveness checking extra credit.
- Fixed typing rule for **`let`** to handle sequenced expressions in body.
- For a program with multiple type errors, require at least one error message, but not necessarily more than one.

February 12, 2009 Original version