The LangF Type System

1 Introduction

This document presents the formal specification of the typing rules for LangF. It is a companion to the Project 2 description, which describes the project of implementing this formal specification in a type checker.1 The type system for LangF is essentially an enrichment of the System F type system.

2 LangF Abstract Syntax

The LangF type system is defined in terms of an abstract syntax, which elides many of the syntactic details found in the concrete syntax. There is a rough correspondence between the compiler’s parse-tree data structures and the abstract syntax, but the abstract syntax omits various derived forms (see Section 8) that can be defined in terms of more primitive syntax.

We start by defining conventions for the various kinds of identifiers in LangF.

\[
\begin{align*}
\text{type-variable identifiers} & : t \in \text{TVID} \\
\text{type-constructor identifiers} & : T \in \text{TYPID} \\
\text{data-constructor identifiers} & : C \in \text{CONID} \\
\text{variable identifiers} & : x, f \in \text{VARID} \\
\text{integer constants} & : n \in \text{INT} \\
\text{string constants} & : s \in \text{STR}
\end{align*}
\]

The abstract syntax of LangF is given in Figure 1. In various places, we use the notation of putting a bar above a meta identifier to represent a sequence of one or more items. For example, “\(\overline{T}\)” represents a sequence of type variables “\(t_1, \ldots, t_n\)” with \(n \geq 1\).

We assume that the abstract syntax satisfies certain syntactic restrictions, which are described in the next section.

---

1Remember, 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 implementation from a specification.
prog ::= def prog  
   | def                     top-level definition
  
  def ::= type T [typ] = typ type-alias definition  
   | data T [typ] = con data-type definition  
   | bind value-binding definition
  
  typ ::= [typ] function  
  
  type  
  
  data  
  
  bind  
  
  con ::= C of typ data constructor  
   | C nullary data constructor
  
  bind ::= fun f fnsig = exp function binding  
   | let spat : typ = exp value binding with constraint  
   | let spat = exp value binding
  
  fnsig ::= [typ] function type parameters  
  
  exp ::= if exp1 then exp2 else exp3 conditional expression  
  
  exp1 ::= exp2 assignment expression  
  
  exp1 :: exp2 list-cons expression  
  
  ! exp dereference expression  
  
  exp1 exp2 application expression  
  
  exp[typ] type-application expression  
  
  (exp1, ..., expn) tuple expression (n > 1)  
  
  { scope } nested scope  
  
  case exp of rule case expression  
  
  x variable  
  
  C data constructor  
  
  lit literal
  
  rule ::= { pat ⇒ scope } match-case rule
  
  pat ::= C spat data-constructor pattern  
  
  spat1 :: spat2 list-cons expression  
  
  (spat1, ..., spatn) tuple pattern (n > 1)  
  
  C nullary-data-constructor pattern  
  
  spat simple pattern
  
  spat ::= x variable pattern  
   | - wild-card pattern
  
  scope ::= bind scope value binding  
   | exp expression

Figure 1: LangF abstract syntax
3 Syntactic Restrictions

There are a number of syntactic restrictions that should be enforced by the type checker. Some of these are properties that could have been specified as part of the grammar in Project 2, but would have made the grammar much more verbose. Others are properties that could be specified as part the typing rules below, but it is easier to specify them separately.

- The type variables in a type or data definition must be distinct.
- The type variables in a type abstraction must be distinct.
- The data constructors in a data-type definition must be distinct.
- The value parameter names of a function definition must be distinct, and the name of the function must not be the same as any of the value parameters.
- The type parameter names of a function definition must be distinct.
- A function definition must have at least one value parameter.
- The variables in a pattern must be distinct.
- The patterns in a `case` expression must be exhaustive and useful. (A pattern $p$ is useful if exists a value that the previous patterns do not match, but $p$ does match). 
- Integers in LangF are represented as 63-bit 2’s-complement numbers, which means that they are in the range $-2^{62}..2^{62} - 1$. Since the scanner only scans positive numbers, we require that integer literals be in the range $0..2^{62} - 1$.
- The last definition in a program must be a function with the name `main` that has the type `List[String] -> Int`.

Your type checker is responsible for checking these properties and reporting an error when they are violated.

4 Semantic Types

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

```plaintext
1 data T = A of Int | B;
2 let x : T = A 1;
3 data T = C of Int | D;
4 let y : T = B;
5 0
```

This program has a type error at line 4 in the declaration `let y : T = B`, because the type of the data constructor expression `B` is the type constructor corresponding to the `data` declaration at line 1, but the type constraint `T` is the type constructor corresponding to the `data` declaration at
\(\alpha, \beta \in \text{TYVAR}\) type variables

\(\theta^{(k)} \in \text{TYCON}\) \(k\)-ary type constructors

\[\tau \ ::= \ \forall \overline{\alpha}(\tau) \quad \text{type abstraction (}|\overline{\alpha}| > 0)\]

\[\tau_1 \rightarrow \tau_2 \quad \text{function type}\]

\[\tau_1 \times \cdots \times \tau_n \quad \text{tuple types (}n > 1\text{)}\]

\[\theta^{(k)}[\overline{\tau}] \quad \text{type constructor instantiation (}k = |\overline{\tau}|\text{)}\]

\(\alpha \) type variable

Figure 2: \textbf{LangF} semantic types

The abstract syntax of \textbf{LangF} semantic types is given in Figure 2 (and represented by the \texttt{Type.ty} datatype in the project seed code). The set of semantic types (\texttt{TYPE}) is built from countable sets of semantic type variables (\texttt{TYVAR}) and semantic type constructors (\texttt{TYCON}). We use \(\tau\) to denote types, \(\alpha\) and \(\beta\) to denote semantic type variables, and \(\theta^{(k)}\) to denote \(k\)-ary type constructors. In the representation, we treat type constants as nullary type constructors, but we will often omit the empty type-argument list in this document (e.g., we write \texttt{Bool(0)} instead of \texttt{Bool(0)[ ]}).

Each binding occurrence of a type variable (respectively, type constructor) will map to a unique semantic type variable (respectively, semantic type constructor) in the AST representation of the program. For example, type checking the \texttt{data} declaration at line 1 will introduce one type constructor, say \(\theta^{(0)}_1\), and type checking the \texttt{data} declaration at line 5 will introduce a different type constructor, say \(\theta^{(0)}_2\). The syntax of semantic types mirrors the concrete syntax, with forms for type abstraction, function types, tuple types, instantiation of type constructors, and type variables.

We use the syntax \(\overline{\alpha}\) to denote a sequence of bound type variables in the term \(\forall \overline{\alpha}(\tau)\) and \(\overline{\tau}\) to denote a (possibly empty) sequence of types in the term \(\theta^{(k)}[\overline{\tau}]\). In the case that \(\overline{\tau}\) is the empty sequence, then \(\forall \overline{\alpha}(\tau) = \tau\). We write \(|\overline{\alpha}|\) to denote the number of elements in the sequence. The capture-free substitution of types \(\tau\) for variables \(\overline{\alpha}\) in a type \(\tau'\) is written as \(\tau'[\overline{\alpha}/\tau]\).

We consider semantic types equal up to renaming of bound type variables.\footnote{This renaming is called \(\alpha\)-conversion.} That is, we will consider the semantic types \(\forall \alpha(\alpha \rightarrow \alpha \rightarrow \texttt{Bool(0)})\) and \(\forall \beta(\beta \rightarrow \beta \rightarrow \texttt{Bool(0)})\) to be equal, whereas the parse trees corresponding to \([a] \ a \rightarrow a \rightarrow \texttt{bool} \) and \([b] \ b \rightarrow b \rightarrow \texttt{bool}\) are not equal, because they use different type variable names.

5 Environments

The typing rules for \textbf{LangF} use a number of different \textit{environments}, which are finite maps from identifiers to information about the identifiers. We write \(\{x \mapsto w\}\) for the finite map that maps \(x\) to \(w\) and we write \(\{\overline{x} \mapsto \overline{w}\}\) for the map that maps elements of the sequence \(\overline{x}\) to the corresponding element of the sequence \(\overline{w}\) (assuming that \(|\overline{x}| = |\overline{w}|\)). If \(E\) and \(E'\) are environments, then we define
the extension of $E$ to be

$$(E \pm E')(x) = \begin{cases} E'(x) & \text{if } x \in \text{dom}(E') \\ E(x) & \text{otherwise} \end{cases}$$

and we write $E \uplus E'$ for the disjoint union of $E$ and $E'$ when $\text{dom}(E) \cap \text{dom}(E') = \emptyset$ (if the domains of $E$ and $E'$ are not disjoint, then $E \uplus E'$ is undefined).

There is a separate environment for each kind of identifier in the parse-tree representation:

- $\text{TyVarEnv} = \text{TyVar} \rightarrow \text{TyVar}$ type-variable environment
- $\text{TyConEnv} = \text{TyId} \rightarrow \text{TyCon} \cup (\text{TyVar}^* \times \text{Type})$ type-constructor environment
- $\text{DataConEnv} = \text{ConId} \rightarrow \text{Type}$ data-constructor environment
- $\text{VarEnv} = \text{ValId} \rightarrow \text{Type}$ variable environment

A type name $T$ can either be bound to a type expression (in a type definition), to a data-type constructor (in a data definition), or to a primitive type constructor. We use the notation $\Lambda \alpha : \tau$ to represent a parameterized type definition in the type-constructor environment, and $\theta^{(k)}$ to denote data-type and primitive type constructors.

Since most of the typing rules involve two or more environments, we define a combined environment.

$$E \in \text{Env} = \text{TyVarEnv} \times \text{TyConEnv} \times \text{DataConEnv} \times \text{VarEnv}$$

We extend the notation on finite maps to the combined environment in the natural way:

$$\langle TVE, TCE, DCE, VE \rangle \pm \langle TVE', TCE', DCE', VE' \rangle = \langle TVE \pm TVE', TCE \pm TCE', DCE \pm DCE', VE \pm VE' \rangle$$

$$\langle TVE, TCE, DCE, VE \rangle \uplus \langle TVE', TCE', DCE', VE' \rangle = \langle TVE \uplus TVE', TCE \uplus TCE', DCE \uplus DCE', VE \uplus VE' \rangle$$

We also use the kind of identifier in the domain as a shorthand for extending an environment with a new binding. For example, by convention $x \in \text{ValId}$, so we will write $E \pm \{x \mapsto \tau\}$ for

$$\langle TVE, TCE, DCE, VE \rangle \pm \{x \mapsto \tau\}$$

where $E = \langle TVE, TCE, DCE, VE \rangle$.

6 Typing Rules

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

$$\text{Context} \vdash \text{Term} \triangleright \text{Descr}$$

which can be read as “in $\text{Context}$, $\text{Term}$ has $\text{Descr}$.” 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 different judgement forms used in the typing rules for $\text{LangF}$ are summarized in Figure 3. Formally, the judgments are smallest relation that satisfies the typing rules.
6.1 Programs

For a program, we check that it is well-formed (i.e., that the types, expressions, and definitions all type check).

For a top-level definition, we check the definition and then check the rest of the program using the enriched environment.

\[
E \vdash \text{def} \triangleright E' \\
E' \vdash \text{pro} \triangleright \text{Ok}
\]

\[
E \vdash \text{def prog} \triangleright \text{Ok}
\]

The body of the program is well-formed if it type checks.

\[
E \vdash \text{exp} \triangleright \tau \\
E \vdash \text{exp} \triangleright \text{Ok}
\]

6.2 Definitions

Type definitions are checked by binding the syntactic type parameters (\( \bar{t} \)) to fresh semantic type variables (\( \bar{\alpha} \)) and then checking the right-hand-side type expression.

\[
\bar{\alpha} \text{ are fresh} \quad |\bar{t}| = |\bar{\alpha}| = k \\
E \pm \{ t \mapsto \bar{\alpha} \} \vdash \text{typ} \triangleright \tau \\
E' = E \pm \{ T \mapsto \Lambda \bar{\alpha} : \tau \}
\]

\[
E \vdash \text{type } T[\bar{t}] = \text{typ} \triangleright E'
\]

Note that the type-variable component of \( E \) will be empty at top level.

Checking a data-type definition requires checking the data-constructor definitions in an environment that has been extended with bindings for both the data-type constructor and type variables.

\[
\bar{\alpha} \text{ are fresh} \quad |\bar{t}| = |\bar{\alpha}| = k \quad \theta^{(k)} \text{ is fresh} \\
E' = E \pm \{ T \mapsto \theta^{(k)} \}
\]

\[
E' \pm \{ t \mapsto \bar{\alpha} \}, \bar{\alpha}, \theta^{(k)} \vdash \text{con}_i \triangleright E_i \text{ for } 1 \leq i \leq n
\]

\[
E \vdash \text{data } T[\bar{t}] = \text{con}_1 \cdots \text{con}_n \triangleright E' \pm (E_1 \uplus \cdots \uplus E_n)
\]
The value binding definition is checked just like a value binding (see Section 6.5).

\[
E \vdash \text{bind} \triangleright E'
\]

\[
E \vdash \text{bind} \triangleright E'
\]

### 6.3 Types

The typing rules for types check for well-formedness and translate the syntactic types to semantic types.

Type checking a type-function type requires introducing fresh semantic type variables \(\bar{\alpha}\) for the syntactic type variables \(\bar{t}\).

\[
\overline{\alpha \text{ are fresh}} \quad E \vdash \{ \bar{t} \mapsto \overline{\bar{\alpha}} \} \vdash \text{typ} \triangleright \tau
\]

\[
E \vdash [\bar{t}]\text{typ} \triangleright \forall \overline{\bar{\alpha}}(\tau)
\]

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

\[
E \vdash \text{typ}_1 \triangleright \tau_1 \quad E \vdash \text{typ}_2 \triangleright \tau_2
\]

\[
E \vdash \text{typ}_1 \rightarrow \text{typ}_2 \triangleright \tau_1 \rightarrow \tau_2
\]

Type checking a tuple type requires checking the component types to form a (semantic) tuple type.

\[
E \vdash \text{typ}_1 \triangleright \tau_1 \quad \cdots \quad E \vdash \text{typ}_n \triangleright \tau_n
\]

\[
E \vdash \text{typ}_1 \star \cdots \star \text{typ}_n \triangleright \tau_1 \times \cdots \times \tau_n
\]

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 data definition (or a built-in abstract type). For type definitions, we check the type arguments and then construct a new (semantic) type by substituting the \(\overline{\tau}\) for the \(\overline{\bar{\alpha}}\).

\[
E(T) = \Lambda \overline{\bar{\alpha}} : \tau_T \quad |\overline{\bar{\alpha}}| = |\text{typ}| \quad E \vdash \text{typ} \triangleright \overline{\tau}
\]

\[
E \vdash T[\text{typ}] \triangleright \tau_T[\overline{\alpha}/\overline{\tau}]
\]

For data definitions (or abstract types), 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.

\[
E(T) = \theta^{(k)} \quad |\text{typ}| = k \quad E \vdash \text{typ} \triangleright \overline{\tau}
\]

\[
E \vdash T[\text{typ}] \triangleright \theta^{(k)}[\overline{\tau}]
\]

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

\[
t \in \text{dom}(E)
\]

\[
E \vdash t \triangleright E(t)
\]
6.4 Data constructors

Checking a data constructor involves checking that its argument type is well-formed.

\[
E \vdash \text{typ} \triangleright \tau
\]

\[
E, \bar{\alpha}, \theta^{(k)} \vdash C \text{ of typ} \triangleright \{ C \mapsto \forall \bar{\alpha} (\tau \to \theta^{(k)}[\bar{\alpha}]) \}\]

Checking nullary data constructors requires no additional checks.

\[
E, \bar{\alpha}, \theta^{(k)} \vdash C \triangleright \{ C \mapsto \forall \bar{\alpha} (\theta^{(k)}[\bar{\alpha}]) \}
\]

6.5 Value Bindings

Type checking a function binding involves first checking its signature, which produces an enriched environment, the function’s type (signature), and the function’s return type. Then we check the return type against the function’s body using the enriched environment extended with the type of \( f \) (we need the type of \( f \) to support recursive functions).

\[
E \vdash \text{fnsig} \triangleright E', \tau, \tau_{\text{ret}} \quad E' \pm \{ f \mapsto \tau \} \vdash \text{exp} \triangleright \tau_{\text{ret}}
\]

\[
E \vdash \text{fun } f \text{ fnsig } = \text{exp} \triangleright E \pm \{ f \mapsto \tau \}
\]

where Return\(\text{Type}(\tau)\) is the return type of the function.

Type checking a variable binding with a type constraint requires checking that the declared type is well formed and that the right-hand-side expression has that type.

\[
E \vdash \text{typ} \triangleright \tau \quad E \vdash \text{exp} \triangleright \tau \quad \tau \vdash \text{spat} \triangleright E'
\]

\[
E \vdash \text{let } \text{spat : typ } = \text{exp} \triangleright E \pm E'
\]

Type checking an unconstrained variable binding requires checking the right-hand-side expression and then using the resulting type as the context for checking the simple pattern.

\[
E \vdash \text{exp} \triangleright \tau \quad \tau \vdash \text{spat} \triangleright E'
\]

\[
E \vdash \text{let } \text{spat } = \text{exp} \triangleright E \pm E'
\]

A value binding that is just an expression \( \text{exp} \) is viewed as syntactic sugar for the binding

\[
\text{let } \_ : \text{Unit } = \text{exp}
\]

which is reflected in its typing rule.

\[
E \vdash \text{exp} \triangleright \text{Unit}^{(0)}
\]

\[
E \vdash \text{exp} \triangleright E
\]
6.6 Function Signatures

Type checking a function signature produces an environment enriched by the function parameter bindings, the type of the function, and the function’s return type.

For type parameters, we bind the names to fresh type variables and define the function’s type to be a type function.

\[ \overline{\alpha} \text{ are fresh} \quad E \vdash \{ t \mapsto \overline{\alpha} \} \vdash fnsig \triangleright E', \tau, \tau_{\text{ret}} \]

\[ E \vdash [t] fnsig \triangleright E', \forall \alpha(\tau), \tau_{\text{ret}} \]

For a value parameter, we check the declared type, bind the name to the type, and define the function’s type to be a function.

\[ E \vdash typ \triangleright \tau \quad E \vdash \{ x \mapsto \tau \} \vdash fnsig \triangleright E', \tau', \tau_{\text{ret}} \]

\[ E \vdash (x : typ) fnsig \triangleright E', \tau \rightarrow \tau', \tau_{\text{ret}} \]

For the return type of the signature, we check that the type is well formed and return it as both the function’s type and the return type.

\[ E \vdash typ \triangleright \tau \quad E \vdash \rightarrow typ \triangleright E, \tau, \tau \]

6.7 Expressions

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.

\[ E \vdash exp_1 \triangleright \text{Bool}^{(0)} \quad E \vdash exp_2 \triangleright \tau \quad E \vdash exp_3 \triangleright \tau \]

\[ E \vdash \text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \triangleright \tau \]

Note that the conditional operators are type-checked as according to their translation into if expressions.

For assignment, we check that the left-hand-side expression has a reference type and that the right-hand-side expression’s type matches the content type of the reference.

\[ E \vdash exp_1 \triangleright \text{Ref}^{(1)}[\tau] \quad E \vdash exp_2 \triangleright \tau \]

\[ E \vdash exp_1 := exp_2 \triangleright \text{Unit}^{(0)} \]

For list construction, we check that the right-hand-side expression has a list type and that the left-hand-side expression’s type matches the element type of the list.

\[ E \vdash exp_1 \triangleright \tau \quad E \vdash exp_2 \triangleright \text{List}^{(1)}[\tau] \]

\[ E \vdash exp_1 :: exp_2 \triangleright \text{List}^{(1)}[\tau] \]

The other infix binary operations are checked as applications to a pair of arguments (see Section 8)
Like assignment, application of the dereferencing operator is a special case to avoid the need for an explicit type application.

\[
E \vdash \text{exp} \triangleright \text{Ref}^{(1)}[\tau] \\
E \vdash ! \text{exp} \triangleright \tau
\]

Application of a function requires checking both expressions and checking that the function expression has a function type whose domain is the same as the type of the argument expression.

\[
E \vdash \text{exp}_1 \triangleright \tau' \rightarrow \tau \quad E \vdash \text{exp}_2 \triangleright \tau' \\
E \vdash \text{exp}_1 \text{exp}_2 \triangleright \tau
\]

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

\[
E \vdash \text{exp} \triangleright \forall \alpha(\tau) \\
E \vdash \text{typ} \triangleright \bar{\tau}' \\
|\alpha| = |\text{typ}| \\
E \vdash \text{exp}[\text{typ}] \triangleright \tau[\alpha/\bar{\tau}']
\]

Checking a tuple expression involves checking each of the subexpressions.

\[
E \vdash \text{exp}_i \triangleright \tau_i \text{ for } 1 \leq i \leq n \\
E \vdash (\text{exp}_1, \ldots, \text{exp}_n) \triangleright \tau_1 \times \cdots \times \tau_n
\]

The rules for type checking the body of a nested scope are given below in Section 6.11. The resulting type is the type of the expression.

\[
E \vdash \text{scope} \triangleright \tau \\
E \vdash \{ \text{scope} \} \triangleright \tau
\]

Typechecking a case expression involves first checking the argument of the case and then using the argument type to check the rules, which must all have the same result type.

\[
E \vdash \text{exp} \triangleright \tau \\
E, \tau \vdash \text{rule}_i \triangleright \tau_i' \text{ for } 1 \leq i \leq n \\
E \vdash \text{case exp of rule}_1 \cdots \text{rule}_n \triangleright \tau'
\]

Variables are mapped to their type in the environment.

\[
x \in \text{dom}(E) \\
E \vdash x \triangleright E(x)
\]

Like variables, data constructors are mapped to their type in the environment.

\[
C \in \text{dom}(E) \\
E \vdash C \triangleright E(C)
\]

Literal expressions have the type appropriate type.

\[
E \vdash n \triangleright \text{Integer}^{(0)} \\
E \vdash s \triangleright \text{String}^{(0)}
\]
6.8 Case Rules

Match-case rules combine pattern matching with a scope. To check them, we first check the pattern against the match-case argument type and then use the resulting environment to check the scope.

\[
E, \tau \vdash \text{pat} \Rightarrow E' \quad E, \tau' \vdash \text{scope} \Rightarrow \tau' \\
E, \tau \vdash \{ \text{pat} \Rightarrow \text{scope} \} \Rightarrow \tau'
\]

6.9 Patterns

Type checking patterns is done in a context that includes both the environment and the expected type (or argument type) of the pattern. The result is an environment that binds the pattern’s variables to their types.

Checking the application of a data constructor to a simple pattern involves checking that the expected type is a type-constructor application and that the type of the data constructor is a (possibly polymorphic) function type with a range that matches the type constructor. We check the argument pattern with an expected type that is the domain of the function type instantiated to the expected argument types. The result environment is the environment defined by the argument pattern.

\[
\tau = \theta(k)[\tau'] \\
E(C) = \forall \alpha (\tau'' \rightarrow \theta(k)[\alpha]) \\
\tau''[\alpha/\tau'] \vdash \text{spat} \Rightarrow E'
\]

The list-constructor pattern requires that the expected type be a list type. We check that the left-hand-side pattern is the element type and the right-hand-side pattern is the list type. The result environment is the disjoint union of the environments defined by the argument patterns.

\[
\tau = \text{List}(1)[\tau'] \\
\tau' \vdash \text{spat}_1 \Rightarrow E_1 \\
\tau \vdash \text{spat}_2 \Rightarrow E_2 \\
E, \tau \vdash \text{spat}_1 :: \text{spat}_2 \Rightarrow E_1 \pm E_2
\]

A tuple pattern requires that the expected type be a tuple of the same arity. We check each subpattern in the context of the corresponding type and then union the resulting environments.

\[
\tau = \tau_1 \times \cdots \times \tau_n \\
\tau_i \vdash \text{spat}_i \Rightarrow E_i \text{ for } 1 \leq i \leq n \\
E' = E_1 \uplus \cdots \uplus E_n \\
E, \tau \vdash \langle \text{spat}_1, \ldots, \text{spat}_n \rangle \Rightarrow E'
\]

Checking a nullary constructor requires matching the constructor’s type (which may be polymorphic) against the argument type given by the context.

\[
\tau = \theta(k)[\tau'] \\
E(C) = \forall \alpha (\theta(k)[\alpha]) \\
E, \tau \vdash C \Rightarrow \emptyset
\]

Simple patterns are checked in a context of the expected type and the resulting environment is the result of checking the pattern.

\[
\tau \vdash \text{spat} \Rightarrow E' \\
E, \tau \vdash \text{spat} \Rightarrow E'
\]
6.10 Simple patterns

Simple patterns are checked in the context of their expected type, which is used to define the resulting environment.

Type checking simple patterns yields an environment that binds the pattern’s variable to the context type.

\[ \tau \vdash x \mapsto \{ x \mapsto \tau \} \]

For wild-card patterns, the resulting environment is empty.

\[ \tau \vdash \_ \mapsto \emptyset \]

6.11 Scopes

Checking a binding extends the environment, which is then used to check the rest of the scope’s body.

\[
\begin{align*}
E \vdash \text{bind} & \quad E' \quad E' \vdash \text{scope} \quad \tau \\
E \vdash \text{bind scope} & \quad \tau
\end{align*}
\]

The type of a scope’s body is the type of the last expression.

\[
\begin{align*}
E \vdash \text{exp} & \quad \tau \\
E \vdash \text{exp} & \quad \tau
\end{align*}
\]

7 LangF Basis

LangF defines a Basis Environment that provides a collection of predefined types, type constructors, data constructors, infix operators, and functions.

7.1 Basis Types

The LangF Basis Environment defines six built-in types. These types are predefined because they require special support from the compiler. The types are

- The abstract type **Int** describes 63-bit 2’s-complement integers. The compiler provides special syntax for integer literals in expressions.
- The abstract type **String** describes sequences of 8-bit characters. The compiler provides special syntax for string literals in expressions.
- The abstract type constructor **Ref[t]** describes mutable reference cells containing values of type t. The compiler supports special typing rules for the assignment and dereferencing operators that infer the type argument.
• The Unit datatype has a single constructor (also called Unit). The compiler supports the special expression syntax “()” for Unit values. The Unit type is also used in certain typing rules.

• The Bool datatype has two constructors: True and False. The Bool type is used in certain typing rules.

• The List[t] type constructor defines a datatype with two data constructors: the nullary Nil and the infix constructor “::” (pronounced “cons”). The typing rule for “::” in expressions infers the type argument.

7.2 Binary Operators

The infix binary operators (other than the conditional operators and assignment) are defined to have the following types:

```plaintext
== : Int * Int -> Bool  integer equality test
!= : Int * Int -> Bool  integer inequality test
< : Int * Int -> Bool   integer less-than test
<= : Int * Int -> Bool  integer less-than-or-equal test
^ : String * String -> String string concatenation
+ : Int * Int -> Int    integer addition
- : Int * Int -> Int    integer subtraction
* : Int * Int -> Int    integer multiplication
/ : Int * Int -> Int    integer division
% : Int * Int -> Int    integer remainder
```

7.3 Unary Operators

The prefix negation operator is defined to have the following type:

```plaintext
- : Int -> Int    integer negation
```

7.4 Predefined Functions

The LangF Basis Environment also includes a small number of built-in functions:

```plaintext
chr : Int -> String  convert an ASCII character code to a string
fail : [t] String -> t terminate with an error message
print : String -> Unit print a string
newRef : [t] t -> Ref[t] create a mutable reference
size : String -> Int string length
sub : String * Int -> Int extract a character from a string
```
8 Derived Forms

The abstract syntax given in Figure 1 omits a number of syntactic forms that are found in the concrete syntax. These forms are viewed as *syntactic sugar* and are defined by translation into the abstract syntax.

- The expression form of top-level value bindings is replaced by a anonymous let binding:

  \[ \text{exp} \implies \text{let } _ : \text{Unit} = \text{exp} \]

  Note that the type constraint enforces the requirement that top-level expressions have *Unit* type.

- The conditional operators are replaced by *if* expressions:

  \[
  \begin{align*}
  \text{exp}_1 \mid \text{exp}_2 & \implies \text{if } \text{exp}_1 \text{ then True else } \text{exp}_2 \\
  \text{exp}_1 \&\& \text{exp}_2 & \implies \text{if } \text{exp}_1 \text{ then } \text{exp}_2 \text{ else False}
  \end{align*}
  \]

- Infix binary operators (other than the conditional operators and assignment) are represented as applications of the operator to a pair of arguments.

- The prefix negation operators is represented as an application.

- The expression form () is replaced by the *Unit* constructor.

With these syntactic derivations, one can derive the typing rules for the syntactic sugar. For example, the typing rule for a binary operator is derived from the rules for application and tuples.

\[
E(\odot) = \tau_1 \times \tau_2 \rightarrow \tau_3 \quad E \vdash \text{exp}_1 \triangleright \tau_1 \quad E \vdash \text{exp}_2 \triangleright \tau_2
\]

\[
E \vdash \text{exp}_1 \odot \text{exp}_2 \triangleright \tau_3
\]

9 Document History

**November 4, 2020**  Fix a few minor typos

**November 3, 2020**  Fix the rule for the dereference operator. The result type is \(\tau\), not Unit\(^{(0)}\).

**October 22, 2020**  Fix a couple of typos.

**October 21, 2020**  Original version.