CSCI-344
Programming Language Concepts (Section 3)

Lecture 16

Kinding Rules for Polymorphic Types

Instructor: Hossein Hojjat

October 12, 2016

Where we are

Done:

- Type Systems
- Typed Impcore

This session:

- Kinding Rules for Polymorphic Types

Recap

- Each language construct (operator, expression, statement, ...) is associated with a **type expression**
- **Type system:** collection of rules for assigning type expressions to constructs
- Type expressions are built from type constructors like `int`, `bool`, `string` and `list`
- **Type-formation** rules tell us what types are legitimate
- Example in Typed Impcore:
  - `int`, `bool`, and `(array int)` are legitimate
  - `(int bool)`, `array`, `(array array)` are not

Type Expression

- General representation of legitimate type expressions:

```
datatype tyex
  = TYCON  of string
  | CONAPP of tyex * tyex list
```

Type Expression

- General representation of legitimate type expressions:

```
datatype tyex
  = TYCON  of string
  | CONAPP of tyex * tyex list
```

- Examples:

`bool`	`TYCON "bool"`
`int list`	`CONAPP (TYCON "list", [TYCON "int"])`

`int * int -> bool`	`CONAPP (TYCON "function",`
	`  [ CONAPP (TYCON "tuple",`
	`  [TYCON "int", TYCON "int"])`
	`  , TYCON "bool" ])`

Type Expression

- General representation of legitimate type expressions:

```
datatype tyex
  = TYCON  of string
  | CONAPP of tyex * tyex list
```

How would you represent an array of pairs of booleans?

```
(bool * bool) array
```

Type Expression

- General representation of legitimate type expressions:

```
datatype tyex
 = TYCON  of string
 | CONAPP of tyex * tyex list
```

How would you represent an array of pairs of booleans?

```
(bool * bool) array
```

```
CONAPP (TYCON "array",
   [ CONAPP (TYCON "tuple",
    [TYCON "bool", TYCON "bool"])
   ])
```

Well-formed Types

We still need to classify type expressions into:

- types that classify terms (e.g. `int`)
- type constructors that build types (e.g. `list`)
- nonsense terms that don't mean anything (e.g., `int int`)

Well-formed Types

We still need to classify type expressions into:

- types that classify terms (e.g. `int`)
- type constructors that build types (e.g. `list`)
- nonsense terms that don't mean anything (e.g., `int int`)

Idea:

- We use types to classify expressions
- Apply the same idea to classify types
- **Kinds** are to types as types are to expressions

Kinds

- Values `"RIT"`, `0`, `false` have types `string`, `int`, `bool`
- Types `string`, `int`, `bool` have kinds $*$

1. `int` constructor:
   - *kind* of constructor that is a type all by itself
2. `list` constructor:
   - *kind* of constructor that has to be applied to a type to make another type

1. Represent first kind with $*$ and pronounce it "type" (nullary type)
2. Represent second kind with $* \Rightarrow *$ and pronounce it "type arrow type"

Kinds

- $*$ is the kind of all nullary type constructors (`string`, `bool`, ...)
- $* \to *$ is the kind of a unary type constructor (`array`, `list`, ...)
- $* \times * \to *$ is the kind of a binary type constructor (`pair`, ...)
- $(* \to *) \to *$ is the kind of a higher-order type operator
  from unary type constructors to nullary types

Kind Formation Rules

$$\frac{}{* \text{ is a kind}} \text{ (KindFormationType)}$$

$$\frac{\kappa_1, \cdots, \kappa_n \text{ are kinds} \qquad \kappa \text{ is a kind}}{\kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \text{ is a kind}} \text{ (KindFormationArrow)}$$

Use kinds to give arities

- Kinds classify type expressions just as types classify terms

Examples:

- `int` :: $*$, `list` :: $* \Rightarrow *$, `pair` :: $* \times * \Rightarrow *$
- Symbol :: is pronounced "has kind"

Non-Examples:

- Have no kind:
- `int int`
- `bool`$\times$ `list`

Kind Environment

- How do we know which type constructors have which kinds?
- Kind environment Δ tracks type constructor names and kinds
- Binding in a kind environment is written using the $::$ symbol

Kinding judgment

- $\Delta \vdash \tau :: \kappa$ Type τ has kind κ
- $\Delta \vdash \tau :: *$ Special case: τ is a type

Kinding Rules for Types

- These three rules replace all type formation rules
- They tell us everything we need to know about formation of types

$$\frac{\mu \in dom(\Delta)}{\Delta \vdash \mathrm{TYCON}(\mu) :: \Delta(\mu)} \text{ (KindIntroCon)}$$

$$\frac{\Delta \vdash \tau :: \kappa_1 \times \cdots \kappa_n \Rightarrow \kappa \qquad \Delta \vdash \tau_i :: \kappa_i,\ 1 \leq i \leq n}{\Delta \vdash \mathrm{CONAPP}(\tau, [\tau_1, \cdots, \tau_n]) :: \kappa} \text{ (KindApp)}$$

$$\frac{\Delta \vdash \tau_i :: *,\ 1 \leq i \leq n \qquad \Delta \vdash \tau :: *}{\Delta \vdash \tau_1 \times \cdots \times \tau_n \to \tau :: *} \text{ (KindFunction)}$$

Monomorphic Type Systems

- Early static type systems (like Pascal and C) were **monomorphic**
- Monomorphism: every expression has a single type
- Every expression has a unique type
  - literals, variables, function arguments and results, operators, ...

Pros:

- Simple type checking

Cons:

- Difficult to write generic code
- Example (Pascal) : no way to write a generic sort procedure that sorts arrays of integers, reals, ...

Monomorphic Type Systems: Limitations

Each new type constructor requires

- Special syntax
- New type rules
- New internal representation (type formation)
- New code in type checker (intro, elim)
- Do another proof of soundness

Polymorphic Type System

- Polymorphism: an expression can have *more* than one type

```
fun length nil = 0
|   length(x :: xs) = 1 +(length xs)
```

- `length` function has type τ `list→int` for all types τ

Type Variables

- To formalize statements like:

  `length` function has type τ `list→int` for all types τ

  it is natural to introduce type variables such as α
- Type variable: placeholder where eventually will be substituted with a real type

  `length`: $\forall\alpha.(\alpha$ `list` $\rightarrow$ `int`$)$
- A type scheme is a type with universally quantified type variables
- Some instantiations of the `length` type scheme:
  - `int list` → `int`
  - `bool list` → `int`
- SML:

```
val 'a length = fn :  'a list -> int
```

Representing quantified types

Two new alternatives for `tyex`

```
datatype tyex
= TYCON of string
| CONAPP of tyex * tyex list
| FORALL of string list * tyex
| TYVAR of string
```

Kinding rules for quantified types

$$\frac{\alpha \in \mathit{dom}(\Delta)}{\Delta \vdash \mathrm{TYVAR}(\alpha) :: \Delta(\alpha)} \text{ (KindIntroVar)}$$

$$\frac{\Delta\{\alpha_1 :: *, \cdots, \alpha_n :: *\} \vdash *}{\Delta \vdash \mathrm{FORALL}(\langle \alpha_1, \cdots, \alpha_n \rangle, \tau) :: *} \text{ (KindAll)}$$

Summary

This Lecture

- Kinds, Polymorphic Type Systems

Next Lecture

- Typed μScheme