Typechecking

98-317: Hype for Types

Due: 24 January 2022 at 8:00 PM

Introduction

In class we introduced inference rules and how they're used to specify type systems. We then described a variant of the Simply Typed Lambda Calculus, called SimpleLC. It's a language with abstractions, products, and bools; and provided inference rules to derive the type of any expression in the language.

In this homework you will implement a typechecker for SimpleLC, 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. The "Non-Required" section will not be collected, but you're welcome to discuss it on Discord with us or other students. Run `make handin.zip` and submit `handin.zip` to the "Lambda Calculus" assignment on Gradescope.

Required

Required Task 1 Send an introduction to the `#introductions` channel on Discord! Tell us your grade, major, and why you decided to take Hype for Types. Also feel free to add anything else you'd like to share.

Required Task 2 Here is the syntax of SimpleLC, replacing $\lambda x : \tau.e$ with $\texttt{fn}\ (x : \tau) \Rightarrow e$.

Type τ	::=	`unit`	unit type
	|	`bool`	bool type
	|	$\tau_1 \times \tau_2$	product type
	|	$\tau_1 \to \tau_2$	function type
Expression e	::=	x	variable
	|	`()`	unit
	|	`false`	false boolean
	|	`true`	true boolean
	|	`if` e_1 `then` e_2 `else` e_3	boolean case analysis
	|	(e_1, e_2)	tuple
	|	$\texttt{fst}(e)$	first tuple element
	|	$\texttt{snd}(e)$	second tuple element
	|	$\texttt{fn}\ (x : \tau) \Rightarrow e$	function abstraction (lambda)
	|	$e_1\ e_2$	function application

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

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}\ (\textsc{unit}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ (\textsc{false})$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\textsc{true}) \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ (\textsc{if})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ (\textsc{tup}) \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}(e) : \tau_1}\ (\textsc{fst}) \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}(e) : \tau_2}\ (\textsc{snd})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}\ (x : \tau_1) \Rightarrow e : \tau_1 \to \tau_2}\ (\textsc{abs}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ (\textsc{app})$$

The syntax of SimpleLC implemented in the file `slc/SimpleLC.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
`unit`	`UnitTy`
`bool`	`Bool`
$\tau_1 \rightarrow \tau_2$	`Arrow` (τ_1, τ_2)
$\tau_1 \times \tau_2$	`Times` (τ_1, τ_2)

Expressions : `exp`

Syntax	Implementation
x	`Variable` "x"
`()`	`Unit`
`true`	`True`
`false`	`False`
`if` b `then` e_1 `else` e_2	`IfThenElse` (b, e_1, e_2)
`fn` $(x : \tau) \Rightarrow e$	`Lambda` $((x, \tau), e)$
$e_1\ e_2$	`Apply` (e_1, e_2)
(e_1, e_2)	`Tuple` (e_1, e_2)
`fst`(e)	`First` e
`snd`(e)	`Second` e

In `checker/SimpleLC_Checker.sml`, complete the typechecking function

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

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

Many of the cases are provided for you; you should edit the ones that currently raise `Fail`, using the inference rules above to guide you.

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

```
$ sml -m ./sources.cm
...
[New bindings added.]
- Top.check "fn (x : bool) => x";
bool -> bool
val it = () : unit
- Top.check "fn (x : bool) => fn (y : unit) => x";
bool -> unit -> bool
val it = () : unit
- Top.check "(fn (x : unit) => (x,x)) ()";
unit * unit
val it = () : unit
- Top.check "fn (x : bool) => if x then false else true";
bool -> bool
val it = () : unit
- Top.check "(fn (x : unit) => x) true";

uncaught exception TypeError
  raised at: checker/SimpleLC_Checker.sml:32.55-32.64
- Top.check "fn (x : unit) => y";

uncaught exception TypeError
  raised at: checker/SimpleLC_Checker.sml:23.27-23.36
```

Useful (Not Required)

Don't turn this in. If you want to talk about the answers to these questions, start a thread on Discord. 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
			`let` $x = e_1$ `in` e_2	let-expression

We define the following static rules for it:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash \bar{n} : \texttt{int}}\ (\textsc{int}) \qquad \frac{}{\Gamma \vdash \bar{s} : \texttt{str}}\ (\textsc{str})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ (\textsc{plus}) \qquad \frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 \hat{} e_2 : \texttt{str}}\ (\textsc{concat})$$

$$\frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}\ (\textsc{let})$$

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:

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

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:

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

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:

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

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 SimpleLC syntax to incorporate n-ary sums and products. If you alter the AST in `slc/SimpleLC.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!)