CMSC 22620/32620 Spring 2015	Implementation of Computer Languages II	Handout 2 April 1, 2015

The Flang Type System

1 Introduction

This document provides a formal description of the Flang type system. 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-type names in a group of data-type definitions must be distinct.
- The data constructors in a group of data-type definitions must be distinct.
- The names of functions in a group of function definitions 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.
- The variables in a pattern must be distinct.
- The patterns in a case expression must be exhaustive and irredundant.
- Integer literals must be in the range $[-2^{62},\ 2^{62}-1]$.

The type checker checks these properties and reports an error when they are violated.

α, β	$\in$	TyVar	type variables
$\theta^{(k)}$	$\in$	TyCon	k-ary type constructors
τ	::=	$\forall\overline{\alpha}(\tau)$	type abstraction ($\lvert\overline{\alpha}\rvert > 0$
	$\mid$	$\tau_1 \rightarrow \tau_2$	function type
	$\mid$	$\tau_1 \times \cdots \times \tau_n$	tuple types ($n > 1$)
	$\mid$	$\theta^{(k)}[\,\overline{\tau}\,]$	type constructor instantiation ($\lvert\overline{\tau}\rvert \geq 0$)
	$\mid$	α	type variable

Figure 1: **Flang** semantic types

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 $\lvert\overline{\alpha}\rvert$ 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.[1] 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

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

[1] This renaming is called α-conversion.

$prog$	::=	$def\ prog$	toplevel definition
	|	exp	program body
def	::=	$\mathbf{type}\, T[\,\overline{t}\,] = typ$	type alias definition
	|	$\mathbf{data}\, data_1\, \mathbf{and}\, \cdots\, \mathbf{and}\, data_n$	data type definitions
	|	$bind$	value binding definition
$data$	::=	$T[\,\overline{t}\,]\, \mathbf{with}\, \overline{con}$	data type definition
typ	::=	$[\,\overline{t}\,]typ$	type function
	|	$typ_1 \to typ_2$	function type
	|	$typ_1 * \cdots * typ_n$	tuple type
	|	$T[\,\overline{typ}\,]$	type constructor
	|	t	type variable
con	::=	$C\, \mathbf{of}\, typ$	data constructor
	|	C	nullary data constuctor
$bind$	::=	$\mathbf{fun}\, fndef_1\, \mathbf{and}\, \cdots\, \mathbf{and}\, fndef_n$	function bindings
	|	$\mathbf{let}\, spat\ :\ typ = exp$	value binding
	|	$\mathbf{let}\, spat = exp$	value binding
$fndef$	::=	$f\ fnsig = exp$	function definition
$fnsig$	::=	$[\,\overline{t}\,]\ fnsig$	function type parameters
	|	$(x : typ)\ fnsig$	function value parameter
	|	$\to typ$	function return type
exp	::=	$\mathbf{if}\, exp_1\, \mathbf{then}\, exp_2\, \mathbf{else}\, exp_3$	conditional expression
	|	$exp_1 :: exp_2$	list-cons expression
	|	$exp_1\ exp_2$	application expression
	|	$exp[\,\overline{typ}\,]$	type-application expression
	|	$(exp_1,\ \ldots,\ exp_n)$	tuple expression
	|	$\{\ scope\ \}$	nested scope
	|	$\mathbf{case}\, exp\, \mathbf{of}\, \overline{rule}$	case expression
	|	$\mathbf{try}\, exp\, \mathbf{catch}\, rule$	try-catch expression
	|	$\mathbf{throw}\, [\ typ\ ]\, exp$	throw expression
	|	x	variable
	|	C	data constructor
	|	lit	literal
$rule$	::=	$\{\ pat\ \Rightarrow\ scope\ \}$	match-case rule
pat	::=	$C\ spat$	data-constructor pattern
	|	$spat_1 :: spat_2$	list-cons expression
	|	$(spat_1,\ \ldots,\ spat_n)$	tuple pattern
	|	C	nullary-data-constructor pattern
	|	$spat$	simple pattern
$spat$	::=	x	variable pattern
	|	_	wild-card pattern
$scope$	::=	$bind\ scope$	value binding
	|	exp	expression

Figure 2: Abstract syntax of **Flang**

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.

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

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

6.1 Programs

$$\boxed{E \vdash prog \blacktriangleright \textbf{Ok}}$$

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.

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

$E \vdash prog \blacktriangleright \mathbf{Ok}$	type checking a program
$E \vdash def \blacktriangleright E'$	type checking a definition
$\vdash data \blacktriangleright TCE$	determine type constructors
$E \vdash data \blacktriangleright E'$	check data constructors
$E \vdash typ \blacktriangleright \tau$	type checking a type
$E, \theta^{(k)}, \overline{\alpha} \vdash con \blacktriangleright DCE$	type checking a data constructor definition
$E \vdash bind \blacktriangleright E'$	type checking a value binding
$E \vdash fndef \blacktriangleright VE$	checking function signatures
$E \vdash fndef \blacktriangleright \mathbf{Ok}$	checking function bodies
$E \vdash fnsig \blacktriangleright E', \tau, \tau_{ret}$	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**

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

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

6.2 Definitions

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

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.

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

Checking a group of mutually recursive data-type definitions requires first constructing a type-constructor environment that maps the type names to their semantic type constructors and then checking the data constructors. Note that the use of $\uplus$ in this rule forces the type names and data-constructor names to be unique.

$$\frac{\begin{array}{c}\vdash data_i \blacktriangleright TCE_i \text{ for } 1 \le i \le n \\ E' = E \pm (TCE_1 \uplus \cdots \uplus TCE_n) \\ E' \vdash data_i \blacktriangleright DCE_i \text{ for } 1 \le i \le n\end{array}}{E \vdash \mathbf{data}\, data_1 \,\mathbf{and}\, \cdots \,\mathbf{and}\, data_n \blacktriangleright E' \pm (DCE_1 \uplus \cdots \uplus DCE_n)}$$

The value binding definition is checked just like a value binding (see Section 6.7).

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

6.3 Data definitions (1)

$\boxed{\vdash \mathit{data} \blacktriangleright \mathit{TCE}}$

The first pass over a data-type definition yields an environment mapping the type name to its corresponding semantic type constructor.

$$\frac{|\overline{t}| = k \qquad \theta^{(k)} \text{ is fresh}}{\vdash T[\,\overline{t}\,]\,\mathbf{with}\; con_1 \cdots con_n \blacktriangleright \{T \mapsto \theta^{(k)}\}}$$

6.4 Data definitions (2)

$\boxed{E \vdash \mathit{data} \blacktriangleright \mathit{DCE}}$

The second pass over a data-type definition yields an environment mapping the data-constructor names to their semantic representation.

$$\frac{\begin{array}{c}\overline{\alpha} \text{ are fresh} \qquad |\overline{\alpha}| = k \qquad E(T) = \theta^{(k)} \\ E \pm \{\overline{t} \mapsto \overline{\alpha}\}, \overline{\alpha}, \theta^{(k)} \vdash con_i \blacktriangleright DCE_i \text{ for } 1 \leq i \leq n\end{array}}{E \vdash T[\,\overline{t}\,]\,\mathbf{with}\; con_1 \cdots con_n \blacktriangleright DCE_1 \uplus \cdots \uplus DCE_n}$$

6.5 Types

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

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 ($\overline{t}$).

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

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

$$\frac{E \vdash \mathit{typ}_1 \blacktriangleright \tau_1 \qquad E \vdash \mathit{typ}_2 \blacktriangleright \tau_2}{E \vdash \mathit{typ}_1 \to \mathit{typ}_2 \blacktriangleright \tau_1 \to \tau_2}$$

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

$$\frac{E \vdash \mathit{typ}_1 \blacktriangleright \tau_1 \qquad \cdots \qquad E \vdash \mathit{typ}_n \blacktriangleright \tau_n}{E \vdash \mathit{typ}_1 * \cdots * \mathit{typ}_n \blacktriangleright \tau_1 \times \cdots \times \tau_n}$$

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 type arguments and then construct a new (semantic) type by substituting the $\overline{\tau}$ for the $\overline{\alpha}$.

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

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.

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

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

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

6.6 Data constructors

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

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.

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

6.7 Value bindings

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

Type checking a group of mutually recursive function definitions involves first checking their signatures to produce a value environment that assigns a type to each of the functions. Then, using that environment, we check that the function bodies are well typed.

$$\frac{\begin{array}{c} E \vdash fndef_i \blacktriangleright VE_i \text{ for } 1 \le i \le n \\ E' = E \pm (VE_1 \uplus \cdots \uplus VE_n) \\ E' \vdash fndef_i \blacktriangleright \mathbf{Ok} \text{ for } 1 \le i \le n \end{array}}{E \vdash \mathbf{fun}\,fndef_1\,\mathbf{and}\,\cdots\,\mathbf{and}\,fndef_n \blacktriangleright E'}$$

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.

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

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.

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

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

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

which is reflected in its typing rule.

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

6.8 Function definitions (1)

$\boxed{E \vdash fndef \blacktriangleright VE}$

The first pass over function definitions checks their signatures, which yields a value environment assigning a type to the functions.

$$\frac{E \vdash fnsig \blacktriangleright E', \tau, \tau_{ret}}{E \vdash f\ fnsig = exp \blacktriangleright \{f \mapsto \tau\}}$$

6.9 Function definitions (2)

$\boxed{E \vdash fndef \blacktriangleright \mathbf{Ok}}$

The second pass checks that the function bodies are well typed and that the declared return type of the function matches the type of the function body.

$$\frac{E \vdash fnsig \blacktriangleright E', \tau, \tau_{ret} \qquad E' \vdash exp \blacktriangleright \tau_{ret}}{E \vdash f\ fnsig = exp \blacktriangleright \mathbf{Ok}}$$

6.10 Function signatures

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

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.

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

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.

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

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.

$$\frac{E \vdash typ \blacktriangleright \tau}{E \vdash \to typ \blacktriangleright E, \tau, \tau}$$

6.11 Expressions

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

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.

$$\frac{E \vdash exp_1 \blacktriangleright \mathbf{Bool}^{(0)} \qquad E \vdash exp_2 \blacktriangleright \tau \qquad E \vdash exp_2 \blacktriangleright \tau}{E \vdash \mathbf{if}\ exp_1\ \mathbf{then}\ exp_2\ \mathbf{else}\ exp_3 \blacktriangleright \tau}$$

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.

$$\frac{E \vdash exp_1 \blacktriangleright \tau \qquad E \vdash exp_2 \blacktriangleright \mathbf{List}^{(1)}[\,\tau\,]}{E \vdash exp_1 :: exp_2 \blacktriangleright \mathbf{List}^{(1)}[\,\tau\,]}$$

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.

$$\frac{E \vdash exp_1 \blacktriangleright \tau' \to \tau \qquad E \vdash exp_2 \blacktriangleright \tau'}{E \vdash exp_1\ exp_2 \blacktriangleright \tau}$$

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.

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

Checking a tuple expression involves checking each of the subexpressions.

$$\frac{E \vdash exp_i \blacktriangleright \tau_i \text{ for } 1 \le i \le n}{E \vdash (exp_1, \ldots, exp_n) \blacktriangleright \tau_1 \times \cdots \times \tau_n}$$

The rules for type checking the body of a nested scope are given below in Section 6.15. The resulting type is the type of the expression.

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

The rule for match cases first checks the type of the argument expression and then uses that type to check each of the rules.

$$\frac{E \vdash exp \blacktriangleright \tau \qquad E, \tau \vdash rule_i \blacktriangleright \tau' \text{ for } 1 \le i \le n}{E \vdash \mathbf{case}\ exp\ \mathbf{of}\ rule_1\ \cdots\ rule_n \blacktriangleright \tau'}$$

Type type of a **`try-catch`** expression is the type of its body, which must match the type of the handler. The pattern in the handler has type $\mathbf{String}^{(0)}$.

$$\frac{E \vdash exp \blacktriangleright \tau \qquad E, \mathbf{String}^{(0)} \vdash rule \blacktriangleright \tau}{E \vdash \mathbf{try}\ exp\ \mathbf{catch}\ rule \blacktriangleright \tau}$$

The **throw** expression has the type that it is applied to; its value argument must be a string.

$$\frac{E \vdash \mathit{typ} \blacktriangleright \tau \qquad E \vdash \mathit{exp} \blacktriangleright \mathbf{String}^{(0)}}{E \vdash \mathbf{throw}\,[\,\mathit{typ}\,]\,\mathit{exp} \blacktriangleright \tau}$$

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 $\mathbf{String}^{(0)}$).

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

6.12 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.

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

6.13 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.

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

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.

$$\frac{\begin{array}{c}\tau = \tau_1 \times \cdots \times \tau_n \\ \tau_i \vdash \mathit{spat}_i \blacktriangleright E_i \text{ for } 1 \leq i \leq n \\ E' = E_1 \uplus \cdots \uplus E_n\end{array}}{E, \tau \vdash (\mathit{spat}_1, \ldots, \mathit{spat}_n) \blacktriangleright E'}$$

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.

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

6.14 Simple patterns

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

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.

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

6.15 Scopes

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

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

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

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:

`||`	:	$\mathbf{Bool}^{(0)} \times \mathbf{Bool}^{(0)} \to \mathbf{Bool}^{(0)}$	or-else conditional operator
`&&`	:	$\mathbf{Bool}^{(0)} \times \mathbf{Bool}^{(0)} \to \mathbf{Bool}^{(0)}$	and-also conditional operator
`==`	:	$\mathbf{Integer}^{(0)} \times \mathbf{Integer}^{(0)} \to \mathbf{Bool}^{(0)}$	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:

$$E_0 = \langle TVE_0, TCE_0, DCE_0, VE_0 \rangle$$

where

$$TVE_0 = \{\}$$

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

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

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

Document History

April 1, 2015 Original version