Homework 5
Check Your Types

98-317: Hype for Types

Due: 20 Februrary 2018 at 11:59 PM

1 Introduction

In class, we discussed two modes of performing type checking and synthesis: given an expression, and the inference rules constituting the definition of the typing judgment, we may confirm that a suspected type for the expression is correct or in some cases produce the type directly.

In both modes, we gave examples of how to perform a proof derivation of the type of an expression starting from the axioms. We showed how to handle variables and references to those variables in expressions.

We remarked that naive approaches at type synthesis may fail or be intractable. We also looked at how a type system can be made non-amenable to checking, including ambiguous/unsound rules, as well as rules that are so weak that we lose all meaning from checking. Finally, we discussed what we might do when type checking fails, such as: giving up, performing implicit casts, or helping the programmer with fixing/adding more types when necessary.

This homework is divided into four parts: Required, Useful, Fun, and Completely Unnecessary But Also Fun. You will receive credit for this homework if you turn in something (not necessarily something working) for the "required" portion.

Turning in the Homework You should submit any code files to Autolab by running the Makefile (type the command `make`) in the hw5 directory and submitting the resulting `hw05.tar` file to the Homework 5 assessment.

You should turn in your written solutions in class, as usual.

2 Required

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

The language from class, augmented with strings, 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:

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

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

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.

Req 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.

Req 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.

Req 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?

Req 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?

3 Useful

We'll now write an implementation of a typechecker. In the code files of the handout, you will find a partially implemented language, named Lambda++. It contains variables, functions, applications, and product and sum types, and we have devised a set of static typing judgments for it:

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

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

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

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

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

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

```
check :  Ctx.dict -> LambdaPlusPlus.exp -> LambdaPlusPlus.typ -> bool
```

Given a typing context (a mapping from variables to types), an expression, and a type, you should return true if the expression is well-typed and false otherwise.

Some Hints:

1. In this language, the base types are strings like "`int`" or "`str`". The SML version of the syntax is in `LambdaPlusPlus.sml`, and you can see

how types and expressions are actually represented. To test your code, remember that you are specifying the expression and an expected type, which can be any valid type you want. For example, one test case might be that `fn x => x` has type `int -> int`.

2. What are the cases that you need to check? Roughly one corresponds to each rule. What happens if the expression is a variable—what would we expect the type to be? What happens if the expression is a tuple—what subcomponents do we need to check first?

3. Make sure to insert into and lookup elements from the context appropriately. Every time you see a variable whose type you want to "remember", it should go into the context, and later you can go into the context to "recall" it. If you look at a rule, the fact that Γ is having some $x : \tau$ added to it in the premise, or removed from it in the conclusion, gives away what context operations you need.

4. The context is represented by the `Ctx` structure, which is essentially a map from variables names (strings) to types. See the comment in the file and the `cmlib` sources for more information.

5. If something does not typecheck, you should return false. That means that you can structure your code very nicely to reject branches that don't match against patterns corresponding to any rule.

4 Fun

Fun Task 1 Now that we have a checker for the Lambda++ language, we can try to synthesize types as well. It will be difficult to do so with the current syntax, since very few type annotations are given. For example, `fn x => x` is naturally polymorphic and we will not be able to give a concrete type for it (since we don't have universal types like SML).

What syntax elements need to be changed so that we have a shot at synthesizing the types? (Which operators need explicit type annotations?)

5 Completely Unnecessary but Also Fun

Unnecessary 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. The parser will likely stop working, so it might be a good idea to remove it from compilation and test the ASTs directly.

Unnecessary Task 2 Based on your answer in Fun Task 1, create a version of Lambda++ (Lambda+++????) that has enough annotations to be able to successfully synthesize the type.

Then, implement a function `synth : Ctx.dict -> LambdaPlusPlusPlus.exp -> LambdaPlusPlusPlus.typ option`, which returns `SOME t` if the expression has type `t` under the context, or `NONE` otherwise.

There are various ways of performing synthesis, the most naive of which would be to just do a search across each rule to see whether the structure matches. But a smarter way would be to start bottom-up, simply recursively synthesizing types for subcomponents and then combining them. You should add enough annotations such that every subexpression has a known type, so that such an approach is viable and polynomial-time. (Of course, you should check the annotations too!)