Adding Polymorphism and Exceptions

YSC3208: Programming Language Design & Implementation

Răzvan Voicu

Week 07, Feb 27 - Mar 3

Generic Code

- Generic code allow code to be reused in a wide range of scenarios.
- Examples:
  - `fun {..} x -> x end`
  - `fun {..} x y -> x end`
  - `fun {..} f x -> f x end`
- What types should they take?

Use Monomorphic Typing

- In monomorphic type, each type denotes only a *particular* type. Thus, when writing generic code, we have to duplicate its codes.
- An example is the identity function:
    - In case of `Int` :

      ```
      fun {Int->Int} x -> x end
      ```
    - In case of `Bool` :

      ```
      fun {Bool->Bool} x -> x end
      ```
    - In case of function `Int->Bool` :

      ```
      fun {((Int->Bool)->(Int->Bool)} x -> x end
      ```
- Problem : Monomorphic type causes code duplication!

Use Dynamic Typing

- One solution : "Drop types altogether".
- Similar to having a type, called `Any`, and having: $\forall t \cdot t <: \texttt{Any}$.
- Thus, earlier identity function could now be written as:
  `fun {Any} x -> x end`
- Similarly, two other polymorphic examples are:
  - `fun {Any} x y -> x end`
  - `fun {Any} f x -> f x end`
- Problem : Errors due to types no longer detected!

Use Dynamic Typing

- Examples of codes with type errors.
- `((fun {Any} x -> x end) true) > 4`
  This is equivalent to `true > 4` which is ill-typed since boolean and integer values are incomparable.
- `((fun {Any} f x -> f x end) 1 2)`
  This is equivalent to `(1 2)` which is ill-typed since 1 is not a function.

Use Dynamic Typing

- Slight improvement.
  - `fun {Any->Any} x -> x end`
  - `fun {Any->Any->Any} x y -> x end`
  - `fun {(Any->Any)->Any->Any} f x -> f x end`
- This typing can reject:
  `((fun {(Any->Any)->Any->Any} f x -> f x end) 1 2)`
- But not `((fun {Any} x -> x end) true) > 4`

Use Polymorphic Typing

- A better solution is to introduce type variables `'a`,`'b`.
- Write codes with polymorphic types:
  - `fun {'a->'a} x -> x end`
  - `fun {'a->'b->'a} x y -> x end`
  - `fun {('a->'b)->'a->'b} f x -> f x end`
- Benefit : Can reject ill-typed codes.
- Question : What exactly is polymorphic type?
- Challenge : Can we infer polymorphic types automatically?

Universally Quantified Types

- Polymorphic types are essentially universally quantified types.
- By default, we assume outermost quantification. Thus:
- `('a->'a)` denotes `(forall 'a. 'a->'a)`
- `('a->'b->'a)` denotes `(forall 'a,'b. 'a->'b->'a)`
- `('a->'b)->'a->'b` denotes
  `(forall 'a,'b. ('a->'b)->'a->'b)`
- This default quantification is referred to as *rank-1 polymorphism*.
- *Rank-0 polymorphism* does not have any quantifiers.
- Types with quantifiers inside arguments are referred to as *higher-ranked types*.

Limits of Rank-1 Polymorphism

- Some examples cannot be handled by Rank-1 polymorphism. An example is:

  `(fun {..} id -> if id false then 3 else id 4 end end)`

- Type-checking fails if we use: `(forall 'a. ('a->'a)->Int)`.
- Reason : Since `id` is used at two locations with different types, it needs to be polymorphic.

Rank-2 Polymorphism

- One solution is to use rank-2 polymorphism of the form: `((forall 'a. 'a->'a)->Int)`, where the quantifier appears in some parameter position, rather than just outermost.
- Using rank-2 type for example below:

  ```
  (fun {(forall 'a. 'a->'a)->Int} id ->
     if id false then 3 else id 4 end end)
  ```

  would allow the function-type parameter `id` to have different types at each of its two locations.

Let Polymorphism

- A simpler solution is to support polymorphism at just `let` construct.
- Assuming that argument of `id` is available, we can re-write the code to:

```
let id = fun {forall 'a. 'a -> 'a} x -> x end
in if id false then 3 else id 4 end
```

- Here, each let-bound variable is *universally quantified*, and could thus be given different types.
- Most strongly typed languages introduce polymorphism here.

Type Rules

- Type Inference for Rank-1 Let Polymorphism is decidable.
- Type Checking for Rank-2 Polymorphism is decidable.
- Complete inference on higher-ranked types is undecidable, but a mix of annotation and inference is possible.
- Type rules may be used for checking and/or inference.
- Type rules of form: $\Gamma \vdash e : t$

Type Rules (Syntax-Directed)

$$\frac{}{\Gamma \vdash \texttt{true} : bool}[Bool_1] \qquad \frac{}{\Gamma \vdash \texttt{false} : bool}[Bool_2]$$

$$\frac{}{\Gamma \vdash n : int}[Int] \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}[Var]$$

Type Rules (Syntax-Directed)

$$\frac{\Gamma[x \mapsto t_1] \vdash e : t_2}{\Gamma \vdash \texttt{fun}\ \{\cdot\}\ x\ \texttt{->}\ e\ \texttt{end} : t_1\texttt{->}t_2}\ [\mathit{Fun}]$$

$$\frac{\Gamma[x \mapsto t_1][f \mapsto t_1\texttt{->}t_2] \vdash e : t_2}{\Gamma \vdash \texttt{recfun}\ \{\cdot\}\ f\ \ x\ \texttt{->}\ e\ \texttt{end} : t_1\texttt{->}t_2}\ [\mathit{RecFun}]$$

Type Rules (Syntax-Directed)

$$\frac{\Gamma \vdash e_1 : t_1\texttt{->}t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}[Appln]$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[x \mapsto t_1] \vdash e_2 : t_2}{\Gamma \vdash \texttt{let x} = e_1\ \texttt{in}\ e_2\ \texttt{end} : t_2}[Let]$$

Type Rules (Structural)

$$\frac{\Gamma \vdash e : \texttt{forall 'a} \cdot t \quad \textit{fresh } \texttt{'b}}{\Gamma \vdash e : [\texttt{'a} \mapsto \texttt{'b}]t} [\textit{Inst}]$$

$$\frac{\Gamma \vdash e : t \quad \texttt{'a} \notin \textit{fv}(\Gamma)}{\Gamma \vdash e : \texttt{forall 'a} \cdot t} [\textit{Gen}]$$

Type Inference

Invented by Hindley-Milner.

Three key observations:

1. When should we perform the structural rules?
   - Introduce polymorphism at Let-construct
   - Instantiate at each variable occurrence
2. Use a fresh type variable if we are unsure of the type of some given subexpression.
3. Use unification as constraint-solving.

Inference Scheme

General Form:

$$\Gamma \vdash e \Longrightarrow t; \phi$$

Expression e under type environment Γ can be inferred to have type t that is valid under equational constraint ϕ.

An Example

$$\Gamma[\mathtt{y} \mapsto \mathtt{int}] \vdash \mathtt{fun}\,\mathtt{x}\;\mathtt{\text{->}}\;\mathtt{x\ y\ end} \Longrightarrow \mathtt{'a\text{->}'b};\ \mathtt{'a} = (\mathtt{int\text{->}'b})$$

Equational constraint allow us to conclude final type is `(int->'b)->'b`

Inference Rules

$$\frac{}{\Gamma \vdash \texttt{true} \Longrightarrow \mathit{bool}; \mathit{true}}$$

$$\frac{}{\Gamma \vdash \texttt{false} \Longrightarrow \mathit{bool}; \mathit{true}}$$

$$\frac{}{\Gamma \vdash n \Longrightarrow \mathit{int}; \mathit{true}}$$

$$\frac{t = \Gamma(x) \quad \mathit{inst}(t) \searrow t_2}{\Gamma \vdash x \Longrightarrow t_2; \mathit{true}}$$

Inference Rules

$$\frac{\textit{fresh}\ \texttt{'a} \quad \Gamma[x \mapsto \texttt{'a}] \vdash e \Longrightarrow t; \phi}{\Gamma \vdash \texttt{fun}\ \{\cdot\}\ x\ \texttt{->}\ e\ \texttt{end} \Longrightarrow \texttt{'a->}t; \phi}$$

$$\frac{\textit{fresh}\ \texttt{'a},\ \texttt{'f} \quad \Gamma[x \mapsto \texttt{'a}][f \mapsto \texttt{'f}] \vdash e \Longrightarrow t_2; \phi}{\Gamma \vdash \texttt{recfun}\ \{\cdot\}\ f\ x\ \texttt{->}\ e\ \texttt{end} \Longrightarrow \texttt{'a->}t_2; \phi \wedge (\texttt{'f} \Cup \texttt{'a->}t_2)}$$

Inference Rules

$$\frac{\Gamma \vdash e_1 \Longrightarrow t_1; \phi_1 \quad \Gamma \vdash e_2 \Longrightarrow t_2; \phi_2 \quad \textit{fresh}\ \texttt{'a}}{\Gamma \vdash e_1\ e_2 \Longrightarrow \texttt{'a}; \phi_1 \wedge \phi_2 \wedge (t_1 \Cup t_2 \texttt{->} \texttt{'a})}$$

$$\frac{\Gamma \vdash e_1 \Longrightarrow t_1; \phi_1 \quad \textit{gen}(\Gamma, t_1) \nearrow t_3 \quad \Gamma[x \mapsto t_3] \vdash e_2 \Longrightarrow t_2; \phi_2}{\Gamma \vdash \texttt{let x} = e_1\ \texttt{in}\ e_2\ \texttt{end} \Longrightarrow t_2; \phi_1 \wedge \phi_2}$$

Inference Rules (Instantiation)

$$\frac{\textit{fresh}\ \texttt{'b}_1 \ldots \texttt{'b}_n}{\textit{inst}(\texttt{forall}\ \texttt{'a}_1 \ldots \texttt{'a}_n \cdot t) \searrow [\texttt{'a}_1 \mapsto \texttt{'b}_1 \cdots \texttt{'a}_n \mapsto \texttt{'b}_n]t}$$

An Example:

$$\frac{\textit{fresh}\ \texttt{'n}_1, \texttt{'n}_2}{\textit{inst}(\texttt{forall}\ \texttt{'a,'b} \cdot (\texttt{'a->'b})\texttt{->'a->'b}) \searrow (\texttt{'n}_1\texttt{->'n}_2)\texttt{->'n}_1\texttt{->'n}_2}$$

Inference Rules (Generalization)

$$\frac{\{\texttt{'a}_1 \ldots \texttt{'a}_n\} = \mathit{fvars}(t) - \mathit{fvars}(\Gamma)}{\mathit{gen}(\Gamma, t) \nearrow \texttt{forall 'a}_1 \ldots \texttt{'a}_n \cdot t}$$

Unification

- We use *unification* to solve equational constraints.
- Given two types t_1 and t_2, we say that the two types can be *unified* if they can be made equal via a substitution ρ such that $\rho t_1 = \rho t_2$. This substitution is often referred to as a *unifier* of the two types.
- For example, given t_1=`(int->'b)` and t_2=`('a->'c)`, they can be unified since we have a substitution $\rho = [$`'a`$\mapsto$`int`, `'b`$\mapsto$`'c`$]$ to make them equal, as follows: $\rho t_1 = \rho t_2 =$`(int->'c)`.

Most General Unifier

- A unifier between two types, t_1 and t_2, is said to be the *most general unifier* (mgu) if all other unifiers are special instances of this one.
- A unifier ρ is said to be a special instance of ρ_g if there exists a substitution λ, such that $\rho = \rho_g o \lambda$.
- Example: $[\texttt{'a}\mapsto\texttt{int}, \texttt{'b}\mapsto\texttt{'c}, \texttt{'c}\mapsto\texttt{int}]$ is a special instance of $[\texttt{'a}\mapsto\texttt{int}, \texttt{'b}\mapsto\texttt{'c}]$

Unification Scheme

- We introduce the following unification scheme

$$\phi \Longrightarrow \rho$$

  where ϕ denotes $t_{1a} \Cup t_{1b} \wedge \cdots \wedge t_{na} \Cup t_{nb}$.
- ρ is either the most general unifier or it is $\perp$ when unification fails.
- Some simple examples:

$$\texttt{int} \Cup \texttt{bool} \Longrightarrow \perp$$

$$\texttt{int} \Cup \texttt{int} \Longrightarrow [\,]$$

$$\texttt{int} \Cup \texttt{'a} \Longrightarrow [\texttt{'a} \mapsto \texttt{int}]$$

Unification

$$\frac{\phi \Longrightarrow \rho}{\phi \ \wedge \ t \Cup t \Longrightarrow \rho}[\mathit{U\text{-}EQ}]$$

$$\frac{\phi \ \wedge \ t_1 \Cup t_3 \wedge t_2 \Cup t_4 \Longrightarrow \rho}{\phi \ \wedge \ t_1\texttt{->}t_2 \Cup t_3\texttt{->}t_4 \Longrightarrow \rho}[\mathit{U\text{-}ARR}]$$

Unification

$$\frac{\phi \wedge \texttt{'a} \Cup t \Longrightarrow \rho}{\phi \wedge t \Cup \texttt{'a} \Longrightarrow \rho}[\textit{U-COMM}]$$

$$\frac{\texttt{'a} \notin t \quad \rho_1 = [\texttt{'a} \mapsto t] \quad \rho_1\phi \Longrightarrow \rho}{\phi \wedge \texttt{'a} \Cup t \Longrightarrow \rho\rho_1}[\textit{U-SUBS}]$$

Unification

$$\frac{t_1 \neq t_2}{\phi \wedge t_1 \Cup t_2 \Longrightarrow \bot}\ [\mathit{U\text{-}NE}]$$

$$\frac{\texttt{'a} \in t \qquad \texttt{'a} \neq t}{\phi \wedge \texttt{'a} \Cup t \Longrightarrow \bot}\ [\mathit{U\text{-}INF}]$$

1. Polymorphism
2. Type Inference
3. Exception Handling
   - Motivation
   - Syntax of Exception Handling
   - Built-in Exceptions
   - Programmer-defined Exceptions
4. Denotational Semantics of simPL1

Motivation

Errors arise from

- Division by zero
- Failure to meet safety pre-condition
- Failure to find needed data structure (see later)
- ...

Handling Exceptions

```
try
   (evaluate input)
catch n with
   if n = 1
   then (evaluate (readNewUserInput))
   else ..
   end
end
```

For simplicity, we use exceptions that are distinguished by integer values, e.g. `throw 1` will raise the exception $\perp(1)$.

Syntax of Exception Handling

$$\frac{E_1 \qquad E_2}{\texttt{try } E_1 \texttt{ catch } n \texttt{ with } E_2 \texttt{ end}}$$

Built-in Exceptions

Division by zero leads to say $\perp(-1)$
Invalid pattern-matching leads to say $\perp(1)$

Examples:

```
4711 / 0
```

raised an exception, denoted by $\perp(-1)$

We allow programmer to throw Exceptions

$$\frac{E}{\texttt{throw } E}$$

but since we allow only integer-valued exceptions, we expect `E` to be of integer type.

Example

```
if percentage > 100
then throw 2
else ... end
```

This exception can then be caught by a surrounding expression and handled appropriately.

Semantic Domains for simPL1 = simPL + exceptions

Sem. domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	**Bool** + **Int** + **Exc** + **Fun**	expressible values
DV	**Bool** + **Int** + **Fun**	denotable values
Id	alphanumeric string	identifiers
Env	**Id** $\rightsquigarrow$ **DV**	environments
Fun	**DV** $\rightsquigarrow$ **EV**	function values
Exc	$\perp$(**Int**)	exceptions

Rules for simPL1

$$\frac{\Delta \Vdash E_1 \rightarrowtail e}{\Delta \Vdash E_1 + E_2 \rightarrowtail e} \text{ if } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v \qquad \Delta \Vdash E_2 \rightarrowtail e}{\Delta \Vdash E_1 + E_2 \rightarrowtail e} \text{ if } v \notin \textbf{Exc} \text{ and } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 + E_2 \rightarrowtail v_1 + v_2} \text{ if } v_1, v_2 \notin \textbf{Exc}$$

Rules for simPL1

$$\frac{\Delta \Vdash E_1 \rightarrowtail e}{\Delta \Vdash E_1/E_2 \rightarrowtail e} \quad \text{if } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v \qquad \Delta \Vdash E_2 \rightarrowtail e}{\Delta \Vdash E_1/E_2 \rightarrowtail e} \quad \text{if } v \notin \textbf{Exc} \text{ and } e \in \textbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1/E_2 \rightarrowtail v_1/v_2} \quad \text{if } v_1, v_2 \notin \textbf{Exc} \text{ and } v_2 \neq 0$$

Rules for simPL1

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail 0}{\Delta \Vdash E_1/E_2 \rightarrowtail e}$$

if $v_1 \notin \textbf{Exc}$ and
where $e = \bot(-1)$, and $e \in \textbf{Exc}$

Rules for simPL1

$$\frac{\Delta \Vdash E \rightarrowtail e}{\Delta \Vdash \texttt{throw}\ E \rightarrowtail e} \quad \textit{if}\ \ e \in \mathbf{Exc}$$

$$\frac{\Delta \Vdash E \rightarrowtail n}{\Delta \Vdash \texttt{throw}\ E \rightarrowtail \bot(n)}$$

Rules for simPL1

$$\frac{\Delta \Vdash E_1 \rightarrowtail v}{\Delta \Vdash \texttt{try}\ E_1\ \texttt{catch}\ n\ \texttt{with}\ E_2\ \texttt{end} \rightarrowtail v}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot(c) \quad \Delta[n \leftarrow c] \Vdash E_2 \rightarrowtail e}{\Delta \Vdash \texttt{try}\ E_1\ \texttt{catch}\ n\ \texttt{with}\ E_2\ \texttt{end} \rightarrowtail e}$$