Thinking Languages

Part 1

Stat 750 Fall 2024

29 Aug 2024

1 Motivation

If you saw Denis Villeneuve's movie *Arrival* (or read Ted Chiang's wonderful story on which it is based), you will be familiar with the idea of *linguistic relativity*: that the structure of a language influences and subtly shapes how a speaker thinks about and perceives reality. Various versions of this idea have been proposed by linguists and continue to be argued over to this day. But when it comes to using abstractions for solving problems, there certainly seems to be something to it.

Writing software, proving theorems, reasoning with logic: at a fundamental level these are the *same* pursuit.[1] With mathematical work, we readily bring new abstractions, new notation, and new patterns to bear in solving problems, and we see their power for shaping how we think about and solve those problems. When it comes to programming, however, we tend to find a comfort zone with a language, with tools, with an ecosystem of resources that we can combine to manage the complexity that software tends to breed.

Yet there is value in taking a different perspective, finding new abstractions and ideas that can change how we think about solving problems. To that end, we will develop in our discussions three **thinking languages** for talking about problems and designing solutions.

- Thinking Language 1 is focused on using *types* to elegantly represent the concepts and entities that we are working with so and inform our design. It is spare and mathematical and allows us to express a variety of deep ideas. It is inspired by several beautiful (real) languages in current use, such as Haskell, Unison, and Idris.

[1]Really they are.

- Thinking Language 2 is an "imperative" pseudo-code for cleanly describing the steps in algorithms. It is relatively loose and informal putting clarity above syntax. It is inspired by scripting languages like Python and Julia.

- Thinking Language 3 is a diagrammatic language that offers a suprisingly powerful tool for thinking about systems. Variants of this language have been used across a wide spectrum of fields, from quantum physics to operations research. We will develop this primarily later in the semester.

Each language provides a way to *express ideas* about our programs and about the entities and operations that are using. They are tools for design and problem solving. We will abbreviate these as `TL1`, `TL2`, and `TL3`.

This document serves as a brief introduction to TL1. We will see more of it (and the others) as we proceed. The goal here is not to focus unduly on syntax and completeness but rather to get the gist of how we can express ideas in this language. So don't feel you need to remember every detail; it will become more natural as we use it. There will be some unfamiliar syntax here, but remember that unfamiliar need not mean complex.

2 Syntax for Function Evaluation

The most common way to denote the evaluation of functions, in mathematics and in programming, is to give the name of the function and then give the function's arguments in a comma-separated list delimited by ()s. So, we write $f(), g(x), h(x, y, z)$. Languages like R, Python, and C use this "infix style", with embellishments like named and default arguments.

But there are other traditions. Consider applying a matrix to a vector: Mv is the *evaluation* of a linear transformation, uMv is a quadratic form. We eschew parentheses (and commas). This "operator style" is used in several mathematical areas and in a number of programming languages.

And programming languages in the LISP tradition use a "prefix style" to represent function evaluation. Instead of `f(x, y, z)` we write `(f x y z)` with the parentheses on the outside and no (or optional) commas. This applies even to operators, e.g., `(+ 2 2 4)`.

The different styles have benefits in the contexts of particular languages or mathematical areas. While these surface-level differences are salient, they are relatively minor when it comes to it.

`TL1` uses the operator style, with space-separated arguments and no outer parentheses. Function evaluation has high precedence, so parentheses may be needed for disambiguating order of operations. This makes for a looser definitional style that interacts well with the *pattern-matching* and *currying* that are described below.

So given a function `f`, we evaluate it with `f 4`, for example. With multiple arguments, we would write `f x y z`. In addition, `TL1` allows the definition of a wide array of operators. For instance, we use `|>` as a "pipe" operator taking a value and passing it to a function `4 |> h |> g |> f` is the same as `f (g (h 4))`. Operators can be used as regular functions by surrounding them in parentheses, e.g., `(+) 2 3` is the same as `2 + 3`. And functions of two arguments can be used as infix operators by surrounding them in backquotes, e.g., ``4 `g` 10``. R shares these capabilities, though with a more limited set of operators (surrounded by %s).

3 Types

When we program in a language like C, we assign "data types" to variables like

```
int index;
double distance;
int counts[10];
```

This tells the compiler how to store the associated data in memory and also what operations we are allowed to perform on the variables.

This embodies two separate ideas: one practical – how to store the data in memory – and one conceptual – what do these entities mean. While the practical level is important for compilers and sometimes for optimizing our code, the conceptual level is our focus. The type of an entity will tell us what the data means for our program, what operations we can perform on it, and what laws or invariants the data must satisfy. A goal of the type system is to allow types that make ***invalid states unrepresentable*** and ***make the semantics salient***. Types can help us think about our computations for design, analysis, and optimization.

Types are a way to categorize data based on the how it is used and the meaning we ascribe to it. Paying attention to types serves several valuable purposes:

1. Explicit types let the compiler (and sometimes you) optimize your code to improve time and memory performance.

2. Explicit types help the compiler and IDE (and you) identify errors *before* your code is run, potentially improving productivity and correctness.

3. Rich and explicit types make salient the concepts and *algebra* underlying your problem and your code, making it easier to think about the problem, design the code, and reuse or generalize the code to other contexts.

Many discussions of types focus on #2, but I think #3 is critical, and that's how we will use types – to make it easier to solve problems. This is a design approach that has tried and tested, and I am confident that with practice, you will find it useful.

A simple model is that a type defines a *set* of possible values, where every element in the same set has the designated properties and semantics. So we might have a type `Integer` that represents arbitrary integer values (with not bound on size as in Python) and represents the set of integers. But we might also have a type `Int` that represents 64-bit machine integers. These are different sets and have subtly different semantics; for example, the latter can "overflow" if we increase an `Int` beyond what it can hold.

We would like to expand the idea of sets to include the operations and laws that values of these types must satisfy. In that sense, we can think of types as *spaces* of values.

3.1 Specifying Types in TL1

If we have a value `x` that has a type `T`, we write `x : T`. The `:` is read as "has type." If we think of types as sets, this is analogous to saying that that the value `x` belongs to the set `T`.

The simplest possible type is analogous to the empty set. We call this type `Void`. It is a type that has *no realizable values.* Although we cannot construct any values with this type, We can, however, make some function that uses it

```
absurd : Void -> a
```

This is a function that takes a value of type `Void` and returns a value of type `a`. The problem of course is that there *are no values of type* `Void`, which makes this oddly well-defined. This reflects the mathematical fact that there is a single function from the empty set to any other set.

The next simplest type has only one possible value. We call this type `Unit`. Because `Unit` has only one value, for every type `a`, there is a unique function of type from `a` to `Unit`. Notice also that there is such a function for every possible value of type `a`; in fact, these functions and the values are in one-to-one correspondence.

Here we first encounter a key point. Types are just *values* in some larger (meta-)type called `Type`. So, `Unit` has type `Type`. To use the value of this type in a program, we need to define its value. In `TL1`, we write:

```
type Unit = ()
```

This defines a type (on the left) and the value of that type (on the right). We have `Unit : Type` and `() : Unit`.

We call `Unit` a **type constructor** because it gives us a type. In this case, it doesn't do much; it just is the type itself. (But stay tuned.) We call `()` a **data constructor** because it gives us a piece of data of a specified type, a *value* that we can use. Again, in this case, there's not much to it, `()` is itself the value.

Think of both the type and data constructors in this case as *nullary functions*, i.e., functions that take no argument. A function of no arguments with a particular codomain can be thought of as having a singleton set as its domain. The set of such functions with the same codomain is isomorphic to the codomain.

The next most complicated type has two values. We give this its common name:

```
type Boolean = False | True
```

This says that a value of type `Boolean` is *either* the value `True` *or* the value `False`. (The `|` is pronounced "or.") Again, `Boolean : Type` and `False : Boolean` and `True : Boolean`. `Boolean` is the type constructor, we write "`Boolean`" to specify the type, and `False` and `True` are data constructors. There is an extended form of this definition that we can use interchangeably:

```
type Boolean where
    True : Boolean
    False : Boolean
```

which simply lists out all the data constructors for the type.

We can use this mechanism to define types that have any specific values:

```
type BasicColor = Red | Green | Blue
type CardinalDirection = North | South | East | West
```

and so on. We use these types by matching the patterns, as described in the next section. In set terms, these types with `|` are analogous to *disjoint unions*. (See F.1.3.) Some languages (e.g., Python, Rust) call these enum types; we will call them *sum types*. Notice that any simple type with two nullary data constructors is isomorphic to `Boolean`, e.g.,

```
type Horizontal = Left | Right
type Vertical = Up | Down
type Outcome = Win | Lose
```

Despite this isomorphism, we prefer to name types conceptually rather than just using `Boolean` for all such choices.

Up to now, our constructors haven't done much construction; they are just values (i.e., nullary functions). Let's look at defining a type that is *parametrized* by another type.

A good example is an "optional" value that has some type *but could be a missing value instead.*

```
type Maybe a = None | Some a
```

Here `a` is *type variable* – a variable whose value is a `Type`, which we typically denote with lowercase letters. This definition says that a value of type `Maybe a` is either the value `None` or a value `Some x` where `x : a`. The extended form of the definition is clearer in some ways, if less concise:

```
type Maybe a where
  None : Maybe a
  Some : a -> Maybe a
```

The the type `Maybe a` is a *concept*: it represents a quantity of a specific tyep `a` that *may not be present*. We cannot elide the distinction between Some and None; it is essential to the concept. This concept is ubiquitous, and this type lets us reason about it. Whenever you can have a value or some other "null" value (like `NA` in R or `None` in Python), you have a value of type `Maybe a`.

We think of the type constructor `Maybe` as a ***function*** that takes a single `Type` as input and returns a `Type`. The type variable `a` is the parameter to this function and is local to this definition; changing it to `b` gives the same type. The `->` notation specifies a function type, as described below. `Some` takes a value of type `a` and returns a value of type `Maybe a`.

As above, there are two types of *constructors* here. A **type constructor** builds a *type*, a function that returns a value of type `Type`. The type constructor `Unit` above is a nullary function that returns a type, which we represent as a value of type `Type`. `Unit` gives that single type. Here, `Maybe` is a unary function that returns a type. It takes a type `a` and returns (i.e., constructs) the type `Maybe a`. So for example, we can construct types `Maybe Integer` and `Maybe String`. We will allow type constructors to take inputs that are not types, as we will discuss in more detail in Part 2.

A **data constructor** builds *values* of a particular type. Here, `None` and `Some` are data constructors, the former is just a value (i.e., it takes no arguments), and the latter is a function that converts a *value* of type `a` to a *value* of type `Maybe a`. So, if `x :  a`, then `Some x` is a `Maybe a` value. For example, `Some 4` has type `Maybe Integer` and `Some "foo"` has type `Maybe String`. We will see how to use values of such types in the next section.

When a type has a single data constructor, we allow the type constructor and data constructor to have *the same name* because it is always possible to infer from context when we want a type and when we want a value. We will use this freedom often, especially with records.

3.1.1 Product Types: Tuples and Records

If `a`, `b`, `c`, `d` et cetera are types then `(a, b)`, `(a, b, c)`, `(a, b, c, d)`, et cetera are *tuple* types, here pairs, triples, and 4-tuples with components of the respective types. Note that these are distinct types. We use ()s for both the type constructor and data constructor, so `(Int, String)` is a type and `(4, "foo")` is a value of that type.

Related to tuples are *records*, which are like tuples but with named fields. For example,

```
type Employee = record Employee where
                    name : String
                    dept : Department
                    asst : Employee
```

defines a type `Employee` with three fields, `name`, `dept`, or `asst` of the relevant types. We can write records equivalently in more compact form if desired

```
type State = record State { current : Int, best : Int }
```

In both these examples, the type constructors and data constructors have the same name, but that is optional. To construct a value of type `Employee`, we specify the fields by name in any order

```
Employee {name="stan", dept=Research, asst=x}
```

where the data-constructor name is necessary. If `e : Employee`, we can access its fields either directly, e.g., `e.name` and `e.dept`, or with accessor functions `Employee.name e` and `Employee.dept e`. (We can just write `name e` and `dept e` when there is no ambiguity.)

Tuples and record correspond to the Cartesian product of sets, and as such we call them *product types*.

3.1.2 Function Types

Function types describe sets of *functions*, and for these we need to give the types of the domain and codomain. The type `a -> b` is the type of functions that take a value of type `a` as input and return a value of type `b`. Here the type constructor is an operator, and as with other operators, we can treat it as a function by surrounding it in ()s: `(->) a b` is equivalently.

The `->` operator is right associative, so the same notation extends to multiple arguments with more arrows: `a -> b -> c`, `a -> b -> c -> d`, et cetera are types of functions that take two, three, et cetera arguments. For example, `a -> b -> c` is a function that takes two values, of type `a` and `b`, and returns a value of type `c`. Right associativity menas that `a -> b -> c` is equivalent to `a -> (b -> c)` and `a -> b -> c -> d` to `a -> (b -> (c -> d))`. The meaning of this and the reason for the `->` notation, besides economy, will be clarified when we discuss *currying* below.

For the moment, we are imagining that we are describing pure and total functions that always return the same value given the same inputs and that always terminate. In reality, we often use functions that have *effects* like producing output, changing global state, or non-termination. We can include effects in the types and discuss that in Part 2.

3.1.3 Recursive Types

A *recursive type* is one specified in terms of the same type. A classic example is the singly linked list:

```
type List a = Nil | Cons a (List a)
```

This specifies that a list of `a`'s is either the value `Nil`, meaning empty, or a value `Cons first rest` containing the head of the list `first : a` and a list of the remaining elements `rest : List a`.

Another example is a binary tree. A binary tree is either a solitary node or a node with two associated binary trees, one on the left and one on the right. One way to represent that as a type is:

```
type Tree a = Leaf | Node (Tree a) a (Tree a)
```

This describes binary trees holding values of type `a`. Notice that the second data constructor takes three inputs. The description of this type could be written instead in extended form as

```
type Tree a where
    Leaf : Tree a
    Node : Tree a -> a -> Tree a -> Tree a
```

3.1.4 Containers

We use `[a]` for a generic sequence of values of type `a`, which is essentially equivalent to `List a` defined above.

Sequences with more specialized requirements generally get their own type. For example, we might want to indicate that a list is non-empty

```
type NonEmptyList a = NonEmptyList a (List a)
```

Alternatively, we might want to specify performance properties for certain operations through the type, such as `Array` (constant-time access), `Queue` (constant-time removal from front and addition from back), or `Deque` (constant-time addition and remove at front and back). , and queues. allow an array's length to be part of its type

3.1.5 Traits, Constraints, and Quantifiers

For the types we have seen so far, those with type parameters like `Maybe a` or `List b`, the type variable can stand for any type. But that is not always what we want. We sometimes need to impose constraints on the type variables for the types and their semantics to make sense.

As an example, if we have a function `aggregate` that combines elements of a list into a single value, we might want to require the elements of the list to be monoidal values:

```
aggregate : Monoid a => List a -> a
```

Here, the condition on the left of the `=>` is a *constraint* on the type at the right, and `Monoid a` is a statement that type `a` must have the `Monoid` *trait*. This means that there is a special value `munit : a` (the identity or unit element) and an operator `(<>) : a -> a -> a` (the assocaitive operator). We can use the same names `munit` and `<>` for all types with the Monoid trait because we can infer from context which type is relevant. As a preview, we might implement `aggregate` as

```
    aggregate Nil = munit
    aggregate (Cons x xs) = x <> aggregate xs
```

For an empty list, we get the monoidal unit; otherwise, we combine the first element of the list with the aggregate of the rest of the list using the monoidal operator. See the next section for details.

If there are multiple constraints, we list them separated by commas and delimited by a pair of ()s, e.g., `(Ordered a, Monoid a) => ...`.

Different languages implement traits differently and with different names Python uses classes and protocols with method implementations. R can use generic functions or more traditional classes like Python. Java uses mixins – classes that implement capabilities that other classes can implement. Clojure calls them protocols with the implementation to use determined by the inspected type of the first element. Rust (traits), Scala (traits), and Haskell (type classes) use type inference to select the implementation automatically. For the purposes of design and description, the ideas are the same.

Examples of common traits include `Eq` for types that support equality comparison, `Ordered` for types that support a partial order `<=` and related operations (including min and max), `Numeric` for types that are some form of numbers. We will touch on the definition of traits in Part 2. The basic idea is that a trait is defined by a set of operations and laws. A type with a particular trait must provide implementations of the operations and (in principle) proofs of the laws. Because trait definitions can also include constraints, traits can be arranged in a hierarchy. For example, the `Numeric` trait requires the `Semiring` constraint as numbers must support semiring operations at the minimum.

We most often apply constraints on functions, but we can do so with type definitions as well. For example,

```
    type Semiring a => Matrix a (n : Nat) (m : Nat)
```

defines the type of an $n \times m$ matrix whose entries must belong to some semiring. The type parametrs `n` and `m` are not values of type `Type` but belong to a (lifted) type of natural numbers, so we specify that explicitly. We could write `Matrix (a : Type) (n : Nat) (m : Nat)` but `Type` is the default for type parameters so that just adds noise. We treat a few special types like `Unit`, `Boolean`, `Nat`, `PosNat`, `Integer`, and certain types of `Lists` as having lifted versions that be used this way. (Others can be used as well, which we will touch on in Part 2.) For example, the definition of this function

```
join : Vector a (n : Nat) -> Vector a (m : Nat) -> Vector a (n + m)
```

is valid and enforces the semantics of concatenating vectors. (To use it in practice, we would need a proof of the length, which can be done, but we are using this for thinking not programming, so the clear type of the function is purely a benefit.)

One additional feature we will sometimes need are local type variables that are universally quantified. For example, when discussing folds, we saw the type

```
type Fold v w r = forall a.
    record Fold where
        lift : (v -> w)
        step : (a -> w -> a)
        init : a
        done : (a -> r)
```

The `forall a.` introduces a local type variable that can be anything and will be determined by inference whenever the `Fold` data record is constructed. There is an implicit `forall` when we define function types like `a -> b`.

3.1.6 Higher-Order Types

One especially useful feature of the type system we are building here is that we can apply these same ideas to any types, even those constructed from other types.

In class, we talked about the fold pattern and the operation of folding a sequence of values. How do we represent types that be folded in this way? Lists, check. Arrays, check. What about streams of data from the network? Or trees? There are many possibilities.

Consider the function

```
foldMonoid : (Monoid a, Foldable t) => t a -> a
```

This takes a container that we can fold over and that contains monoidal values and combines all the values (using `munit` and `<>`) to produce a combination. The particularly interesting here is that the type variable `t` is a container, i.e., it has type `Type -> Type` and we put a constraint on it through a `Foldable` trait. This `t` is a *higher-order type*, and we can use it as we do other type variables.

The *type* `t a` has type `Type` and represents the container. This allows us to define the operation of folding over a wide variety of containers with the same code.

3.1.7 Flexibility and Laws

This is a *thinking* language, so we do not have to be completely formal. We can where needed specify ideas through types even if the practical definition of those types is complicated. For example, we could write

```
Numeric a => StochasticMatrix a (n : PosNat)
```

to represent the type of $n \times n$ stochastic matrices with entries of a numeric type `a`. This satisfies specific laws: it's entries are non-negative and rows sum to one. Defining such a type in a concrete language is complicated, but we can use this idea as needed.

In general, we want a type system that can make invalid values unrepresentable. We would like the types to encompass the laws and operations that describe those values. In most current languages, this is difficult to do in practice, but we are not bound by such concerns here.

4 Functions, Pattern Matching, and Locals

We have seen how to apply functions but not how to define them. We have seen how to construct types but not how to "deconstruct" them and use them. These two ideas are related.

We start with defining functions. Although much of our use of TL1 will focus on defining the types of entities, it will also be useful to be able to express functions and operations in the language.

Functions are defined via simple expressions. We first give the type of the function and then specify one or more expressions that define it. For example,

```
inc : Integer -> Integer
inc x = x + 1
```

We could even generalize the type with the same definition.

```
inc : Numeric a => a -> a
inc x = x + 1
```

Specific values of the input can be specified as part of the defintion. For instance,

```
fib : Nat -> Nat
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

The order of the expressions matters, as the third line will not be used when the argument is 0 or 1. (There are better ways to define `fib` here; this is to illustrate the syntax.) We can also give *guard expressions* to determine what part of the definition applies

```
thresh9 : Integer -> Integer
thresh9 x | x < -9 = -9
          | x > 9  = 9
          | otherwise = x
```

After each `|` is a Boolean expression in terms of the arguments where `otherwise` is just equal to True. The expressions are tested in order and the definition with the first true guard is used. Another example for illustrative purposes

```
choose : Nat -> Nat -> Nat
choose _ 0 = 1
choose n 1 = n
choose n k
   | k <= n = choose (n - 1) k + choose (n - 1) (k - 1)
   | k > n  = 0
```

We define `choose` as though we were writing it on the blackboard: If the second argument is 0, return 1; no matter what the first argument is. If the second argument is 1, we just return the first argument. Otherwise, we check a condition. If `k > n`, we return 0; else, we compute the value based on the value of the function for smaller arguments. Here's a more general and efficient version

```
choose : Numeric a => a -> Integer -> a
choose _ 0 = 1
choose r 1 = r
choose r k
   | k < 0  = 0
   | otherwise = (r / k) * choose (r - 1) (k - 1)
```

When the input is a type defined by a data constructor, we can *pattern match* on the input to deconstruct the components. For example:

```
maybeInc : Numeric a => Maybe a -> Maybe a
maybeInc None = None
maybeInc (Some x) = Some (x + 1)
```

And similarly

```
head : List a -> Maybe a
head Nil = None
head (Cons x _) = x
```

where _ is used as a placeholder. And again

```
crossProduct : (Real, Real, Real) -> (Real, Real, Real)
crossProduct (a, b, c) (x, y, z) = ( b * z - c * y
                                   , c * x - a * z
                                   , a * y - b * x
                                   )
```

This pattern matching can be used with a `match` expression anywhere inside a function as well:

```
f : Maybe Int -> Int
f z = match z where
        None -> -999
        Some x -> x
```

takes a `z : Maybe Int` and returns an `Int`. Each case in the `match` produces the value after the `->`. Guard expressions before the `->` are allowed here analogously to those in a function definition.

Conditionals are *expressions* `if c then x else y` not statements:

```
relu x = if x > 0 then x else 0
```

We can define local values (which can be functions!) either before an expression with `let` or at the end of a function definition with `where`.

```
centerAbs x c = let z = abs (x - c) in z * z
```

This defines one or more local variables that are defined in the scope of the expression after `in`. In general, we use whitespace similarly to (but more loosely than) Python: subsidiary expressions must be consistently indented more than the parent expression. But we allow multiple expressions in {}s separated by ;s if desired.

```
recurrence f n = let u = f(n - 3)
                     v = f(n - 2)
                     w = f(n - 1)
                 in
                     2 * u + 3 * v - w
-- or more ugly:   (this is a comment btw)
```

```
recurrence f n = let {u = f(n - 3); v = f(n - 2) w = f(n - 1)}
                 in
                     2 * u + 3 * v - w
```

Here the `in` is part of the `let` construct and so can have equal indentation, though the entire construct can also be on one line. The braces are not mistaken for a record here both because of ;s and because there is no data-constructor name.

We can also define expressions local to an entire function with a `where` clause.

```
shift f n = shifted
  where
      shifted x = f(x - n)
```

We can specify the types of such functions if it is not obvious (or even if it is):

```
reverse : List a -> List a
reverse lst = rev lst []
  where
    rev : List a -> List a -> List a
    rev Nil acc = acc
    rev (Cons x xs) acc = rev xs (Cons x acc)
```

Here we define an *auxilliary function* `rev` that does the real work. It uses an "accumulator," `acc` to build up the reversed list one step at a time, calling itself with a smaller and smaller list.

4.1 Currying and Partial Evaluation

There is a one-to-one correspondence between a function of pairs and a function that takes two arguments. More specifically, the types `(a, b) -> c` and `a -> b -> c` are isomorphic. We can construct this correspondence explicitly:

```
curry : ((a, b) -> c) -> (a -> b -> c)
curry f = fc
  where fc a b = f (a, b)

uncurry : (a -> b -> c) -> ((a, b) -> c)
uncurry fc = f
  where f (a, b) = fc a b
```

In TL1, all functions are really functions of *one argument*. If `f : a -> b -> c` and `x : a`, then `f x` has type `b -> c`. If `y : b`, then `(f x) y` has type `c`. However, functional evaluation has highest precedence in TL1, so those parenthese are not necessary: `f x y` has type `c` and is equivalent to `(f x) y`. In Python or R, if you had a function like this you would evaluate it $f(x)(y)$. Same idea, fewer parentheses.

It follows that the `->` operator on types is *right associative*; `a -> b -> c -> d` is equivalent to `a -> (b -> (c -> d))` et cetera.

We can take advantage of this when defining and using funtions. Start with a function that unwraps a `Maybe`

```
fromMaybe : a -> Maybe a -> a
fromMaybe _ (Some x) = x
fromMaybe default None = default
```

with a default value, where again `_` is a special placeholder for a value not referenced. Now, we can define a function to unwrap integers with a fixed default 0:

```
unwrap : Maybe Integer -> Integer
unwrap = fromMaybe 0
```

We defined the function without even specifying its arguments! That's because `fromMaybe 0` has type `Maybe a -> a`, which is the type we want.

With binary operators, we sometimes want to partially evaluate them, and we can use the placeholder as a *hold*:

```
lift : (a -> b) -> Maybe a -> Maybe b
lift f (Some x) = Some (f x)
lift _ None = None

maybeInc = lift (_ + 1)
```

Here `(_ + 1)` is a function of one argument that fills in the hole and adds one to it. It's a bit of syntactic sugar that gives us `(_ + 1) y == y + 1`. We can apply `maybeInc` to values without checking if a result is available; it just short circuits to `None` when no result is available.

Puzzle: Referring back to the `List` type and remembering that data constructors are functions: if we take `4 : Integer`, what is the type of `Cons 4`? Refering to `choose` above, what is the type of `choose 5`, assuming `5 : Integer`.

5 An Example: Kadane's Algorithm

In class, we discussed Kadane's algorithm. Here we will develop two distinct versions of that algorithm using `TL1`, the second will only require cell contents that are an ordered monoid. We will use the `Fold` type that we developed in class.

First, the algorithm in terms of a `Fold` uses a record to keep track of the current and best value so far. For simplicity, we assume a trait `HaveAgg` so that a type with `HaveAgg c` has a function `agg : c -> c -> c` for doing the aggregation (*with reset*) and an initial value `start : c`.

```
type State c = record State { total : c, biggest : c }

kadane : (Foldable t, Ordered c, HaveAgg c) => t c -> c
kadane = fold (Fold id step (State start start) State.biggest)
  where
    step : Ordered c => State c -> c -> State c
    step (State cur best) value = State next (max best next)
        where
            next = agg cur value
```

Notice that in the context of this function, we have

```
     fold : (Foldable t) => Fold c (State c) c -> t c -> c
```

and in `kadane` we supply the first argument returning a function `t c -> c` as desired.

Second, we give a leaner version that builds the `Fold` implicitly based on the given monoid and requires no extra operators or data. In this case, `kadane` takes a foldable collection and returns the result.

```
    bestSoFar : (Ordered c, Foldable t) => Fold a c c -> t a -> c
    bestSoFar f as = scan f as |> max

    kadane : (Monoid a, Ordered a, Foldable t) => t a -> a
    kadane = bestSoFar f
        where
            next s a = max munit (s <> a)
            f = Fold {wrap=id, step=next, init=munit, done=id}
```

We use `scan` to do the fold while collecting all intermediate results. Again, because `kadane` supplies the first argument to `bestSoFar`, we get a function `t c -> c` in the end.

6 Exercises

1. Pick a piece of code you've recently written (or a reasonable part thereof). Identify the key data entities and functions and give a `TL1` description of their types.

2. Guess how you might use each of the following types:

```
type Outcome a b = Failure b | Success a
type Validation a b = Errors (List b) | Correct a
type Environment r a = Env (r -> a)
```

Show how you would use these definitions to construct a value of each type.

3. The function

```
f : a -> a
```

is *polymorphic*: it accepts a value of any type and must return a value of the same type, but it does not know any details of the type of object it has. So it's computation cannot depend on the details of its input. What must this function be?

4. Write a function `partial` in R or Python that takes a function of two arguments and returns a function of one argument that returns a function of one argument.

7 Teaser: The Algebra of Types

There is a strong connection between the structure of sum and product types along with the special types `Void` and `Unit`. They form an algebra that looks a lot like a semiring. In addition, function types `a -> b` act like *exponentials* in this context. (Consider how many functions there are with type `Boolean -> Fin n` where `Fin (n : Nat)` is the type of values from 0 to $n-1$.) There are deep connections that we will explore and exploit later.