Type systems

COMP 105 Assignment

Due Friday, March 17, 2017 at 11:59PM

Contents

Overview

This assignment will help you learn about type systems and polymorphism. You will understand how a type checker works, and you will be able to translate formal type-system rules into code for a type checker. You will add typed primitives to an interpreter. And you will write a couple of explicitly typed, polymorphic functions.

Setup

If you have not done so already, clone the book code:

```
git clone linux.cs.tufts.edu:/comp/105/build-prove-compare
```

You will modify two interpreters: `build-prove-compare/bare/tuscheme/tuscheme.sml` and `build-prove-compare/bare/timpcore/timpcore.sml`. You will compile your work with, e.g.,

```
mosmlc -o timpcore timpcore.sml
mosmlc -o tuscheme tuscheme.sml
```

Dire warnings

Your modified `timpcore.sml` and `tuscheme.sml` must compile using `mosmlc` without errors or warnings.

As in the ML homework, you must not use the functions `null`, `hd`, and `tl`. Use pattern matching.

Your `typed-funs.scm` must load into `tuscheme` without warnings or errors, as in

```
tuscheme -q < typed-funs.scm
```

All the homework problems

Reading comprehension (10 percent)

These problems will help guide you through the reading. We recommend that you complete them before starting the other problems below. You can download the questions[1].

1. Read Section 6.3, which describes how Typed Impcore is extended with arrays. Examine code chunk 403, which shows the cases that have to be added to the type checker. For each case, name the corresponding type rule, and explain the rule in informal English:

   - The rule for case `| ty (AAT (a, i)) = ...` is:

     Your explanation of the rule:

   - The rule for case `| ty (APUT (a, i, e)) = ...` is:

     Your explanation of the rule:

   - The rule for case `| ty (AMAKE (len, init)) = ...` is:

     Your explanation of the rule:

   - The rule for case `| ty (ASIZE a) = ...` is:

     Your explanation of the rule:

   You are now ready for exercise 2 in the pair problems.

[1] ./cqs.typesys.txt

2. Exercise 8 asks you to design new syntax and type rules for lists. In μScheme, we operate on lists by calling polymorphic functions. But in a monomorphic language, you must design *new syntax* for each list operation.

   (a) In the initial basis of μScheme, what four functions on lists would be sufficient to implement all the others?

   Exercise 8 asks you to label each of your rules as a formation rule, an introduction rule, or an elimination rule. Read the sidebar on "Formation, introduction, and elimination" on page 404.

   (b) Suppose that for each of the functions in part (a), you design a new syntactic form of expression. For each function, give its name, and say whether you expect the corresponding syntactic form to be an introduction form or an elimination form. (An introduction form creates new data, and an elimination form interrogates or "takes apart" old data.)

      i.

      ii.

      iii.

      iv.

   (c) How many formation rules will you need, and why?

   (d) You will need at least one rule for each of the syntactic forms in part (b), and you will need whatever formation rules you have listed in part (c). Do you expect to need any other rules? If so, why?

   You are now ready for Exercise 8.

3. Read Section 6.6.3 on quantified types in Typed μScheme.

   (a) Variable `syms` holds a list of symbols. What expression do you write to compute its length?

   (b) You are given a function `larger?` of type `(int -> bool)`. Using the predefined function `o`, what code do you write to compose `larger?` with `not`?

   (c) In testing, we sometimes use a three-argument function `third` that ignores its first two arguments and returns its third argument. Such a function has type

   ```
   (forall ('a 'b 'c) ('a 'b 'c -> 'c))
   ```

   There is only one sensible function that has this type. Using a `val` definition, define function `third` in Typed μScheme.

   You are ready for exercise TD.

4. Read about type equivalence on pages 430 and 431.

   You are given ML values `tau1` and `tau2`, which represent the respective Typed μScheme types `(forall ['a] 'a)` and `(forall ['b] 'b)`. These types are equivalent, but if you code `tau1 = tau2`, you'll get `false`. In your type checker, what code do you write to determine if `tau1` and `tau2` are equivalent?

   You will soon be ready for Exercise 23, but you first need to complete the next comprehension question.

5. Read Section 6.6.5 on typing rules for expressions in Typed μScheme. For each of the expressions below, say if it is well typed, and if so what its type is. If the expression is not well typed, say what typing rule fails and why.

```
; (a)
(if #t 1 #f)

; (b)
(let ([x 1]
      [y 2])
   (+ x y))

; (c)
(lambda ([x : int]) x)

; (d)
(lambda ([x : 'a]) x)

; (e)
(type-lambda ['a] (lambda ([x : 'a]) x))
```

   You are now ready for Exercise 23.

6. Exercise 24 on page 458 calls for you to add a primitive queue type to Typed μScheme. Read it. Then read "Primitive type constructors of Typed uScheme" in Section 6.6.9.

   (a) Which existing primitive type most resembles a queue type, and why?

   (b) When you add a primitive type constructor for queues, what chunk of the source code do you intend to add it to? (Give the page number, and if applicable, the letter.)

   Read "Primitives of Typed μScheme" in section M.4 of Appendix M on page 1294.

   (c) Which set of primitive functions most resembles the functions you will need to add for queues, and why?

   (d) When you add primitives functions that operate on queues, what chunk of the source code do you intend to add it to? (Give the page number, and if applicable, the letter.)

   You are ready for Exercise 24.

Problems to do by yourself (27 percent)

On your own, please work Exercise 8 on page 452 of Ramsey and problem **TD** described below.

8. *Adding lists to Typed Impcore*. Do Exercise 8 on page 452 of Ramsey. The exercise requires you to **design new syntax** and to **write type rules** for lists.

Your typing rules must be *deterministic*. This means that in any given typing environment, any given expression has at most one type, and the type must be computable by a function that is given the abstract syntax and the typing environment as inputs.

Related reading:

- Study the new abstract syntax for arrays in Section 6.3.2, which starts on page 400. Be sure you understand that you are seeing new syntactic forms, *not* functions.
- Each new form in code chunk 402a comes with a typing rule, which can be found in Section 6.3.3, which starts on page 403. As long as you keep in mind the differences between lists and arrays, this section will help you imagine the sorts of rules you will need to write for lists.
- For another example of new forms and corresponding rules, study the sum-introduction forms left and right in Section 6.4 near page 406.
- Finally, for help classifying rules, see the sidebar on "Formation, introduction, and elimination" on page 404.

*Hint: This exercise is more difficult than it first appears. I encourage you to scrutinize the lecture notes for similar cases, and to remember that you have to be able to type check **every** expression* at compile time. I recommend that you **do Pair Exercise 2 first**. It will give you more of a feel for monomorphic type systems.

Here are some things to watch out for:

- It's easy to conflate syntax, types, and values. In this respect, doing theory is significantly harder than doing implementation, because there's no friendly ML compiler to tell you that you have a type clash among `exp`, `tyex`, and `value`.
- It's especially easy to get confused about `cons`. You need to create a **new syntax** for `cons`. This syntax needs to be *different* from the `PAIR` constructor that is what `cons` *evaluates* to.
- Here's a good mental test case: it should be possible to write a recursive `reverse` function, and if `ns` is a list of integers, then `(car (reverse ns))` is an expression that should have a type.
- The empty list presents a challenge. Typed Impcore is monomorphic, which implies that any given piece of syntax has at most one type. But you want to allow for empty lists of *different* types. The easy way out is to design your syntax so that you have many different expressions, of different types, that all evaluate to empty lists. The most common mistake is to design a syntax that requires nondeterminism to compute the type of any term involving the empty list. But the problem requires a *deterministic* type system, because deterministic type systems are much easier to implement.

  You might consider whether similar difficulties arise with other kinds of data structures or whether lists are somehow unique.
- You might want to see what happens to an ML program when you try to type-check operations on empty and nonempty lists. For this exercise to be helpful, you have to understand the *phase distinction* between a type error and a run-time error. For example, `hd 3` results in a type error, but `hd []` is well-typed—and results in a run-time error.

TD. *Polymorphic functions in Typed uScheme*. To hold your solution, create a file `typed-funs.scm`. Implement, in Typed μScheme, fully typed versions of these two functions:

- Function `take` from the Scheme homework[2], which takes a given *number* of elements from a list
- Function `dropwhile`, from Exercise 10 on page 200, which drops elements that satisfy a given *predicate*

[2] ./scheme.html#take

The problem has four parts:

(a) Write, in a `check-type`, the polymorphic type you expect `take` to have.

(b) Write a definition of `take`.

(c) Write, in a `check-type`, the polymorphic type you expect `dropwhile` to have.

(d) Write a definition of `dropwhile`.

If you are not able to write implementations of `take` and `dropwhile` with the proper types, you may get partial credit by submitting parts (a) and (c) with `check-type` commented out.

Related reading: Read Section 6.6.3 on quantified types. Look especially at the definitions of `list2`, `list3`, `length`, and `revapp`. If you are not yet confident, go to Section M.5 in Appendix M and study the definitions of `append`, `filter`, and `map`.

Problems you can work on with a partner (63 percent)

Please complete Exercise 2 on page 449, Exercise 23 on page 458, Exercise **T** described below, and Exercise 24 on page 458. You may work by yourself or with a partner. (Most students prefer to work with a partner.)

2. *Type-checking arrays*. Do Exercise 2 on page 449 of Ramsey. My solution to this problem is 21 lines of ML.

Related reading:

- Study Section 6.2.1, which starts on page 392. Understand the structure of function `typeof`, which takes three explicit typing environments, and internal function `ty`, which has access to those environments even though it takes only one parameter. Study the cases for `SET`, `IFX`, `EQ`, and `PRINT`. Develop an idea how typing rules and code are related.
- Look at how the `ARRAYTY` value constructor is defined in chunk 384g. An ML value constructed with `ARRAYTY` represents an array type in Typed Impcore. When you need to recognize an array type, you will pattern match using `ARRAYTY`. When you need to construct an array type, you will apply `ARRAYTY` to another ML value of type `ty`.
- Understand the typing rules in Section 6.3.3, which starts on page 403.
- For a broader view of how Typed Impcore is extended with arrays, study Section 6.3, which starts on page 400.

23. *Type checking Typed uScheme*. Do Exercise 23 on page 458 of Ramsey: write a type checker for Typed uScheme. You will submit a modified interpreter and a file containing regression tests. Don't worry about the quality of your error messages, but do remember that your ML code must compile *without errors or warnings*.

Follow the step-by-step instructions listed below under the heading "How to build a type checker," which tells you how to build both the type checker and the regression tests.

My type checker is about 120 lines of ML. It is very similar to the type checker for Typed Impcore that appears in the book. The code could have been a little shorter, but I put some effort into error messages.

Related reading:

- Study Section 6.6.5, which starts on page 422—it gives the typing rules for expressions and definitions. You will implement each of these rules.
- Section 6.6.6, which starts on page 430, contains a long song and dance about type equivalence. You do not need to understand any of the song and dance—you will get the important aspects later in the term—but you *do* need to understand functions `eqType` and `eqTypes` well enough to know how to use them.
- In Section 6.6.7, which starts on page 435, there is even more song and dance. To implement your type checker successfully, you need to know only how to use functions `freetyvarsGamma` and `instantiate`.
- In Section 6.6.9, which starts on page 440, you need to know how to use function `asType`.
- Study the instructions "How to build a type checker" below.

T. *Unit tests for type checkers*. Create a file `type-tests.scm`, and in that file, write three unit tests for Typed µScheme type checkers. Each test must use either `check-type` or `check-type-error`. If you wish, your file may include `val` bindings or `val-rec` bindings of names used in the tests. *Your file must load and pass all tests using the reference implementation of Typed µScheme:*

```
tuscheme -q < type-tests.scm
```

If you submit more than three tests, we will use only the *first* three.

Here is a complete example `type-tests.scm` file, with five tests:

```
(check-type cons (forall ('a) ('a (list 'a) -> (list 'a))))
(check-type (@ car int) ((list int) -> int))
(check-type
   (type-lambda ['a] (lambda ([x : 'a]) x))
   (forall ('a) ('a -> 'a)))
(check-type-error (+ 1 #t))                          ; extra example
(check-type-error (lambda ([x : int]) (cons x x))) ; another extra example
```

You may, if you wish, submit any of these example tests, provided you attribute them properly to me. But your tests will be evaluated on how well they find bugs in the type checkers everyone writes—so new tests are more likely to earn high grades.

Related reading: To be able to write `check-type` and `check-error` tests, you need to know the concrete syntax for *unit-test* and *type-exp*, which is shown in Figure 6.2 on page 410.

24. *Polymorphic queue primitives for Typed µScheme*. Do Exercise 24 on page 458 of Ramsey: extend Typed uScheme with a type constructor for queues and with primitives that operate on queues. As it says in the exercise, **do not change** the abstract syntax, the values, the `eval` function, or the type checker. If you change any of these parts of the interpreter, your solution will earn **No Credit**.

Parts (a) and (b) ask you to write a kind and a type. The answers will appear in your code, but so we can find them, please also put the answers in your README file. Even if the code isn't perfect, you'll get partial credit for a good kind and good types.

I recommend that you represent each queue as a list constructed using the `PAIR` and `NIL` value constructors of Typed µScheme's `value` type. If you do this, you will be able to use the following primitive implementation of `put`:

```
let fun put (x, NIL)          = PAIR (x, NIL)
      | put (x, PAIR (y, ys)) = PAIR (y, put (x, ys))
      | put (x, _)            = raise BugInTypeChecking "non-queue passed to put"
in  put
end
```

Hint: you will modify two parts of the code that build the initial basis. Both parts are shown under "Building the initial basis" on page 443. Your definitions of `empty?`, `put`, `get-first`, and `get-rest` can go next to primitives `null?`, `cons`, `car`, and `cdr`. But because `empty-queue` is *not* a function, you will add its definition in a different place, as ⟨*primitives that aren't functions, for Typed μScheme* `::` 1295e⟩. In the code, look for the comment

```
(* primitives that aren't functions, for \tuscheme\ [[::]] 1295e *)
```

A couple of lines below, you will see an empty list. Edit the list to add a triple containing the name, value, and type of `empty-queue`.

My solution to this problem, including the implementation of `put` above, is under 20 lines of ML.

Related reading: Read "Primitive type constructors of Typed μScheme" in Section 6.6.9, which starts on page 440. Look at "Primitives of Typed μScheme" in section M.4 of Appendix M on page 1294 of Ramsey. Focus on primitives that manipulate `NIL` and `PAIR` values, such as `null?`, `cons`, `car`, and `cdr` in code chunk 1295c.

What and how to submit: Individual problems

You should submit these files:

- A `README` file containing
  - A list of the problems you completed
  - The names of the people with whom you collaborated
- File `cqs.typesys.txt`[3] containing your answers to the reading-comprehension questions
- A PDF file `lists.pdf` containing your work on Exercise 8.
- A file `typed-funs.scm` containing your code for problem **TD** above

As soon as you have the files listed above, run `submit105-typesys-solo` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

What and how to submit: Pair problems

For your joint work with your partner, *one* of you should submit these files:

- A `README` file containing
  - Your answers to parts (a) and (b) of Exercise 24 (the kind and the types will also be in your `tuscheme.sml`, but we want to see them in a readable notation)
  - A high-level description of the design and implemenation of your solutions
  - The names of the people with whom you collaborated

[3] ./cqs.typesys.txt

- A file `timpcore.sml` containing your code for Exercise 2
- A file `tuscheme.sml` containing your code for Exercises 24 and 23
- A file `regression.scm` containing the regression tests for Exercise 23. Each group of tests should be identified by a comment saying which step of the testing process the tests belong to.
- A file `type-tests.scm` containing your tests for Exercise T

As soon as you have the files listed above, run `submit105-typesys-pair` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

How your work will be evaluated

We will evaluate the functional correctness of your code by extensive testing.

We will evaluate your test cases by using them to look for bugs in other people's code. The more bugs your tests find, the better they are.

We will evaluate the structure and organization of your Typed μScheme code using the same criteria as used in previous homework assignments. We will evaluate the structure and organization of your ML code using similar criteria for naming and documentation. For indentation and layout, we'll look for conformance to the Style Guide for Standard ML Programmers,[4] within the constraints imposed by the code from the book.

General advice about type-related code

Here's some generic advice for writing any of the type-checking code, but especially the queues:

1. Compile early (you could use the command `mosmlc -o tuscheme tuscheme.sml`).
2. Compile insanely often.
3. Compile from within your editor, and use an editor that can jump straight to the location of the first error. With Vim, use `:make`, and with Emacs, use `M-x compile`.
4. Come up with examples in Typed μScheme.
5. Figure out how those examples are *represented* in ML.
6. Keep in mind the distinction between the term language (values of queue type, values of function type, values of list type) and the type language (queue types, function types, list types).
7. If you're talking about a thing in the term language, you should be able to give its type.
8. If you're talking about a thing in the type language, you should be able to give its kind.

How to build a type checker

Building a type checker is the first COMP 105 exercise of significant scope. You must approach it systematically. *Do not copy and paste the Typed Impcore code into Typed μScheme*. Copying and pasting would be a grave strategic error. You will be *much* better off adding a brand new type checker to the `tuscheme.sml` interpreter, one step at a time.

[4] ../handouts/mlstyle.pdf

Writing the whole type checker before running any of it will make you miserable. Use the techniques presented in class, start small, and build one piece at a time.

Follow these steps:

1. The initial basis contains code for predefined functions that you will not be able to typecheck until your work is complete. Your first step should therefore be to disable those functions. I suggest that you find the line in the source code that corresponds to the binding of value `fundefs` on page 443 of the book:

   ```
   val fundefs   =
   (* predefined {\tuscheme} functions, as strings (generated by a script) *)
   ```

   Replace the line `val fundefs =` with these two lines:

   ```
   val predefined_included = false
   val fundefs = if not predefined_included then [] else
   ```

   Verify that your modified interpreter compiles with `mosmlc`.

2. Start function `typeof`. I recommend defining an internal function `ty`, just as in the type checker for Typed Impcore. Create the first draft of `ty` by writing a clausal definition that has one case for each syntactic form of Typed μScheme. On the right-hand side of each clause, raise the `LeftAsExercise` exception.

   Verify that your modified interpreter compiles with `mosmlc`.

3. Write a function `literal` that computes the type of a literal value. Start with just numbers, Booleans, and symbols—you can add types for list literals later.

   Verify that your modified interpreter compiles with `mosmlc`.

4. Write the case for `typeof`/`ty` that handles `LITERAL` expressions—it should call `literal`.

5. Create a test file `regression.scm` containing a comment and three unit tests:

   ```
   ;; step 5

   (check-type 3 int)
   (check-type #t bool)
   (check-type 'hello sym)
   ```

   Verify that your modified interpreter compiles with `mosmlc`.

   Verify that your interpreter correctly typechecks the literals used in the tests above. Run

   ```
   ./tuscheme -q < regression.scm
   ```

   You **must** remember the `./` in `./tuscheme`, or otherwise you will be testing my code, not your own code.

   If you are working on a departmental server, you can try the command

   ```
   regression-test-tuscheme
   ```

   As you build your type checker, you will continually add "regression" tests to file `regression.scm`. They are called "regression" tests because they are designed to prevent *regressions*—a regression is a bug introduced into previously working code.

6. Write the case for `typeof` that handles IF-expressions, which I plan to show in class. Add regression tests for a few IF-expressions that have different types. Also add tests for some IF-expressions that are ill-typed.

   - Add the comment `;; step 6` to your `regression.scm` file.
   - Add some `check-type` unit tests for `if` to your `regression.scm` file.
   - Add some `check-type-error` unit tests for `if` to your `regression.scm` file.
   - *Verify that your interpreter compiles and passes all its unit tests.* If something goes wrong with a unit test, make sure the unit test is OK—test it by running `/comp/105/bin/tuscheme -q < regression.scm`.

7. Implement the VAR rule. Add commented regression tests that check the types of some primitive functions.

   Verify that your interpreter compiles and passes all its regression tests.

8. Now turn your attention to function `elabdef`, which is right next to `typeof`. It takes a true definition, a kind environment, and a typing environment, and it returns a new typing environment and a string.

   - The new typing environment contains a binding for whatever name is defined.
   - The string shows the *type* of whatever name is defined, which you get by applying function `typeString` to the type.

   Write four clauses for `elabdef`, each to raise `LeftAsExercise`. There should be one clause each for `VAL`, `VALREC`, `EXP`, and `DEFINE`.

   Verify that your interpreter compiles and passes all its regression tests.

9. Continuing work with `elabdef`, implement the VAL rule for definitions. Then the EXP rule.

   Add a "step 9" comment and a couple of `val` bindings to your regression-test file, along with `check-type` and `check-error` tests that use those bindings.

   Verify that your interpreter compiles and passes all its regression tests.

10. Return to `typeof`. Implement the rule for function application. Add regression tests that apply functions. Include both `check-type` and `check-type-error` tests. You should be able to apply some primitive arithmetic and comparison functions.

    Verify that your interpreter compiles and passes all its regression tests.

11. Implement LET binding. The Scheme version is slightly more general than what I plan to cover in class. Be careful with your contexts.

    Verify that your interpreter compiles and passes all its regression tests.

12. Once you've got LET working, LAMBDA should be quite similar.

    Add suitable regression tests, and verify that your interpreter compiles and passes its regression tests.

13. Knock off SET, WHILE, and BEGIN.

Add suitable regression tests, and verify that your interpreter compiles and passes its regression tests.

14. There are a couple of different ways to handle LETSTAR. As usual, the simplest way is to treat it as syntactic sugar for nested LETs. Implement type checking for `LETSTAR`.

    Add suitable regression tests, and verify that your interpreter compiles and passes its regression tests.

15. Go back to `elabdef`, and knock off the definition forms VALREC and DEFINE. (Remember that DEFINE is syntactic sugar for VALREC.)

    Don't overlook the side condition; in

    ```
    (val-rec x e)
    ```

    it is necessary to be sure that `e` can be evaluated without evaluating `x`. Many students forget this side condition, which can be implemented very easily with the help of the function `appearsUnprotectedIn`, which should be listed in the code index of your book.

    Add `val-rec` and `define` definitions to your regression-test file, and add regression tests for the names you define.

    Verify that your interpreter compiles and passes all its regression tests.

    Your `elabdef` is now complete.

16. Return to `typeof` and implement TYAPPLY and TYLAMBDA. Save these cases for after the last class lecture on the topic. (Those are the *only* parts that have to wait until the last lecture; you can have your entire type checker, except for those two constructs, finished before the last class.)

    Add suitable regression tests, and verify that your interpreter compiles and passes its regression tests.

17. Complete your `literal` function by making sure it handles list literals formed with `PAIR` or `NIL`.

    Add suitable regression tests, and verify that your interpreter compiles and passes its regression tests.

    Your `typeof` function is now complete.

    Your entire type checker is now complete.

18. Return to the code you modified in step 1. Bind

    ```
    val basis_included = true
    ```

    Verify that your interpreter compiles and that it can typecheck the predefined functions of Typed μScheme.

Avoid common mistakes

In Exercise 8, it's a common mistake to try to create a type system that prevents programmers from applying `car` or `cdr` to the empty list. Don't do this! Such a type system is too complicated for COMP 105.

As in ML, taking `car` or `cdr` of the empty list should be a well-typed term that causes an error at run time.

In Exercise 8, it's common to write a nondeterministic type system by accident. The rules, typing context, and syntax have to work together to determine the type of every expression. But you're free to choose whatever rules, context, and syntax you want.

In Exercise 8, it's inexplicably common to forget to write a typing rule for the construct that tests to see if a list is empty.

There are already interpreters on your PATH with the same name as the interpreters you are working on. So remember to get the version from your current working directory, as in

```
ledit ./timpcore
```

Just plain `timpcore` will get the system version.

ML equality is broken! The = sign gives equality of *representation*, which may or may not be what you want. For example, in Typed uScheme, you must use the `eqType` function to see if two types are equal. If you use built-in equality, **you will get wrong answers**.

It's a common mistake to call `ListPair.foldr` and `ListPair.foldl` when what you really meant was `ListPair.foldrEq` or `ListPair.foldlEq`.

It's not a common mistake, but it can be devastating: when you're writing the type of a polymorphic primitive function, write the type variable with an ASCII quote mark, as in `'a`, not with a Unicode right quote mark, as in `'a`.

It's not a common mistake, but don't define any new exceptions. And don't raise any exceptions besides `TypeError`. (If you don't finish, you might also raise `LeftAsExercise`.)

What is and is not hard or time-consuming

In Exercise 8 on page 452, I am asking you to create new type rules on your own. Many students find this exercise easy, but many find it very difficult. The "difficult" people have my sympathy; you haven't had much practice *creating* new rules of your own. You'll get it now.

Problem **TD**, writing `take` and `dropwhile` in Typed μScheme, requires that you really understand *instantiation* of polymorphic values. Once you get that, the problem is not difficult, but the type checker is persnickety. A little of this kind of programming goes a long way.

Exercise 2, type-checking arrays in Typed Impcore, has a lot of related reading—you'll fill in any ideas or details that you missed in class. But aside from the amount of reading, this exercise is probably the easiest exercise on the homework. You need to be able to duplicate the kind of reasoning and programming that we will do in class for the language of expressions with LET and VAR.

Exercise 23, the full type checker for Typed μScheme, presents two kinds of difficulty:

- You have to understand the connection between typing judgments, typing rules, and code.
- You have to understand a moderately sophisticated ML program (the interpreter) and then build a relatively big and independent extension of it.

For the first item, we'll talk a lot in class about the concepts and the connection between type theory and type checking. For the second item, it's not so difficult **provided** you remember what you've learned about building big software: don't write it all in one go. Instead, start with a tiny language and grow it very slowly, testing at each step—just as instructed in the guide above. As in yoga, the slow way is the fastest.

Exercise 24, adding queues to Typed μScheme, requires you to understand how primitive type constructors and values are added to the initial basis. And it requires you to write *ML code* that manipulates *μScheme representations*. The task is not inherently difficult, but there are two challenges:

- Because the task is not inherently difficult, it won't get any air time in class. You'll rely on the book.
- Understanding how *ML* code relates to a *μScheme* primitive is not trivial

To address these challenges, your best bets are to study the way the existing primitives are implemented and to emulate the code that you see.