Assignment 6

# Type Checking & Subtyping

**Code Due:** 5 pm, Wednesday, November 24, 2004

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 a 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 6 and 8. You should begin by copying the files in `/cs/cs131/src/ass6`, which contain 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 and typeful language that people write programs in, and a lower-level, untyped language that is actually executed (e.g., in C and C++, the untyped language is typically machine code).

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, strings, and bools. Importantly, it *does not* and *will not* directly support lists or pairs.

  The core language also does not directly support recursion. Recursion is achieved through the use of the `Y` combinator. The core language also has no "operators" and no **if**...**then**...**else**, but it provides functions, such as `_plus_` and `_cond_` that can serve the same purpose.[2] It does have **let**...**in**...**end**, largely for readability.

- 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).

1. Do *not* use the code from Lab 8.
2. The names have underscores to reduce the risk that a programmer might inadvertently reuse the name of a vital function.

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 language implements lists and pairs using features that exist in the core language—specifically, functions. It uses function-based *encodings* that mirror the lambda calculus encodings we saw in class. The typed language "knows what it has encoded", whereas the untyped core language does not know and does not care.

## Running Code

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

1. The system first loads some additional code from `preamble.mml` to provide the necessary encodings of lists, pairs, etc. The process is 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 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: First, the code was written in the richer, typed language. Second, we never performed any actual type checking on this code.

   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 Run.initialTenv.

2. The parser for the typed language reads in an expression from the desired file (such as `okay2.mml`) using Parser.parseExpr.

3. The expression is typechecked, by Typecheck.typeofExpr in `typecheck.sml`. If typechecking is successful, we know the type of the expression.

4. The expression is translated into an expression in the untyped core language by Transform.transDecls.

5. The type of the expression is used to generate code to print a value of that type. 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 will be *encoded* as a rather confusing-looking anonymous function, but *we know* that it is a pair, and thus how to make a function that'll print it so that it looks like a pair.

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.[3] This expression is evaluated by the core-language evaluator, resulting in a string value. This string is then printed.

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`.

# 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 writing type names in upper case. Mini-ML has the basic types `INT`, `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`.

The subtyping rules for MiniML are as follows:

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

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

$$\frac{t_1 \sqsubseteq ty_2}{ty_1\ \texttt{LIST} \sqsubseteq ty_2\ \texttt{LIST}} \qquad \frac{t_3 \sqsubseteq ty_1 \qquad ty_2 \sqsubseteq ty_4}{ty_1 \rightarrow ty_2 \sqsubseteq ty_3 \rightarrow ty_4}$$

The `ANY` and `NONE` types are actually surprisingly useless. There are *no values* that belong to the `NONE` type. A function of type `NONE -> INT` can thus never be called because you can never have a suitable argument value to pass it. Similarly, a value of type `ANY` could be literally anything. Because it could be anything, there is nothing that you can actually do with it, other than pass it around. The `NONE` type is used for situations where there is no value to worry about (e.g., the `error` function never returns, and so has type `STRING -> NONE`—because it never returns, it can claim that "when it

3. Because the code doesn't actually do the printing, but just makes a string, the implementation actually calls it a "stringizer".

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`.

does return, it'll give you the mythical value of type NONE", similarly nil has type NONE LIST—the fact that there can be no values of type NONE doesn't matter because there aren't any in the (empty) list). The ANY type is useful if the value is irrelevant (e.g., the length function on lists has type `ANY LIST -> INT`—it doesn't care what's in the list).

The abstract syntax for types is provided by the type *ty* in `typedabsyn.sml`.

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.*

- 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 is 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 is 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 typechecks an expression 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.

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.

Check your work by running some of the test cases.

## 3 Implementing case

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

1. Extend the typed abstract syntax to include **case** statements, including 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 for guidance.)

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

3. Extend the parser to parse **case** statements. Your grammar rules should be *very rigid and literal*. The only valid **case** statement is one that lists **nil** as the first arm. The parser, and only the parser, should reject syntactically invalid **case** statements.

4. Extend the translator to turn a **case** statement into a suitable combination of calls to `_isempty_`, `_head_`, `_tail_`, and `_cond_`. Remember that the core-language abstract syntax does support **let**...**in**...**end**, which may be convenient.

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

Check your work by running `okay8.mml`.

## Bonus: Implementing REAL

If you're eager for more, you can implement `REAL`, too, where `INT` $\sqsubseteq$ `REAL`. If you're interested, send email for details...