Strong Typing

M-J. Dominus

Plover Systems Co.

`mjd-perl-yak-id-i0n+bd8c8jw+@plover.com`

Talk given to Philadelphia and St. Louis Perl mongers September 22, 1999

You might want to read this while following the original slides for the talk.

Slide 2: On Holy Wars

Sometimes when you read Usenet or whatever, and you see folks arguing endlessly about the same religious topics that they've been arguing about for years, it can be hard to believe that it could ever come to an end. But sometimes these holy wars do come to an end.

For example, the biggest holy war of holy wars was the war over structured programming. It started in the 1960s, with Edsger Dijkstra's famous letter to the *Communications of the ACM* titled *GOTO Statement Considered Harmful*. For years and years it seemed as though this argument was never going to come to an end, but it has, and the bodies of the `goto` folks are buried pretty deep.

(At this point in the talk, Walt claimed that the `goto` folks had not lost completely, because for example you could find use of `goto` all through the Linux kernel. I replied that the mere fact that Walt could say that was evidence of the tremendous magnitude of the anti-`goto` forces: It was only because he was unfamiliar with 1960's programming styles that he thought that one `goto` every fifty lines was a frequent use of `goto`.)

Anyway, the real point is that nobody disbelieves the structured programming manifestos any more. Everyone thinks it is better to structure your program into explicit `while` and `for` loops and to reserve use of `goto` for special circumstances. When people want to argue the other side, they argue for extremely feeble positions, like ''Well, sometimes `goto` is useful when you want to break out of a deeply nested conditional.'' That is the extent of the support for `goto` these days.

A holy war that's even further back is the subroutines vs. no subroutines war. On one side, programmers who say that use of subroutines improves modularity and maintainability. On the other side, the programmers who say that the gains are not that great, and that subroutine calls are slow. You know who won that one.

Probably the oldest holy war of all is high-level languages vs. assembly language. From our vantage point in 1999, it can be hard to imagine that this was ever a subject for debate at all. But I caught the tail end of this one when I was first learning to program in the 1970s.

Here's one that isn't resolved yet: Are formal verification methods and correctness proofs valuable? I think that it's not yet clear. Certainly they aren't very useful *today*. But just as I can easily imagine a future in which correctness proof research has been abandoned, I can easily imagine a future in which everyone takes for granted the idea that a program should come with a formal correctness proof. I just can't see either of those things happening in less than twenty-five years.

One of these holy wars that I find particularly interesting, and the one I'm going to talk about, is the war over 'strong type checking'. What makes this one interesting is that many people think it's over, but it isn't. I'm going to spend a little time talking about the history of type checking up to about thirty years ago, and then I'll move on to the state of the art.

Slide 3: Why Types?

Why have types at all? What are they for?

Well, imagine that you are a programmer in 1962, so you are writing an assembly language program. You store the string `"More yummy eels, please"` into the computer's memory, say at locations 2001-2024. Some time later, a mistake occurs, because your program has a bug. The four-byte quantity at locations 2012-2015 is loaded into a floating-point register and used as a floating-point number. What happens when the four bytes `eels` are used as a floating-point number? 1.87292264408661e+31, that's what. Well, the actual value will vary depending on your computer, but it won't be anything sensible.

Slide 4: Why Types?

In fact, we could interpret that four-byte quantity as a four-byte integer also, in which case it turns into the number 1,936,483,685. We would like to prevent this sort of error from occurring.

The idea we have is that the computer is going to keep track of the *type* of value stored at each part of its memory, whether it's part of a string, or an integer, or a floating-point number, or whatever. Each operation such as multiplication will also have a type. The compiler or assembler can check to make sure that the types of an operator and its operands match, and raise a compile-time error if they don't, and so hopefully detect some sorts of bugs early on before they have a chance to cause real trouble.

As far as I can tell, this idea first appeared on COBOL.

Slide 5: Early Type Systems: Fortran

To give you a flavor of what early type systems were like, we'll look at Fortran, a very popular language that originally dates from about 1955. Fortran had a bunch of integer types in 2, 4, and 8-byte sizes, called respectively `INTEGER*2`, `INTEGER*4`, and `INTEGER*8`. It had several sizes of boolean types, which in Fortran were called `LOGICAL*1` and so forth. It had real number types, also in several sizes, called

`REAL*4` and so on. And it had complex number types of various sizes, from `COMPLEX*8` through `COMPLEX*32`. You could also omit the sizes (the `*8` parts) and get a default size.

If you wrote a Fortran program that contained the code

```
INTEGER I
REAL R, S

R = I + S
```

the compiler could generate the correct instructions, including the implicit conversion of the value of the integer variable `I` to the entirely different floating-point representation of the same number.

This is called *static* type checking, because it is performed once at compile time.

Slide 6: Fortran Continues

The Fortran type system had some other interesting properties. You could omit the explicit type declarations, in which case variables defaulted to `INTEGER` if their names began with `I`, `J`, `K`, `L`, `M`, or `N`, and to `REAL` otherwise. Fortran had array types also; the declaration

```
INTEGER A(10)
```

declared `A` to be an array of ten integers, of which `A(3)` was the third. And functions also had types, so for example:

```
FUNCTION F(X)
INTEGER F, X
F = X+1
RETURN

N = F(37)
```

Here the compiler can tell that both the argument to `F` and its return value are integers and check that `F` is being used appropriately.

Once again, this is all happening at compile time. The assembly language output of the compiler has no type checking at all. The compiler has already made sure that you did everything right, so rechecking at run time would be a waste of time. This strategy is called static type checking.

Slide 7: Early Type Systems: Lisp

Since we're taking a tour of the programming world of the 1950's, let's look at the other successful language from that era, Lisp. Lisp is quite different from Fortran. Instead of static type checking, it has dynamic type checking. This means that each value carries around its type with it, and that each time an operation is invoked, it checks to make sure its operands have the correct types. If not, it raises a run-time type error. For example:

```
(+ 1 2)
        3
```

```
(+ 1 2.0)
        3.0

(+ 1 "eels")
        Error in +: "eels" is not a number.
```

This run-time type checking incurs a performance penalty, of course, but it also lends tremendous flexibility to the language. It means that you can write one function that does the 'right thing' for many different types of values, and it means that you can extend the type system at run time, which in turn enables techniques such as object-oriented programming. All of these are impossible (or at least very difficult) in Fortran.

Slide 8: Typing in ALGOL-based languages

The ALGOL language of 1960 was tremendously influential, and spawned a number of successful descendants, of which Pascal and C (and C++) are among the best-known. The type systems of C, Pascal, and ALGOL 60 are all quite similar. ALGOL attempted to extend type systems beyond simple scalars to *aggregates* of various sorts. They contain:

- `array of` *type*
- `pointer to` *type* (Algol uses 'reference' here)
- `set of` *type* (Pascal only)
- `record of` *type* (C calls this a `struct`)
- `function returning` *type*

And also arbitrary compositions of these, so for example we have the C type

```
int *((*pf[17])(int));
```

which is an array of 17 pointers to functions that take integer arguments and which each return a pointer to an integer.

Slide 9: Typing: Hard to Get Right

The problem with all this was that it turns out to be more complicated to get 'right' than scalar types were. What do I mean by 'right'? Well, remember that the goal here is to enable compile-time checking of the soundness of your program. If it compiles, and there are no type errors, you'd like to be able to feel safe about your code. Now there are two kinds of possible failures wrong here. There might be some real type error that is not caught by the compiler; that's a 'false negative'. It's bad because then you run your program and it does something bizarre, like interpreting `eels` as the number 1.87292264408661e+31.

It's also possible that the compiler might report an error where there isn't one; that's a 'false positive'. It's also bad because you are trying to do something reasonable, and the compiler refuses to compile your program because it is afraid something will go wrong. Another way in which this is bad is that it encourages you to ignore the correct error messages when they do appear---that's the 'Boy Who Cried Wolf' syndrome.

Pascal is noted for crying Wolf. Here is an example of a common problem with the type checking in Pascal:

```
var s : array [1..10] of character;
s := 'hello';                        { You wish }
```

You declare `s` to be an array of ten characters, and then you try to assign the string `hello` to `s`. No, sorry; this is illegal. `hello` is an array of five characters, not ten, so its type does not match that of `s` and you are not allowed to make the assignment, or even to compile the program.

Here is a similar example: We are trying to write a function to emit an error message. We define a `string` type for the argument to this function:

```
type string = array [1..40] of character;
procedure error (c: string)
begin
   write('ERROR: ');
   write(c);
   writeln('');
end;
```

Now you would like to issue an error message this way:

```
error('File not found');
```

but you can't, because the argument is an array of 14 characters, and is is required to be an array of 40 characters. You have to write this instead:

```
error('File not found                          ');
error('Please just kill me, Mr. Wirth!         ');
```

Well, Wirth agrees that this was a mistake. I imagine he fixed it in his later languages, although I am not sure.

Slide 10: Typing: Hard to Get Right

I see the C programmers snickering. Just to show that this is not a problem that is unique to Pascal, here's an example from C.

```
int main(void)
 {
   unsigned char *c;
   float f = 10;

   for (c = (char *)&f; c < sizeof(float) + (char *)&f; c++) {
     printf("%u ",  *c);
   }
   putchar('\n');

   return 0;
 }
```

This is one of the programs that I wrote while I was computing that the representation for `eels` was the same as for 1.87292264408661e+31. It does the opposite conversion, and finds the four-character string whose representation is the same as that of the float `10.0`. When I compiled this with a popular compiler, I got this error message:

```
float.c: In function 'main':
float.c:10: warning: comparison of distinct pointer types
            lacks a cast
```

Now this is very interesting. What does it mean? After some investigation, I discovered that it was complaining about the comparison

```
c < sizeof(float) + (char *)&f
```

Why the complaint? The expression on the right of < has type 'pointer to char', whereas `c` on the left has type 'pointer to unsigned char'.

Now this is a totally spurious error message, because the sizes of `char` and `unsigned char` are guaranteed to be the same, so no harm can come of making the comparison between the pointers.

(Note that this is different from comparing a signed and an unsigned `char` value, which really *is* unsafe; here I am only comparing the *pointers* to such values.)

But okay, let's stipulate that this might be an indication of a real mistake, and that it is worth warning about this so-called problem that can never cause anything bad to actually happen. Fine. Then why, if this is so terrible, why did the compiler complain about the comparison, and remain completely silent about the assignment

```
c = (char *)&f
```

on the same line, which commits the exact same type error? Surely if the pointers are so incompatible that they cannot even be compared, then it must be even worse to assign one to the other?

Well, this is all nitpicking. We could argue all day about `(char *)` versus `(unsigned char *)` and whether it should have emitted one warning or two, but this is ignoring the real failure here. The real failure here is that the entire program is one gigantic type violation; the whole point of the program is to take a `float` and print it out as if it were an array of `char`s. The program is printing out floats as if they were strings, and all the compiler can find to complain about is this ridiculous triviality about the signedness of the characters I want to print out.

I call that a big fat failure.

Slide 11: Static Typing in Pascal and C is a Failure

Okay, my conclusion is that type checking, as practiced by C and Pascal, is a failure. We have seen several examples of basically spurious errors, where the compiler complained about something that was not a problem at all, or should not have been a problem if the type system-had been better designed.

These spurious errors waste the programmer's time; they have to be investigated, and understood, and worked around. Then when the real errors come, the programmer is tempted to ignore them or work around them in the same ways.

As a larger and more general sort of argument to show that the type systems of C and Pascal are failures, I'm going to point out that each language has several mechanisms specifically designed for disabling or otherwise working around the brokenness of the type system. For example:

- Casts `(char *)&f` (C only)
- Automatic conversions (C only)

  ```
  int i;
  i = 1.42857;
  ```

  (`1.42857` is silently truncated here to 1.)
- Variadic functions (C only)
- Union types (Both C and Pascal)

  ```
  var u: case tag: integer of
           0: (intval: integer);
           1: (realval: real);
           2: (stringval: array [1..20] of character);
           3: (boolval: boolean);
         end;
      r : real;

  u.intval = 1428457;
  r = u.realval;
  ```

  We committed a type error here.
- One I forgot to put on the slide: If you define a function in one file and use it in another file, Pascal has no way to check that the types are consistent. This is a hole in the type system. It wouldn't be fair to hold it against Pascal, except that it has been used *and advertised* over the years specifically as a way to evade the requirements of the Pascal type system.

This proliferation of methods for evading or disabling the type systems is demonstration that those type systems are failures. If they worked properly, then why do we need all these ways to evade them?

Slide 12: Coping With Failure

So given my conclusion, that static typing, as implemented by languages like C and Pascal, is a failure, what can we do about it?

One strategy is to simply give up and forget about static typing. This strategy has been very successful. Languages that do this include APL, the perennially popular Lisp, the Unix scanning language AWK, and of course Perl. In fact, Perl gives up even more than these other languages, because instead of raising a type error when you try to add a number to a string, it just silently converts the string to a number and proceeds.

```
+(8/2).".".0.0.0
```

Yields 4.00 in Perl. (This is due to Abigail, who was at the talk.)

So that worked pretty well, and a lot of people draw the conclusion from this that type checking is basically a failure, and that the trend in the future will be away from type checking.

Those that don't come to this conclusion still have the idea that type checking has been pushed as far as it can go and it is at its pinnacle with Pascal and C. If you ask people to name a strongly typed language, they frequently mention Pascal. This is unfortunate, because almost everyone hates the Pascal type system. And why not? It is terrible. But that rotten image is what strong typing has to contend with.

However, there is another strategy that you can use to cope with the failure of Pascal. You can try to do better. Pascal dates from around 1968. In the last thirty years, research has not stood still, and we can in fact do better, as evidenced by languages like ML, Haskell, and Miranda. It may come as a surprise to you that this strategy has *also* worked pretty well. The next part of the talk is going to show that type checking does not have to be a failure, and, contrary to what you might think, Pascal is *not* a strongly typed language, but a weakly typed one.

Slide 13: Strong Typing that Works

We saw from the examples that the typing in C and Pascal failed for several reasons. It was too fine-grained, as in Pascal's useless distinction between an array of twelve characters and an array of thirteen characters. It led to many spurious error messages, which means warnings that are ignored and waste everyone's time. It was too easy to violate the type systems though union types and casts, and it *had* to be so, because of the preponderance of spurious errors. In places it was too coarse-grained, as in C's `structs`; a pointer to any sort of `struct` is required to be equivalent to a pointer to any other sort of `struct`. And finally, they are inconvenient to use because you have to cover your program with zillions of type declarations everywhere.

People are so used to C and Pascal that they take these problems for granted and assume that they are inherent in the idea of static typing. Then what I say that you can solve this problem by making the type system stronger, they are surprised. If Pascal's type system is a problem, then surely a stronger type system is an even bigger problem. But no, that's not right, because most of the problem is with the clumsiness of the system; it is always getting in your way and preventing you from doing what you want. If the type system were smarter, so that it always did what you wanted, you would be happy that it was stronger. Neither C nor Pascal is state of the art; they are both thirty years old, and it shows. All the problems of their type systems are surmountable, not some time in the future, but with languages that are available today.

Slide 14: The ML Programming Language

I am going to show examples from the ML programming language. I picked ML because I felt that it made the point, and because I was most familiar with ML at the time. Since then I've learned Haskell, which is a more recent development; I like it even better than ML. But ML is good enough.

ML dates from about 1970, and was originally a research language used for developing theorem-proving systems. It is very strongly typed, and its type system solves the problems that C and Pascal had.

Slide 15: Types in ML

ML has the usual sorts of scalars: strings, integers (called `int`), real numbers (called `real`), and also a boolean type, called `bool` with values `true` and `false`.

```
17                                int
17.3                              real
"brain d foy"                     string
true                              bool
```

Note that in ML, unlike in C or Pascal, `string` is a basic scalar type, and not an array or a pointer or anything like that.

ML also has *tuple types*, including pairs and triples:

```
(17, "foo")                       int * string
(12.5, 13.5, 9)                   real * real * int
(true, false, true)               bool * bool * bool
```

Note that each of these has its own type, such as `real * real * int`. (ML has record types too, but they are much less frequently used than the tuples.)

ML has lists:

```
[true, false, true]               bool list
[true, false, true, false]        bool list
[1,2,3,4,5]                       int list
["brain", "d", "foy"]             string list
[17, "foo"]                       ILLEGAL
[ [1,2,3], [4,6], [0,233] ]       int list list
```

They are something like lists in Lisp; they have a head and a tail like Lisp lists. But in ML, all the elements of a list must have the same type. This is necessary, or else when you pulled an element out of a list you wouldn't know what you were going to get. In contrast, you can mix types in tuples, because a tuple has a fixed length and each position is typed separately, so you still always know what you are going to get.

```
[]                                'a list
[ [1,2,3], [], [] ]               int list list
[ ["b", "d", "f"], [], [] ]       string list list
```

ML also has *polymorphic types*. The simplest example of a value with a polymorphic type is the empty list, which has type `'a list` The `'a` here is pronounced 'alpha', and is a *type variable* which could be any type at all. That's because the empty list of integers looks the same as the empty list of strings or the empty list of anything else. So it is okay to include the empty list as a member of a list of int lists, or as a member of a list of string lists, as we did above. We'll see more examples of this later.

Finally, these types can be composed arbitrarily. For example, you might have the type `(bool * 'a list) list`, which is the type of lists whose elements are pairs where the first item in the pair is a boolean and the second item is a list of some unspecified type.

Slide 16: Types in ML

In ML, unlike most languages, there are *no* implicit conversions. An int is an int and a real is a real and if you want to add an int to a real, you have to use an explicit function that converts ints to reals.

```
3 + 4.5
```

This is a compile-time error because the types don't match.

```
real(3) + 4.5
```

This works, and the result is the real 7.5.

`real` is a built-in function which takes an `int` argument and yields a `real` result, so its type is `int->real`. The arrow here indicates a *function type*. The types of some other built-in functions are:

```
floor                         real -> int
sqrt                          real -> real
not                           bool -> bool
explode                       string -> string list
```

`explode` takes a string and breaks it up into a list of strings of length 1.

```
mod                           int * int -> int
```

`mod` is declared to be an *infix* operator, so you must write it in between its two `int` arguments, like `12 mod 5` instead of `mod(12,5)`.

```
rev                           'a list -> 'a list
```

Notice that `rev`, which reverses a list, is polymorphic. Its argument is any sort of list, and it produces another list of the same type as its result.

```
::                            'a * 'a list -> 'a list
```

`::` is the 'cons' operator. It takes a head of type `'a`, and a tail, which is a list of `'a`s, and puts together a new list of `'a`s with the specified head and tail. That's how lists are normally constructed, and the usual syntax is defined to be syntactic sugar for a sequence of cons operations. For example

```
1::2::3::[]
```

and

```
[1,2,3]
```

are exactly the same.

Slide 17: Big Deal?

Okay, we've now seen the ML type system. It is rather complicated, and there are a lot of types. But what is the big deal?

The big deal is that in spite of the complexity of the type system, it is all automatic. We do not have to put any type declarations into our programs.

Here's a little story about what it is like to program in ML. We want to write a factorial function, so we type in the following:

```
fun fact 0 = 1
  | fact n = n * fact(n-1);
```

The | here is pronounced 'or'. The first clause says that the factorial of `0` is `1`, and the second clause says that the factorial of some number `n` (other than 0) is `n * fact(n-1)`.

Notice how there are no type declarations. Nevertheless, the compiler sees the 0, and says, ''Oh, `0` is an `int`, so the function must be a function on `int`s.'' This means that `n` must also be an `int`. Since `n` is an `int`, the expression `n-1` is legal.

Now, `*`, the multiplication operator, requires two arguments of the same type, and `n` is an `int`, so the return value from `fact` must also be an `int`. The return value of `1` in the first clause is consistent with that.

Everything has checked out, so the compiler accepts the definition, and prints out:

```
val fact = fn : int -> int
```

This means that `fact` is a function with type `int -> int`. The compiler has figured this out all by itself. Now all you have to do is check to make sure that it is what you were expecting. It is, so you go on to the next function.

If the type deduced by the compiler was not what you expected, it almost always means your program has a bug. Not a pernickety annoying bug like some string that is the wrong length, but a real bug, one you would be glad to find out about because it means that your program was really going to generate the wrong answers.

Slide 18: Type Elaboration Example 2

Here's another example. It's a function to add up the elements of a list.

```
fun sumof []     = 0
  | sumof (h::t) = h + sumof t;
```

The sum of the elements in the empty list is `0`; the sum of the elements in a list of the form `h::t` is computed by adding `h` to the sum of the elements in `t`.

Notice again that there are no type declarations. What does the compiler do with this?

It sees that the argument in the first clause is `[]`, so the function must operate on some kind of list, say an `'a list`, where `'a` is presently unknown. In the second clause, the argument is `h::t`, which must also be an `'a list`, and since the compiler knows the type of `::` it can deduce that `h` must be an `'a` and `t` must be an `'a list`.

Now the return value in the first clause is `0`, which is obviously an `int`, and that means that the function returns an `int`. Since `h` is being added to this return value in the second clause, `h` must also be an `int`. We now know what `'a` represents: It is `int`, and `t` and `h::t` are both `int list`. Everything else is consistent with this. The compiler then prints out

```
val sumof = fn : int list -> int
```

`sumof` is a function that takes an `int list` as argument and produces an `int` as its result. This is exactly correct.

If we were to replace the `0` with a `0.0`, it would have deduced `real list -> real` instead.

If we replaced the `0` with `true`, the compiler would have signaled a type error.

Slide 19: Type Elaboration Example 3: Union Types

Union types were big trouble in Pascal and C, because they let us store a value of one type and read it back again as another type. Let's see how ML handles union types.

```
datatype MyNum = IV of int | FV of real;
```

Here we create a union type; the new type is called a `MyNum`. It can contain an integer or a real value. Every `MyNum` either has the form `IV i` for some `int i`, or `FV f` for some `real f`. We can write either of these in our program and the result is a value of type `MyNum`:

```
val n = IV 5;
```

This creates a variable named `n` bound to the `MyNum` with value `IV 5`. The `IV` is a *constructor* which is a lexical marker for a literal value of `MyNum` type; you can also think of it as a function for converting `int`s into `MyNum`s. `FV` is similar. When we create this value, ML prints out a message that its type is `MyNum`.

This data type really is useful as a union type. We can't make a function that accepts both `int` and `float` arguments, because a function must have exactly one type. But we can certainly make a function that is defined on all values of the type `MyNum`. For example, here is a function that converts a `MyNum` to a `real`:

```
fun Num_to_real (IV i) = real i
  | Num_to_real (FV f) = f ;
```

The compiler deduces the type `MyNum -> real` for this function.

Or we could write a function that takes the square root of a `MyNum`, yielding a new `MyNum` as the result:

```
fun Numsqrt n = FV(sqrt(Num_to_real n));
```

The compiler says this has type `MyNum -> MyNum`.

Or we could construct a function that turns a `MyNum` into an integer, throwing away the decimal part if it has one:

```
fun Numtrunc (IV i) = i
  | Numtrunc (FV f) = trunc f ;
```

if the `MyNum` has the form `IV i`, then we just return `i`; if it has the form `FV f` then we take `f`, which is a `real`, and truncate it and return the result. This function has type `MyNum -> int`.

Slide 20: Union Type Sabotage

Now let's see if we can abuse the union type the way we did in Pascal and C, by storing one type of value into the union and extracting a different type.

Here's our first try:

```
fun intval (IV x) = x;
```

It's a function that takes a `MyNum` in the form `IV x` and just returns the integer `x` value. We suspect that having defined this function, we can pass it a `MyNum` of the form `FV f` and it will return the `real` value `f` but think that it is an `int`. But things start to go wrong right away: We get a non-fatal warning:

```
Warning: match nonexhaustive
          IV x => ...
```

This means that we forgot to define the `intval` function in all possible cases, which is correct. But it also compiles the function and elaborates the correct type, which is `MyNum -> int`.

Now let's go ahead with our plan of handing `intval` a value that it is not prepared to deal with:

```
intval (FV 3.5);
```

ML does not interpret `3.5` as an `int`. Instead, we get a run-time fatal error:

```
        uncaught exception Match...
```

In ML, you can make a union type, but it is *impossible* to store a value of one type and read it back as another type. The `intval` function extracts the integer part of a `MyNum`, but only in a type-safe way; it refuses to work if there was no integer part to extract.

We did get a run-time error, which was a shame. But we got a compile-time advance warning that run-time errors were possible, because we did not define the function for all the possible values it was supposed to operate on. ML can do this sort of case coverage analysis for lists also:

```
fun Nth (0, (h::t)) = h
  | Nth (n, (h::t)) = Nth ((n-1), t) ;

Warning: match nonexhaustive ...
```

Here ML is warning us that our function is undefined when the second component of the tuple is the empty list.

Slide 21: Union Type Sabotage

Let's try again to screw up the union types.

```
fun intval (IV i) = i
  | intval (FV f) = f ;
```

This function takes a `MyNum` and just extracts the data part of it. Maybe ML will get confused and treat both parts the same? No, this doesn't even compile; instead, it raises a type error:

```
Error: rules don't agree (tycon mismatch)
  expected: MyNum -> int
  found:    MyNum -> real
  rule:
    FV x => x
```

The first clause implies that this function is going to take a `MyNum` and produce an `int`. But the second clause says to take a `MyNum` and to produce a `real`. This is inconsistent, and if ML let us define this function we would not know what kind of value we were getting back from it.

Let's make one final try: We'll construct a `MyNum` based on an integer, and try to pass it to the square root function:

```
val n = IV(3);
sqrt(n);
```

No, this fails. We can construct `n`, but when we try to take the square root, ML refuses:

```
Error: operator and operand don't agree (tycon mismatch)
          operator domain: real
          operand:         MyNum
          in expression:
            sqrt (n)
```

It was expecting a `real`, and we gave it a `MyNum`.

Slide 22: Final Note About Union Types

Let's skip this slide; I think it's belaboring the point.

Slide 23: Type Elaboration Example 4

It may not be obvious at first how powerful ML's polymorphic types can be. Here's an example.

```
fun map(f, [])      = []
  | map(f,(h::t))   = f(h)::(map(f, t));
```

This is something like Perl's `map` function. It takes a function `f` and a list, applies `f` to every member of the list, and produces a list of the results. How does ML deduce the type of this function?

The arguments to `map` are `f`, which has some totally unknown type, and `[]` which has type `'a list` for some `'a`. Since the arguments in the second clause must match, the compiler knows that `h` has type `'a` and `t` has type 'a list.

On the right-hand side of the second clause, `h` is used as an argument to `f`, so `f` must be a function of type `'a -> 'b` for some type `'b`, which is pronounced 'beta'. The return value of `f` is used as the first argument to the `::` operator, and the second argument is the return value from `map` itself, so `map` must be returning a `'b list`. This is consistent with the return value from the first clause.

Everything else checks out okay, and no new constraints are discovered, so ML reports that the type is

```
('a -> 'b) * 'a list -> 'b list
```

It takes a pair whose first element is a function of type `('a -> 'b)` and whose second element is a list of `'a`s, and it returns a list of `'b`s. Here `'a` and `'b` could be any types at all. It is now perfectly legal to write

```
map (sqrt, [1.0,2.0,3.0])
```

and get back a list of reals, or to write

```
map (rev, [[1,2,3],[4,5,6],[7,8,9]])
```

and get back a list of `int list`s. `map` will work on lists of any type, but in a type-safe way.

Slide 24: Type Elaboration Example 4 Continued

Actually the `map` function I just showed is not the way you would normally do it. ML supports something called *currying*, which is easier to show than to explain. The normal definition of `map` is like this:

```
fun map f  []      = []
  | map f (h::t)  = (f h)::(map f t);

       val map = fn : ('a -> 'b) -> ('a list -> 'b list)
```

The type here says that `map` gets one argument, which is a function of type `'a -> 'b`, and returns another function, which is of type `'a list -> 'b list`. If we apply `map` to `sqrt`, which has type `real -> real`, we get as the result a function of type `real list -> real list`. We could re-invoke this result function on a `real list` directly:

```
map sqrt [1.0, 2.0, 3.0, 4.0, 5.0];

        val it = [1.0, 1.414, 1.732, 2.0, 2.236] : real list
```

in which case it doesn't look too different from the version on the previous slide. But we can also use the returned function as a value:

```
val sqrtall = map sqrt;

        val sqrtall = fn : real list -> real list
```

Here `map` has transformed the `sqrt` function into a `sqrtall` function that operates on lists of reals:

```
sqrtall [1.0,4.0,9.0];
```

```
val it = [1.0, 2.0, 3.0] : real list
```

So a curried function is more flexible than a regular function: We can invoke it on all of its arguments in the normal way, or just on the first few arguments, in which case it yields a function that is a special case of the original curried function.

Slide 25: Type Elaboration Example 5

I find the ML type system really interesting, and I shouldn't have put so many examples in the talk, because it was too long. But I got carried away. I think this slide has low value, and if I were doing it over I would leave it out.

Slide 26: Life with ML

What's life like as an ML programmer?

As we've seen, ML has a lot of unspectacular successes. Day after day, you get a lot of type errors, and they indicate real problems. I made a lot of typos while I was making up the examples for the slides, omitted parentheses and commas and the like, and even when the result was syntactically correct, the ML type checker usually caught the error. For example, I wrote an example that should have looked like this:

```
fun first (Node(d,Nil      ,_)) = d
  | first (Node(_,left_tree,_)) = first left_tree;
```

But instead I wrote this:

```
fun first Node(d,Nil      ,_) = d
  | first Node(_,left_tree,_) = first left_tree;
```

This is syntactically correct, but doesn't mean what it appears to; it defines a curried function `first` of two arguments, of which the first argument is taken to be the `Node` function. So this would define `first` as a function, which, given the `Node` function, yields a function which, when given one of the specified triples, yielded the value on the right hand side. Don't worry if you don't understand this, because it doesn't make any sense. And because it didn't make any sense, it didn't pass the type checker.

Programming in ML is very pleasant. More than almost any other language I know, you can just write down the definitions of the functions as they come into your head. You don't need to bother with declarations; everything is just figured out for you automatically. And you *do* get a lot of type errors, both actual failures and also places where the type emitted by the compiler is not what you thought it should be. But unlike in C or Pascal, *every* one of those errors indicates a real, serious problem in your program, not merely something you have to groan over and try to work around the compiler's stupidity yet again.

Slide 27: A Spectacular Example

As I said, ML has many unspectacular successes, finding mistake after mistake in your programs every day. But here's a really wonderful example of a spectacular success, which I got from Andrew R.

Koenig of Bell Labs. Koenig wanted to write a merge sort function. To do merge sort, we break our input list into two lists, sort each one separately, and merge the sorted lists together.

For concreteness, let's assume that we're going to write a merge sort function that works on lists of integers only, so we will expect it to end up with type `int list -> int list`.

Slide 28: Splitting

```
fun split []        = ([], [])
  | split [h]       = ([h], [])
  | split (x::y::t) = let val (s1, s2) = split t
                       in (x::s1, y::s2)
                      end
```

Splitting apart the input list is not too hard. You should be able to read off the first two cases. The third case makes use of a feature we haven't seen before: `let`. `let` says to compute the value of `split t`, and to bind the two values in the result to the variables `s1` and `s2`, and then return as the result the expression following the `in`. So in that third case, where the input list is more than one element long, the first two elements get named `x` and `y`, and the tail of the list is `t`. We split up `t`, and then append `x` and `y` to the front of the two result lists, and that's the final answer.

Now to merge two lists back together is about equally simple:

```
fun merge ([], x) = x:int list
  | merge (x, []) = x
  | merge (h1::t1, h2::t2) =
    if h1 < h2    then h1::merge(    t1, h2::t2)
                  else h2::merge(h1::t1,     t2);
```

If either of the input lists is empty, then the result of merging it with the other list is just the other list. And in the third clause, if neither list is empty, we compare the heads of the two lists, and whichever is less gets put on the head of the result.

That `x:int list` in the first clause is a type declaration for `x`. I lied about the type declarations. On certain very rare occasions you have to put one in. Why can't it deduce the type in this case? It *almost* can; the only problem is that < is ambiguous, and could operate on either `int`s or `real`s, and ML has no way to tell which. One way to solve this problem is by requiring an occasional type declaration, as here, to disambiguate the situation; the other solution would have been to use a different symbol for the `real` version of <, in which case < would no longer be ambiguous and the type declaration would be unnecessary. The designers of ML chose the first solution, but it's not clear that it was the right choice. (Some dialects of ML, such as OCaml, do it the other way, and so the type annotation would be unnecessary.)

In any case, `merge` gets the type `int list * int list -> int list`.

Slide 29: Merge Sort

Now we have `split` and `merge`, we can build our `sort` function:

```
fun sort [] = []
```

```
  | sort x  = let val (p, q) = split x
               in merge (sort p, sort q)
              end;
```

The result of sorting the empty list is just the empty list. And to compute the result of sorting some other list `x`, we first split `x` into `p` and `q`, we sort `p` and `q`, and we merge the results.

All fine, except that the type that comes out of the compiler is a shock. We expected it to be `int list -> int list`, and instead we get `'a list -> int list`.

This says that we can put in any kind of list at all; that is a surprise because we hardwired the < operator into `merge`, and it shouldn't work properly for a list of, say, `MyNum`. The type also says that no matter what we put in, even a list of `MyNum`s, we will always get an `int list` out. This is impossible.

Slide 30: Huh???

Has the vaunted ML type checker screwed up? It certainly has produced an impossible result. Will we have to disable it or work around it to get our program to compile, like we had to in C and Pascal? No. The type checker has the type correct in this case. The fact that the type is not what we expected means that there is a bug in our program, not in the type checker.

How can the impossible occur? One way is if it never actually occurs. That's why you can promise to renew your NSI registration when Hell freezes over. Are you going to have to renew? No, because Hell is not going to freeze over.

If we put a list of strings into our `sort` function, the compiler says that when it returns, it will have turned it into a list of `int`s. This is impossible. One possible explanation is that the `sort` function never returns. And in fact this is the case. The `sort` function, as we wrote it, goes into an infinite loop on any input except the empty list.

This means that the *type checker* has found an *infinite loop bug* in our program. Ponder that for a minute.

Now try to imagine the Pascal type checker detecting an infinite loop.

Slide 31: Where's the Bug?

Where is the bug, anyway?

Suppose we are trying to sort the one-element list `[x]`.

It's not empty, so we split it into two lists, sort the split lists, and merge them back together.

But when we split the list, it splits into `[x]` and `[]`. Sorting `[]` works out fine, but when we try to sort `[x]` it does indeed put us into an infinite loop.

We need to add another special case to `sort` to tell it how to sort a one-element list without recursing. We just add another clause to the definition:

```
fun sort []  = []
  | sort [x] = [x]
  | sort x   = let val (p, q) = split x
                in merge (sort p, sort q)
               end;
```

Now the type is

```
int list -> int list
```

as we expected.

We could make the `sort` function a little better by having it take the comparison operator as an argument. Then the type becomes

```
('a * 'a -> bool) -> 'a list -> 'a list
```

Incidentally we can get rid of the type declaration in `merge` if we do this. Then we can use our `sort` function like this:

```
sort string_less_than ["fred", "bill", "andrew"];
        val ["andrew", "bill", "fred"] : string list
```

or we can use it to manufacture sorting functions:

```
val sort_strings = sort string_less_than;
        val sort_strings = fn: string list -> string list

sort_strings ["fred", "bill", "andrew"];
        val ["andrew", "bill", "fred"] : string list
```

Slide 32: Type Checking in Perl

OK, this was supposed to be a talk about Perl, and I finished talking for two hours about Cobol, Fortran, Lisp, and ML with hardly a word about Perl. This part of the talk is the part about Perl.

We saw that ML is pretty cool, and that it has this fabulous type-checking system, and that type-checking systems don't have to suck, which is quite a surprise after seeing Pascal. Perl is also pretty cool, and has a history of adopting good ideas from other languages, so maybe we can borrow the ML type checker for Perl.

Well, it's a nice thought, but I really can't see how to make it work. Perl and ML are just too different, and Perl has just gone too far in the other direction. Perl hardly has any type checking at all, not even at run time.

Slide 33: Type Checking in Perl: Gaaaack

Here are some of the things that you'd have to give up in Perl if you wanted to put in ML-style type checking.

```
while (<>) {
```

```
    $total += $_;
  }
```

Here `$_` is a string, read from a file, and is being implicitly converted to a number. That is going to foil your type checking.

```
  @a = (1, 3.1416, "A painted lotus flower");
```

Here we have a list with many different types of items in it. You cannot know at compile time what type `$a[1]` is going to turn out to have later on. In an ML list, all the elements have the same type.

```
  @a = localtime();
  $s = localtime();
```

Here the `localtime()` function returns two totally different things depending on context. That is going to give the ML type checker fits. More to the point, if any possible use is legal, there are no errors to raise, and there is no point of checking if there are no erroneous circumstances to report.

```
  print ;
```

Here's a good one. We print out the value of `$_`, regardless of whether it is a number or a string or a reference or whatever, and it is automatically converted to a string. If you want type checking, you want this to be an error unless `$_` is a string. But who would want that?

```
  if (@a) { ... }
  $s = @a;
```

Here we're using an array like a scalar. Again, if you want this to be an error, you don't want Perl. And if you don't want this to be an error, then what do you mean when you say you want type checking?

```
  $a[100000] = 'big!';
  $h{nosuchkey} = 'It means "to chop"';
```

Finally, here we are assigning values to out-of-range and nonexistent array and hash elements. You could make this into an error, but the result wouldn't be very Perl-like.

The conclusion here: Type checking is nice, and so is Perl, but not in the same way. Adding type-checking to Perl would be like making a hot fudge sundae with mashed potatoes instead of ice cream and with gravy instead of fudge. It might be perfectly edible, but it wouldn't be a hot fudge sundae. Perl's present design is simply not consistent with most of the goals of compile-time type checking.

Slide 34: Run-Time Type Checking

Having said that, let's look at some of the places where Perl *does* manage to do a little type checking. The most important is the

```
  use strict 'refs';
```

declaration. Normally, Perl is happy to convert a string to a reference; if you use the string `foo` where Perl expects to see a hash reference, for example, `foo` is taken to be a reference to the hash `%foo`. The `use strict 'refs'` declaration makes this into a run-time error.

The `-w` optional warning flag also enables warnings for some type errors. For example, if `-w` is enabled, then this code:

```
$s = 'some string';
$r = $s + 13;
```

generates the warning `Argument "some string" isn't numeric in add`.

Slide 35: Compile-Time Type Checking

Perl is even getting a little bit of compile-time type checking. Here's the idea behind this very new feature: People use hashes for objects all the time in Perl. The hash keys are member data names, and the corresponding values are the object's member data. The member data is then looked up in the hash by key, every time the program does a member data access.

This is inefficient, because the object probably only has a few keys, and they have to get looked up over and over again. Also, it wastes memory, because Perl's hashes are designed to be expanded dynamically and to support all sorts of operations like `each` and `delete` that are actually irrelevant for objects. So we would like to use an array instead of a hash, and look up the attributes by number instead of by name. This will save time and memory.

Of course, the down side is that it is now hard to remember which number goes with which attribute. So we will have a compromise: The object will be an array, but the first element of the array will be a hash that maps the attribute names to array indices. If the expected keys are declared at compile time, you can write `$object->{NAME}` and the compiler can pretend that you wrote `$object->[3]` instead. For keys that vary at run-time, Perl can look up the key in the hash at run time and use that to figure out the right index.

Slide 36: Compile-Time Type Checking

So here is an example. A typical object looks like this:

```
$octopus =
   [ { tentacles => 1, hearts => 2, favorite_food => 3 },
     8,
     3,
     "crab cakes",
   ];
```

This octopus has eight tentacles, three hearts, and its favorite food is crab cakes. We can write

```
$octopus->{tentacles};
```

and get 8, because when the Perl sees `$octopus->{tentacles}` it looks up `tentacles` in the `$octopus->[0]` hash, finds the index 1, and then returns `$octopus->[1]`. Similarly,

```
$octopus->{noses} = 1;
```

can generate a run-time error. (``No such array field'')

Slide 37: Compile-Time Type Checking

Now for the compile-time checking. We have to warn the compiler which attribute names to expect. We do that with two declarations. First, in the package that defines our `Octopus` class, we include a `use fields` declaration:

```
package Octopus;
use fields qw(tentacles hearts favorite_food);
sub new { ... }
```

This declares the field names `tentacles`, `hearts`, and `favorite_food` so that they can be resolved at compile time.

Now, in the main program, we have to warn Perl that a particular object is an `Octopus`:

```
use Octopus;
my $fenchurch = new Octopus;
my Octopus $junko = new Octopus;
```

Notice the type declaration in the third line; this is new.

```
$fenchurch->{tentacles} = 8;
```

This is resolved at run-time, by looking up `tentacles` in the hash `$fenchurch->[0]`, and then indexing `$fenchurch` with the resulting number.

```
$junko->{tentacles} = 8;
```

However, because of the declaration, this is simply resolved to `$junko->[1] = 8` at compile time, as if we had written it that way all along. So the `$junko` version is much more efficient than the `$fenchurch` version, even though they look the same.

This code will generate a run-time error `No such array field`:

```
$fenchurch->{noses} = 1;
```

because `$fenchurch->[0]` does not contain an entry for `noses`. But the corresponding code for `$junko` will generate a *compile-time* error:

```
$junko->{noses} = 1;
```

This says `No such field "noses" in variable $var of type Octopus`.

Conclusion

1. Type checking in C and Pascal sucks.

2. Just because static type checking is a failure in C and Pascal doesn't mean you have to give up on

the idea.

3. Languages like ML have powerful compile-time type checking that is successful beyond the wildest imaginings of people who suffered from Pascal.

4. It is probably impossible to get static, ML-like type checking into Perl without altering it beyond recognition.

5. However, Perl does have some type checking mechanisms, and more are coming up.

Return to: Universe of Discourse main page | Perl Paraphernalia | Yak Yak Page

mjd-perl-yak-id-i0n+bd8c8jw+@plover.com