

## Type Abstraction

Suppose that we wanted to define a new abstract type and the operations that could be performed on it.

Here's how we might add this to ELL (the syntax is in the book):

```
lettype myint = int
  myint zero () = 1
  myint succ (myint x) = add1(x)
  myint pred (myint x) = sub1(x)
  bool iszero (myint x) = zero?(sub1(x))
in (iszero? (succ (succ (zero))))
```

The idea is that we define the name of the new abstract type (`myint`), what type it really is (`int`), and the operations you can perform on it.

How would we implement enforcing the type abstraction?

Who gets to know that `myint` is an `int`, and who doesn't?

## Type Inference

With type inference, the programmer doesn't have to declare the types of things—the interpreter/compiler *infers* the types of things from how they are used.

One way of thinking of type inference is in terms of *type equations*:

- Every time we see a new variable or a new expression that returns a value, we make up a new type (a *type variable*) for that variable or expression
- Every time we have a compound expression (something with subexpressions that we type check), we generate a *type equation*
- We then go through the type equations, inferring the type variables until we have them all resolved—this is called *unification*.

## Example

Expression to type check:

```
proc (f, x) (f +(1, x) zero?(x))
```

Let's make up type variables for all of the types (subexpressions) involved here:

The result of the body of the proc	t1
The type of the variable f	tf
The type of the variable x	tx
The type of the subexpression +(1, x)	t2
The type of the subexpression zero?(x)	t3
The type of the proc expression	(tf * tx) → t1

Already known:

The type of +	(int * int) → int
The type of zero?	(int) → bool

**Example (cont'd)**

Type equations generated when type checking each subexpression:

Subexpression $+(1, x)$	$(\text{int} * \text{int}) \rightarrow \text{int} = (\text{int} * tx) \rightarrow t2$
Subexpression $\text{zero?}(x)$	$(\text{int}) \rightarrow \text{bool} = (\text{int}) \rightarrow t3$
Subexpression $(f \ +(1, x) \ \text{zero?}(x))$	$tf = (t2 * t3) \rightarrow t1$

The process from here is to go through the type equations (repeatedly if necessary) to fill in all of the unknown type variables.

In this case, we can determine all of the types except  $t1$ —this is said to be *polymorphic on type  $t1$* . (In other words, this `proc` works regardless of the return type of the actual parameter passed to `f`).

## Common Implementation

- Generate a new “unknown” value ? in the type environments for new variables.
- In the process of type checking, any time you have a type equality check of this form:

$$t = ?$$

the check succeeds and as a side effect changes the unknown type to the known one.

- Repeatedly type check the tree until all of the information propagates throughout the tree (information may travel up one branch and then down another).