Assignment 7

Type Checking & Subtyping

All Parts Due: 11:59 PM, Monday, March 28, 2016

This assignment gives you experience with type checking, subtyping, and bootstrapping.

Instructions

You should begin by copying the files from the hw7 directory in the usual way. See the course wiki for details.

All the usual policies apply. Work in pairs or alone, use subversion, and so forth. Remember as you code that your grade will depend not just on correctness, but on clarity (e.g., comments and the ability of the graders to easily understand how your code works) and elegance (e.g., factoring out common code rather than duplicating sections of code).

The Moses Language

Moses is a simple functional language, similar to Haskell (it is called “Moses” after Moses Schönfinkel, inventor of schönfinkelization, much as Haskell is named after Haskell Curry, inventor of currying). It closely matches the syntax we have seen in examples in class. Like Haskell, we can see Moses as a more useable incarnation of the λ-calculus.

Features of Moses include

- An expressive functional language, with familiar features such as λ-expressions, let expressions, binary operators, etc. See Figure 1

- A type system that supports subtyping, as well as tuples and lists. See Figures 2 and 3

Some of the most salient limitations of Moses compared to Haskell include

- Very limited pattern matching. We can break apart tuples in a let expression, but nested tuples will require nested lets.

- A quite different type system. Haskell allows type variables representing unknowns in types (e.g., id :: a → a), whereas Moses does not.

```mose
letrec fact = \n :: INT => if n == 0 then 1 else n * fact (n - 1)
in    fact 20
```

Figure 1: Factorial function in Moses
letrec sum = \l :: [REAL] => case l of [] -> 0
| n : ns -> n + sum ns
in sum [54.2, -29.3, 42.1, -25]

Figure 2: An example of lists and subtyping in Moses

letrec posneg = \l :: [INT] =>
case l of [] -> ([],[])
         | n : ns -> let (pos,neg) = posneg ns
           in if n >= 0 then (n:pos, neg)
            else (pos, n:neg)
in (posneg ([54, -29] @ [42, -25, -3]), "foo" ++ "bar")

Figure 3: An example of tuple creation and unpacking in Moses

- Whitespace-independent syntax that is more restrictive than Haskell's. You cannot
give multiple definitions inside a let or letrec, instead you must nest them.

- Moses uses different operators for list append and string concatenation. Moses uses
@ for list append (compared to ++ in Haskell), in Moses ++ is always string concate-
nation. (Strings in Moses are not lists of characters.)

More examples of Moses code can be found in the homework directory.

Running Moses Programs

The easiest way to get familiar with Moses is to run it. You can compile a command-line
version of the language by running

ghc -O Moses.hs
./Moses fact.moses
./Moses -e "42+3"
./Moses -i
Moses> 1+1
... descriptive output ...
2
Moses> quit

The first example runs code from a file, the second runs code given on the command line,
and the third starts an interactive session. You can also run :load Moses.hs in ghci, and
then use the run or runfile functions to execute code. The provided version of Moses is
incomplete, but you can use a fully operational version by running /cs/cs131/bin/Moses
instead of ./Moses.

The Moses system prints a lot of information about the steps of execution. It first
shows the parsed input (pretty printed), then it typechecks the code (well, it should, for
now, it doesn’t do much), then it converts all the nice constructs to vanilla lambda calculus, converts that to SK combinators, and then executes the program, and prints the result. Because it shows all the steps, you probably want to make sure you a terminal that lets you scroll back (not a problem if you’re using the lab Macs).

One key feature of the Moses system is that it uses the type of the expression to know how to print it out. It actually synthesizes printing code based on the type, thus an integer list (i.e., [INT]) is actually printed using a list printing higher-order function that is passed an int printer. Unfortunately, all but the very simplest programs don’t print any useful output on the command line, because the typechecker gives expressions almost everything the type ANY, which is always printed out as <???-who-knows-what-???>.¹

If you run things in ghci, you’ll find that programs that produce simple values (such as integers) are visible in the returned value from the run or runfile functions, allowing us to “cheat” and see the results even without type checking. Thus, fact.mose will produce its answer. More complex types, such as lists are not simple values and so you won’t be able to see their output printed until the type checker works.

If you want to see the full power of the printing code before the typechecker is complete, there are three special built-in values in Moses, example1, example2, and example3, of type [REAL], ([INT], (REAL,B00L), ([STRING]), and [INT] respectively. The synthesized printing code for example2 is quite complex (it’s quite a complex type!), but is actually generated following fairly simple rules. Be aware that example3 is an infinite list of the natural numbers—it will never finish printing.

**Structure of the Moses Language Implementation**

Many languages have a rich high-level language that people write programs in, and a lower-level language that is actually executed (e.g., in C and C++, the low-level language is typically machine code). As you saw in the previous section when you ran some Moses programs, Moses follows this pattern: the higher-level language is a richly typed language, whereas the lower-level language is far more basic “untyped” language.

In our Moses implementation,

- The untyped core is a simple combinator-based language. In addition to the combinators S, K and I, it does have a few built-in types and functions supporting integers, reals, and strings. Importantly, it does not and will not directly support booleans, lists, or pairs. (This kind of approach is a common one; for example, when C++ is compiled to X86 machine code, that code has no idea what C++ classes are.)

Similarly, the core does not directly support recursion (it uses the Y combinator instead). It also has no “operators” and no if...then...else, instead providing functions, such as _plus_ and _cond_ to serve the same purpose.²

¹Because Moses is lazy, and because there is nothing useful that can be done with an ANY value, the command-line version of Moses will never do any computation on anything that types as ANY. This property isn’t always true when using run or runfile in ghci, because they return the value produced by evaluation.

²The names have underscores to reduce the risk that a programmer might inadvertently reuse the name
The typed language provides lists, tuples, and a static type system, as well as niceties such as variables, let expressions, and so forth (the parser for the typed language also provides the familiar square-bracket list notation as syntactic sugar for colon notation as well as a few other niceties not actually present in the typed abstract syntax).

All programs in the typed language are translated into programs in the untyped core language. The only evaluator is the untyped–core-language evaluator.

The typed high-level language supports additional kinds of value beyond those directly supported by the low-level language—booleans, lists, and tuples are provided using the function-based (i.e., λ-calculus style) encodings we saw earlier in the semester. The untyped core language doesn’t know what these encodings are being used for, but the typed language does, because it has type information about the program.

Some of Moses is written in itself. For example, the @ operator is turned into a call to a function _append_ that appends two lists. There is nothing especially difficult about appending two lists, and so the code for _append_ is written in Moses. Other languages do similar things—much of Haskell is written in Haskell (including the list append function), and similarly C’s and C++ have standard libraries written in C and C++. The file prelude.mosedefs includes definitions for both private internal functions and publicly available functions, all written in Moses code.

**PHASES OF EXECUTION**

When Moses runs a program from a file (such as when you type runfile "tests/okay-value1.mose" or execute run "let x = 20 in x+x"), the following stages occur:

1. The system first loads code from prelude.mosedefs to provide the necessary encodings of booleans, lists, and pairs, as well as a variety of other support functions that are written in Moses itself. Some of the functions (e.g., _cons_ for lists) are internal functions (in code, users write x:y not _cons_ x y), whereas others are intended to be visible and usable. For this latter group of functions, the file provides type declarations describing the functions. No actual type checking happens here at all, by design. Because we say what the types are, we can define a function, and then say “don’t see this as a function, see it as a BOOL”. Of course, this does assume that when we tell Moses what the types are, we’re saying something meaningful.

2. The parser for the typed language reads in an expression from the desired file (such as okay-value1.mose). Any parse errors will abort with an error.

3. The expression is typechecked, by typeofExpr from Typecheck.hs. If typechecking is successful, we know the type of the expression. Any type-checking errors will abort with an error.

4. The expression is translated into an expression in the untyped lower-level core language by code in Transform.hs and SKRun.hs.

Note: This is the only file you will need to make changes to in the assignment. You may want to look at TypedAbsyn .hs too though.
5. The type of the expression is used to generate code to print a value of that type (much like you wrote printing code in the metaprogramming assignment, but easier because we don’t have user-defined types). This printing functionality is necessary because neither the core-language evaluator, nor Haskell know what our encodings mean. Something as simple as a pair of booleans will be encoded as a rather confusing-looking function (such as \(\lambda a (\lambda b. c) (\lambda d. e. d)\)), but we know that in this case, these functions represent a pair of booleans, and thus how to make a function that will print it so that it looks like a pair.³

6. The translated expression is evaluated by the core-language evaluator, resulting in a final value.

7. An expression is created that applies the “printer” code to the value from the previous step. This expression is evaluated by the core-language evaluator, thereby printing program’s result in human-readable form. (The potentially non–human-readable \(SKAbsyn.Value\) that represents the program’s result is returned by \(run\) and \(runfile\).

That’s the theory, anyway….

In practice, the typechecker is unfinished and so only typechecks very simple expressions. The biggest effect of this problem is the lack of a correct type for most kinds of expression, which means no printer for that kind of expression. The type checker operates correctly on \(okay-value1.mose\) through \(okay-value4.mose\), and \(error-var1.mose\). For the others, the lack of good typechecking results in late catching of errors and no good print function.⁴

**Your Task, Part 1: Subtyping**

Moses has a different type system from Haskell; it has subtyping and parametric types (e.g., lists of int and lists of string), but does not have parametric polymorphism (i.e., type variables). In addition, to make types stand out, Moses adopts the convention of writing type names in upper case in the concrete syntax.⁵ Moses has the basic types \(\text{INT}\), \(\text{REAL}\), \(\text{STRING}\), and \(\text{BOOL}\), as well as tuples, such as \((\text{INT}, \text{BOOL})\) or \((\text{BOOL}, \text{STRING}, \text{INT})\). It also has homogenous lists, such as \(\text{[INT]}\) or \(\text{[(STRING, BOOL)]}\). There are also function types, such as \(\text{INT} \rightarrow \text{STRING}\). Finally, it has two special types, ANY and NONE.

A value belonging to the ANY type is a value that is a member of the universal set—thus, if you have an ANY value, it could be anything. Thus all types are subtypes of the

---

³When we know that this function is encoding a pair of booleans, we can determine that it holds the pair (false, true), but the same representation could also encode other values for other types, such as the pair of (false,[])—without type information, we have no hope of meaningfully printing data encoded inside an arbitrary function.

⁴In some cases, you can cheat and read the result of evaluation from the untyped value that \(runfile\) returns to the Haskell interactive loop, but try that with \(okay6.mose\).

⁵In our abstract-syntax representation of types, which is provided by the type \(Ty\) in the \(TypedAbsyn\) module, we represent types such as \(\text{INT}\) and \((\text{REAL, INT LIST})\) as \(\text{RealType}\) and \(\text{TupleType}\) \([\text{RealType}, \text{ListType(IntType)}]\), respectively.
ANY type. An ANY value is so underspecified that we can’t do anything useful with it. But the ANY type does have a few uses—for example, a function that ignores its argument and returns 42 is best described as having type ANY \rightarrow INT since it doesn’t care what its argument is. Think “a useless ANY value”.

A value belonging to the NONE type is a value that is a member of the empty set—thus, if you have a NONE value, you have something pretty miraculous, because there is clearly no such thing. Because no NONE values actually exist, every NONE that does exist (all none of them) can be used as anything you want. Thus, NONE is a subtype of all types because a mythical NONE value can do anything. Anyone who promises you an actual NONE value is lying, but NONEs do have their uses—for example, the empty list has type NONE LIST because, being empty, it doesn’t actually contain any NONEs. Similarly, the error function has type STRING \rightarrow NONE because it prints the error-message string and terminates the program—because the function never returns it can make the false promise that it will return something miraculous. Think “a mythical NONE value”.

The subtyping rules for Moses are as follows:

\[\begin{align*}
\text{ty} \subseteq \text{ty} & \\
\text{t}_1 \subseteq \text{t}_2 & \quad [\text{ty}_1] \subseteq [\text{ty}_2] \\
\text{NONE} \subseteq \text{ty} & \\
\text{t}_1 \subseteq \text{t}_3 & \quad \text{t}_2 \subseteq \text{t}_4 & \quad (\text{ty}_1, \text{ty}_2) \subseteq (\text{ty}_3, \text{ty}_4) \\
\text{ty} \subseteq \text{ANY} & \\
\text{ty}_3 \subseteq \text{ty}_1 & \quad \text{ty}_2 \subseteq \text{ty}_4 & \quad \text{ty}_1 \rightarrow \text{ty}_2 \subseteq \text{ty}_3 \rightarrow \text{ty}_4
\end{align*}\]

I’ve only shown the rule for pairs, but larger tuples are defined analogously.

To provide support for subtyping, the Typecheck module provides the functions, isSubtype, commonSupertype, and commonSubtype. Unfortunately the code for these functions is unfinished. You must finish them.

- isSubtype \(x\ t\ y\) returns true when \(x \subseteq y\).
- commonSupertype \(x\ t\ y\) returns a type \(t\) such that
  \[
  x \subseteq t \land y \subseteq t \land (\forall t' \neq t : t' \subseteq t \land x \subseteq t' \land y \subseteq t')
  \]
  In other words, it returns the “least upper bound” of \(x\) and \(y\). Such a \(t\) always exists—in the worst case, everything falls under the supertype ANY.

- commonSubtype \(x\ t\ y\) returns a type \(t\) such that
  \[
  t \subseteq x \land t \subseteq y \land (\forall t' \neq t : t \subseteq t' \land t' \subseteq x \land t' \subseteq y)
  \]
  In other words, it returns the “greatest lower bound” of \(x\) and \(y\). Such a \(t\) always exists—in the worst case, everything is above the subtype NONE.
When you write `isSubtype`, the “backwardness” of the subtyping rule for function arguments will be fully in your mind (it is, after all, in the subtyping-inference-rule functions given above). But similar “backwardness” applies when writing `commonSupertype`—there is a reason I asked you to define `commonSubtype` as well (they’ll end up mutually recursive!).

As a reminder, the full abstract syntax for types (defined in `TypedAbsyn.hs`) is

```haskell
data Ty = IntTy
      | RealTy
      | StringTy
      | BoolTy
      | AnyTy
      | NoneTy
      | FunTy Ty Ty Ty  -- ... -> ...
      | TupleTy [Ty]  -- (... , ..., ...)
      | ListTy Ty  -- [...]
```

TIP: When I was doing this part myself, I found the following Haskell list functions quite handy: `zip`, `all`, and `map`.

**Your Task, Part 2: Type Checking**

The function `typeofExpr` checks that an expression is well typed and returns its type. Again, this function is unfinished. Adapt the type rules from class to finish the typechecker. You should always return the “tightest” type reasonably possible—`ANY` might be a valid type for everything but it isn’t very useful. (Also, remember that the mythical `NONE` value is allowed wherever you need any kind of value.)

As you write rules for the different expression cases, you’ll find yourself wanting to do the many of the same things each time. Rather than copy and paste code, try to make your life easier by abstracting out common functionality into helper functions.

Proceed as follows:

1. Typecheck primitive values (Exactly …). Actually, that’s done already. Yay.
2. Typecheck variables by looking them up in the type environment (Var …). Actually, that’s done already, too.
3. Typecheck tuples (Tuple …). This one is quite straightforward; with `TupleTy` and Haskell’s `mapM` function, you can be done in moments. (See the hints and tips page if the monad-ness of things spins your head a little.)
4. Typecheck λ-expressions. With `Map.insert` and a recursive call, it’s looking good.
5. Typecheck simple (non-tuple) let-expressions. Not much different from λ-expressions
6. Typecheck conditionals. We want a `BOOL` for the condition. `typeofExprBounded` is likely to be handy here. For the result type, `commonSupertype` will also be useful.
NOTE: Because Haskell is lazy, if we weren’t using our Fallible monad, it would never actually run your checks on the boolean condition (because the result type does not depend on this calculation). But thanks to our use of a monad, we can ensure the sequencing we desire.

7. Typecheck the list cons operator (:). We single this one out for special treatment because, unlike all the other operators, its arguments have different types. Once again, commonSupertype is your friend (e.g., 1 : ["Foo"] is legal, and has type [ANY]), as is typeofExprBounded to make sure the right-hand side is a list.

8. Typecheck case expressions. This is code is similar to the code for simple lets and the code for if statements. Again, commonSupertype is your friend.

9. Typecheck tuple-match let-expressions. The rhs needs to be a tuple of the same size as the number of variables. You might find the Haskell functions zip and foldl handy when extending the type environment.

10. Typecheck all the remaining the binary operators. There are a lot, and some are similar, some different. The partial code gives a hint as to one way to handle the constraints and checking. Note that the relational operators are overloaded, they either take two REALs, or they take two STRINGs—the Fallible type is a member of the Alternative typeclass, and thus the <> operator (a.k.a. the mplus function) may be useful when handling this either/or situation.

11. Typecheck function application. The first expression needs to be “some kind of function” (typeofExprBounded is your friend!), and the second expression needs to be compatible with the function argument.

12. Typecheck letrec. When typechecking the definition, use NONE as the type of the variable. Then in the body, use the type you actually calculated when you type checked the definition.

Recursive Declarations, Revisited

Our rule for recursive declarations “works,” but has a subtle bug. It types the following declaration as INT -> INT:

```
letrec silly = \n :: INT => if n == 42 then 54 else silly
in silly
```

This type isn’t really correct, because if silly has type INT -> INT, then the type of the if statement is the common supertype of INT and INT -> INT, which is ANY, thus the real type of the expression ought to have been INT -> ANY.

The easiest “fix” is relatively simple and simply involves “doing things twice”. But you need to understand how the original version worked and why it failed. What matters in this case is that you produce a type that is not obviously wrong.
Extra Credit

In Haskell, functions like map have type \((a \to b) \to [a] \to [b]\), but Moses does not have type variables (a.k.a. parametric polymorphism). But, just as we have for operators, we could add special type rules for a few select functions. Develop a clean and general way to check functions like map, foldl, foldr, all, takeWhile, and id as special cases. In particular, the “official type” of the functions remains limited; it is only when they are applied to an argument that we cheat and figure out a more general type.