FMFP - Lecture Notes Week 5

Ruben Schenk, ruben.schenk@inf.ethz.ch

April 1, 2022

1 Typing

1.1 Overview

Type checking should prevent "dangerous expressions", such as `2 + True`, `[2] : [3]`, etc. Dangerous expressions lead to *runtime errors.*

The objectives for a type checker are as follows:

- Quick, decidable, static analysis
- Permit as much generality / re-usability as possible
- Prevent runtime errors

1.2 Mini-Haskell

1.2.1 Syntax

Programs are **terms** (for now, let variables $\mathcal{V}$ and integers $\mathcal{Z}$ be given):

$$\begin{aligned} t \ ::= & \mathcal{V} \,|\, (\lambda x.t) \,|\, (t_1\, t_2) \,|\, \\ & True \,|\, False \,|\, (\text{iszero } t) \,|\, \\ & \mathcal{Z} \,|\, (t_1 + t_2) \,|\, (t_1 * t_2) \,|\, (\text{if } t_0 \text{ then } t_1 \text{ else } t_2) \,|\, \\ & (t_1,\, t_2) \,|\, (\text{fst } t) \,|\, (\text{snd } t) \end{aligned}$$

The core of Mini-Haskell is λ-calculus: variables, abstractions, and applications. Additional syntax and types can be easily added, e.g. `&&`, `Strings`, etc.

We employ some syntactic sugar, like omitting parenthesis (e.g. `x y z` instead of `((x y) z)`).

1.2.2 Typing

We consider **types**, given $\mathcal{V}_\tau$ is a set of variables like a, b, etc., such that

$$\tau \ ::= \ \mathcal{V}_\tau \,|\, Bool \,|\, Int \,|\, (\tau,\, \tau) \,|\, (\tau \to \tau)$$

The type system notation is based on **typing judgements** of the following form:

$$\Gamma \vdash t :: \tau,$$

where:

- Γ is a set of bindings $x_i : \tau_i$, mapping variables to types. Intuitively, Γ represents a kind of typing "symbol table".
- t is a *term*
- τ is a *type*

Example:

$$x : int \vdash x + 2 :: Int$$

$$x : Int,\ f : Bool \to Bool \nvdash f\,x :: Bool$$

1.2.3 Proof System

Proof rules are formulated in terms of type judgements J:

$$\frac{J_1 \quad \cdots \quad J-n}{J}$$

For example, one rule could be, given $op \in \{+, *\}$, the $BinOp$ rule:

$$\frac{\Gamma \vdash t_1 :: Int \quad \Gamma \vdash t_2 :: Int}{\Gamma \vdash (t_1 \, op \, t_2) :: Int}$$

1.2.4 Rules For Core λ-Calculus

We introduce the following rules for the core λ-calculus:

Axiom :

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau} \; \textit{Var}$$

Abstraction $(x \notin \Gamma)$:

$$\frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. \, t) :: \sigma \to \tau} \; \textit{Abs}$$

Application :

$$\frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \; t_2) :: \tau} \; \textit{App}$$

1.2.5 Further Typing Rules

Base types
$$\frac{}{\Gamma \vdash n :: \texttt{Int}}\,\textit{Int} \qquad \frac{}{\Gamma \vdash \texttt{True} :: \texttt{Bool}}\,\textit{True} \qquad \frac{}{\Gamma \vdash \texttt{False} :: \texttt{Bool}}\,\textit{False}$$

Operations ($\textbf{op} \in \{+, *\}$):
$$\frac{\Gamma \vdash t :: \texttt{Int}}{\Gamma \vdash (\textbf{iszero}\ t) :: \texttt{Bool}}\,\textit{iszero} \qquad \frac{\Gamma \vdash t_1 :: \texttt{Int} \qquad \Gamma \vdash t_2 :: \texttt{Int}}{\Gamma \vdash (t_1\ \textbf{op}\ t_2) :: \texttt{Int}}\,\textit{BinOp}$$

$$\frac{\Gamma \vdash t_0 :: \texttt{Bool} \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash (\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2) :: \tau}\,\textit{if}$$

Tuples
$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)}\,\textit{Tuple} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash (\textbf{fst}\ t) :: \tau_1}\,\textit{fst} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash (\textbf{snd}\ t) :: \tau_2}\,\textit{snd}$$

Example
$$\dfrac{\dfrac{\dfrac{}{x : \texttt{Int} \vdash x :: \texttt{Int}}\,\textit{Var} \qquad \dfrac{}{x : \texttt{Int} \vdash 2 :: \texttt{Int}}\,\textit{Int}}{x : \texttt{Int} \vdash x + 2 :: \texttt{Int}}\,\textit{BinOp}}{\vdash \lambda x.\, x + 2 :: \texttt{Int} \to \texttt{Int}}\,\textit{Abs}$$

1.3 Type Inference

Syntax-directed typing rules specify an algorithm for computing the type of expressions:

1. Start with judgement $\vdash t :: \tau_0$ with type variable τ_0.
2. Build the derivation tree bottom-up by applying the available rules. Introduce fresh type variables and collect constraints if needed.
3. Solve constraints to get possible types.

Example:

Type inference example

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x :: ((\tau_3 \to \texttt{Int}), \tau_4)}\,\textit{Var}}{\Gamma \vdash \textsf{fst}\ x :: \tau_3 \to \texttt{Int}}\,\textit{fst} \qquad \dfrac{\dfrac{}{\Gamma \vdash 2 :: \tau_5}\,\textit{Int} \quad \dfrac{}{\Gamma \vdash \texttt{True} :: \tau_6}\,\textit{True}}{\Gamma \vdash (2, \texttt{True}) :: \tau_3}\,\textit{Tuple}}{\Gamma \vdash (\textsf{fst}\ x)\ (2, \texttt{True}) :: \texttt{Int}}\,\textit{App}}{\dfrac{\overbrace{x : \tau_1}^{\Gamma=} \vdash \textsf{iszero}\ ((\textsf{fst}\ x)\ (2, \texttt{True})) :: \tau_2}{\vdash \lambda x.\,\textsf{iszero}\ ((\textsf{fst}\ x)\ (2, \texttt{True})) :: \tau_0}\,\textit{Abs}}\,\textit{iszero}
$$

Constraints:

$\tau_0 = \tau_1 \to \tau_2$

$\tau_2 = \texttt{Bool}$

$\tau_1 = ((\tau_3 \to \texttt{Int}), \tau_4)$

$\tau_3 = (\tau_5, \tau_6)$

$\tau_5 = \texttt{Int}$

$\tau_6 = \texttt{Bool}$

Most general type:

$\tau_0 = (((\texttt{Int}, \texttt{Bool}) \to \texttt{Int}), a) \to \texttt{Bool}$

1.4 Type Classes

1.4.1 Monomorphic vs. Polymorphic

We can distinguish between monomorphic and polymorphic functions. Some **monomorphic** functions:

```
xor x y = (x || y) && (not (x && y))

? :type xor
xor :: Bool -> Bool -> Bool
```

Others are **polymorphic:**

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

? :type (++)
(++) :: [a] -> [a] -> [a]
```

1.4.2 Type Classes - The Middle Way

Type classes allow for polymorphism to be restricted using class constraints. Example:

```
allEqual :: Eq a => a -> a -> a -> Bool
allEqual x y z = (x == y) && (y == z)
```

Functions for precisely those types a that belong to the **class** Eq. For example, the definition for the Eq class is given as follows:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x /= y = not (x == y)
```

The definition includes:

1. Class name: *Eq*
2. Signature: List of function names and types
3. Default implementations (optional): Can be overwritten later

Elements of a class are called **instances.** `instance` builds instances by "interpreting" signature functions:

```
instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False
```