CS-320

Computer Language Processing

Exercise Session 4

November 1, 2017
Overview

Today you will get some more practice in understanding and designing type systems:

- Exploring a typing derivation in Amyrli
- Amy’s pattern matching rule
- A type system for physical units
Recap: Type-checking a simple program

Consider the Amy-like language of arithmetic, logical connectives and if expressions from the lecture:

\[ t := \text{true} \mid \text{false} \mid c_l \mid f(t_1, \ldots, t_n) \mid \text{if} (t) \ t_1 \ \text{else} \ t_2 \]

where \( c_l \) denotes integer literals.

We also saw some of its typing rules, for instance:

\[
\text{If-Then-Else} \\
\Gamma \vdash b : \text{Bool} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash (\text{if} \ (b) \ t_1 \ \text{else} \ t_2) : \tau
\]
Finding a typing-derivation for a simple program

Exercise 1

Given the type system we saw for this language, type-check and show the typing derivations for the following program:

\[ p_{\text{fun}} = (e, \text{fun}(2)) \]

where \( e(\text{fun}) = (n, \text{Int}, \text{if } (n \leq 1) 1 \text{ else } n \ast \text{fun}(n), \text{Int}) \)
Recap: When do we say a program type-checks?

Given initial program \((e, t)\) define

\[
\Gamma_0 = \{ (f, \tau_1 \times \cdots \times \tau_n \rightarrow \tau_0) \mid (f, \_\_\_, (\tau_1, \ldots, \tau_n), t_f, \tau_0) \in e \}
\]

We say program type checks iff:

(1) the top-level expression type checks:

\[
\Gamma_0 \vdash t : \tau
\]

and

(2) each function body type checks:

\[
\Gamma_0 \oplus \{ (x_1, \tau_1), \ldots, (x_n, \tau_n) \} \vdash t_f : \tau_0
\]

for each \((f, (x_1, \ldots, x_n), (\tau_1, \ldots, \tau_n), t_f, \tau_0) \in e.\)
Finding a typing-derivation for a simple program

Exercise 1 (solution)

⇒ We have to check whether
a) $\Gamma_0 \vdash fun(2) : T$ for some type $T$, and
b) $\Gamma'_0 \vdash \text{if } (n \leq 1) 1 \text{ else } n \ast fun(n) : \text{Int}$

where $\Gamma_0 = \{\ldots(builtins), (fun, \text{Int} \Rightarrow \text{Int})\}$ and $\Gamma'_0 = \Gamma_0 \oplus \{(n, \text{Int})\}$.

Typing derivation for $\Gamma_0 \vdash fun(2) : T$:

$$
\frac{(fun, \text{Int} \Rightarrow \text{Int}) \in \Gamma_0 \quad \Gamma_0 \vdash 2 : \text{Int}}{\Gamma_0 \vdash fun : \text{Int} \Rightarrow \text{Int} \quad \Gamma_0 \vdash fun(2) : \text{Int}}
$$
Finding a typing-derivation for a simple program

Exercise 1 (solution)

Typing derivation for $\Gamma_0 \vdash \text{if } (n \leq 1) \ 1 \ \text{else} \ n \ast \text{fun}(n) : \text{Int}$ where $\Gamma_0' = \Gamma_0 \oplus \{(n, \text{Int})\}$:

\[
\begin{array}{c}
\Gamma_0' \vdash n \leq 1 : \text{Bool} \\
\Gamma_0' \vdash 1 : \text{Int} \\
\Gamma_0' \vdash n \ast \text{fun}(n) : \text{Int}
\end{array}
\]

\[
\Gamma_0' \vdash \text{if } (n \leq 1) \ 1 \ \text{else} \ n \ast \text{fun}(n) : \text{Int}
\]
Finding a typing-derivation for a simple program

Exercise 1 (solution)

Typing derivation for $\Gamma_0' \vdash n \leq 1 : Bool$.

\[
\begin{array}{c}
(\leq, (Int \times Int) \Rightarrow Bool) \in \Gamma_0' \\
\Gamma_0' \vdash \leq : (Int \times Int) \Rightarrow Bool \\
(n, Int) \in \Gamma_0' \\
\Gamma_0' \vdash n : Int \\
\Gamma_0' \vdash 1 : Int \\
\hline
\Gamma_0' \vdash n \leq 1 : Bool
\end{array}
\]
Finding a typing-derivation for a simple program

Exercise 1 (solution)

Typing derivation for $\Gamma_0 \vdash n \ast \text{fun}(n) : \text{Int}$.

\[
\begin{align*}
(\ast, (\text{Int} \times \text{Int}) \Rightarrow \text{Int}) & \in \Gamma'_0 \\
\Gamma'_0 \vdash \ast : (\text{Int} \times \text{Int}) \Rightarrow \text{Int} \\
(n, \text{Int}) & \in \Gamma'_0 \\
\Gamma'_0 \vdash n : \text{Int} \\
\end{align*}
\]

\[
\begin{align*}
(\text{fun}, \text{Int} \Rightarrow \text{Int}) & \in \Gamma'_0 \\
\Gamma'_0 \vdash \text{fun} : \text{Int} \Rightarrow \text{Int} \\
(n, \text{Int}) & \in \Gamma'_0 \\
\Gamma'_0 \vdash n : \text{Int} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma' \vdash \text{fun}(n) : \text{Int} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma'_0 \vdash n \ast \text{fun}(n) : \text{Int} \\
\end{align*}
\]
Finding a typing-derivation for a simple program

Exercise 1

▷ We have shown that the program type-checks, but did you notice any other problem with it?
Typing rules for pattern matching in Amy

We have seen a typing rule for if expressions, but how can we type more advanced control constructs like pattern matches? Let’s see a corresponding rule for the Amy language:

**Pattern Matching**

\[
\begin{align*}
\Gamma & \vdash e : T_s \\
\forall i \in [1, n]. & \quad \Gamma \vdash p_i : T_s \triangleright \Gamma_{p_i} \quad \Gamma \oplus \Gamma_{p_i} \vdash e_i : T_c \\\n\Gamma & \vdash e \text{ match } \{ \text{ case } p_1 \Rightarrow e_1 \ldots \text{ case } p_n \Rightarrow e_n \} : T_c
\end{align*}
\]

Note that we use auxiliary *extraction* judgments of the form

\[
\Gamma \vdash p : T \triangleright \Gamma_p
\]

to *check* that pattern \( p \) matches a type \( T \), while also *extracting* its bindings \( \Gamma_p \).
Typing rules for pattern matching in Amy

We have seen a typing rule for if expressions, but how can we type more advanced control constructs like pattern matches? Let’s see a corresponding rule for the Amy language:

\[
\begin{align*}
\text{Pattern Matching} & \quad \Gamma \vdash e : T_s \\
& \quad \forall i \in [1, n]. \quad \Gamma \vdash p_i : T_s \triangleright \Gamma_{p_i} \quad \Gamma \oplus \Gamma_{p_i} \vdash e_i : T_c \\
& \quad \Gamma \vdash e \text{ match } \{ \text{ case } p_1 \Rightarrow e_1 \ldots \text{ case } p_n \Rightarrow e_n \} : T_c
\end{align*}
\]

Note that we use auxiliary *extraction* judgments of the form

\[
\Gamma \vdash p : T \triangleright \Gamma_p
\]

to *check* that pattern \( p \) matches a type \( T \), while also *extracting* its bindings \( \Gamma_p \).
Typing rules for pattern matching in Amy

We have seen a typing rule for if expressions, but how can we type more advanced control constructs like pattern matches? Let’s see a corresponding rule for the Amy language:

**Pattern Matching**

\[
\begin{align*}
\Gamma & \vdash e : T_s \\
\forall i \in [1, n]. & \quad \Gamma \vdash p_i : T_s \triangleright \Gamma_{p_i} \quad \Gamma \triangleleft \Gamma_{p_i} \vdash e_i : T_c \\
\Gamma & \vdash e \text{ match } \{ \text{ case } p_1 = \rightarrow e_1 \ldots \text{ case } p_n = \rightarrow e_n \} : T_c
\end{align*}
\]

Note that we use auxiliary *extraction* judgments of the form

\[
\Gamma \vdash p : T \triangleright \Gamma_p
\]

to check that pattern \( p \) matches a type \( T \), while also *extracting* its bindings \( \Gamma_p \).
Typing rules for pattern matching in Amy

We have seen a typing rule for if expressions, but how can we type more advanced control constructs like pattern matches? Let’s see a corresponding rule for the Amy language:

\[
\begin{align*}
\Gamma &\vdash e : T_s \\
\forall i \in [1, n]. &\quad \Gamma \vdash p_i : T_s \triangleright \Gamma_{p_i} \\
\Gamma \oplus \Gamma_{p_i} &\vdash e_i : T_c
\end{align*}
\]

\[
\Gamma \vdash e \text{ match } \{ \text{ case } p_1 \Rightarrow e_1 \ldots \text{ case } p_n \Rightarrow e_n \} : T_c
\]

Note that we use auxiliary extraction judgments of the form

\[
\Gamma \vdash p : T \triangleright \Gamma_p
\]

to check that pattern \( p \) matches a type \( T \), while also extracting its bindings \( \Gamma_p \).
Typing rules for pattern matching in Amy (2)

We define the following extraction rules for patterns:

**Wildcard Pattern**
\[
\left\{\text{\texttt{\_}} : T \Rightarrow \emptyset\right\}
\]

**Identifier Pattern**
\[
\left\{\text{\texttt{\textit{v}}} : T \Rightarrow \{(\text{\textit{v}}, T)\}\right\}
\]

**Case Class Pattern**
\[
\begin{align*}
\Gamma & \vdash p_1 : T_1 \Rightarrow \Gamma_{p_1} \\
& \vdots \\
\Gamma & \vdash p_n : T_n \Rightarrow \Gamma_{p_n} \\
\Gamma & \vdash C : (T_1, \ldots, T_n) \Rightarrow T \\
\Gamma & \vdash C(p_1, \ldots, p_n) : T \Rightarrow \Gamma_{p_1} \oplus \cdots \oplus \Gamma_{p_n}
\end{align*}
\]
Type-checking pattern matching expressions

Exercise 2

▷ Find a typing derivation for the body of function \( \text{len} \) in the following program:

```scala
abstract class List
case class Nil() extends List
case class Cons(x: Int, xs: List) extends List

def len(xs: List): Int = xs match {
  case Nil() => 0
  case Cons(_, rest) => len(rest) + 1
}
```
Consider the following language of integral additions, multiplications, divisions:

\[
t := c_R \mid m \mid s \mid t + t \mid t \cdot t \mid t / t \mid \sqrt{t}
\]

\[
T := 1 \mid \text{meter} \mid \text{second} \mid T \ast T \mid T^{-1}
\]

where \(c_R\) denotes a real literal and \(m, s\) are used to introduce meters and seconds as units.

For instance:

\[
3 : 1 \quad 4 \cdot m : \text{meter} \quad 3 \cdot m/s : \text{meter} \ast \text{second}^{-1}
\]
A type-system for physical units

Exercise 3

Note that \((\text{meter} \times \text{second} \times \text{meter}^{-1})\) and \((\text{second})\) are not syntactically equivalent!

⇒ We will implicitly normalize our types and use a shorthand:

\[
\text{Dim } m \, n \equiv 1 \times \text{meter}^m \times \text{second}^n
\]

For instance:

\[
1 \equiv \text{Dim } 0 \, 0 \quad \text{meter} \equiv \text{Dim } 1 \, 0 \quad \text{second}^{-1} \times \text{meter} \equiv \text{Dim } 1 \, -1
\]

\[
\text{meter} \times (\text{second} \times \text{meter})^{-1} \times \text{second} \equiv \text{Dim } 0 \, 0
\]
A type-system for physical units

Exercise 3a

▷ Design typing rules that track the units of expressions and only permit adding expressions of the same unit. Furthermore, make sure that sqrt will only accept square meters.

▷ Write a function

\[ \text{dist} : (\text{meter} \times \text{second}^{-1} \times \text{second}) \Rightarrow \text{meter} \]

and show its typing derivation.
A type-system for physical units

Exercise 3a (solution)

\[
\begin{align*}
T-Lit & \quad T-Met & \quad T-Sec \\
\vdash c_R : 1 & \quad \vdash m : \text{meter} & \quad \vdash s : \text{second} \\
T-Add & \quad \vdash t_1 : \text{Dim } m \ n & \quad \vdash t_2 : \text{Dim } m \ n \\
& \quad \vdash t_1 + t_2 : \text{Dim } m \ n \\
T-Mul & \quad \vdash t_1 : \text{Dim } m_1 \ n_1 & \quad \vdash t_2 : \text{Dim } m_2 \ n_2 \\
& \quad \vdash t_1 \cdot t_2 : \text{Dim } (m_1 + m_2) \ (n_1 + n_2) \\
T-Div & \quad \vdash t_1 : \text{Dim } m_1 \ n_1 & \quad \vdash t_2 : \text{Dim } m_2 \ n_2 \\
& \quad \vdash t_1 / t_2 : \text{Dim } (m_1 - m_2) \ (n_1 - n_2) \\
T-WeirdSqrt & \quad \vdash t : \text{Dim } 2m \ 0 \\
& \quad \vdash \sqrt{t} : \text{Dim } m \ 0
\end{align*}
\]
Using your typing rules, find a typing derivation for the following (top-level) expression:

\[ \sqrt{m \cdot 4 \cdot m} + \frac{1}{s} \cdot 10 \cdot m \cdot s \]
A type-system for physical units

Exercise 3b

▷ Using your typing rules, find a typing derivation for the following (top-level) expression:

\[
\sqrt{((m \cdot 4) \cdot m) + (((1/s) \cdot 10) \cdot m) \cdot s}
\]