CMSC 22610	Implementation of	Project 3
Winter 2015	Computer Languages I	February 9, 2015

Flang typechecker

Due: February 27, 2015

1 Introduction

The third project is to implement a type checker for **Flang**, 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 an ML-ULex based scanner (but you may also use your hand-written scanner from Project 1), an ML-Antlr based parser (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 **Flang**.[1] The type system for **Flang** is essentially an enrichment of the *System F* type system.

2 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 irredundant.

[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 implementation from a specification.

$$
\begin{array}{rcll}
\alpha, \beta & \in & \textsc{TyVar} & \text{type variables} \\
\theta^{(k)} & \in & \textsc{TyCon} & k\text{-ary type constructors} \\
\tau & ::= & \forall\overline{\alpha}(\tau) & \text{type abstraction } (|\overline{\alpha}| > 0) \\
 & | & \tau_1 \to \tau_2 & \text{function type} \\
 & | & \tau_1 \times \cdots \times \tau_n & \text{tuple types } (n > 1) \\
 & | & \theta^{(k)}[\,\overline{\tau}\,] & \text{type constructor instantiation } (|\overline{\tau}| \geq 0) \\
 & | & \alpha & \text{type variable}
\end{array}
$$

Figure 1: **Flang** semantic types

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

3 Flang types

In the **Flang** 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 **Flang** program:

```
1  data T with
2    con A of Integer
3    con B;
4  let x : T = A 1;
5  data T with
6    con C of Integer
7    con D;
8  let y : T = B;
9  0
```

This program has a type error at line 8 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 line 5. The second **`data`** declaration at line 5 *shadows* the earlier declaration at line 1. In the parse-tree representation, however, all instances of `T` correspond to the same type constructor name (that is, as values of the `Atom.atom` type.

The abstract syntax of **Flang** semantic types is given in Figure 1 (and represented by the `Type.ty` datatype in the project seed code). The set of semantic types (TYPE) is built from countable sets of semantic type variables (TYVAR) and semantic type constructors (TYCON). We use τ to denote types, α and β 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 $\mathbf{Bool}^{(0)}$ instead of $\mathbf{Bool}^{(0)}[\,]$).

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 **`data`** declaration at line 1 will introduce one type constructor, say $\theta_1^{(0)}$, and type checking the **`data`** declaration at line 5 will introduce a different type constructor, say $\theta_2^{(0)}$. 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 $\overline{\alpha}$ 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 $\overline{\tau}$ for variables $\overline{\alpha}$ in a type τ' is written as $\tau'[\overline{\alpha}/\overline{\tau}]$.

We consider semantic types equal up to renaming of bound type variables.[2] That is, we will consider the semantic types $\forall\alpha(\alpha \to \alpha \to \mathbf{Bool}^{(0)})$ and $\forall\beta(\beta \to \beta \to \mathbf{Bool}^{(0)})$ 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 Flang abstract syntax

The typing rules for **Flang** are defined over an abstraction of the concrete syntax. We use the following naming conventions for variables in the abstract syntax:

T	$\in$	TyId	Type constructor identifiers
t	$\in$	TyVar	Type variable identifiers
C	$\in$	ConId	Data constructor identifiers
f, x	$\in$	ValId	Value identifiers

The grammar, which is given in Figure 2, roughly corresponds to the datatypes defined in the `ParseTree` module from Project 2. As with the syntax of semantic types, we use the overbear notation to denote sequences of syntactic objects (*e.g.*, $\overline{t}$ to represent sequences of type variables and $\overline{typ}$ to represent sequences of types).

binary expressions

5 Environments

The typing rules for **Flang** use a number of different *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

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

and we write $E \uplus E'$ for the *disjoint union* of E and E' when $\mathrm{dom}(E) \cap \mathrm{dom}(E') = \emptyset$ (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:

TyVarEnv	$=$	TyVar $\rightharpoonup$ TyVar	type-variable environment
TyConEnv	$=$	TyId $\rightharpoonup$ TyCon $\cup$ (TyVar* $\times$ Type)	type-constructor environment
DConEnv	$=$	ConId $\rightharpoonup$ Type	data-constructor environment
VarEnv	$=$	ValId $\rightharpoonup$ Type	variable environment

[2] This renaming is called α-conversion.

$prog$	$::=$	$def\ prog$	toplevel definition
	$\mid$	exp	program body
def	$::=$	$\mathbf{type}\, T[\,\overline{t}\,] = typ$	type alias definition
	$\mid$	$\mathbf{data}\, T[\,\overline{t}\,]\, \mathbf{with}\, \overline{con}$	data type definition
	$\mid$	$bind$	value binding definition
typ	$::=$	$[\,\overline{t}\,]typ$	type function
	$\mid$	$typ_1 \to typ_2$	function type
	$\mid$	$typ_1 * \cdots * typ_n$	tuple type
	$\mid$	$T[\,\overline{typ}\,]$	type constructor
	$\mid$	t	type variable
con	$::=$	$C\, \mathbf{of}\, typ$	data constructor
	$\mid$	C	nullary data constuctor
$bind$	$::=$	$\mathbf{fun}\, f\ fnsig = exp$	function binding
	$\mid$	$\mathbf{let}\, spat\ :\ typ = exp$	value binding
	$\mid$	$\mathbf{let}\, spat = exp$	value binding
$fnsig$	$::=$	$[\,\overline{t}\,]\ fnsig$	function type parameters
	$\mid$	$(x : typ)\ fnsig$	function value parameter
	$\mid$	$\to typ$	function return type
exp	$::=$	$\mathbf{if}\, exp_1\, \mathbf{then}\, exp_2\, \mathbf{else}\, exp_3$	conditional expression
	$\mid$	$exp_1 :: exp_2$	list-cons expression
	$\mid$	$exp_1\ exp_2$	application expression
	$\mid$	$exp[\,\overline{typ}\,]$	type-application expression
	$\mid$	$(exp_1,\ \ldots,\ exp_n)$	tuple expression
	$\mid$	$\{\ scope\ \}$	nested scope
	$\mid$	$\mathbf{case}\, exp\, \mathbf{of}\, \overline{rule}$	case expression
	$\mid$	x	variable
	$\mid$	C	data constructor
	$\mid$	lit	literal
$rule$	$::=$	$\{\ pat\ \Rightarrow\ scope\ \}$	match-case rule
pat	$::=$	$C\ spat$	data-constructor pattern
	$\mid$	$spat_1 :: spat_2$	list-cons expression
	$\mid$	$(spat_1,\ \ldots,\ spat_n)$	tuple pattern
	$\mid$	C	nullary-data-constructor pattern
	$\mid$	$spat$	simple pattern
$spat$	$::=$	x	variable pattern
	$\mid$	$\_$	wild-card pattern
$scope$	$::=$	$bind\ scope$	value binding
	$\mid$	exp	expression

Figure 2: Abstract syntax of **Flang**

$E \vdash prog \blacktriangleright \mathbf{Ok}$	type checking a program
$E \vdash def \blacktriangleright E'$	type checking a definition
$E \vdash typ \blacktriangleright \tau$	type checking a type
$E, \theta^{(k)}, \overline{\alpha} \vdash con \blacktriangleright E'$	type checking a data constructor definition
$E \vdash bind \blacktriangleright E'$	type checking a value binding
$E \vdash fnsig \blacktriangleright E', \tau, \tau_{ret}$	type checking a function signature
$E \vdash exp \blacktriangleright \tau'$	type checking an expression
$E, \tau \vdash rule \blacktriangleright \tau$	type checking a match-case rule
$E, \tau \vdash pat \blacktriangleright E$	type checking a pattern
$\tau \vdash spat \blacktriangleright E'$	type checking a simple pattern
$E \vdash scope \blacktriangleright \tau$	type checking a scope

Figure 3: Typing judgments for **Flang**

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 $\Lambda\overline{\alpha} : \tau$ 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.

$$E \in \textsc{Env} = \textsc{TyVarEnv} \times \textsc{TyConEnv} \times \textsc{DConEnv} \times \textsc{VarEnv}$$

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

$$\begin{aligned}&\langle TVE, TCE, DCE, VE\rangle \pm \langle TVE', TCE', DCE', VE'\rangle \\ &\quad = \langle TVE \pm TVE', TCE \pm TCE', DCE \pm DCE', VE \pm VE'\rangle\end{aligned}$$

$$\begin{aligned}&\langle TVE, TCE, DCE, VE\rangle \uplus \langle TVE', TCE', DCE', VE'\rangle \\ &\quad = \langle TVE \uplus TVE', TCE \uplus TCE', DCE \uplus DCE', VE \uplus VE'\rangle\end{aligned}$$

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

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

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

6 Typing rules

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

$$Context \vdash Term \blacktriangleright Descr$$

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 different judgement forms used in the typing rules for **Flang** are summarized in Figure 3. Formally, the judgments are smallest relation that satisfies the typing rules.

The typing rules for **Flang** are *syntax directed*, which means that there is a typing rule for each (major) syntactic form in the parse-tree representation of **Flang** programs. For each of the syntactic forms in the abstract syntax, there is a typing rule written in a *natural deduction* style.

$$\frac{\mathit{premise}_1 \qquad \cdots \qquad \mathit{primise}_n}{\mathit{conclusion}}$$

where the conclusion will be the typing judgment for the syntactic form in question.

6.1 Programs

$\boxed{E \vdash \mathit{prog} \blacktriangleright \mathbf{Ok}}$

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

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

$$\frac{E \vdash \mathit{def} \blacktriangleright E' \qquad E' \vdash \mathit{prog} \blacktriangleright \mathbf{Ok}}{E \vdash \mathit{def}\ \mathit{prog} \blacktriangleright \mathbf{Ok}}$$

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

$$\frac{E \vdash \mathit{exp} \blacktriangleright \tau}{E \vdash \mathit{exp} \blacktriangleright \mathbf{Ok}}$$

6.2 Definitions

$\boxed{E \vdash \mathit{def} \blacktriangleright E'}$

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

$$\frac{\overline{\alpha} \text{ are fresh} \qquad |\overline{t}| = |\overline{\alpha}| = k \qquad E \pm \{\overline{t} \mapsto \overline{\alpha}\} \vdash \mathit{typ} \blacktriangleright \tau \qquad E' = E \pm \{T \mapsto \Lambda\overline{\alpha} : \tau\}}{E \vdash \mathbf{type}\, T[\,\overline{t}\,] = \mathit{typ} \blacktriangleright E'}$$

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.

$$\frac{\begin{array}{c}\overline{\alpha} \text{ are fresh} \qquad |\overline{t}| = |\overline{\alpha}| = k \qquad \theta^{(k)} \text{ is fresh} \qquad E' = E \pm \{T \mapsto \theta^{(k)}\} \\ E' \pm \{\overline{t} \mapsto \overline{\alpha}\}, \overline{\alpha}, \theta^{(k)} \vdash \mathit{con}_i \blacktriangleright E_i \text{ for } 1 \leq i \leq n\end{array}}{E \vdash \mathbf{data}\, T[\,\overline{t}\,]\, \mathbf{with}\, \mathit{con}_1 \cdots \mathit{con}_n \blacktriangleright E' \pm (E_1 \uplus \cdots \uplus E_n)}$$

The value binding definition is checked just like a value binding (see Section 6.5).

$$\frac{E \vdash \mathit{bind} \blacktriangleright E'}{E \vdash \mathit{bind} \blacktriangleright E'}$$

6.3 Types

$\boxed{E \vdash typ \blacktriangleright \tau}$

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 ($\overline{\alpha}$) for the syntactic type variables ($\overline{t}$).

$$\frac{\overline{\alpha}\ \textit{are fresh} \qquad E \pm \{\overline{t} \mapsto \overline{\alpha}\} \vdash typ \blacktriangleright \tau}{E \vdash [\,\overline{t}\,]typ \blacktriangleright \forall\overline{\alpha}(\tau)}$$

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

$$\frac{E \vdash typ_1 \blacktriangleright \tau_1 \qquad E \vdash typ_2 \blacktriangleright \tau_2}{E \vdash typ_1 \to typ_2 \blacktriangleright \tau_1 \to \tau_2}$$

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

$$\frac{E \vdash typ_1 \blacktriangleright \tau_1 \qquad \cdots \qquad E \vdash typ_n \blacktriangleright \tau_n}{E \vdash typ_1 * \cdots * typ_n \blacktriangleright \tau_1 \times \cdots \times \tau_n}$$

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 builtin 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.

$$\frac{E(T) = \theta^{(k)} \qquad |\overline{typ}| = k \qquad E \vdash \overline{typ} \blacktriangleright \overline{\tau}}{E \vdash T[\,\overline{typ}\,] \blacktriangleright \theta^{(k)}[\,\overline{\tau}\,]}$$

For **`data`** definitions (or abstract types), we check the type arguments and then construct a new (semantic) type by substituting the $\overline{\tau}$ for the $\overline{\alpha}$.

$$\frac{E(T) = \Lambda\overline{\alpha} : \tau_T \qquad |\overline{\alpha}| = |\overline{typ}| \qquad E \vdash \overline{typ} \blacktriangleright \overline{\tau}}{E \vdash T[\,\overline{typ}\,] \blacktriangleright \tau_T[\overline{\alpha}/\overline{\tau}]}$$

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

$$\frac{t \in \mathrm{dom}(E)}{E \vdash t \blacktriangleright E(t)}$$

6.4 Data constructors

$\boxed{E, \overline{\alpha}, \theta^{(k)} \vdash con \blacktriangleright E'}$

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

$$\frac{E \vdash typ \blacktriangleright \tau}{E, \overline{\alpha}, \theta^{(k)} \vdash C\ \mathbf{of}\ typ \blacktriangleright \{C \mapsto \forall\overline{\alpha}(\tau \to \theta^{(k)}[\,\overline{\alpha}\,])\}}$$

Checking nullify data constructors requires no additional checks.

$$\frac{}{E, \overline{\alpha}, \theta^{(k)} \vdash C \blacktriangleright \{C \mapsto \forall\overline{\alpha}(\theta^{(k)}[\,\overline{\alpha}\,])\}}$$

6.5 Value bindings

$\boxed{E \vdash \mathit{bind} \blacktriangleright E'}$

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).

$$\frac{E \vdash \mathit{fnsig} \blacktriangleright E', \tau, \tau_{\mathit{ret}} \qquad E' \pm \{f \mapsto \tau\} \vdash \mathit{exp} \blacktriangleright \tau_{\mathit{ret}}}{E \vdash \mathbf{fun}\ f\ \mathit{fnsig} = \mathit{exp} \blacktriangleright E \pm \{f \mapsto \tau\}}$$

where $\mathrm{ReturnType}(\tau)$ 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.

$$\frac{E \vdash \mathit{typ} \blacktriangleright \tau \qquad E \vdash \mathit{exp} \blacktriangleright \tau \qquad \tau \vdash \mathit{spat} \blacktriangleright E'}{E \vdash \mathbf{let}\ \mathit{spat}\ :\ \mathit{typ} = \mathit{exp} \blacktriangleright E \pm E'}$$

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.

$$\frac{E \vdash \mathit{exp} \blacktriangleright \tau \qquad \tau \vdash \mathit{spat} \blacktriangleright E'}{E \vdash \mathbf{let}\ \mathit{spat} = \mathit{exp} \blacktriangleright E \pm E'}$$

A value binding that is just an expression exp is viewed as syntactic sugar for the binding

$$\mathbf{let}\ \_\ :\ \mathrm{Unit} = \mathit{exp}$$

which is reflected in its typing rule.

$$\frac{E \vdash \mathit{exp} \blacktriangleright \mathbf{Unit}^{(0)}}{E \vdash \mathit{exp} \blacktriangleright E}$$

6.6 Function signatures

$\boxed{E \vdash \mathit{fnsig} \blacktriangleright E', \tau, \tau_{\mathit{ret}}}$

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.

$$\frac{\overline{\alpha}\ \text{are fresh} \qquad E \pm \{\overline{t} \mapsto \overline{\alpha}\} \vdash \mathit{fnsig} \blacktriangleright E', \tau, \tau_{\mathit{ret}}}{E \vdash [\,\overline{t}\,]\ \mathit{fnsig} \blacktriangleright E', \forall\overline{\alpha}(\tau), \tau_{\mathit{ret}}}$$

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.

$$\frac{E \vdash \mathit{typ} \blacktriangleright \tau \qquad E \pm \{x \mapsto \tau\} \vdash \mathit{fnsig} \blacktriangleright E', \tau', \tau_{\mathit{ret}}}{E \vdash (x : \mathit{typ})\ \mathit{fnsig} \blacktriangleright E', \tau \to \tau', \tau_{\mathit{ret}}}$$

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.

$$\frac{E \vdash \mathit{typ} \blacktriangleright \tau}{E \vdash \rightarrow \mathit{typ} \blacktriangleright E, \tau, \tau}$$

6.7 Expressions

$\boxed{E \vdash \mathit{exp} \blacktriangleright \tau}$

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.

$$\frac{E \vdash \mathit{exp}_1 \blacktriangleright \mathbf{Bool}^{(0)} \qquad E \vdash \mathit{exp}_2 \blacktriangleright \tau \qquad E \vdash \mathit{exp}_2 \blacktriangleright \tau}{E \vdash \mathbf{if}\ \mathit{exp}_1\ \mathbf{then}\ \mathit{exp}_2\ \mathbf{else}\ \mathit{exp}_3 \blacktriangleright \tau}$$

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.

$$\frac{E \vdash \mathit{exp}_1 \blacktriangleright \tau \qquad E \vdash \mathit{exp}_2 \blacktriangleright \mathbf{List}^{(1)}[\,\tau\,]}{E \vdash \mathit{exp}_1 :: \mathit{exp}_2 \blacktriangleright \mathbf{List}^{(1)}[\,\tau\,]}$$

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.

$$\frac{E \vdash \mathit{exp}_1 \blacktriangleright \tau' \rightarrow \tau \qquad E \vdash \mathit{exp}_2 \blacktriangleright \tau'}{E \vdash \mathit{exp}_1\ \mathit{exp}_2 \blacktriangleright \tau}$$

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.

$$\frac{E \vdash \mathit{exp} \blacktriangleright \forall \overline{\alpha}(\tau) \qquad E \vdash \overline{\mathit{typ}} \blacktriangleright \overline{\tau}' \qquad |\overline{\alpha}| = |\overline{\mathit{typ}}|}{E \vdash \mathit{exp}[\,\overline{\mathit{typ}}\,] \blacktriangleright \tau[\overline{\alpha}/\overline{\tau}']}$$

Checking a tuple expression involves checking each of the subexpressions.

$$\frac{E \vdash \mathit{exp}_i \blacktriangleright \tau_i \text{ for } 1 \leq i \leq n}{E \vdash (\mathit{exp}_1, \ldots, \mathit{exp}_n) \blacktriangleright \tau_1 \times \cdots \times \tau_n}$$

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.

$$\frac{E \vdash \mathit{scope} \blacktriangleright \tau}{E \vdash \{\ \mathit{scope}\ \} \blacktriangleright \tau}$$

$$\frac{E \vdash \mathit{exp} \blacktriangleright \tau \qquad E, \tau \vdash \mathit{rule}_i \blacktriangleright \tau' \text{ for } 1 \leq i \leq n}{E \vdash \mathbf{case}\ \mathit{exp}\ \mathbf{of}\ \mathit{rule}_1 \cdots \mathit{rule}_n \blacktriangleright \tau'}$$

Variables are mapped to their type in the environment.

$$\frac{x \in \mathrm{dom}(E)}{E \vdash x \blacktriangleright E(x)}$$

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

$$\frac{C \in \mathrm{dom}(E)}{E \vdash C \blacktriangleright E(C)}$$

To type check literal expressions, we assume a function TypeOf that maps literals to their type (*i.e.*, ether $\mathbf{Integer}^{(0)}$ or $\mathbf{String}^{(0)}$).

$$\frac{}{E \vdash \mathit{lit} \blacktriangleright \mathrm{TypeOf}(\mathit{lit})}$$

6.8 Match-case rules

$\boxed{E, \tau \vdash \mathit{rule} \blacktriangleright \tau'}$

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.

$$\frac{E, \tau \vdash \mathit{pat} \blacktriangleright E' \qquad E \pm E' \vdash \mathit{scope} \blacktriangleright \tau'}{E, \tau \vdash \{\, \mathit{pat} \;\Rightarrow\; \mathit{scope} \,\} \blacktriangleright \tau'}$$

6.9 Patterns

$\boxed{E, \tau \vdash \mathit{pat} \blacktriangleright E'}$

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.

$$\frac{\tau = \theta^{(k)}[\,\overline{\tau}'\,] \qquad E(C) = \forall \overline{\alpha}(\tau'' \rightarrow \theta^{(k)}[\,\overline{\alpha}\,]) \qquad \tau''[\overline{\alpha}/\overline{\tau}'] \vdash \mathit{spat} \blacktriangleright E'}{E, \tau \vdash C\ \mathit{spat} \blacktriangleright E'}$$

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.

$$\frac{\tau = \mathbf{List}^{(1)}[\,\tau'\,] \qquad \tau' \vdash \mathit{spat}_1 \blacktriangleright E_1 \qquad \tau \vdash \mathit{spat}_2 \blacktriangleright E_2}{E, \tau \vdash \mathit{spat}_1 :: \mathit{spat}_2 \blacktriangleright E_1 \pm E_2}$$

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.

$$\frac{\begin{array}{c}\tau = \tau_1 \times \cdots \times \tau_n \\ \tau_i \vdash spat_i \blacktriangleright E_i \text{ for } 1 \leq i \leq n \\ E' = E_1 \uplus \cdots \uplus E_n\end{array}}{E, \tau \vdash (spat_1, \ \ldots, \ spat_n) \blacktriangleright E'}$$

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

$$\frac{\tau = \theta^{(k)}[\,\tau'\,] \qquad E(C) = \forall\overline{\alpha}(\theta^{(k)}[\,\overline{\alpha}\,])}{E, \tau \vdash C \blacktriangleright \emptyset}$$

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

$$\frac{\tau \vdash spat \blacktriangleright E'}{E, \tau \vdash spat \blacktriangleright E'}$$

6.10 Simple patterns

$\boxed{\tau \vdash spat \blacktriangleright E'}$

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.

$$\frac{}{\tau \vdash x \blacktriangleright \{x \mapsto \tau\}}$$

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

$$\frac{}{\tau \vdash \_ \blacktriangleright \emptyset}$$

6.11 Scopes

$\boxed{E \vdash scope \blacktriangleright \tau}$

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

$$\frac{E \vdash bind \blacktriangleright E' \qquad E' \vdash scope \blacktriangleright \tau}{E \vdash bind\ scope \blacktriangleright \tau}$$

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

$$\frac{E \vdash exp \blacktriangleright \tau}{E \vdash exp \blacktriangleright \tau}$$

7 Predefined identifiers

Flang programs can use a number of predefined types, constructors, operators, and functions. This section details the types for these.

7.1 Binary operators

The grammar given in Figure 2 does not include a form for infix binary operators. Instead, we treat the expression $exp_1 \odot exp_2$ as shorthand for $\odot\,(exp_1,\ exp_2)$. The binary operators have the following types:

`==`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Bool}^{(0)}$	integer equality
`<=`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Bool}^{(0)}$	integer less-than-or-equal
`<`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Bool}^{(0)}$	integer less-than
`@`	:	$\mathbf{String}^{(0)} \times \mathbf{String}^{(0)} \to \mathbf{String}^{(0)}$	string concatenation
`+`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Integer}^{(0)}$	integer addition
`-`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Integer}^{(0)}$	integer substraction
`*`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Integer}^{(0)}$	integer multiplication
`/`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Integer}^{(0)}$	integer division
`%`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Integer}^{(0)}$	integer modulo

Note that the infix list-constructor `::` has its own syntactic form for both expressions and patterns, along with special typing rules.

7.2 The Flang basis

A **Flang** program is checked in the context of an initial environment (also known as a basis environment) E_0 that provides predefined type constructors, data constructors, and variables. This initial environment is defined as follows:

$$E_0 = \langle TCE_0,\, TCE_0,\, DCE_0,\, VE_0 \rangle$$

where

$$TCE_0 = \{\}$$

$$TCE_0 = \left\{ \begin{array}{rcl} \texttt{Bool} & \mapsto & \mathbf{Bool}^{(0)} \\ \texttt{Integer} & \mapsto & \mathbf{Integer}^{(0)} \\ \texttt{List} & \mapsto & \mathbf{List}^{(1)} \\ \texttt{String} & \mapsto & \mathbf{String}^{(0)} \\ \texttt{Unit} & \mapsto & \mathbf{Unit}^{(0)} \end{array} \right\}$$

$$DCE_0 = \left\{ \begin{array}{rcl} \texttt{False} & \mapsto & \mathbf{Bool}^{(0)} \\ \texttt{Nil} & \mapsto & \forall\alpha(\mathbf{List}^{(1)}[\,\alpha\,]) \\ \texttt{True} & \mapsto & \mathbf{Bool}^{(0)} \\ \texttt{Unit} & \mapsto & \mathbf{Unit}^{(0)} \end{array} \right\}$$

$$VE_0 = \left\{ \begin{array}{rcl} \texttt{argc} & \mapsto & \mathbf{Unit}^{(0)} \to \mathbf{Integer}^{(0)} \\ \texttt{arg} & \mapsto & \mathbf{Integer}^{(0)} \to \mathbf{String}^{(0)} \\ \texttt{fail} & \mapsto & \forall\alpha(\mathbf{String}^{(0)} \to \alpha) \\ \texttt{ignore} & \mapsto & \forall\alpha(\alpha \to \mathbf{Unit}^{(0)}) \\ \texttt{neg} & \mapsto & \mathbf{Integer}^{(0)} \to \mathbf{Integer}^{(0)} \\ \texttt{not} & \mapsto & \mathbf{Bool}^{(0)} \to \mathbf{Bool}^{(0)} \\ \texttt{print} & \mapsto & \mathbf{String}^{(0)} \to \mathbf{Unit}^{(0)} \\ \texttt{size} & \mapsto & \mathbf{String}^{(0)} \to \mathbf{Integer}^{(0)} \\ \texttt{sub} & \mapsto & \mathbf{String}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Integer}^{(0)} \end{array} \right\}$$

The provided `Basis` module defines the semantic objects for these identifiers.

8 Building the abstract syntax tree

In addition to checking the program for type correctness, your program is also responsible for converting the parse tree representation of the program into the abstract syntax tree (AST) representation. The typed abstract syntax tree (AST) is a further simplification of the program representation.

- Unlike the parse tree, the AST does not include source location information.
- AST type variables, type constructors, data constructors, and variables in the AST representation are implemented the types `TyVar.t`, `TyCon.t`, `DataCon.t`, and `Var.t` (respectively). 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 names. Each of the provided modules includes a `new` function for creating unique identifiers.
- Variables bound in simple patterns include their type.
- Wild cards are replaced with unique variables.
- There is no **`type`** declaration form; all type abbreviations will have been expanded during type checking and conversion to the abstract syntax tree.
- The **`let`** declaration form has no type constraint, since the representation of AST variables includes the variable's type.
- Polymorphic data constructors in patterns are applied to their type arguments.
- The body of program is represented as a single expression.

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 6.3:

$E \vdash typ \blacktriangleright \tau$ in the environment E, the parse tree type typ is well-formed and translates to the AST type τ.

We can imagine other judgements that combine type checking with translation to the AST:

$E \vdash exp \blacktriangleright \tau; e$	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.
$E \vdash def \blacktriangleright E'; d$	in the environment E, the parse tree declaration *Decl* returns the environment E′ and translates to the abstract syntax tree declaration d.
$E \vdash prog \blacktriangleright p$	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 6, except they will construct an appropriate output abstract syntax tree form.

9 Hints

- This project is the most difficult and substantial of the projects, so budget your time appropriately and start early.
- Study the sample code carefully. You will need to be familiar with the types and operations in the `ParseTree`, `Type`, and `AST` modules.
- Think about how to represent the environment(s). The SML/NJ Library `AtomMap` structure provides an implementation of finite maps on `Atom.atom` keys. You may find it useful to define a "context" type that groups the envronments with the current span and the error stream.
- 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 6. 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.

10 Requirements

We will seed a directory, called `proj3`, in your `phoenixforge` repository. The directory will contain subdirectories for the parser and scanner (`proj3/parser`), type checker (`proj3/typechecker`), common utility code (`proj3/util`), and the driver (`proj3/driver`).

Your task is to implement the type checker according to the rules given in this document. We will collect the projects at **10pm** on **Friday February 27th** from the SVN repositories, so make sure that you have committed your final version before then.

Document History

February 10, 2015 Fixed a number of minor typos

February 9, 2015 Original version