Homework 4: Hype for Sorts

98-317: Hype for Types

Due: 13 February 2018 at 11:59 PM

1 Introduction

In class, we discussed the idea of coming up with syntax that made it impossible to write down illegal programs. We quickly discovered that this was too restrictive, and prevented most interesting programs from being written. And so, we discovered why types are necessary! We then went on to learn how to implement parsers. On this homework, you'll be exploring syntax more and, if you choose to, writing part of a parser for a simple language.

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 turn in your written solutions in class as usual. Additionally, tar up any SML files that you've modified and submit them to the Homework 4 Autolab.

2 Required

Consider the following syntax:

```
Sort   Name       Operators
Var    x    ::=   x
Num    n    ::=   Zero
                  One
                  Plus(n, n)
Fun    f    ::=   Fun(x, n)
App    a    ::=   App(f, n)
```

This syntax is quite restrictive. For example, Plus can only have subtrees that are of sort Num - any ASTs of sort Fun, App, or Var are not allowed. Similarly, Fun's second subtree must be either Zero, One, or Plus(n, n). It may not be an AST of sort Var, Fun, or App.

Some examples of programs that use this syntax are as follows:

```
App(Fun(x, Zero), Zero)
Plus(Plus(Plus(Zero, One), One), One)
Fun(x, Plus(Plus(Zero, One), One))
```

Req Task 1 In SML, we can define the same kinds of expressions:

```
Our Syntax    SML
x             x
Zero          0
One           1
Plus(n, n)    n + n
Fun(x, n)     fn x => n
App(f, n)     f n
```

but SML allows us to write many more programs than this system does. Give an example of a program that you can write using the SML expressions written above that would not be allowed in our syntax.

Req Task 2 What is the arity of `App`?

Req Task 3 Draw the abstract syntax tree for `App(Fun(x, Zero), Zero)`. What is the sort of `App(Fun(x, Zero), Zero)`?

3 Useful

In this section, you will implement part of a parser for the Hype language. Hype consists of simple arithmetic operations, recursive, single-argument functions, and conditionals. We will give you the AST for Hype, and it is your job to come up with syntax for some elements of the language and write functions to parse that syntax.

3.1 Distribution Code

You will be parsing strings into an AST defined by the program datatype, described in the file `hype_abt.sml`. It is reproduced below.

```
type var = string

datatype typ = Num
             | Bool
             | Arrow of typ * typ

datatype exp = Plus of exp * exp
             | Minus of exp * exp
             | Less of exp * exp
             | True
             | False
             | Number of int
             | If of exp * exp * exp
             | Var of var
             | App of var * exp

datatype decl = Val of var * typ * exp
             |  Fun of var * (var * typ) * typ * exp

type program = decl list
```

Many valid programs can already be parsed. We have implemented parsers for numbers and arithmetic, comparisons, functions and applications, and types. However, Booleans and value declarations have been left unimplemented. Take a look at the code in the `HypeParser` structure to see this. We've used the SML Parcom library to implement this, as shown in class. You'll be using this library to implement your part of the parser.

Additionally, we've provided you with a typechecker and an interpreter for this language. You can access it by calling

```
Interpreter.repl()
```

which opens a REPL for Hype. You can type in declarations or expressions at the prompt and it will parse, typecheck, and evaluate them. Or you can call

```
Interpreter.runFile f
```

which parses, typechecks, and runs code from a file f.

3.2 Problems

Examples of already-implemented Hype syntax can be found in the file `examples.hyp`. However, this syntax is incomplete. The parser does not currently allow Boolean constants or value declarations. It's your job to decide on a syntax for these two things and to implement parsers for them.

Useful Problem 1 Decide on what syntax you want users of the Hype language to have to use when writing down the Boolean constants True and False. The syntax SML uses is `true` and `false`, but you can use whatever you'd like. Then, in `hype.sml` implement the parsers

```
val true_parser : (exp, t) parser
val false_parser : (exp, t) parser
```

That is, implement a parser that parses a string that contains your syntax for True and returns True and likewise for False.

Hint 1: There is a list of reserved names in the HypeDef structure. You can add your Boolean syntax to that and then use the `reserved <string>` parser.
Hint 2: Look at how `bool_parser` is implemented for inspiration.

Useful Problem 2 Decide what syntax you'd like to use for declaring variables. In SML, this is

```
val x : tau = e
```

but you can use any syntax you'd like. However, the syntax must contain 1) the name of the variable 2) the type of the variable and 3) the expression that the variable is equal to, so that you can parse it to be a value of the form Val (x, t, e).
Then implement the parser for value declarations:

```
val val_parser : (stmt, t) parser
```

That is, implement a parser that parses a string that contains your syntax for value declarations and returns `Val (x, tau, e)`.

Hint: Take a look at `fun_parser` and format your code similarly.

Useful Problem 3 Implement 3 values using your syntax (we'll use these to test your parser):

1. `Val ("x", Bool, True)`
2. `Val ("y", Bool, False)`
3. `Val ("z", Num, If (True, Number 1, Number 0))`

You should put them at the bottom of `examples.hyp`.

4 Fun

4.1 Semantics

The semantics of Hype are as follows:

- Plus(e1, e2) evaluates e1 and e2 and then adds together the results.
- Minus(e1, e2) evaluates e1 and e2 and then subtracts the results.
- Less(e1, e2) evaluates e1 and e2 and then returns True if e1 is less than e2 and False otherwise.
- If(e1, e2, e3) evaluates e1. If e1 is True, then it returns e2. Otherwise, it returns e3.
- Plus(e1, e2), Minus(e1, e2), and Number have type Num, assuming their subtrees have type Num.
- Less(e1, e2), True, and False have type Bool.
- If has whatever type e2 and e3 have.
- Val(x, t, e) binds an expression e of type t to a variable x.
- Fun (f, (x, t), tr, e) creates a function f with argument x of type t, return type tr, and body e.
- App(f, e) applies the function f to argument e.

For example, the program

```
[Fun (f, ("x", Num), Num, Plus(Var "x", Number 10)),
 Val ("y", Num, App("f", Number 1))]
```

declares a function f, which takes in a number and adds 10 to it, and a value y which equals 11. We could rewrite this in ML-like syntax as

```
fun f (x : num) : num = x + 10
val y : num = f(1)
```

though your syntax may differ for the declaration of y.

Fun Task 1 Implement a function in hype that multiplies a number by 10

```
num mult10 (n : num) = ...
```

Fun Task 2 Think about why you can't implement a function in Hype that multiplies two numbers together. How would you fix this problem?

Fun Task 3 Change up the syntax in the parser! Make all the syntax wacky, esoteric, and interesting. Or make it better and more usable.

5 Completely Unnecessary But Also Fun

Unnecessary Task 1 Currently, HYPE doesn't allow shadowing at all. This prevents a particular bug called *capture* from occurring in the HYPE interpreter. Another way of avoiding this bug is to use a locally nameless ABT. This removes all variable names and replaces them with number of binding sites ago they were initially bound (de Bruijn indices).

In a new file, `nameless.sml` translate the ABT into locally nameless form.

Unnecessary Task 2 Currently, HYPE is crippled by the fact that it only has single-argument functions. Rewrite the parser, typechecker, and interpreter to allow multi-argument functions.