Assignment 6

Type Checking & Subtyping

Code Due: 11:59 pm, Wednesday, November 16, 2005

> This assignment gives you experience with type checking, subtyping, and bootstrapping. The result will be a language with ML-like syntax, lazy evaluation, and subtype polymorphism rather than ML's parametric polymorphism. In many ways, the resulting system is more like a compiler than a traditional interpreter, because it performs both type analysis and code generation.

Instructions

This assignment extends the Mini-ML interpreter you've seen in class and in Labs 7 and 8. You should begin by copying the files in `/cs/cs131/src/ass6`, which contains the latest version of the Mini-ML system.[1]

All the usual policies apply. Submit using the submit system, and 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).

Structure of the Mini-ML System

As you learned in Lab 8, 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). In this assignment, we increase the distinctions between these languages by making the higher-level language a *typed* language, and the lower-level language *untyped*.

In Lab 8 and this assignment, both languages are variants of "mini-ML", where

- The untyped core language is essentially mini-ML restricted down to little more than the lambda calculus, although 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.

  In Lab 8, you saw some of the other ways the core language is simpler. It does not directly support recursion (it uses the `Y` combinator). It also has no "operators" and no **if**...**then**...**else**, instead providing functions, such as `_plus_`, `_true_`, and `_false_` to serve the same purpose.[2] It does have **let**...**in**...**end**, largely for readability.

1. The code you saw in Lab 8 was essentially the code for this assignment with all of the typechecking code ripped out.

2. The names have underscores to reduce the risk that a programmer might inadvertently reuse the name of a vital function.

- The typed language extends our previous mini-ML interpreter with lists, pairs, and a static type system (the parser for the typed language also provides the familiar square-bracket list notation as syntactic sugar for colon-colon notation).

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

Recall that the typed high-level language supports additional kinds of value beyond those directly supported by the low-level language—booleans, lists, and pairs are provided using the function-based (i.e., lambda-calculus style) encodings we saw in class and lab. 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.

Running Code

When Mini-ML runs a program from a file (such as when you run `Run.run "okay1.mml"` or `Run.run "okay2.mml"`), the following stages occur:

1. The system first loads code from `preamble.mml` to provide the necessary encodings of booleans, lists, and pairs, as well as a variety of other support functions that are written in Mini-ML itself. This process proceeds as follows:

   (a) The parser for the typed language reads in the list of declarations (using `Parser.parseDecls` in `parser.sml` and the language syntax defined in both `mini-ml.lex` and `mini-ml.grm`).

   (b) These declarations are transformed into declarations in the untyped lower-level core language (by `Transform.transDecls` in `transform.sml`).

   (c) The declarations are evaluated by the core-language evaluator (by `Eval.declare` in `eval.sig` and `eval.sml`), and added to the initial environment for the core-language evaluator (which is `Eval.builtins`).

   The result of this stage is that we have an environment for the core-language evaluator that contains all the support functions we need. Observe two things about the preamble code: First, the code was written in the richer, high-level language. Second, we never performed any actual type checking on this mini-ML support code—we only perform type checking on user code, in part because some of our support functions (such as the `Y` combinator) can't be given a good type in our types system, and aren't directly called by user code anyway, so don't need to be given an explicit type.

   Most of these support functions are never referred to *directly* in user code. The few functions that are exposed have their types are specified by hand in `preamble.specs` (which is parsed with a call to `Parser.parseSpecs`).

2. The parser for the typed language reads in an expression from the desired file (such as `okay2.mml`) using `Parser.parseExpr`. Any parse errors will abort `run` with an exception.

3. The expression is typechecked, by Typecheck.typeofExpr in `typecheck.sml`. If typechecking is successful, we know the type of the expression. Any type-checking errors will abort run with an exception.

4. The expression is translated into an expression in the untyped lower-level core language by Transform.transExpr.

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 Standard ML know what our encodings mean. Something as simple as a pair of booleans will be *encoded* as a rather confusing-looking function (such as λa.a (λb.λc.c) (λd.λe.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 *UntypedAbsyn.value* that represents the program's result is returned by Run.run).

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. Tests `okay1.mml`, `okay2.mml`, `okay3.mml`, and `error1.mml` do typecheck correctly. For the others, the lack of good typechecking results in late catching of errors and no good print function.[4]

Also, the parser doesn't understand **case** statements, so it fails to parse `okay8.mml` and `okay11.mml`–`okay14.mml`.

1 Subtyping

Mini-ML has subtyping and parametric types, but does not have parametric polymorphism. In addition, to make types stand out, Mini-ML adopts the convention of writ-

3. When 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,nil)—without type information, we have no hope of meaningfully printing data encoded inside an arbitrary function.

4. In many cases, you can cheat and read the result of evaluation from the untyped value that Run.run returns to the Standard ML interactive loop, but try that with `okay6.mml`.

ing type names in upper case in the concrete syntax.[5] Mini-ML has the basic types INT, REAL, STRING, and BOOL, as well as pairs, such as INT * BOOL or (BOOL * STRING) * INT. It also has homogenous lists, such as INT LIST or (STRING * BOOL) LIST. There are also function types, such as INT -> 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 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 -> 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 -> 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 MiniML are as follows:

$$\frac{}{ty \sqsubseteq ty} \qquad \frac{t_1 \sqsubseteq ty_2}{ty_1\ \texttt{LIST} \sqsubseteq ty_2\ \texttt{LIST}}$$

$$\frac{}{\texttt{NONE} \sqsubseteq ty} \qquad \frac{t_1 \sqsubseteq ty_3 \qquad ty_2 \sqsubseteq ty_4}{ty_1 \times ty_2 \sqsubseteq ty_3 \times ty_4}$$

$$\frac{}{ty \sqsubseteq \texttt{ANY}}$$

$$\frac{}{\texttt{INT} \sqsubseteq \texttt{REAL}} \qquad \frac{ty_3 \sqsubseteq ty_1 \qquad ty_2 \sqsubseteq ty_4}{ty_1 \rightarrow ty_2 \sqsubseteq ty_3 \rightarrow ty_4}$$

To provide support for subtyping, the Typecheck structure (in `typecheck.sml`) provides the functions, isSubtype, commonSupertype, and commonSubtype. Unfortunately the code for these functions is unfinished. *You* must finish them.

5. In our abstract-syntax representation of types, which is provided by the type *ty* in the TypedAbsyn structure (in `typedabsyn.sml`), we represent types such as INT and REAL * INT LIST as RealType and PairType(RealType, ListType(IntType)), respectively.

- isSubtype(x,y) returns true when $x \sqsubseteq y$.
- commonSupertype(x,y) returns a type t such that

  $$x \sqsubseteq t \ \wedge \ y \sqsubseteq t \ \wedge \ (\nexists\, t' \neq t : t' \sqsubseteq t \ \wedge \ x \sqsubseteq t' \ \wedge \ y \sqsubseteq t')$$

  In other words, it returns the "most specific generalization" or "least upper bound" of x and y. Such a t always exists—in the worst case, everything falls under the supertype ANY.
- commonSubtype(x,y) returns a type t such that

  $$t \sqsubseteq x \ \wedge \ t \sqsubseteq y \ \wedge \ (\nexists\, t' \neq t : t \sqsubseteq t' \ \wedge \ t' \sqsubseteq x \ \wedge \ t' \sqsubseteq y)$$

  In other words, it returns the "most general specialization" or "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 on the previous page). But similar "backwardness" applies when writing commonSupertype—there is a reason I asked you to define commonSubtype as well and made the two of them mutually recursive!

2 Type Checking

The function Typecheck.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. Your rule for **if** *must* use commonSupertype. Typecheck.typeofExpr 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. The tests `none-*.mml` provide examples.

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.

Check your work by running some of the test cases.

2.1 Recursive Declarations

The rule for recursive declarations "works", but has a subtle bug. It types the following declaration as `INT -> INT`:

```
val rec silly = fn n : INT => if false then 7 else silly
```

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*. If you're not sure, you can leave the code for recursive declarations alone and come back to it later.

3 Implementing case

The language we saw in class had **case** statements, and the test cases `okay8.mml` and `okay11.mml`–`okay14.mml` use such a **case** statement, but Mini-ML does not (yet) support them. You will need to remedy this deficiency.

Your abstract syntax and grammar rules for **case** should be *very rigid and literal*. The only valid **case** statement is one with exactly two arms, where the **nil** case is the first arm and the second arm has a pattern of the form v_1 :: v_2—more complex patterns are *not* supported. The parser, and only the parser, should reject syntactically invalid **case** statements.

1. Extend the typed abstract syntax to include **case** statements. Also extend the pretty-printing code in `expr2frags`. (You may limit yourself to using `Frags` and `Piece`, though, if you don't want to fiddle with making "pretty" output. Look at the other pretty-printing code and `easy-pp.sml` for guidance.)

2. Extend the grammar description to add the tokens `CASE`, `OF`, and `BAR`.

3. Extend the lexer to recognize **case**, **of**, and the vertical bar character **|**.

4. Extend the parser to parse **case** statements.

5. Extend the translator to turn a **case** statement into a suitable low-level code. The representation of lists used by the interpreter (specified in `preamble.mml`) corresponds to the last method covered in Lab 8 (the one most people didn't get to do). In this representation, the code

   case L **of**
   nil => M
   | h :: t => N

   should translate to $L\ (M)\ (\lambda h.\lambda t.N)$.[6]

6. Extend the typechecker to typecheck **case** statements (the code should be much like your code for **if**).

Check your work by running `okay8.mml` and `okay11.mml`–`okay14.mml`. If you want to check your **case** implementation before you've written the typechecking rules for **case** (or perhaps even before you've written *any* typechecking rules), you can use `Run.untypedrun` instead of `Run.run`.

6. If you don't like the idea of hard-wiring the code to a particular representation of lists, you could instead output code to call a builtin function _case_, with suitable arguments, and devise code for this _case_ function and add it to `preamble.mml`.

4 Extra Credit: Using Types in Compilation

As the language stands now, equality is limited to values that the low-level core language understands, (i.e., strings, integers, and reals). Equality on boolean values is not supported, nor is equality on complex types such as pairs or lists. In the same way that we construct a print function for an arbitrary and potentially complex type, we could also construct "the right equality test function". In fact, `TraversalCodeGenerator.tyEqualsFunc` takes a type and returns an expression that can be used as an equality-test function on values of that type. Your mission in this extra-credit portion of the assignment, should you choose to accept it, is to let people write "x `=` y", and have Mini-Ml "do the right thing" and generate a call to a suitably constructed equality-test function.

The difficulty will be in placing this type-specific equality-test code in the right place—we need to put it where we would otherwise have put `_equals_`, but that turns out to be a bit of a challenge. The problem is that the call to `_equals_` is inserted by the `RelOp` case of `transExpr`—a function that has no type information to hand for us to use. In contrast, our value-printer code was easier to integrate, because we just applied the function to the user-supplied expression (i.e., at the outermost level), and we had the type of the entire expression available from performing typechecking.

One trick for solving this problem is to add a mutable annotation to the high-level abstract syntax for `RelOp`s, specifically a $\cdots ? \cdots$ *option ref* (you'll have to decide what the question-mark part should be). When the abstract syntax tree is created, there is no annotation and so the initial value of the annotation is **ref** `NONE` (that's Standard ML's `NONE` value from the *option* type, not Mini-ML's `NONE` type). Add code to the `RelOp` case of `typeofExpr` to assign `SOME` $\cdots ? \cdots$ to this reference to store information that you can later use during the translation phase of the `RelOp`. Extend the `RelOp` case of `transExpr` to use this information and "do the right thing".

Once you have equality working, you might want to add support for the other relational operators, too. Try to avoid duplicating too much code.