Typechecking

98-317: Hype for Types

Due: 29 January 2019 at 6:30 PM

Introduction

In class we introduced inference rules and how they're used to specify type systems. We then gave the specification of Lambda++, a language with abstractions, products, and sums; and we implemented a typechecker for Lambda++ based on this specification.

In this homework you will implement this typechecker yourself, based on the same inference rules. We've also included optional exercises which should help improve your understanding of type systems.

Turning in the Homework We'll only be collecting your solution to the "Required" section of this homework. We would be happy to separately discuss your solutions to the other sections of the homework with you if you'd like. Submit `LPPChecker.sml` to the "Typechecking" assignment on Autolab. Do not submit a tar for this assignment.

Required

As you'll recall from class, here is the syntax of Lambda++.

Type	τ	$::=$	α	base type
			$\tau \to \tau$	arrow type
			$\tau + \tau$	sum type
			$\tau \times \tau$	product type
Expression	e	$::=$	x	variable
			$\texttt{fn}\ (x : \tau) \Rightarrow e$	abstraction
			$e\ e$	application
			(e, e)	pair
			$\#1\ e$	left projection
			$\#2\ e$	right projection
			$\texttt{INL}\ e\ \texttt{into}\ \tau + \tau$	left injection
			$\texttt{INR}\ e\ \texttt{into}\ \tau + \tau$	right injection
			$\texttt{case}\ e\ \texttt{of}\ \texttt{INL}\ x \Rightarrow e\ \texttt{\|}\ \texttt{INR}\ x \Rightarrow e$	case analysis

Here are the static semantics of Lambda++. These semantics are identical to those we presented in class.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}\ (x : \tau_1) \Rightarrow e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e' : \tau_1 \quad \Gamma \vdash e : \tau_1 \to \tau_2}{\Gamma \vdash e\ e' : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#1\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#2\ e : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{INL}\ e\ \texttt{into}\ \tau_1 + \tau_2 : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{INR}\ e\ \texttt{into}\ \tau_1 + \tau_2 : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{INL}\ x_1 \Rightarrow e_1\ \texttt{|}\ \texttt{INR}\ x_2 \Rightarrow e_2 : \tau}$$

The syntax of Lambda++ implemented in the file `LambdaPlusPlus.sml`. The correspondence between the syntax and the implementation is fairly straightforward, but it's explicitly laid out in the following table just in case.

Types : `typ`

Syntax	Implementation	
α	`BaseType` "τ"	base type
$\tau_1 \to \tau_2$	`Arrow` (τ_1, τ_2)	arrow type
$\tau_1 + \tau_2$	`Plus` (τ_1, τ_2)	sum type
$\tau_1 \times \tau_2$	`Times` (τ_1, τ_2)	product type

Expressions : `exp`

Syntax	Implementation	
x	`Variable` "x"	variable
`fn` $(x : \tau) \Rightarrow e$	`Lambda` $((x, \tau), e)$	abstraction
$e_1\ e_2$	`Apply` (e_1, e_2)	application
(e_1, e_2)	`Tuple` (e_1, e_2)	pair
$\#1\ e$	`First` e	left projection
$\#2\ e$	`Second` e	right projection
`INL` e `into` $\tau_1 + \tau_2$	`Left` $(e, (\tau_1, \tau_2))$	left injection
`INR` e `into` $\tau_1 + \tau_2$	`Right` $(e, (\tau_1, \tau_2))$	right injection
`case` e `of INL` $x_1 \Rightarrow e_1$ `|` `INR` $x_2 \Rightarrow e_2$	`Case` $(e, (x_1, e_1), (x_2, e_2))$	case analysis

Required Task 1 In `LPPChecker.sml`, complete the typechecking function

```
check : LambdaPlusPlus.exp -> LambdaPlusPlus.typ
```

such that `check` e returns τ if $\vdash e : \tau$ is derivable under the given static semantics, and raises `TypeError` otherwise.

You can use the `Top.check` function to test your typechecker:

```
$ sml -m ./sources.cm
...
[New bindings added.]
- Top.check "fn (x : A) => x";
  A -> A
  val it = () : unit
- Top.check "fn (x : A) => fn (y : B) => x";
  A -> B -> A
  val it = () : unit
- Top.check "fn (x : A * B) => (#2 x, #1 x)";
  A * B -> B * A
  val it = () : unit
- Top.check "
  fn (x : A + B) =>
    case x of
      INL y => INR y into B + A
    | INR z => INL z into B + A
";
  A + B -> B + A
  val it = () : unit
- Top.check "fn (x : A + B) => #1 x";
  uncaught exception TypeError
    raised at: LPPChecker.sml:...
- Top.check "fn (x : A) => y";
  uncaught exception TypeError
    raised at: LPPChecker.sml:...
```

Useful (Not Required)

In this section, you'll be looking a simple language and answering some questions about how type checking will work in that language.

The syntax of this language is:

Type	τ	$::=$	`int`	integer
			`str`	string
Expression	e	$::=$	x	variable
			$\bar{n}$	integer literal
			$\bar{s}$	string literal
			$e_1 + e_2$	addition
			$e_1 \,\hat{}\, e_2$	concatenation
			$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	let-expression

We define the following static rules for it:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \bar{n} : \texttt{int}} \qquad \frac{}{\Gamma \vdash \bar{s} : \texttt{str}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 \,\hat{}\, e_2 : \texttt{str}} \qquad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

Namely:

1. Variables have the type that they have been assigned in the context.
2. Number and string literals are numbers and strings respectively.
3. An addition of numbers is a number. A concatenation of strings is a string.
4. A `let`-expression finds the type of e_1, then assuming x has that same type, finds the type of e_2. That type is then the type of the overall `let`-expression.

Useful Task 1 Give a derivation of the following claim:

$$(\texttt{let}\ x = 1 + (\texttt{let}\ y = \text{“98”}\ \texttt{in}\ 2)\ \texttt{in}\ 317 + x) : \texttt{int}$$

No need to be super formal, or even bother with typesetting the proof tree if it's too complicated. Just lay out which claims you're checking, and which rules you apply at every step.

Useful Task 2 The following expression is not well-typed:

$$\text{“vi”} \,\hat{}\, (\texttt{let}\ x = \text{“vijay”}\ \texttt{in}\ (\texttt{let}\ y = 2\ \texttt{in}\ x + y))$$

By inspection we can clearly tell that it is ill-formed. At which step (which rule) does an attempt at synthesizing the type fail? There might be more than one answer to this question, but try to give a justifiable one.

Useful Task 3 Suppose we added the following rule to the typing judgment:

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 + e_2 : \texttt{str}}$$

This would not be a very good idea. From the perspective of soundness (the type system making logical sense), why is it a bad idea? From the perspective of implementation (how we would actually go about checking and synthesizing types), how would this negatively impact us?

Useful Task 4 Suppose we deleted the rule for string literals:

$$\frac{}{\Gamma \vdash \bar{s} : \texttt{str}}$$

How would this affect the typechecking of the two examples in the first two required tasks? Would the output of a synthesis attempt be different than they were before?

Fun (Not Required)

Fun Task 1 The language we gave you contains binary sums and products. Not much effort is necessary to have it support n-ary sums and products, where essentially we would have a list for the type and the expression of sums and products. Try to rework the Lambda++ syntax to incorporate n-ary sums and products. If you alter the AST in `LambdaPlusPlus.sml` the parser will likely stop working, so it might be a good idea to remove it from compilation and test the ASTs directly if you choose to do this.

Rework the typing rules to support n-ary sums and products, and update the typechecker you've written to match this. (Make sure not to submit this reworked typechecker as your solution to the Required section!)