CSE 341, Spring 2011, Lecture 17 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

One of the biggest differences between Scheme and ML is that ML has a type system that rejects many programs "before they run" meaning they are not ML programs at all. (Ruby and Java have an analogous difference.) The purpose of this lecture is to give some perspective on what it means to have a type system, how one can judge a particular type system, and in general some of the advantages and disadvantages of having a type system.

Let's first discuss the difference between *strong typing* and *weak typing*. In a language with weak typing, there exist programs that the implementation must accept as legal (i.e., it must run them) but that then can do *anything* including produce arbitrary output, corrupt data, delete files, or set the computer on fire. C and C++ are the most common examples – if you have a type-cast that is incorrect (the result of evaluting an expression is not the type you claim), you don't get an exception like in Java. Instead, anything can happen, including things that make no sense from the programmer's perspective. A strongly typed language does not have this property: A value of a certain type (e.g., `int`) always behaves as it should (e.g., as an integer), which prevents such arbitrary behavior. In this sense, both ML and Scheme are strongly typed. Even though Scheme does not reject ill-typed programs until they run, it is still well-behaved enough never to treat a string as a procedure.

In the past, strong typing was dismissed with the slogan, "strong types for weak minds" which meant that programmers should be smarter than any particular type system so languages should have unchecked type casts for programmers who know what they are doing. You hear this view less these days since arbitrary behavior sounds pretty bad for a society that relies on software for nearly everything, and experience shows that programmers often make mistakes. The "advantage" of unchecked type casts is that they are more efficient at run-time (i.e., zero cost), just like the "advantage" of unchecked array-indexing is more efficient. Yet this sort of undefined behavior costs the world economy billions of dollars a year. For the rest of the lecture, we will consider only strong typing.

The relevant difference between ML and Scheme is that ML is *statically typed* and Scheme is *dynamically typed*. To understand the difference, consider that in ML `"hi" - "mom"` is an error and in Scheme `(- "hi" "mom")` is an error, but these errors arise at different times. An ML "program" containing string subtraction is not a program at all — the type-checker rejects it statically, i.e., "at compile-time", i.e., before execution begins. A Scheme program containing string subtraction is a perfect fine program, but if dynamically, i.e., "at run-time", the subtraction is actually performed, an error will occur since `-` checks that its arguments are numbers. Similarly, `car` and `cdr` check that their arguments are pairs whereas ML's `hd` and `tl` do not need to check — the type-checker guarantees it.

An example like `"hi" - "mom"` clearly demonstrates an advantage of static typing — such an expression is always wrong so it's helpful to know that before the program starts running since it can't possibly be useful. However, static type systems are always approximate. They also reject programs that are useful just to be sure that they don't allow bad programs. Before we discuss exactly why and some important terminology about this issue, here is an example:

```
(define (f g x y)
  (if (g x)
      (string-length y)
      (+ y 1)))
```

```
fun f (g,x,y) =
  if g x
  then String.size y
  else y + 1 (* type-error! *)
```

The ML code will not type-check because `y` has to have type `string` in one place and type `int` in another. However, the function `f` is fine *if* callers use it correctly. In particular, they must pass an int for `y` whenever `g x` will produce `#f` and a string for `y` otherwise.

In general, one big advantage of a static type system is that it catches certain bugs without testing, i.e., earlier in the software-development cycle. However, it's *impossible* for a static type system to reject

exactly all the buggy programs and accept exactly all the correct programs, so any type system is just an approximation. Here are some reasons why:

- For an arbitrary program, it's impossible (in the sense of *undecidable* as studied in CSE 311) to know at compile-time what code will execute, what values variables will have, etc. So a type system will give *false positives* if it thinks for example that an expression might evaluate to a string when in fact it will always evaluate to an int.
- Just because a program "type-checks" does not mean it is correct. Algorithm bugs can remain since surely a type system cannot "know" you meant to write `-` where you wrote `+`.

This is not meant to dismiss the advantages of static typing, but simply to emphasize that type systems are always an approximation of what you actually want to check for. Approximations can be extraordinarily useful, especially when they are fast, automatic, and *sound* (see below, roughly, "always erring on the side of caution").

So far we have been very informal about a key question for any type system: *What is it checking, i.e., what does it intend to prevent for any program that type-checks?* This depends on the language, and you simply cannot talk about whether static typing is "good" or "bad" without being clear about what is being checked. For example, here are several errors one can have in an ML program:

1. Apply a primitive (e.g., `+`) to arguments of the wrong values (e.g., functions)
2. Violate a module signature (e.g., access a private function)
3. Have a redundant pattern in a case-expression
4. Apply `hd` to the empty list
5. Retrieve the 10th element of an array that has only 5 elements

It turns out that the ML type system prevents 1–3 (they will *never* happen in a well-typed program) but not 4–5. This division of what is checked is "pretty normal" — it's about what one typically expects from a type system these days — but a different language might have a different list. As a programmer, it's essential to know what the type-checker prevents and what it does not. When I program in ML, I am fairly sloppy when I write down constructor names in patterns because I know the type-checker will catch any typos. But I am very careful when I compute array indices because I know only testing will find the errors. These considerations also affect what test cases I write — there is no sense in testing for something that the type-system ensures cannot happen.

Once we have a type system and have enumerated what it is supposed to prevent, we can ask two important questions related to technical definitions you should know:

1. Is the type system *sound*? A sound type system never accepts a program that can do something the type system is supposed to prevent. Such a program is a *false negative*: the type-checker missed a bad program by letting it run.
2. Is the type system *complete*? A complete type system never rejects a program that cannot do something the type system is supposed to prevent. Such a program is a *false positive*: the type-checked rejected a bad program that would be fine to run.

In CSE 311 you learn that any nontrivial property of a program is *undecidable*, meaning it is impossible for another always-terminating program to determine the property soundly and completely. A direct corollary is that every type system has to be either unsound or incomplete (or both). The typical engineering choice is to provide a sound but incomplete type system, i.e., it rejects some extra programs in order to be sound. Provided we "do not compromise on soundness" the goal of type-system design then is to provide a convenient type system. Convenience involves trade-offs: we want the system to be simple so programmers can understand type errors and type-checking is efficient, but we also do not want too many false positives.

Unsound type systems are generally frowned upon for programming languages, but they can still be useful for finding bugs. The problem is when an unsound type system accepts a program, you do not have any guarantee about how the program behaves.

As ML and Scheme demonstrate, the typical points at which to prevent a "bad thing" are "compile-time" and "run-time". However, it's worth realizing that there is really a continuum of eagerness about when we declare something an error. Consider for sake of example something that most type systems do not prevent statically: division-by-zero. If we have some function containing the expression `(/ 3 0)`, when could we cause an error:

- Keystroke-time: Adjust the editor so that one cannot even write down a division with a denominator of 0. This is approximate because maybe we were about to write 0.33, but we were not allowed to write the 0.
- Compile-time: As soon as we see the expression. This is approximate because maybe the context is `(if #f (/ 3 0) 42)`.
- Link-time: Once we see the function containing `(/ 3 0)` might be called from some "main" function. This is less approximate than compile-time since some code might never be used, but we still have to approximate what code may be called.
- Run-time: As soon as we execute the division.
- Even later: Rather than raise an error, we could just return some sort of value indicated division-by-zero and not raise an error until that value was used for something where we needed an actual number, like indexing into an array.

While the "even later" option might seem too permissive at first, it's exactly what floating-point numbers do. `(/ 3.0 0.0)` produces `+inf.0`, which can still be computed with but cannot be converted to an exact number. In scientific computing this is very useful to avoid lots of extra cases: maybe we do something like take the tangent of $\pi/2$ but only when this will end up not being used in the final answer.

Now that we know what static and dynamic typing are, let's wade into the decades-old argument about which is better. Rather than try to resolve this subjective question, we'll consider five specific arguments and consider objectively the trade-offs that static and dynamic typing resolve differently. These are all separate from the basic issues discussed above that static typing catches many errors for you early, soundness ensures certain kinds of errors do not remain, and incompleteness means some perfectly fine programs are rejected.

Convenience:

Dynamic and static typing can both claim to be more convenient. For example, for dynamic typing consider a simple Scheme function that just returns either a number or a boolean:

```
(define (f x) (if (> x 0) (* 2 x) #f))
```

There is no extra work to do in this function, and callers can simply use a run-time test to act accordingly for either return type, for example:

```
(let ([ans (f y)]) (if (number? ans) e1 e2))
```

In ML, just to have a function return like this, we need to define a datatype and use constructors:

```
datatype intOrBool = Int of int | Bool of bool
fun f x = if x > 0 then Int (2*x) else Bool false
```

And callers still need to use a run-time test (in this case pattern-matching):

```
case f y of
  Int ans => e1
| Bool _ => e2
```

In essence, Scheme's "one big datatype" lets us just return what we want.

On the other hand, static typing makes it more convenient to write libraries that require their arguments to have certain types. As a simple example, a function like `fun cube x = x * x * x` does not have to guard against a non-number argument; the ML type system ensures all callers pass a number. In Scheme if we wanted this guarantee, we have to check at run-time:

```
(define (cube x) (if (not (number? x))
                     (error "bad arguments")
                     (* x x x)))
```

Notice that without this check, the error would arise in the `*` function. In practice, such errors (e.g., applying `car` to a non-list) can arise deep inside many library calls making it harder to identify the misuse of some library. Therefore, it's good in Scheme to check argument types at entry points to a library, but this is an inconvenience often unnecessary in ML.

Preventing useful programs:

Incompleteness ensures that a type system rejects some programs that it doesn't need to, but the real question is whether it reject programs that people want to write. That depends on many things, including how powerful the type system is. Suppose ML did not have parametric polymorphism. Then you would need a different list library for every element type you were using for some list in your program. Reimplementing `length`, `map`, `append`, etc. would get burdensome. Since ML does have such polymorphism, the number of times you find yourself duplicating effort just to appease the type-checker is considerably reduced, but probably not to zero.

Proponents of static typing have another argument against proponents of Scheme-style dynamic typing: You can program that way in ML if you want; you just have to use lots of manual uses of constructors and pattern-matching. Consider this datatype binding:

```
datatype SchemeVal = Int of int | String of string | Bool of bool
                   | Fun of SchemeVal -> SchemeVal
                   | Cons of SchemeVal * SchemeVal
                   (* more cases for other kinds of Scheme things *)
```

This is basically Scheme's "one big datatype." Then we can just have all our functions take and return values of this type. For example, consider this function of type `SchemeVal->SchemeVal` that returns either a "function" (wrapped with the `Fun` constructor) or a "pair" (wrapped with the `Cons` constructor):

```
fun f x =
  if (case x of Bool b => b)
  then Fun (fn y => (case y of Int i => Int (i * i * i)))
  else Cons (Int 7, String "hi")
```

This function is like Scheme's

```
(define (f x) (if x (lambda (i) (* i i i)) (cons 7 "hi")))
```

What it does is pattern-match whenever it needs what kind of thing some `SchemeVal` is and use a constructor whenever it builds a `SchemeVal`. The ML code will get warnings for the incomplete pattern matches, but we can ignore those. This example is also instructive because it makes explicit exactly where Scheme is implicitly testing the types of things and inserting constructors (or "tags"). Note the following version of the ML code is more concise by using patterns in function arguments:

```
fun f (Bool x) =
  if x
  then Fun (fn Int i => Int (i * i * i))
  else Cons (Int 7, String "hi")
```

Code evolution:

Lots of software development is concerned with changing code that is already written to fix mistakes or add new functionality. Static and dynamic typing can both claim advantages with respect to how easy it is to evolve code that already exists. As a very small example, suppose in version 1 of some software we have a function that takes and returns integers, say `fun f x = x * 2` in ML or `(define (f x) (* x 2))` in Scheme. Now suppose we want new functionality where `f` can also take a string and append it to itself. In Scheme, we can add this as follows:

```
(define (f x)
  (if (number? x)
      (* x 2)
      (string-append x x)))
```

Notice that none of our existing call-sites need to change. Callers like `(f 7)` continue to work without knowing that `f` has evolved. In ML, the story is less pleasant since we need `f` to take a datatype:

```
datatype t = I of int | S of string
fun f x =
   case x of
      I i => I (i * 2)
    | S s => S (s ^ s)
```

Therefore, we need to go change all the existing call-sites, for example from `f 7` to `case f (I 7) of I i => i` (or more to avoid the inexhaustive pattern-match warning that results from this change).

However, for code evolution where call-sites *do* need to change, static typing can be a huge help. Suppose we wanted to make this change in Scheme instead:

```
(define (f x)
   (if (> x 0)
       (* x 2)
       "negative number"))
```

Now we need to find all the places are Scheme code calls `f` to make sure that it is not assuming a number is returned (unless it knows the argument is positive). In ML, the corresponding change:

```
datatype t = I of int | S of string
fun f x =
   if x > 0
   then I (x * 2)
   else S "negative number"
```

also requires changing all calls to `f`. But if we just try to compile the code after making the change above, the type-checker will automatically produce a "to-do list" of all the places we need to make a change. This is incredibly convenient for making sure we do not miss any.

A related example that often arises is adding a new constructor to a datatype binding. After the addition, the type-checker will probably report many incomplete pattern matches, and again this is a convenient to-do list of what we need to evolve. Notice, though, that if the original software used wildcard patterns as default cases, this programming technique is less effective.

Reusing code:

Dynamic typing arguably makes it easier to reuse library functions. After all, if you build lots of different kinds of data out of cons cells, you can just keep using `car`, `cdr`, `cadr`, etc. to get the pieces out rather than defining lots of different getter functions for each data structure. On the other hand, this can mask bugs.

For example, suppose you accidentally pass a list to a function that expects a tree. If `cadr` works on both of them, you might just get the wrong answer or cause a mysterious error later, whereas using different types for lists and trees could catch the error sooner.

This is really an interesting design issue more general than just static vs. dynamic typing. Often it is good to reuse a library or data structure you already have especially since you get to reuse all the functions available for it. Other times it makes it too difficult to keep separate things that are really different conceptually so it is better to define a new type. That way the type-checker (or at least in Racket using `define-struct` a run-time error) can catch when you put the wrong thing in the wrong place.

Performance:

Lastly, both dynamic typing and static typing can claim to be more efficient. For dynamic typing, because programmers do not have to write their code in a certain style to appease the type-checker, they have fewer restrictions on how they write their algorithms. For static typing, the main argument is that the implementation does not need to have so many implicit type-tests. After all, in ML, for `x + y`, the interpreter *knows* `x` and `y` will be numbers, so it can just add them. In Scheme, for `(+ x y)`, the interpreter has to ask `(and (number? x) (number? y))` and then get out the "actual numbers underneath." This takes time and space (for storing the tags necessary for implementing `number?`). Dynamic typing proponents would counter that often the implementation can optimize away much of this overhead and you have the same overhead in ML when you use datatypes. Static typing proponents would counter that by pointing out that ML programmers have control over when they use datatypes and when they do not.

In any case, there are many factors that influence performance and it is likely that these low-level issues are often not the dominating factors.

Finally:

Everyone is certainly entitled to his/her own opinion on whether ML-style type-checking (or Java-style or whatever) is more of a benefit or a hindrance, but it's also important to understand the facts underlying the arguments on each side above. Also remember that every language checks some things statically and others dynamically (even Scheme won't allow a program with mismatched parentheses), so it is really a question of *what* you want to check *when* and how *approximate* you are.