# Assignment #4:
# Polymorphic Type Inference

Michael Sullivan, Nathan Harmata, for 15-312: Concepts of Programming Languages

Out: Thursday, March 17
Due:
Theory: Thursday, March 24 1:30pm
Implementation: Thursday, March 31 3 11:59pm

## 1 Introduction

### 1.1 Some Advice

You may find this assignment much more involved and challenging than previous homeworks. Be sure to start early. The implementation portion will be a relatively large amount of work but will (hopefully) be very rewarding when completed. We hope you enjoy this assignment!

As an organizational note, Section 2.1 is a stand-alone discussion with questions on equivalences. The remainder of this assignment deals with type inference.

The implementation portion of this assignment deals with a language for which we have provided you with a parser[1]. The parser works using `ml-yacc` and `ml-lex` and as such you will need to have them installed on your machine. In order to ensure compatibility, we recommend testing on Andrew UNIX.

### 1.2 Submission

Solutions to the theory questions should be handed in in class on Thursday, March 24.
Submit solutions to the programming problems by 11:59 PM, Thursday, March 31. by placing your `*.sml` files in your hand-in directory

`/afs/andrew.cmu.edu/course/15/312-sp11/handin/<yourandrewid>/assn4/`

You are not required to typeset the written portion of the assignment, but it is *highly encouraged* . If you instead choose to handwrite your solutions, please try to be as legible as possible. Turn in your solutions at the beginning of class on the due date.

## 2 Theory

### 2.1 Extensional Equivalence

We've devoted several lectures to a rather detailed look at equivalences. We had this discussion with the language Gödel's T in mind. This question focuses on *extensional equivalence*. Refer to your notes and/or

[1] Details about the parser are provided in the appendix

PFPL Ch 51.2 for details (especially notation).

Recall in lecture we proved that extensional equivalence is *reflexive* [2] . Formally,

**Reflexivity**

If $\vdash e : \tau$

then $e \sim e : \tau$

We then claimed that extensional equivalence is also symmetric and transitive. You will now prove this claim (for closed terms under the empty context). As always, develop clear proofs that explicitly show the reasoning and rules used. You will want to prove symmetry and transitivity properties via "structural induction" on types[3]. This is for two reasons. First, it's because we defined extensional equivalence inductively on the types. Second, the proofs become relatively straightforward when done in this manner.

**T**ask 2.1 (6%). Give a formal statement of the symmetry of extensional equivalence and then prove the statement.

**T**ask 2.2 (10%). Give a formal statement of the transitivity of extensional equivalence and then prove the statement.

[2] Also recall that this result gave us a slick proof of termination of all Gödel's T programs

[3] You might be concerned here since have shunned structural induction in the past in favor of rule induction. This isn't an issue here because extensional equivalence is actually defined inductively on the structure of types!

## 2.2 Motivation for Type Inference

Type inference is convenient feature of modern programming languages that allows the user to omit explicit type annotations and instead have them be inferred by the compiler. We already have good intuition of the convenience and use of type inference in strongly typed functional languages like SML; more recently languages like C# have begun to incorporate type inference. The languages we've studied so far in 15-312 have required their programs to have explicit type annotations everywhere (think the type annotation of a lambda argument), even in places where it can be "annoying" at times (think the type annotation of a left or right injection into a sum). In this assignment, we will develop polymorphic type inference for a small (but hopefully representative) subset of ML to which we will refer as MinML.

## 2.3 MinML

MinML is a small fragment of core ML which contains some of the constructs you would expect. Its abstract syntax [4] is given by the following grammar.

$$
\begin{array}{rcl}
e & ::= & x \\
& | & \lambda x.e \\
& | & e\,e \\
& | & \mathsf{fix}(x.e) \\
& | & \mathsf{let}(e, x.e) \\
& | & \langle\rangle \\
& | & \langle e, e\rangle \\
& | & \mathsf{fst}(e) \\
& | & \mathsf{snd}(e) \\
& | & \mathsf{inl}(e) \\
& | & \mathsf{inr}(e) \\
& | & \mathsf{case}(e, x.e, x.e) \\
& | & \mathsf{roll}_{\mathsf{list}}(e) \\
& | & \mathsf{unroll}_{\mathsf{list}}(e)
\end{array}
$$

Little in this language should look particularly surprising, modulo minor syntactic differences. The two unusual constructs are are $\mathsf{roll}_{\mathsf{list}}(e)$ and $\mathsf{unroll}_{\mathsf{list}}(e)$. For simplicity, we do not provide full recursive types. Instead, we provide a specialized type for one particularly useful recursive type: lists (natural numbers can then be viewed being of type unit list). Rules for these constructs in the context of the typed language can be found the in appendix.

Note that this language has no explicit types! Programs can be given types through the process of type inference.

## 2.4 Type Inference

For our purposes, type inference on a MinML program is the process of determining both its type and annotating the program with this type. In the presence of polymorphism, we actually want to make type instantiations à la System-F where appropriate. We will accomplish both these goals by simultaneously inferring a program's type and elaborating the program to an explicitly polymorphically typed version of MinML, to which we will refer as XMinML. The abstract syntax of XMinML is given by

[4] See the appendix for details on the concrete syntax of this language

$$
\begin{array}{rcl}
\tau & ::= & \alpha \\
& | & \forall \alpha.\tau \\
& | & \tau \to \tau \\
& | & \mathsf{unit} \\
& | & \tau \times \tau \\
& | & \tau + \tau \\
& | & \tau\ \mathsf{list} \\
\\
e & ::= & x \\
& | & \lambda x{:}\tau.e \\
& | & e\ e \\
& | & \Lambda \alpha.e \\
& | & e[\tau] \\
& | & \mathsf{fix}(x{:}\tau.e) \\
& | & \mathsf{let}(e, x.e) \\
& | & \langle\rangle \\
& | & \langle e, e \rangle \\
& | & \mathsf{fst}(e) \\
& | & \mathsf{snd}(e) \\
& | & \mathsf{inl}^{\tau}(e) \\
& | & \mathsf{inr}^{\tau}(e) \\
& | & \mathsf{case}(e, x.e, x.e) \\
& | & \mathsf{roll}_{\tau\ \mathsf{list}}(e) \\
& | & \mathsf{unroll}_{\mathsf{list}}(e)
\end{array}
$$

Before we discuss type inference, we need to have a discussion about "unification."

## 2.5 Unification

Unification is the process of constructing a set of substitutions that make two terms equal[5]. Unification is a general procedure that comes up in many different places, so we'll abstract away from the specific language under consideration. We will do this by discussing *operators*, *variables*, and *constants*[6]. For example, we'll write the term $\mathsf{f}(x, \mathsf{b})$ to mean the operator $f$ applied to the variable $x$ and the term constant $\mathsf{b}$.

So, $\mathsf{f}(x, \mathsf{b})$ is equal to $\mathsf{f}(\mathsf{c}, \mathsf{b})$ under the set of substitutions $\{x \to \mathsf{c}\}$. Note that unification isn't necessarily unique; consider the task of unifying $\mathsf{f}(x, \mathsf{b})$ with $\mathsf{f}(y, z)$. Clearly we need to have the substitution $\{z \to \mathsf{b}\}$ but we have (infinitely) many choices for how to unify $x$ and $y$; we could make the substitution $\{x \to y\}$, or we could instead perform the set of substitutions $\{x \to u, y \to u\}$, for any fresh variable $u$. Also note that unification fails on terms that are simply unequal, such as $\mathsf{f}(x, \mathsf{b})$ and $\mathsf{f}(x, \mathsf{c})$.

Let's work out a couple more examples. Can $x$ unify with $x$? Yes - they already are the same! But what about trying to unify $x$ with $\mathsf{f}(x)$? Think about this for a moment. The answer is that unification cannot happen in this case. To see why, suppose for the sake of argument that $x$ is able to be unified with $\mathsf{f}(x)$. Then, there must exist some set of substitutions under which this unification happens. Since there is only one variable under discussion and $x$ and $\mathsf{f}(x)$ aren't already equal, this substitution must be of the form

[5] We mean alpha equivalent here.

[6] This is sufficiently expressive for our purpose. For example, we can define an operator $p$ such that $p(x, y)$ constructs the MinML expression $\langle x, y \rangle$

$\{x \to e\}$, for some term $e$. But, performing this substitution on $x$ and $\mathtt{f}(x)$ we get $e$ and $\mathtt{f}(e)$, which aren't equal since $f$ is a non trivial operator. In practice, an unification algorithm actually never gets this far. When implemented naively, such an algorithm won't terminate on this example because it will keep trying to unify $x$ with larger and larger super-expressions of $x$. For example, we might naively try to unify $x$ with $\mathtt{f}(x)$ using the substitution $\{x \to \mathtt{f}(x)\}$ because this seems to make the terms equal. But, working this out, we see that $\mathtt{f}(x)$ becomes $\mathtt{f}(\mathtt{f}(x))$ which becomes $\mathtt{f}(\mathtt{f}(\mathtt{f}(x)))$, and so on. Thinking deeper, we see that the true issue with trying to unify $x$ and $\mathtt{f}(x)$ is that there is no way for a variable to unify with a larger term that has that variable as a subexpression. More succinctly, $x$ can unify with $x$ trivially (see previous example) but it cannot unify with $\mathtt{f}(x)$, where $\mathtt{f}(x)$ is a non identity operator, since $x$ occurs in $f(x)$. Because of this, when performing unification, we need to perform the appropriately named "occurs check" to ensure that we don't run into these issues.

To summarize,

- A variable trivially unifies with itself.
- Unequal terms can never unify.
- Since the result of unification is a substitution, multiple occurrences of the same variable need to unify to the same term.
- We need to perform the "occurs check" when unifying a variable with a larger term.

Terminology note: the terms "existential variable" and "unification variable" and used interchangeably in this document.

There are two general approaches to unification: the unification context approach, and the imperative approach. Many of you are familiar with the former from 15-212; we will focus on the imperative approach in this assignment, but we will give first give a brief description of the unification context approach.

### 2.5.1 A Theoretical Perspective

Since the goal of unification is to produce a set of substitutions that under which two terms unify, one way we can do so is by constructing this set piece by piece by explicitly tracking the substitutions that need to be made and applying them when necessary. The advantages of the scheme are that the algorithm (which we have glossed over) is quite elegant and fairly easy to specify.

### 2.5.2 A More Efficient Approach

While having the output of unification be a substitution is a clean approach, in practice it has a number of downsides. Carrying the substitution around can be cumbersome and repeated applications of the substitution may be inefficient.

A more practical approach, employed by some production compilers, is to instead use an *imperative* unification algorithm. In imperative unification, we dispense with producing a substitution in favor of making unification variables *mutable*.

Unification variables then become reference cells that contain either a free variable identifier or some type that the variable has been unified with.

As a concrete example, the datatype for types in the system you will be implementing is:

```
datatype typ =
       ...
       | TEVar of evar
     and bind =
         Free of int
       | Unified of typ
withtype evar = bind ref
```

When a fresh unification variable is needed, we create a new ref cell containing `Free k` where k is an integer unique across unification variables.

Then, when it comes time to unify a unification variable with another type, the variable's reference is simply updated with that type, updating all copies of that variable in-place in one fell swoop.

There is one snag, though. When operating on types, we must account for the possibility that the type is a unification variable pointing to another type. Worse, there could be a long chain of unification variables that must be followed before finding a concrete type. To deal with this, whenever we case analyze a type that may contain unification variables, we "simplify" it by following the chain of unification variables until reaching a concrete type. As part of simplification, we also perform *path compression* by updating every unification variable in a chain to point to the eventual destination. [7]

## 2.6 A Note on Variables

When doing polymorphic type checking, there are two flavors [8] that are in play. There are unification variables, which are introduced at typechecking time and unified away. They are an auxiliary notion and not part of the actual type system being considered. There are then normal type variables (which will be referred to in this document simply as "type variables") which are part of the type system of the language. Type variables are bound by $\forall$ and $\Lambda$ and are used for polymorphism.

It is important to keep these concepts separate. In fact, the core of polymorphic is generalization, which is the process by which we determine which unification variables can be converted into type variables.

## 2.7 Type Inference Algorithm

With all the preceding machinery in place, we can now have a formal discussion of type inference that nicely lends itself to implementation. The key idea of the algorithm is that whenever we want to know a type that is not immediately obvious, we introduce a unification variable in place of the type. Then, whenever we would compare types for equality in a normal typechecking algorithm, we instead perform unification. If it succeeds, unification will potentially constrain some of the unification variables, leading to a more specific type.

We assume some notion/implementation of unification and we write $\tau_1 \doteq \tau_2$ to mean "$\tau_1$ unifies with $\tau_2$". We introduce a new judgment for type inference:

$$\Gamma \vdash e \Rightarrow \tau$$

[7] This path compression is the same path compression performed in the union find data structure. In fact, this unification algorithm is implicitly using union find. If we wanted to, we could also perform union-by-rank, for an even better asymptotic speed up. We don't really want to.

[8] Flavor is one of the few unreserved words in math for describing "flavors" of things: type, kind, set, sort, category, and variety are all taken.

pronounced "$e$ is inferred to have $\tau$ under $\Gamma$". Here, $e$ is an expression in (the implicitly typed) MinML and $\tau$ is its corresponding type in the explicitly typed language. We start with some of the simpler rules

$$\frac{}{\Gamma \vdash \langle\rangle \Rightarrow \mathsf{unit}} \text{ infer-unit} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \Rightarrow \tau_1 \times \tau_2} \text{ infer-pair}$$

Now, consider performing type inference on a lambda expression of the form $\lambda x.e$. We don't know the type of the bound variable $x$, but what we can do is infer the type of $e$ under the assumption that $x : A$ for some fresh unification type variable $A$.

$$\frac{A \,\mathsf{fresh} \quad \Gamma, x : A \vdash e \Rightarrow \tau}{\Gamma \vdash \lambda x.e \Rightarrow A \to \tau} \text{ infer-lam}$$

Let's turn to inferring the type of $e_1\, e_2$. Our first (incorrect) attempt at a rule for this might look like the familiar typing rule for application

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 \Rightarrow \tau_1}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau_2} \text{ infer-app-WRONG}$$

but this won't be sufficient. If type inference on $e_1$ produced a unification variable as its result, it will not be of the form $\tau_1 \to \tau_2$. Now, from inferring the type of $e_2$, we know the argument type of the function. Thus, we create a new unification variable to stand in for the result type of the function and we unify:

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash e_2 \Rightarrow \tau_2 \quad A \,\mathsf{fresh} \quad \tau_1 \doteq \tau_2 \to A}{\Gamma \vdash e_1\, e_2 \Rightarrow A} \text{ infer-app}$$

And fix:

$$\frac{A \,\mathsf{fresh} \quad \Gamma, x : A \vdash e \Rightarrow \tau \quad A \doteq \tau}{\Gamma \vdash \mathsf{fix}(x.e) \Rightarrow \tau} \text{ infer-fix}$$

Note the use of unification in rule `infer-fix` to ensure that the inferred type of the body of the `fix` expression is consistent with its assumed type. Now that we have some good intuition, the rest of the rules for type inference are fairly straightforward.

**T**ask 2.3 (9%). Give the rules for performing type inference on case, inl, and $\mathsf{roll}_{\mathsf{list}}$ expressions. You should only need to develop one rule for each expression.

You will also want to develop the other rules anyway as you will need them for the implementation portion of this assignment.

## 2.8 Let Polymorphism

"Let polymorphism" is the flavor of polymorphic type systems employed by core ML and also in our MinML language. In this system, polymorphism is only introduced at let bindings and is eliminated by introducing type application where variables are used. This results in polymorphic types where all of the $\forall$s appear at the top level. This is known as the *prenex restriction*. Furthermore, polymorphic types may only be instantiated at concrete types. This is in contrast to the polymorphic systems discussed in lecture, where any $\Lambda$ expression could be type instantiated with any type, including $\forall$ types.

### 2.8.1 Generalization

Generalization is the heart of let polymorphism. Types produced during typechecking may still have unification variables in them. To generalize a type $\tau$ underneath a context $\Gamma$, we perform the following procedure:

1. Find all unification variables in $\tau$ that are not mentioned in any of the types contained in the context $\Gamma$. Call them $X_1, \cdots X_n$.
2. Generate new type variables $\alpha_1, \cdots, \alpha_n$ and substitute them in for the unification variables, yielding $\tau' = [\alpha_1, \cdots, \alpha_n / X_1, \cdots, X_n]\tau$
3. Bind the new type variables underneath foralls, yielding $\mathsf{Generalize}(\Gamma, \tau) = \forall\alpha_1 \cdots \forall\alpha_n.\tau'$.

There is the restriction that we only generalize type variables that do not appear in the context because those represent actual constraints on $\tau$; some other term also mentions that unification variable and would not appreciate having it generalized. By only generalizing unification variables that are not mentioned in the context, we ensure that the only unification variables that are generalized are those that were introduced in the process of inferring $\tau$.

Generalization comes into play whenever we see a let binding. At a let binding, we infer the type of let bound term, generalize it, and then put the generalized type in the context for typechecking the body of the let. Or, in inference rule form:

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \mathsf{Generalize}(\Gamma, \tau_1) = \tau_1' \quad \Gamma,\, x : \tau_1' \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash \mathsf{let}(e_1, x.e_2) \Rightarrow \tau_2} \text{ infer-let}$$

### 2.8.2 Specialization

Now that we have figured out how to introduce polymorphism into our language, we also need to figure out how to *use* it.

Since all polymorphic generalization occurs at let bindings, we know that the only place where we can encounter a polymorphic term is at a variable. When we encounter a variable polymorphic type, we need to instantiated it at some particular type. While we might not yet know what type to instantiate it at (it might even end up being another polymorphic type), we have a good way to deal with this: use fresh unification variables.

Our rule for performing type inference on variables then becomes:

$$\frac{\Gamma(x) = \forall\alpha_1 \cdots \forall\alpha_n.\tau \quad E_1\ \mathsf{fresh} \cdots E_n\ \mathsf{fresh}}{\Gamma \vdash x \Rightarrow [E_1 \cdots E_n / \alpha_1 \cdots \alpha_n]\tau} \text{ infer-var}$$

## 2.9 Elaboration

As mentioned earlier, our goal is to develop polymorphic type inference for MinML *and* also a translation between MinML and XMinML. We accomplished the former in the previous sections. The latter is known as "elaboration".

The idea behind elaboration is to use information gained from type inference (namely, the inferred type) of an expression to "elaborate" it into the typed language. For example, if we can infer that $\lambda x.e$ has type $\tau_1 \to \tau_2$ under $\Gamma$, that directs us towards the proper translation for the lambda expression; for example, the type of the bound variable $x$ of the lambda expression should be $\tau_1$. Since we desire to use typing

information gained from type inference, elaboration is an example of a "type directed translation.". It then follows the elaboration needs access to the same typing context $\Gamma$ as type inference. Continuing our example of elaborating $\lambda x.e$ given that $\Gamma \vdash \lambda x.e \Rightarrow \tau_1 \rightarrow \tau_2$, we still need to elaborate the body $e$ in order to finish the translation. We can do this by recursively elaborating $e$ under the same context $\Gamma$ augmented with the assumption that $x : \tau_1$. Supposing that the result of elaborating $e$ in this manner is $e'$, we can say that elaboration of $\lambda x.e$ is $\lambda x{:}\tau_1.e'$.

You will need to figure out the details of this and the remaining cases of elaboration for the coding portion of the assignment. In your implementation, you will perform elaboration and type inference simultaneously.

# 3 Implementation

## 3.1 Overview

For the implementation portion of this assignment, you will be implementing polymorphic type inference for MinML, a translation between MinML and XMinML, and an evaluator for XMinML. We have provided you with a parser and a top-level interpreter environment. You will be able to write programs in concrete MinML syntax, load them up into the environment, evaluate them, and interactively evaluate more programs; your experience will be very similar to coding in SML. In addition, we have made some improvements to the test harness in order to make it simpler to use. Combined with the ability to write and test MinML programs in a natural way, you should find testing and debugging your code to be a much more enjoyable experience than in previous labs. Upon completion of this assignment, you will have a working implementation of MinML!

## 3.2 Overview of the supplied code

Before you begin, you should read through the provided code (especially the signatures) to gain an understanding of the setup. All of the necessary SML files are listed in `sources.cm`, and you can build the project in SML/NJ by typing `CM.make ''sources.cm''`. The following files are included:

| | |
|---|---|
| `minml-untyped.sml` | Abstract syntax of MinML |
| `minml-typed.sml` | Abstract syntax of XMinML |
| `untyped-util-sig.sml` | Signature for pretty printers, substitution, etc for MinML |
| `untyped-util.sml` | Implementation pretty printers, substitution, etc for MinML |
| `typed-util-sig.sml` | Signature for pretty printers, substitution, etc for XMinML |
| `typed-util.sml` | (To be partially filled in by you) |
| `context-sig.sml` | Signature for an abstract context for use with de Bruijn form |
| `context.sml` | Implementation of contexts |
| `infer-sig.sml` | Signature for type inference on MinML |
| `infer.sml` | (To be filled in by you) |
| `typecheck-sig.sml` | Signature for type checking on XMinML |
| `typecheck.sml` | Implementation of typechecker |
| `evaluator-sig.sml` | Signature for evaluator |
| `evaluator.sml` | (To be filled in by you) |
| `test-sig.sml` | Signature for test framework |
| `test.sml` | Test framework and tests |
| `toplevel-sig.sml` | Signature for interactive environment |
| `toplevel.sml` | Top-level interactive environment |
| `interpreter/*` | Various support files for the top-level environment |
| `tests/*.mml` | Assorted MinML programs with which you can test your MinML implementation |

## 3.3 Language Representation

The `MinMLUntyped` and `MinMLTyped` structures provide representations of the appropriate languages. Note that both representations use De Bruijn indices for variables; `MinMLUntyped` uses them for term variables, and `MinMLTyped` uses them for both term and type variables. Using De Bruijn indices in

complex (relative to previous assignments) languages like MinML can be tricky and prone to error. Pay special attention when using De Bruijn indices in your code. Even the TAs had trouble with this!

Note that variables count the number of abstractions are between them and their binding site. This includes both type and term abstractions. To reiterate: both type and term variables count the total number of abstractions (both type and term) between them and their binding site.

To this end, we have provided you with some helpful utility functions in the structures `UntypedUtil : UNTYPED_UTIL` and `TypedUtil : TYPED_UTIL`. Take at look at the specifications of these functions in the appropriate signature files. If you find yourself needing to write utility functions on the internal representations of MinML/XMinML, chances are that they are already present in `UntypedUtil` or `TypedUtil`.

### 3.4 Type Simplification

You need to implement type simplification with path compression as discussed above.

**T**ask 3.1 (5%). Write the `simplify` function in the `TypedUtil` structure.

### 3.5 Type inference

**T**ask 3.2 (60%). Fill out the `Inference` structure ascribing to the `INFERENCE` signature. Note the specifications presented in the `INFERENCE` signature, especially how the `infer` function returns both the inferred type and the elaborated term. You will have to figure out how to combine the algorithms presented earlier in this document, but this approach leads to much more concise code.

### 3.6 Typechecker

We have provided you with an implementation of a typechecker for XMinML. This could be useful to you as a sanity check and for debugging purposes. The interactive environment (more on that later) performs this sanity check for you.

### 3.7 Evaluator

**T**ask 3.3 (10%). Fill out the `Evaluator` structure ascribing to the `EVALUATOR` signature. Note that the evaluator is for XMinML and you are use the big-step evaluation dynamics as presented in the appendix.

### 3.8 Testing Your Code

Because of the much more substantial language with which you are dealing for the assignment, we provide a parser.

The `TopLevel` structure provides a handful of useful testing functions. `TopLevel.repl` will run a read-eval-print loop that accepts expressions and declarations, type infers and evaluates them, and returns the result. It behaves much like the SML/NJ repl. Like SML/NJ, evaluation is delayed until a semicolon is encountered. `TopLevel.replFile` processes the contents of a file and then drops into an interactive session. Semi-colons are not needed between declarations in the file.

You may find it useful to modify Toplevel to pretty print additional information during testing. `evalCore` is a good place to start when doing such changes.

We have provided a small test harness in the structure `Test` which ascribes to the signature `TEST` (these can be found in `test.sml` and `test-sig.sml`, respectively). The functionality described in `TEST` is pretty self-explanatory. You should find it easier to write your own tests, as we have made some enhancements to the harness structure over previous homeworks.

`Test.runAll` runs all the tests in the harness, while `Test.runTests` runs a provided list of tests. When you run tests, the test names and success/failure messages will be printed to the console.

`Test.printInfoN testList k` prints information about the $k^{th}$ test in `testList`.

There are a number of other functions in `TEST` that you may find useful. Read the source for more details.

The test suite is *not exhaustive. Since we have provided you with such a large infrastructure (parser, repl, file support etc), the provided test harness is rather small.* You can (and should!) add your own tests to the suite; look at the comments at the beginning of the `test.sml` file. You should also write your own `.mml` programs and play around with them in the interpreter environment.

### 3.9 Tips

- Start early! The programming portion of this assignment is *substantially* more difficult than that of previous assignments. Less of the algorithm is fully specified and there are more subtle considerations.
- Test your code. Write some interesting code of your own using the parser to do so. A lot of interesting functions can be written in MinML.
- Be very careful about De Bruijn indexes. In particular, you will probably find it necessary to use the `shift` functions. If you are adding binders to a term or type that may have free variables, and you do not want those variables to be captured, you must shift the term. This is made trickier because sometimes you *do* want the variables to be captured.
- While you should perform unification by imperatively updating the reference cells in unification variables, you should actually substitute for the unification variables while generalizing. To see why, consider what happens when a type variable is written into a unification variable that appears under a different number of binders at different places in the program. (This can't happen during unification, since there should be no type variables in types being unified.)
- Start early and ask us for help if you get stuck: email the TAs or come to office hours.

# A MinML

## A.1 Concrete Syntax

The concrete syntax of MinML is very similar to that of SML, with one exception being the syntax for `case` expressions, which requires surrounding parentheses. (MinML's **fn** construct also allows multiple variables, like SML's *should*). In fact, any MinML program that does not use multiple variable **fn**, does not use numeral shorthand for `unit lists`, and does not run afoul of the value restriction should also be valid SML when loaded with the declarations in `testing/sml_compat.sml`.

Pieces of MinML syntax are presented in **bold**. Note that MinML has the same associativity and precedence rules as SML, namely function application binds to the *left*.

```
program  ::=  decls

decls    ::=  decl
           |  decl decls

decl     ::=  val ident = expr
           |  fun ident funargs = expr

funargs  ::=  ident
           |  ident funargs

expr       |  ident
           |  (expr)
           |  ()
           |  let decls in expr end
           |  fn  funargs => expr
           |  expr expr
           |  (expr , expr)
           |  fst expr
           |  snd expr
           |  inl expr
           |  inr expr
           |  (case expr of  inl  ident => expr | inr ident => expr)
           |  roll expr
           |  unroll expr
```

SML-style nested comments are also supported.

## A.2 Parser

We've provided a parser that parses and lexes MinML programs into the abstract syntax. You don't need to worry about any of the details, but we will share a few in case you are curious; for example, declarations get translated to nested let expressions and `fun`s get translated to a combination of `fix` and lambda expressions.

### A.3 Notes about the interactive environment

While the language does not have a type of naturals numbers, nats may be simulated with `unit list`. The parser will transform integer constants appropriately. The toplevel's pretty printer will format lists nicely and knows to interpret `unit list` as `nat`.

### A.4 Abstract Syntax

$$
\begin{array}{rcl}
e & ::= & x \\
& | & \lambda x.e \\
& | & e\,e \\
& | & \mathsf{fix}(x.e) \\
& | & \mathsf{let}(e, x.e) \\
& | & \langle\rangle \\
& | & \langle e, e\rangle \\
& | & \mathsf{fst}(e) \\
& | & \mathsf{snd}(e) \\
& | & \mathsf{inl}(e) \\
& | & \mathsf{inr}(e) \\
& | & \mathsf{case}(e, x.e, x.e) \\
& | & \mathsf{roll}_{\mathsf{list}}(e) \\
& | & \mathsf{unroll}_{\mathsf{list}}(e)
\end{array}
$$

# B XMinML

## B.1 Abstract Syntax

$$
\begin{array}{rcl}
\tau & ::= & \alpha \\
& | & \forall \alpha.\tau \\
& | & \tau \to \tau \\
& | & \mathsf{unit} \\
& | & \tau \times \tau \\
& | & \tau + \tau \\
& | & \tau\ \mathsf{list}
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & x \\
& | & \lambda x{:}\tau.e \\
& | & e\ e \\
& | & \Lambda \alpha.e \\
& | & e[\tau] \\
& | & \mathsf{fix}(x{:}\tau.e) \\
& | & \mathsf{let}(e, x.e) \\
& | & \langle\rangle \\
& | & \langle e, e \rangle \\
& | & \mathsf{fst}(e) \\
& | & \mathsf{snd}(e) \\
& | & \mathsf{inl}^{\tau}(e) \\
& | & \mathsf{inr}^{\tau}(e) \\
& | & \mathsf{case}(e, x.e, x.e) \\
& | & \mathsf{roll}_{\tau\ \mathsf{list}}(e) \\
& | & \mathsf{unroll}_{\mathsf{list}}(e)
\end{array}
$$

## B.2 Static Semantics

In these rules, we make the tacit assumption that contexts $\Gamma$ are unordered and have at most one judgement of the form $x : \tau$ for each $x$. When adding judgements to the context, the latter assumption can be easily maintained by alpha varying terms. We make similar assumptions for contexts $\Delta$ of judgements of the form $\alpha\ \mathsf{type}$.

$$\frac{}{\Delta,\ \alpha\ \mathsf{type} \vdash \alpha\ \mathsf{type}}\ \text{istype-var} \qquad \frac{}{\Delta \vdash \mathsf{unit}\ \mathsf{type}}\ \text{istype-unit} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \tau_1 \to \tau_2\ \mathsf{type}}\ \text{istype-arr}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \tau_1 \times \tau_2\ \mathsf{type}}\ \text{istype-prod} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \tau_1 + \tau_2\ \mathsf{type}}\ \text{istype-sum} \qquad \frac{\Delta,\ \alpha\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \forall \alpha.\tau\ \mathsf{type}}\ \text{istype-forall}$$

$$\frac{}{\Delta; \Gamma,\ x : \tau \vdash x : \tau}\ \text{typ-var} \qquad \frac{\Delta; \Gamma,\ x : \tau \vdash e : \tau}{\Delta; \Gamma \vdash \mathsf{fix}(x{:}\tau.e) : \tau}\ \text{typ-fix}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash e_1\, e_2 : \tau_2}\ \text{typ-app} \qquad \frac{\Delta;\Gamma,\, x : \tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \text{typ-lam}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type} \quad \Delta;\Gamma \vdash e : \forall\alpha.\tau'}{\Delta;\Gamma \vdash e[\tau] : [\tau/\alpha]\tau'}\ \text{typ-papp} \qquad \frac{\Delta,\, \alpha\ \mathsf{type};\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \text{typ-plam}$$

$$\frac{}{\Delta;\Gamma \vdash \langle\rangle : \mathsf{unit}}\ \text{typ-unit} \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}\ \text{typ-pair}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \mathsf{fst}(e) : \tau_1}\ \text{typ-fst} \qquad \frac{\Delta;\Gamma \vdash e : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \mathsf{snd}(e) : \tau_2}\ \text{typ-snd}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_1}{\Delta;\Gamma \vdash \mathsf{inl}^{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}\ \text{typ-inl} \qquad \frac{\Delta;\Gamma \vdash e : \tau_2}{\Delta;\Gamma \vdash \mathsf{inr}^{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}\ \text{typ-inr}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_1 + \tau_2 \quad \Delta;\Gamma,\, x_1 : \tau_1 \vdash e_1 : \tau \quad \Delta;\Gamma,\, x_2 : \tau_2 \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{case}(e, x_1.e_1, x_2.e_2) : \tau}\ \text{typ-case}$$

$$\frac{\Delta;\Gamma \vdash e : \mathsf{unit} + (\tau \times \tau\ \mathsf{list})}{\Delta;\Gamma \vdash \mathsf{roll}_{\tau\ \mathsf{list}}(e) : \tau\ \mathsf{list}}\ \text{typ-listroll} \qquad \frac{\Delta;\Gamma \vdash e : \tau\ \mathsf{list}}{\Delta;\Gamma \vdash \mathsf{unroll}_{\mathsf{list}}(e) : \mathsf{unit} + (\tau \times \tau\ \mathsf{list})}\ \text{typ-listunroll}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau' \quad \Delta;\Gamma,\, x : \tau' \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{let}(e_1, x.e_2) : \tau}\ \text{typ-let}$$

### B.3 Dynamic Semantics - Big Step

$$\frac{[\mathsf{fix}(x{:}\tau.e)/x]e \Downarrow v}{\mathsf{fix}(x{:}\tau.e) \Downarrow v}\ \text{bigsteps-fix} \qquad \frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v_2}{\mathsf{let}(e_1, x.e_2) \Downarrow v_2}\ \text{bigsteps-let}$$

$$\frac{}{\lambda x{:}\tau.e \Downarrow \lambda x{:}\tau.e}\ \text{bigsteps-lam} \qquad \frac{e_1 \Downarrow \lambda x{:}\tau.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\, e_2 \Downarrow v}\ \text{bigsteps-app}$$

$$\frac{}{\Lambda\alpha.e \Downarrow \Lambda\alpha.e}\ \text{bigsteps-plam} \qquad \frac{e \Downarrow \Lambda\alpha.e' \quad [\tau/\alpha]e' \Downarrow v}{e[\tau] \Downarrow v}\ \text{bigsteps-papp}$$

$$\frac{}{\langle\rangle \Downarrow \langle\rangle}\ \text{bigsteps-unit} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\langle e_1, e_2\rangle \Downarrow \langle v_1, v_2\rangle}\ \text{bigsteps-pair}$$

$$\frac{e \Downarrow \langle v_1, v_2\rangle}{\mathsf{fst}(e) \Downarrow v_1}\ \text{bigsteps-fst} \qquad \frac{e \Downarrow \langle v_1, v_2\rangle}{\mathsf{snd}(e) \Downarrow v_2}\ \text{bigsteps-snd}$$

$$\frac{e \Downarrow v}{\mathsf{inl}^{\tau}(e) \Downarrow \mathsf{inl}^{\tau}(v)}\ \text{bigsteps-inl} \qquad \frac{e \Downarrow v}{\mathsf{inr}^{\tau}(e) \Downarrow \mathsf{inr}^{\tau}(v)}\ \text{bigsteps-inr}$$

$$\frac{e \Downarrow \mathsf{inl}^{\tau}(e') \quad [e'/x_1]e_1 \Downarrow v}{\mathsf{case}(e, x_1.e_1, x_2.e_2) \Downarrow v}\ \text{bigsteps-case-l} \qquad \frac{e \Downarrow \mathsf{inr}^{\tau}(e') \quad [e'/x_2]e_2 \Downarrow v}{\mathsf{case}(e, x_1.e_1, x_2.e_2) \Downarrow v}\ \text{bigsteps-case-r}$$

$$\frac{e \Downarrow v}{\mathsf{roll}_{\mathsf{list}}(e) \Downarrow \mathsf{roll}_{\mathsf{list}}(v)}\ \text{bigsteps-listroll} \qquad \frac{e \Downarrow \mathsf{roll}_{\mathsf{list}}(v)}{\mathsf{unroll}_{\mathsf{list}}(e) \Downarrow v}\ \text{bigsteps-listunrollroll}$$