Programming Languages

Types

Dr Russ Ross

Utah Tech University—Department of Computing

Fall 2024

Introduction to Types

We will use the term *type* to refer to a *static* check, i.e., one that can be done purely with the program source. This means:

- types cannot refer to dynamic conditions
- they may suffer from either false-positive or false-negative errors

In return:

- they give us guarantees without ever having to run the program
- every part of the program can be checked, not just parts that have test coverage

Like objects, types are not well standardized. Many languages do not have them and those that do often disagree on their form. However, there are parts with widespread agreement.

A Standard Model of Types

Types are an abstraction of run-time values. We can have many distinct numbers, strings, etc., but when considering types we are concerned with the distinctions between categories and not the distinctions within them.

Let's start with a basic interpreter:

```
(define-type BinOp
  [plus])

(define-type Expr
  [binE (operator : BinOp)
        (left : Exp)
        (right : Exp)]
  [numE (value : Number)])
```

```
(calc : (Exp -> Number))
(define (calc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (+ (calc l) (calc r))])]
    [(numE v) v]))

(test (calc (binE (plus) (numE 5) (numE 6))) 11)
```

A Standard Model of Types

What needs to happen with a type-checker? The name “checker” suggests that the job of a type-checker is to *pass judgement* on a programs, i.e., to determine whether or not they are type-correct. Thus a natural type for the checker would be:

```
(tc : (Exp -> Boolean))
```

(In practice, of course, we would want more information in case the program is not type-correct, i.e., we’d like an error diagnostic. But we’re ignoring human factors considerations here.) With this type, we can now rewrite the relevant parts of the interpreter above:

```
(define (tc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (and (tc l) (tc r))])]
    [(numE v) #true]))
```

```
(test (tc (binE (plus) (numE 5) (numE 6))) #true)
```

Look at this for a moment. Given a number, it returns `#true`. In the recursive case, it checks the pieces and `and`s the result. There is no way to return `#false`, so every program is always type-correct.

A Standard Model of Types

The problem is that we only have one type (numbers) and one operation on numbers (plus) so there is nothing that *could* go wrong.

We need more types and operations:

```
(define-type BinOp
  [plus] [++])

(define-type Expr
  [binE (operator : BinOp)
        (left : Exp)
        (right : Exp)]
  [numE (value : Number)]
  [strE (value : String)])
```

Various things break, and need to be fixed. How about this?

```
(define (tc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (and (tc l) (tc r))]
       [(++)   (and (tc l) (tc r))])]
    [(numE v) #true]
    [(strE v) #true]))

(test (tc (binE (++) (strE "hello")
                     (strE "world"))) #true)
```

So this looks pretty good, right?

A Standard Model of Types

Here are two tests that demonstrate *desirable* behavior:

```
(test (tc (binE (++) (numE 5) (numE 6))) #false)
(test (tc (binE (plus) (strE "hello") (strE "world"))) #false)
```

The first string-concatenates two numbers, the second adds two strings. Therefore, both should be rejected by the type-checker. Yet both of them pass (i.e., the tests above fail).

What is the core problem here? Given an expression, we only know

- *whether* its sub-expressions typed correctly,
- but not *what* their types are.

This is insufficient to determine whether the current expression is type-correct. For instance:

- the ++ operator needs to check not only whether its two sub-expressions are well-typed, but also
- whether they produce strings;
- if they did not, then the concatenation is erroneous.

A Standard Model of Types

We need the type-checker to have a richer type: it must instead be

```
(tc : (Exp -> Type))
```

That is, the type “checker” must actually be a type *calculator*:

- it closely parallels the evaluator
- but over the universe of abstracted values (types)
- rather than concrete ones

Following convention, however, we will continue to call it a checker, because it *also* checks in the process of calculating types.

A Standard Model of Types

Type is a new (plait type) definition that records the possible types:

```
(define-type Type [numT] [strT])
```

With this, we can rewrite our type-“checker”:

```
(define (tc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (if (and (numT? (tc l)) (numT? (tc r)))
                   (numT)
                   (error 'tc "not both numbers"))]
       [(++)   (if (and (strT? (tc l)) (strT? (tc r)))
                   (strT)
                   (error 'tc "not both strings"))])]
    [(numE v) (numT)]
    [(strE v) (strT)]))
```

A Standard Model of Types

Now it passes some useful tests:

```
(test (tc (binE (plus) (numE 5) (numE 6))) (numT))
(test (tc (binE (++) (strE "hello") (strE "world"))) (strT))

(test/exn (tc (binE (++) (numE 5) (numE 6))) "strings")
(test/exn (tc (binE (plus) (strE "hello") (strE "world"))) "numbers")
```

There are three take-aways from this:

1. The type-checker follows the same implementation schema as the interpreter: an algebraic datatype to represent the AST, and structural recursion to process it. This is the schema we're calling SImPl.
2. A type-checker, unlike an interpreter, operates with "weak" values: note, for instance, how the `numE` case ignores the actual numeric values. Both the strengths and weaknesses of traditional type-checking arise from this ignorance.
3. In mathematical terms, the upgrade we performed in going from a type-checker to a type-calculator was a process of strengthening the inductive hypothesis: instead of returning only a `Boolean`, we had to return the actual type of each expression. This may not seem like a literal strengthening; but it is inasmuch as the former `#true` has been replaced by a `Type` and the `#false` by an `error`.

A Concise Notation

As we extend our type system, it is increasingly unwieldy to write everything out as code. Instead, we will adopt a notation commonly used in the world of types (though it can also be used for interpreters and other SImPl programs). We will write terms of the form

```
|- e : T
```

where the `e` are expressions, `T` are types, and `:` is pronounced as "has type": i.e., the notation above says "`e` has type `T`". For now we won't pronounce `|-` as anything at all; later, we will see that it should be read as "proves".

A Concise Notation

First, we can very concisely say that all numeric expressions have numeric type and all string expressions have string type:

```
|- n : Num
|- s : Str
```

where `n` stands for all the syntactic terms with the syntax of numbers, and `s` likewise for strings.

(We can think of this as an infinite number of rules, one for each number and each string. We're in the realm of mathematics, so what's an infinite number of rules between friends?) The former is exactly equivalent to writing

```
[(numC n) (numT)]
```

but much more concisely.

A Concise Notation

When we get to Booleans, we have a choice: we can either write

```
|- b : Bool
```

where `b` stands for all the syntactic terms with the syntax of Booleans, or—because there are only two of them—just enumerate them explicitly:

```
|- true : Bool
|- false : Bool
```

That covers the base cases of the type-checker. These are called *axioms*. Next we will look at the *conditional* cases, which are called *(typing) rules*.

A Concise Notation

Remember our code for typing addition:

```
[(plus) (if (and (numT? (tc l)) (numT? (tc r)))
            (numT)
            (error 'tc "not both numbers"))]
```

We can write it in this notation very concisely as follows:

```
|- e1 : Num    |- e2 : Num
--------------------------
|- (+ e1 e2) : Num
```

We read the line as "if (what's above) then (what's below)", and the space as "and".

So this says: "if `e1` has type `Num` and `e2` has type `Num`, then `(+ e1 e2)` has type `Num`". This is of course the exact same thing the code says, but with rather less noise.

Terminology: The part above is called the antecedent (that which goes before) and the part below is called the consequent (that which comes after). Don't call these the numerator and denominator!

Handling Division

Addition, multiplication, and subtraction are *total* functions over numbers: they consume two numbers and produce one. In contrast, division is a *partial* function: it isn't defined when the denominator is zero.

There are a few strategies for handling this:

1. Declare that division doesn't *return a number* but instead something else that captures its partiality.
    - Works, but every single use of division will need to check which kind of result it got
2. Declare that division only *consumes* non-zero numbers in its second argument.
    - A major change to the type system, because now numbers are not all a single type
    - Affects all callers of division, who must now prove that they are not dividing by zero. The type checker cannot prove this on its own in all cases (see Rice's Theorem)
3. Give it the same type as other binary numeric operations, but declare that division by zero will be handled by an exception or error
    - This puts the burden on the rest of the program, which must be aware of this possibility and handle it

Handling Division

Most languages take the third option (exceptions/panics), but some are exploring the other two.

- They get around Rice's Theorem by proving non-zero-ness when they can, and putting the burden of proof on the programmer the rest of the time
- This creates more effort for the programmer, but increases the program's robustness

Another Perspective on Types

We have seen one way to think about types:

- An abstraction of values
- Type-checking as running a program over these abstract values

Another perspective is to think of types:

- As a *static discipline*
- A way of making judgements about programs

This is kind of like parsing:

- A parser statically (before the program runs)
- passes judgement (decides some programs are good and some bad)

Types can be viewed as an extension of this idea.

Aside: In computability theory terms, parsers are usually context-free, whereas types usually reflect context-sensitive constraints. Computability theory then helps us understand why we might separate these checks into two separate phases, and in particular why we might do one before the other. Essentially, the type-checker only needs to deal with programs that have already passed the parsing, i.e., context-free check, so it has much less complexity than if it had to do everything. We already saw this: our previous checker only consumed `Exprs`, which are produced by the parser.

From Axioms and Rules to Judgements

When we need to apply type rules to a program, we compose them recursively, just as the type-checker runs. Consider this program:

```
(+ 5 (+ 6 7))
```

To decide its type, we will use our current rules:

- It does not fit any axiom, because the program does not match the syntax of a single number or string.
- We have to use a conditional rule.

We have seen only one so far, and fortunately this term does match the consequent:

- it requires two terms, and we have two terms, so `e1` is `5` and `e2` is `(+ 6 7)`
- Therefore, applying this conditional rule, we get:

```
|- 5 : Num    |- (+ 6 7) : Num
----------------------
|- (+ 5 (+ 6 7)) : Num
```

From Axioms and Rules to Judgements

Let's look at the two terms in the antecedent. The first matches an axiom, so we are finished with it:

```
|- 5 : Num    |- (+ 6 7) : Num
----------------------
|- (+ 5 (+ 6 7)) : Num
```

For the other, we have to apply the same conditional rule again:

```
              |- 6 : Num    |- 7 : Num
              ------------------------
|- 5 : Num    |- (+ 6 7) : Num
----------------------
|- (+ 5 (+ 6 7)) : Num
```

From Axioms and Rules to Judgements

These new terms also match the axiom for numbers:

```
               |- 6 : Num    |- 7 : Num
               ------------------------
|- 5 : Num     |- (+ 6 7) : Num
----------------------
|- (+ 5 (+ 6 7)) : Num
```

Every part of the tree now terminates in an axiom, therefore:

- We consider this program to have successfully type-checked
- This tree is called a *judgement*, because it passes judgement on the initial term
- It is judged to have type-checked and to produce a value of type `Num`

This is the same pattern of execution we had with the type-checker program

- We were able to skip the tedious details of passing and returning things
- We just used pattern-matching
- This will save work going forward

Judgements and Errors

Let's see another example:

```
(+ 5 (+ 6 "hi"))
```

This starts out similar to the previous example. We attempt a judgement:

```
               |- 6 : Num    |- "hi" : Num
               ---------------------------
|- 5 : Num     |- (+ 6 "hi") : Num
-------------------------
|- (+ 5 (+ 6 "hi")) : Num
```

But now we have a problem: we need to type-check

```
|- "hi" : Num
```

but *we don't have a rule that matches*. Therefore we *cannot* construct a successful tree.

Judgements and Errors

We *cannot* construct a successful tree:

```
               |- 6 : Num    |- "hi" : Num
               ---------------------------
|- 5 : Num     |- (+ 6 "hi") :
-------------------------
|- (+ 5 (+ 6 "hi")) :
```

Remember the “if … and … then” interpretation. Because we cannot satisfy all the antecedents, we cannot prove anything about the consequents, leaving the tree incomplete.

A type error is simply a failure to construct a judgment.

- It may not be the most satisfying user feedback, but our concern here is with a concise way of expressing ideas;
- going from this to an implementation is not too hard, and the user interface details can be added to the latter.

Judgements and Errors

This requires some clarification.

- We only call it a judgment if the tree is "checked off" completely:
  - every antecedent is generated using given rules, and
  - all the leaves are actual axioms.
- In this example, we are unable to check off the tree:
  - there is no available rule or axiom that lets us conclude that `"hi"` is a `Num`.
  - Therefore, we cannot "judge" the initial expression.

This is a technical meaning of the word "judgment", not to be confused with the broader use of this term. Imagine that we started with this program:

```
(+ 5 (- 6 7))
```

We would get this far:

```
|- 5 : Num     |- (- 6 7) : Num
----------------------
|- (+ 5 (- 6 7)) : Num
```

Again we would fail, this time because we haven't provided a (conditional) rule for `(- e1 e2)`. Obviously it's not difficult to define one; we just haven't done so yet, so our pattern-matcher would fail.

Typing Conditionals

Now we're ready to add a rule for `if`. Languages have different rules for what can go in the conditional clause.

- The goal of a type-checker is to catch type errors, so it is common for languages with type-checkers to demand that the conditional be a Boolean (without a truthy/falsy set of Boolean values).
- Our goal here is not to make a value judgment but rather to illustrate how we would add a type rule for it.

By now, we can see that we will need a conditional rule (because we want to type-check more than just constants); following SImPl, and we will need the antecedent to say something about the sub-expressions. We need at least:

```
|- C : Bool    …
------------------
|- (if C T E) : …
```

Typing Conditionals

Okay, what now? What is the type of the entire conditional expression?

- Technically, it should be whatever type is returned by the branch that was executed.
- However, a type-checker can't know which branch will be executed; over time, both might.
- We have to somehow capture the uncertainty in this situation.

There are two common solutions:

1. Introduce a new kind of type that stands for "this type or that type" (a *union*). This is easy to introduce but creates a burden for every piece of code that will consume such a value.

2. Just rule that both branches should have the same type. The latter is a very elegant solution, because it eliminates the uncertainty entirely.

Typing Conditionals

Okay, so we need to do the following things:

- Compute the type of `T`.
- Compute the type of `E`.
- Make sure `T` and `E` have the same type.
- Make this (same) type the result of the conditional.

That seems like a lot: how will we express all that? Very easily, actually:

```
|- C : Bool    |- T : U    |- E : U
-----------------------------------
|- (if C T E) : U
```

Here, `U` is a placeholder:

- it isn't a concrete type but rather stands for whatever type might go in that place.
- The repeated use of `U` accomplishes all of our goals above.

Typing Conditionals

```
|- C : Bool    |- T : U    |- E : U
-----------------------------------
|- (if C T E) : U
```

Read this as:

- if `C` has type `Bool` and
- `T` has type `U` and
- `E` has [the same] type `U`,
- then `(if C T E)` has [the same] type `U`

Typing Conditionals

Let's see this in action on the following program:

```
(if true 1 2)
```

We get:

```
|- true : Bool    |- 1 : U    |- 2 : U
--------------------------------------
|- (if true 1 2) : U
```

Either of the axioms for the other two antecedents tells us what `U` must be, which lets us fill in the result of `U` everywhere:

```
|- true : Bool    |- 1 : Num    |- 2 : Num
------------------------------------------
|- (if true 1 2) : Num
```

Typing Conditionals

Fortunately, the other two antecedents are also axioms:

```
|- true : Bool    |- 1 : Num    |- 2 : Num
------------------------------------------
|- (if true 1 2) : Num
```

This lets us conclude that the overall term is well-typed, and that it has type `Num`.

Typing Conditionals

Now let's look at:

```
(if 4 1 2)
```

Applying the conditional rule gives us:

```
|- 4 : Bool    |- 1 : U    |- 2 : U
-----------------------------------
|- (if 4 1 2) : U
```

However, we do not have any axiom or conditional rule that lets us conclude that `4` has type `Bool` (because, in fact, it does not). Therefore, we cannot complete the judgment and the program is (rightly) judged to have a type error.

Typing Conditionals

One last example:

```
(if true 1 "hi")
```

Again, applying the conditional rule and checking off the first antecedent:

```
|- true : Bool    |- 1 : U    |- "hi" : U
-----------------------------------------
|- (if true 1 "hi") : U
```

But now we have a problem. If we apply the axiom for numbers, we replace all instances of `U` with `Num` to get:

```
|- true : Bool    |- 1 : Num    |- "hi" : Num
---------------------------------------------
|- (if true 1 "hi") : Num
```

Typing Conditionals

Maybe we just tried the wrong axiom? We do have one more option!

However, it ends up with the same net effect:

```
|- true : Bool    |- 1 : Str    |- "hi" : Str
---------------------------------------------
|- (if true 1 "hi") : Str
```

Because there is no way to construct a judgment for this program, it too has a type error.

Exercise: Let's now add functions. We need two new constructs: one to introduce them (`lambda`) and one to use them (function application). Write down judgments for each.

Where Types Diverge from Evaluation

Something very important, and subtle, happened above.

Compare the type rule for a conditional with the evaluation process. If the rule is too abstract, just look at the example judgments (or failed judgments) above.

- The evaluator evaluates only one branch out of `T` and `E`; indeed, that is the entire point of a conditional.
- The type-checker, in contrast, traverses both branches! In other words, it looks at code that might evaluate, not only code that absolutely does evaluate.

In other words, the idea that a type-checker is like an “evaluator that runs over simple values” is a convenient starting analogy, but it is in fact false.

- An evaluator and type-checker follow different traversal strategies.
- That is why a program like `(if true 1 "hi")` might run without any difficulty but is rejected by a type-checker.

Where Types Diverge from Evaluation

While this particular example may make the type-checker look overly pedantic, what if the same program were

```
(if (is-full-moon) 1 "hi")
```

Should the type-checker pass the program every month? Should it consider the moon's phase at the time of type-checking or at execution? Unfortunately, the type-checker doesn't know when the program will run; indeed, the program is type-checked once but may run an arbitrary number of times. Therefore, a type-checker must necessarily be *conservative*.

Where Types Diverge from Evaluation

This also lets us relate type-checking to **testing**.

- In software testing, making sure that all branches are visited is called *branch coverage*, and making sure all branches have coverage is both important and very difficult (because each branch may have additional branches which in turn may have even more branches which…).
- In contrast, a type-checker effortlessly covers both branches. The trade-off is that it does so only at the *type* level (and indeed, the abstraction of values to types is precisely what enables it to do this).

Thus, testing and type-checking are complementary.

- Type-checking provides code coverage at a lightweight level;
- testing typically provides only partial coverage but at the deep level of specific values.

In recent years, people have invented a notion of *concolic*—i.e., “concrete” + “symbolic”—testing to try to create the best of both worlds.

Growing Types: Typing Functions

Let us grow our language further to include functions. Concepts like functions tend to come in pairs:

- a way to introduce them, a `lambda` form in our case
- a way to use (“eliminate”) them, namely function application

We use syntactic sugar over functions to obtain forms like `let`, so typing functions covers those cases as well.

Typing Function Applications

A function application has two parts: the function and the arguments. We will work with single-argument functions.

Because functions are first-class values, the function position is itself an expression. We have to check each sub-expression before we can type the whole expression. Therefore, function applications are conditional rules with two terms in the antecedent:

```
|- F : ???    |- A : ???
------------------------
|- (F A) : ???
```

Typing Function Applications

First, let's notice that functions are different kinds of values than other values:

- a function is not itself a number, or string, or Boolean
- it may produce one of those, but it is not itself one of those (an important distinction).
- Therefore, we need a different type for functions, which reflects what functions consume and what they produce.

A natural idea is to assume functions have some "function" type, here called Fun:

```
|- F : Fun     |- A : ???
------------------------
|- (F A) : ???
```

What do we know about the argument expression (the actual parameter)?

- It had better match the type demanded by the formal parameter.
- But how do we check that here?

We've collapsed all functions in the world into a single type, `Fun`. That's far too coarse.

Typing Function Applications

Instead, following convention, we'll use the "arrow" syntax for functions:

```
|- F : (??? -> ???)    |- A : ???
---------------------------------
|- (F A) : ???
```

(Technically, the arrow is a constructor of function types. It's a two-place constructor, for reasons we will see below.)

With this, we can now say that the function's formal parameter's type had better match up with the type of the actual argument.

Which type, exactly? Functions could consume numbers, strings, even other functions...all we know is that these should be consistent.

This is very similar to the consistency we expected of the branches of a conditional. We can again encode this by using the same placeholder in both places:

```
|- F : (T -> ???)    |- A : T
-----------------------------
|- (F A) : ???
```

Typing Function Applications

Now, what about what the function returns? Again, it could return values of any type. Whatever that type is, that is what the entire application produces. Again, we use a common placeholder to reflect this:

```
|- F : (T -> U)     |- A : T
---------------------------
|- (F A) : U
```

So here's how we read this:

- Type-check the `F` position. Make sure it's a function type (`->`). Assuming it is, call the formal parameter's type `T` and the return type `U`.
- Type-check the actual parameter (the argument). Make sure it has the same type as what the function is expecting in its formal parameter.
- If both of those hold, then the function's return type is the type of the entire application.

This list of steps is what a conventional type-checker would implement. Observe that again, a type error is the result of a failure to construct a judgment. If, for instance, the actual argument's type doesn't match that of the formal parameter, then the conditional rule above doesn't apply (it applies only when we can write the same type for the `T` placeholder), which is how we learn that the program has a type error.

Typing Function Definitions

Now we're ready to type `lambda`. Here, we have to be careful about how many sub-expressions there are.

- Given `(lambda V B)`, it is tempting to think that there are two:
  1. `V` (the formal parameter) and
  2. `B` (the body).
- This is wrong! The formal parameter is a literal name, not an expression: we can't replace that name with some larger expression, which is what it would mean for it to be an expression.
- Furthermore, we can't evaluate it: it would (most likely) produce an unbound variable error, because its whole job is to bind that variable, so it can't assume it has already been bound.

Therefore, there is only one sub-expression, the body.

Typing Function Definitions

Therefore, we expect to end up with a conditional rule that looks like this:

```
|- B : ???
---------------------
|- (lambda V B) : ???
```

If we think about this for a moment, we can see that there's going to be a problem.

- We just said that the `lambda` introduces a binding for the variable in the `V` position.
- This is precisely so that the body, `B`, can make use of that variable.

So let's imagine the simplest function:

```
|- x : ???
---------------------
|- (lambda x x) : ???
```

But we don't have any typing rule that covers variables! Furthermore, we have no way of knowing what the type of any old variable will be. So we have a problem.

Typing Variables

Remember how we addressed this problem in our interpreter:

- we had an environment for recording the value bound to each variable.

We will use this same idea again:

- we'll have a type environment for recording the type of each variable.

That is, just as our interpreter had the type

```
(interp : (Exp Env -> Value))
```

our type-checker will have the type

```
(tc : (Exp TEnv -> Type))
```

Typing Variables

In our type-checker notation, we will use a slightly different way of writing it, which will finally make make `|-` stop being silent and take it proper pronounciation, “proves”: all type rules will have the form

```
Γ |- e : T
```

where `Γ`, the capital Greek letter gamma, is conventionally used for the environment.

We read this as “the environment `Γ` proves that `e` has type `T`”. So in fact there’s been an environment hiding in all our judgments, but we didn’t have to worry about it when we didn’t have variables.

But now we do, so from now on we have to make it explicit. Fortunately, in most cases the environment is unchanged, and just passes recursively to the sub-terms, as you would expect from writing the interpreter.

With this, we can write a type for variables. What is the type of a variable? It’s whatever the environment says it is! We’ll treat the environment as a function, so we can just write the following axiom (where `v` stands for all the syntactically valid variable names):

```
Γ |- v : Γ(v)
```

Back to Typing Function Definitions

Now we're in a position to fill in the holes. When we check the body of the function, we should do it in an extended environment:

```
Γ[V <- ???] |- B : ???
-----------------------
Γ |- (lambda V B) : ???
```

where `Γ[V <- _]` is how we write "Γ is extended with `V` bound to `_`": this is the same environment-extension function that we've written before, for type environments instead of value environments, but operationally the same.

Okay, but two questions: extend *which* environment, and extend it with *what*?

Which is easy: it's the environment of the function definition (static scope!). The repetition of Γ in both the consequent and antecedent accomplishes that.

Back to Typing Function Definitions

In terms of what: We need to provide a type for the variable so that, when we try to look up its type, the environment can return something.

But we don't know what to extend it with! The type-checker needs the *programmer to tell it* what type the function is expecting. This is one of the reasons why programming languages expect annotations in function and method definitions.

(Another—equally good—reason is because it better documents the function for people who have to use it and maintain it.)

Therefore, we have to extend the syntax of functions to include a type annotation:

```
(lambda V : T B)
```

which says that `V` is expecting to be bound to a value of type `T` in body `B`.

Back to Typing Function Definitions

Once we accept this modification, we can make progress on the conditional rule:

```
Γ[V <- T] |- B : ???
---------------------------
Γ |- (lambda V : T B) : ???
```

What type are we expecting for the function definition? Clearly it must be a function type:

```
Γ[V <- T] |- B : ???
------------------------------------
Γ |- (lambda V : T B) : (??? -> ???)
```

Furthermore, we know that the type expected by the function must be `T`:

```
Γ[V <- T] |- B : ???
----------------------------------
Γ |- (lambda V : T B) : (T -> ???)
```

Back to Typing Function Definitions

Given a value of type `T`, the function will return whatever the body produces:

```
Γ[V <- T] |- B : U
--------------------------------
Γ |- (lambda V : T B) : (T -> U)
```

And that gives us our final rule for function definitions.

More Divergence Between Types and Evaluation

It is interesting to contrast the above pair of typing rules with the corresponding evaluation rules.

- In the evaluator, we visit the body of the function on every *application*—which could be as many as an infinite number of times in a program.
- In contrast, we visit the body of the function on *definition*, which happens only once.
- Therefore, even if the program runs forever, the type-checker is guaranteed to terminate!

Why can we get away with this?

- The evaluator has to run the body with the *specific* value it was given.
- The type-checker, however, has abstracted the concrete values away.
- Therefore, it only needs to make one pass through the body with the "abstract value", the type.

Aside: Earlier, when we proposed the type `Fun`, we said that it collapsed all functions in the world into one type. This was too coarse, and we had to refine the type of a function. However, we are *still* collapsing an infinite number of functions into each of those function types—just as we collapse an infinite number of strings into `Str`, and so on. Both the strength and weakness of type-checking lies in this collapsing.

For the same reason, observe that a function application rule only cares about the *type* of the function, not *which* specific function is being applied. Therefore, any function that has that type can be used. For that same reason, the type-checker *cannot* traverse the function's body at application time—it doesn't even know which function might be used! All communication between the function body and application must happen entirely through the type boundary.

Assume-Guarantee Reasoning

There is a delicate dance going on between these typing rules for application and definition (now updated to have the environment):

```
Γ |- F : (T -> U)    Γ |- A : T
-------------------------------
Γ |- (F A) : U
```

```
Γ[V <- T] |- B : U
--------------------------------
Γ |- (lambda V : T B) : (T -> U)
```

The rule for `lambda` *assumes* the parameter will be given a value of type `T`; the application rule *guarantees* that that the actual parameter will indeed have the expected type. The application rule *assumes* that the function, if given a `T`, will produce a `U` (because the type is `(T -> U)`); the `lambda` rule *guarantees* that the function will indeed perform that way.

Aside: The notation `(T -> U)` is not chosen at random. The `->` may remind you of the notation for implication in mathematics. That's intentional. We can read the type as "giving the function a `T` implies that it will produce a `U`" (not giving it a `T` implies nothing about what it will do…). It is that *implication* that is assumed in the application rule, and that is guaranteed by the rule for `lambda`.

This assume-guarantee reasoning shows up in many places, so look out for this pattern in other places as well.

Recursion and Infinite Loops

We alluded, earlier, to how we can desugar more interesting features into functions and application. Let's take a look at a very specific feature: an infinite loop. Let's first confirm that we can write an infinite loop. Here's a program that does it:

```
fun f():
  f()

f()
```

But this assumes we already have recursion. Can we write it without recursion? Actually we can! We'll use historical names (ω is the lower-case Greek omega):

```
(let ([ω (lambda (x) (x x))])
  (ω ω))
```

Run this in Racket and confirm that it runs forever!

Recursion and Infinite Loops

Now let's see what happens when we try to type this. We have to provide a type annotation:

```
(let ([ω (lambda (x : ???) (x x))])
  (ω ω))
```

Historically, the overall term is called Ω (the capital Greek omega).

Okay, so what is the annotation? To determine a type for `x`, we have to see how it's used. It's used twice. One use is in a function application position, so we know that the type must be of the form `(T -> U)`; now we have to determine what `T` and `U` are. Let's focus on the parameter type, `T`. But what are we passing in? We're passing in `x`, whose type is `(T -> U)`. So we need a solution to the equation

```
T = (T -> U)
```

with one coming from the application position and the other from the argument position. Of course, there is no finite type that can fit this equation! Therefore, it appears that this program cannot be typed!

Of course, this is not a proof. However, there is a formal property associated with this programming language, which is called the Simply Typed Lambda Calculus (STLC): the property is called *strong normalization*, and it means that *all programs in this language terminate*.

Aside: If you have heard about the Halting Problem, how does that square with what you just read?

Recursion and Infinite Loops

It may seem rather useless to have a language in which all programs terminate—you can't write an operating system, or Web server, or many other programs in such a language. However, that misses two things.

1. There are many cases where we want programs to always terminate.
    - You don't want a network packet filter or a device driver or a compiler or a type-checker or … to run forever. Of course we also want them to run quickly, but it would be nice if we had a guarantee that no matter what we did, we cannot create an infinite loop.
    - The STLC is very useful in some of these settings.
    - Another example of a place where we want guaranteed termination is in program linking, and the module language of Standard ML is therefore built atop the STLC: it lets you even write higher-order programs, but the type language guarantees that all module compositions (linkages) will terminate.
2. Second, many long-running programs are actually a composition of an infinite loop and a short-running program.
    - Think about an operating system with device drivers, a Web server with a Web application, a GUI with callbacks, etc. In each case, there is a "spine" of an infinite loop that simply keeps the program reactive, and "ribs" of short computations that do a little specific work and terminate.
    - In fact, on the Web these programs must terminate quickly, otherwise the Web browser thinks the server has hung and offers to kill the window!
    - These kinds of reactive systems are therefore a composition of a very generic infinite loop calling out to specific programs for which a termination guarantee will often be very useful.

Finally, observe that we've learned something profound. Until now, we have probably thought of types as just a convenience or as a way of eliminating basic errors. However, we have just now seen that adding a type system can change the expressive power of a language. That is, these types are "semantic".

Typing Recursion

What went wrong above? The problem is that each application “uses up an arrow” in a function type; because a program text must be finite, it can contain at most a finite number of “arrows”, so eventually the program must terminate. To get around this, we need a way to effectively have an “infinite quiver”.

We typically do this by adding a recursive function construct to the language, and create a custom type for it. Let’s start with a type rule for the analogous, but simpler, `let`:

```
Γ |- E : T    Γ[V <- T] |- B : U
--------------------------------
Γ |- (let V : T E B) : U
```

Note that we’re going to expect an annotation in `let` for the same reason we do for function definitions. So this says that we’ll check that `E` actually does have the type promised in the declaration, `T`; when we extend the type environment with the `V` having type `T`, if the body `B` produces type `U`, then that’s the type of the whole expression.

Aside: Notice that there’s an assume-guarantee pair in the antecedent: the first term is guaranteeing the annotation, which the second term is assuming.

Aside: Technically, the type of `E` could be calculated. Therefore, the `T` annotation is not strictly necessary.

Typing Recursion

Observe that this is basically the type rule we would get from expanding the syntactic sugar for `let`. Therefore, this still doesn't let us write a recursive definition. We need something more. Let's introduce a new construct, `rec`, for recursive definitions. An example of a `rec` (in an untyped setting) might be

```
(rec ([inf-loop (lambda (n) (inf-loop n))])
  (inf-loop 0))

(rec ([fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))])
  (fact 10))
```

Typing Recursion

In the typed world, we'll want `rec` to have the form

```
(rec V : T E B)
```

so we'd instead have to write

```
(rec inf-loop : (Number -> Number)
     (lambda (n) (inf-loop n))
  (inf-loop 0))

(rec fact : (Number -> Number)
     (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))
  (fact 10))
```

where `V` is `fact`, `T` is `(Number -> Number)`, `E` is the big `lambda` term, and `B` is `(fact 10)`.

Typing Recursion

So this introduces a recursive definition, and then uses it. How might we type this?

```
???
------------------------
Γ |- (rec V : T E B) : U
```

Well, clearly one part of it must be the same: we have to type the body in the extended environment, and the environment must be extended with the annotated type:

```
???    Γ[V <- T] |- B : U
-------------------------
Γ |- (rec V : T E B) : U
```

We also know that we need to confirm that the annotation is correct:

```
??? |- E : T    Γ[V <- T] |- B : U
----------------------------------
Γ |- (rec V : T E B) : U
```

But clearly, something needs to be different, otherwise we've just reproduced `let`.

Typing Recursion

Look at the example use of `rec`: the `E` term also needs to have `V` bound in it! In other words, both `E` and `B` are typed in the same environment:

```
Γ[V <- T] |- E : T    Γ[V <- T] |- B : U
----------------------------------------
Γ |- (rec V : T E B) : U
```

From the type, we can read off how the recursion happens: the extended environment for `B` initiates the recursion, while that for `E` sustains it. Essentially, the environment of `E` enables arbitrary recursive depth.

In short, to obtain arbitrary recursion—and hence infinite loops—we have to add a special construct to the language and its type-checker; we cannot obtain it just through desugaring. Once we add `rec` to the STLC, however, we obtain a conventional programming language again.

Safety and Soundness

A critical component of SMoL is the concept of safety:

- some operations are partial over the set of all values
- a SMoL language enforces this by reporting violations

Typical examples of partiality may include + applying only to certain types of values. However, I intentionally write "operations" rather than, say, "functions", because these could be primitive operations like application (expecting the first position to be a function or method) as well. In fact, in some languages like JavaScript, there are very few violations, as the Wat talk shows.

How must these be enforced? It can be either statically or dynamically. In Python and JavaScript, for instance, all safety violations are reported dynamically. In Java or OCaml, most of them are reported statically. Either way, safety means that data have integrity: there is some notion of "what they are", and that identity is respected by operations. Put differently, data are not misinterpreted.

These are all very abstract statements, which we will soon concretize.

Revisiting the Basic Calculator

We will start with a very basic calculator that has two types, numbers and strings, and an operation (addition and concatenation, respectively) on them. Note that it helps to have more than one type if we want to talk about safety. We will skip most of the boilerplate code and focus on the core of the calculator:

```
(define-type Exp
  [num (n : Number)]
  [str (s : String)]
  [plus (l : Exp) (r : Exp)]
  [cat (l : Exp) (r : Exp)])

(define-type Value
  (numV (n : Number))
  (strV (s : String)))

(calc : (Exp -> Value))

(define (calc e)
  (type-case Exp e
    [(num n) (numV n)]
    [(str s) (strV s)]
    [(plus l r) (num+ (calc l) (calc r))]
    [(cat l r) (str++ (calc l) (calc r))]))
```

```
(define (num+ lv rv)
  (type-case Value lv
    ((numV ln)
     (type-case Value rv
       ((numV rn) (numV (+ ln rn)))
       (else (error '+ "right not a number"))))
    (else (error '+ "left not a number"))))

(define (str++ lv rv)
  (type-case Value lv
    ((strV ls)
     (type-case Value rv
       ((strV rs) (strV (string-append ls rs)))
       (else (error '++ "right not a string"))))
    (else (error '++ "left not a string"))))
```

Revisiting the Basic Calculator

With a suitable parser, we can run tests such as the following:

```
(test (calc (plus (num 1) (num 2))) (numV 3))
(test (calc (plus (num 1) (plus (num 2) (num 3)))) (numV 6))
(test (calc (cat (str "hel") (str "lo"))) (strV "hello"))
(test (calc (cat (cat (str "hel") (str "l")) (str "o"))) (strV "hello"))
(test/exn (calc (cat (num 1) (str "hello"))) "left")
(test/exn (calc (plus (num 1) (str "hello"))) "right")
```

The last two, in particular, show that the language is safe. The checks inside the primitives—in `num+`, for instance—are called *safety checks*.

Making Memory Explicit (Unsafely)

Now we're going to do something fun: we're going to make the memory allocation of values explicit. As we go through this, remember what we've said before: a value in SMoL is just a memory address.

Let's do this in stages. First, we'll use a vector to represent memory:

```
(define MEMORY (make-vector 100 -1))
```

The value `-1` is useful for identifying parts of memory that have not yet been touched (assuming, of course, we don't write a program that produces `-1`—which we can avoid doing easily enough in this illustration).

Making Memory Explicit (Unsafely)

It will be useful to have a helper to use the next available bit of memory:

```
(define next-addr 0)
(define (write-and-bump v)
  (let ([n next-addr])
    (begin
      (vector-set! MEMORY n v)
      (set! next-addr (add1 next-addr))
      n)))
```

Making Memory Explicit (Unsafely)

Now let's say we want to store a number in memory. We put it in the next available memory place, and return the *address* of the place where the number was stored. Be careful here: the number we return is a memory address (which, here, is represented as an array index), which is not at all necessarily the same as the *numeric value* being stored.

```
(define (store-num n)
  (write-and-bump n))
```

Correspondingly, when we want to read a number, we simply return what is at the address corresponding to the number.

```
(define (read-num a)
  (vector-ref MEMORY a))
```

We want the property that when we `read-num` from the address where we `store-num` a number, we get back that same number: for all `N`,

```
(read-num (store-num N)) is N
```

Making Memory Explicit (Unsafely)

Now let's look at strings. We are going to convert the string into a sequence of character codes, and store those codes explicitly:

```
(define (store-str s)
  (let ([a0 (write-and-bump (string-length s))])
    (begin
      (map write-and-bump
           (map char->integer (string->list s)))
      a0)))
```

In particular, the value stored at the address representing the string is the *length* of the string, followed by the individual characters. (Endless blood has been spent over whether strings should store their lengths at the front, or whether they should only be delimited by a special value, or both. The question is uninteresting here.)

Making Memory Explicit (Unsafely)

Thus, suppose with a fresh memory we run

```
(store-str "hello")
```

this would return the address `0`. The resulting value of `MEMORY` would be

```
'#(5
   104
   101
   108
   108
   111
   -1
   -1
   -1
   ...)
```

That is, at address `0` we have the length of the string, followed by five character codes; these six memory entries together constitute the five-character string `"hello"`. The rest of the memory remains untouched.

Making Memory Explicit (Unsafely)

To read a string we have to reassemble it:

```
(define (read-str a)
  (letrec ([loop
            (lambda (count a)
              (if (zero? count)
                  empty
                  (cons (vector-ref MEMORY a)
                        (loop (sub1 count) (add1 a)))))])
    (list->string
     (map integer->char
          (loop (vector-ref MEMORY a) (add1 a))))))
```

Once again, we want the result of reading a written string to give us the same string.

Making Memory Explicit (Unsafely)

Now let's update the calculator. First, we're in for a surprise: we no longer need (*or want*) a fancy Racket datatype to track values, because values are just addresses (i.e., array indices)! So:

```
(define-type-alias Value Number)
```

The type of the calculator doesn't change; it still produces values. It's just that the representation of values has changed…dramatically. (Recall, again, that these `Numbers` are addresses, not numeric values *in* the interpreted language.)

The calculator remains the same. What has changed is in the helper functions. In the primitive value cases, we have to explicitly allocate them—which is what we were doing when we called the previous definitions of `numV` and `strV` (which store data on the heap), except it may not have been so evident. We will make it explicit as follows:

```
(define numV store-num)
(define strV store-str)
```

Making Memory Explicit (Unsafely)

Okay, now to update the helper functions. Let's focus on `num+`. The core logic is currently

```
[(numV rn) (numV (+ ln rn))]
```

Observe that now we're calling it on the result of calling `calc`, i.e., on `Values`. That means `num+` is going to get two addresses as arguments, and it needs to look up the corresponding numbers in memory, and then produce the resulting number:

```
(define (num+ la ra)
  (numV (+ (read-num la) (read-num ra))))
```

Analogously, we can define concatenation as well:

```
(define (str++ la ra)
  (strV (string-append (read-str la) (read-str ra))))
```

Making Memory Explicit (Unsafely)

Finally, we have to update our tests as well. Because `calc` now returns *addresses*, all our answers appear to be incorrect. Instead, we have to obtain the corresponding numbers or strings at those addresses. Once we do so, `calc` passes the tests:

```
(test (read-num (calc (plus (num 1) (num 2)))) 3)
(test (read-num (calc (plus (num 1) (plus (num 2) (num 3))))) 6)
(test (read-str (calc (cat (str "hel") (str "lo")))) "hello")
(test (read-str (calc (cat (cat (str "hel") (str "l")) (str "o")))) "hello")
```

Except...does it? These two tests do not pass:

```
(test/exn (calc (cat (num 1) (str "hello"))) "left")
(test/exn (calc (plus (num 1) (str "hello"))) "right")
```

In fact, how *can* they? In all the above code, there are no errors left!

Making Memory Explicit (Unsafely)

Rather, when we run

```
(calc (cat (num 1) (str "hello")))
```

we get an address back (maybe 69; it depends on what you ran earlier and hence what is in `MEMORY`). In fact, we can decide how we want to treat this: as a number?

```
> (read-num 69)
- Number
6
> (read-str 69)
- String
"\u0005hello"
```

How can something be both a number and a string? Well, actually, the situation is a bit more confusing than that: 69 above is just an address in memory from which we can read off whatever we want *however we want it* (i.e., the content of that address is *interpreted* by the function that reads from it), which can result in garbage.

Making Memory Explicit (Unsafely)

It can get even worse:

```
> (read-num (calc (plus (num 1) (str "hello"))))
- Number
6
> (read-str (calc (plus (num 1) (str "hello"))))
- String
. . integer->char: contract violation
    expected: valid-unicode-scalar-value?
    given: -1
```

That is, we've tried to read "off the end of memory". It was dumb luck that we had a `-1` as the initial value; the `-1` triggered an error when we tried to convert it to a character *because Racket's primitives are safe*, which halted the program. If `integer->char` did not have a safety check, we would have gotten some garbled string instead.

In short, what we have created is an *unsafe* language. Data have no integrity. Any value can be treated as any kind of datum. This, in short, is the memory model of C, and it's largely proven to be a disaster for modern programming, which is why SMoL languages evolved.

Recovering Safety

Fortunately, it does not take too much work to make the language safe again. What we've just written holds the key:

- every value needs to record what kind of value it is.
- And any use of that value needs to check that it's the right kind of value.

This information is called a *tag*; it takes a fixed amount of space, and represents *metadata* about the subsequent datum. All subsequent values are interpreted in accordance with the tag. We need two tags for the two kinds of values. Let's use

```
(define NUMBER-TAG 1337)
(define STRING-TAG 5712)
```

It's important that the two tags be different, so they are unambiguous. However, we don't need to worry about the tags themselves being confused with other data (e.g., numbers), because the tags will never be processed directly as program data (unless, of course, there is a bug in our implementation that accidentally does so…which is why language implementations need to be tested extensively).

Recovering Safety

Now, when we allocate a number, we write its *tag* into the first address, followed by the actual numeric value:

```
(define (store-num n)
  (let ([a0 (write-and-bump NUMBER-TAG)])
    (begin
      (write-and-bump n)
      a0)))
```

And when we try to read a number, we *first* check that it really is a number, and only then obtain the actual numeric value:

```
(define (safe-read-num a)
  (if (= (vector-ref MEMORY a) NUMBER-TAG)
      (vector-ref MEMORY (add1 a))
      (error 'number (number->string a))))
```

Recovering Safety

Strings are analogous:

```
(define (store-str s)
  (let ([a0 (write-and-bump STRING-TAG)])
    (begin
    (write-and-bump (string-length s))
    (map write-and-bump
         (map char->integer (string->list s)))
    a0)))

(define (safe-read-str a)
  (if (= (vector-ref MEMORY a) STRING-TAG)
      (letrec ([loop
                 (lambda (count a)
                   (if (zero? count)
                       empty
                       (cons (vector-ref MEMORY a)
                             (loop (sub1 count) (add1 a)))))])
        (list->string
         (map integer->char
              (loop (vector-ref MEMORY (add1 a)) (+ a 2)))))
      (error 'string (number->string a))))
```

Recovering Safety

So now, starting from a fresh memory, running

```
(store-str "hello")
```

still produces `0`, but the content of `MEMORY` looks a bit different:

```
'#(5712
   5
   104
   101
   108
   108
   111
   -1
   -1
   -1
   …)
```

That is, at address `0` we first encounter the tag for strings. Only then do we get the string's length, followed by its contents.

Recovering Safety

Observe that now, storing the length up front makes even more sense:

- the first two locations contain the tag and the length,
- both of which are metadata that help us interpret what comes later,
- with the second (the length) *refining* the first (the tag).

With this change, the interpreter stays unchanged, and effectively so do the helpers, other than using the new names we've chosen:

```
(define (num+ la ra)
  (store-num (+ (safe-read-num la) (safe-read-num ra))))

(define (str++ la ra)
  (store-str (string-append (safe-read-str la) (safe-read-str ra))))
```

All our "good" tests still pass, but interestingly, our "bad" tests now fail:

```
(test/exn (calc (cat (num 1) (str "hello"))) "string")
(test/exn (calc (plus (num 1) (str "hello"))) "number")
```

What Price Safety?

Our safe evaluator has, however, come at a price relative to the unsafe evaluator:

- In terms of running **time**, we are now clearly paying for the overhead of safety checks.
- In terms of **space**, we are paying for the tags.
- Thus, we have had to get worse space *and* time.

Nevertheless, the price of unsafe languages is so high—e.g., in the form of security problems—and the cost of safety is often so low, that programmers gladly pay this price (or do so without even particularly noticing it).

What Price Safety?

Still, it would be nice if we didn't have to pay the price at all. And there is a way to accomplish that: *types*.

Look at our "bad" programs:

- These are programs that can *statically* be rejected by a type-checker.
- If we could reject all such programs, then—since no "bad" programs would be left—we can then run the program on the unsafe evaluator without worrying about negative consequences.
- This, in effect, is what most typed languages, like Java and OCaml, do.

Thus we find another use for types: to improve program performance. But this requires care.

Soundness

Running on an unsafe evaluator is, as the name suggests, dangerous. Therefore, we should only do it if we can be sure that nothing can go wrong. That means that our type system needs to come with a *guarantee*.

The way this guarantee is usually formulated is as follows. Suppose we have

`e : t`

and suppose we evaluate it and find that

`e -> v`

The latter—its value—is the ground truth. The type checker's job is to make sure it matches what the evaluator produces. That is, we would ideally like that

`e : t` if and only if `e -> v` and `v : t`

This says that the type checker's job is to perfectly mirror the evaluator: whatever type the program's result value has is the same type the type-checker says it has.

Soundness

Unfortunately, for a Turing-complete language, this full guarantee is impossible to obtain, because of Rice's Theorem. Instead, we have to compromise and see if we can get at least one of the two directions. When we think about it, we realize that, in a typed language, we're only really interested in programs that pass the type-checker (i.e., have a type). Therefore, we expect that

```
If e : t then
       if e -> v, then v : t
```

This says that whatever type the type-checker predicted is exactly the type that the program has. That means we can rely on the type-checker's prediction. Which in turn means that we can be sure there are no type violations. Which tells us we can safely run the program atop an unsafe evaluator! This property is called *type soundness*.

Note that soundness is not a given: it's a property that must be formally, mathematically *proven* of a given type-checker and evaluator. The proof can be quite complex. This is because the "shape" of program evaluation and that of type-checking can be very different, as we have seen before for conditionals and functions.

And failure to prove it correctly—i.e., claiming it holds when in fact it doesn't—means we've allowed a vulnerability to slip through. This can manifest as uncaught exceptions, crashes, segmentation faults, etc. In addition, a clever attacker can construct a program that exploits the vulnerability, and our system can be subjected to a security or other attack. Thus, any soundness violations are emergencies and result in panic.

Generic Printing

One of the consequences of our tagged representation is that when extracting a value from memory, we don't have to know whether to use `safe-read-num` or `safe-read-str`; the tag at the address can tell us which to use. That is, we can define

```
(define (generic-read a)
  (let ([tag (vector-ref MEMORY a)])
    (cond
      [(= tag NUMBER-TAG) (safe-read-num a)]
      [(= tag STRING-TAG) (safe-read-str a)]
      [else (error 'generic-print "invalid tag")])))
```

Unfortunately, this code can't be typed by plait because the two branches return different types.

Generic Printing

We can solve this in two ways:

1. We can use a hack: use `#lang plait #:untyped`, which provides the same syntactic language, features, and run-time behavior, but turns off the type-checker. (Curiously, we were using the type-checker to keep us disciplined: so that the only values we could store in `MEMORY` would be numbers! Therefore, it's good to not use the untyped version often.)
2. Notice that in the end, what printers do is essentially print a string.

Therefore, we just need to return a string in all cases:

```
(define (generic-read a)
  (let ([tag (vector-ref MEMORY a)])
    (cond
      [(= tag NUMBER-TAG) (number->string (safe-read-num a))]
      [(= tag STRING-TAG) (safe-read-str a)]
      [else (error 'generic-print "invalid tag")])))
```

Generic Printing

A consequence of having this function is that we can rewrite our tests to be more proper:

```
(test (generic-read (calc (plus (num 1) (num 2)))) "3")
(test (generic-read (calc (plus (num 1) (plus (num 2) (num 3))))) "6")
(test (generic-read (calc (cat (str "hel") (str "lo")))) "hello")
(test (generic-read (calc (cat (cat (str "hel") (str "l")) (str "o"))))
"hello")
```

This is much closer to how we would write the test in the original interpreter; the only difference here is that the evaluator produces an address as the value, but we would like to inspect the value in a human-readable and -writable form, so we use `generic-read`.

Unannotated Programs and Types

Consider the following plait program:

```
(lambda (x y)
  (if x
    (+ y 1)
    (+ y 2)))
```

If we enter this program into plait, e.g., as follows, something remarkable happens:

```
> (lambda (x y)
    (if x
      (+ y 1)
      (+ y 2)))
- (Boolean Number -> Number)
#<procedure>
```

In response, plait *figures out* the type of this function *without* our having to provide any annotations. This is in contrast to the type-checker we just wrote, which required us to extend the syntax just to provide (required) type annotations. That tells us that something different—and more—must be happening under plait.

Unannotated Programs and Types

In contrast, consider another example:

```
(lambda (x)
  (if x
    (+ x 1)
    (+ x 2)))
```

This produces an error, observing that we are using x both in a position that requires it to be a Boolean (in `if`) and a number (in the two additions). Again, plait has figured this out without our having to write any annotations at all!

The algorithm that sits underneath plait is essentially the same algorithm under OCaml, Haskell, and several other programming languages. These languages provide *type inference*: figuring out (inferring) types automatically from the program source. Now we're going to see how this works.

Unannotated Programs and Types

The key idea is to break this seemingly very complex problem into two rather simple parts.

- In the first, we recursively visit each sub-expression of the program (following SImPl) and generate a set of *constraints* that formally do what we've been doing informally above.
- The second phase *solves* this set of constraints, using a process that is a generalization of the process you used for solving "systems of simultaneous equations" in school.

The solution is a *type* for each variable. That lets us fill in the annotations that the programmer left blank.

The process of generation will also have applied the type constraints, so there will be no further need to type-check the program;

- But we can use the annotations, for instance, in an IDE for tool-tips, in a compiler for optimization, etc.
- That is, with inference, we can program as if we're in a "scripty" language without annotations, yet achieve most of the benefits of types.
- (I say "most" because one of the benefits is documentation; leaving off all annotations makes programs harder to read and understand. For that reason, inference should be used sparingly.)

Imagining a Solution

Until now, our type checker has required us to annotate the parameter of every function. But let's imagine someone handed us a piece of code without annotations; can we figure out the type anyway? For instance, consider:

```
(+ 1 2)
```

We clearly know the type of this; even our type-checker can calculate it for us without any annotations. But of course that's not surprising: there are no variables to annotate. So now consider this expression:

```
(lambda (x : ___) (+ x 1))
```

With a moment's inspection, we can tell that the function has type `(Num -> Num)`. But our type-checker couldn't have calculated that, because it would have tripped on the empty annotation. So how can *we* figure it out?

Imagining a Solution

Well, let's see. First we have to figure out the type of `x`. To determine its type, we should look for uses of `x`. There is only one, and it's used in an addition. But the rule for addition

```
Γ |- e1 : Num    Γ |- e2 : Num
------------------------------
Γ |- (+ e1 e2) : Num
```

tells us that the term in that position must have type `Num`.

- There is no additional information we have about `x` (this remark will become clearer in a moment).
- Therefore, we can determine that its type must be `Num`.
- Furthermore, we know that the result of an addition is also a `Num`.

From that, we can conclude that the function has type `(Num -> Num)`.

Unique Variable Names

In what follows, we will assume that all variable names in the program are unique. That is, a given variable name is bound in at most one place in a program. This greatly simplifies the presentation below, because we can speak of the type of a variable and know *which* variable it refers to, instead of having to constantly qualify which variable of that name we mean.

This restriction does not actually preclude any programs in a language with static scope. Consider this program, which produces 7:

```
(let ([x 3])
  (+ (let ([x 4])
       x)
     x))
```

We can just as well *consistently rename* one of the xs to something else (heck, we can even use the DrRacket interface to have Racket do the renaming for us), and leave the program meaning exactly the same:

```
(let ([x 3])
  (+ (let ([y 4])
       y)
     x))
```

This renaming process is called *alpha conversion* or *alpha renaming*.

More Informal Examples

With that important detail out of the way, let's return to our process of *inferring* or *reconstructing* the types of variables from the way they're used in a program. Here's another example with a two-parameter function:

```
(lambda ((x : ___) (y : ___))
  (if x
      (+ y 1)
      (+ y 2)))
```

Once again, we can't just calculate its type with our type-checker; instead, we must reconstruct the type from the function body. Let's do that. What can we tell?

Let's again refer to the conditional rule:

```
Γ |- C : Bool    Γ |- T : U    Γ |- E : U
-----------------------------------------
Γ |- (if C T E) : U
```

- This tells us that what's in the `C` position—here, `x`—must be a `Bool`.
- Furthermore, both branches `(+ y 1)` and `(+ y 2)` must have the same type.

That's all we can learn from the rule for `if`!

More Informal Examples

But now we can (and must) recur into the sub-expressions.

```
(lambda ((x : ___) (y : ___))
  (if x
      (+ y 1)
      (+ y 2)))
```

- Each one is an addition, and the addition rule tells us that both arguments must be `Num`s.
- Both of these indicate that the type of `y` must be `Num`.

Furthermore, both indicate that the overall addition returns a `Num`. From that we can tell that the entire expression must have the type

```
(Bool Num -> Num)
```

By this process,

- we can figure out what types to put in the missing annotations.
- More subtly, notice that by running through this process, we have effectively applied all the typing rules;
- therefore, if we have successfully reconstructed the type annotations, we need not bother type-checking the program with those annotations: it will have to type-check.

More Informal Examples

Now let's consider a slight variation on the above program:

```
(lambda (x : ___)
  (if x
      (+ x 1)
      (+ x 2)))
```

Now let's figure out everything we can learn about `x` from the function's body:

- `x` is used in the conditional position of an `if`. Therefore, it must have type `Bool`.
- `x` is used as a parameter to `+`. Therefore, it must have type `Num`.
- `x` is again used as a parameter to `+`. Therefore, it must have type `Num`.

Notice that each of these conclusions is perfectly fine on its own. However, when we *put them together* (which is what we meant by "additional information" above), there's a problem:

- `x` cannot be both of those.
- That is, we are unable to find a single type for `x`.
- This inability to find a type for `x` means that the program has a *type error*.

And indeed, there is no type we could have given that would have enabled this program to execute safely.

More Informal Examples

Observe something subtle.

- While we can report that the program clearly has a type error, our error message must necessarily be much more ambiguous.
- Previously, when we had a type annotation on `x`, we could pinpoint where the error occurred.
- Now, all we can say is that the program is not type-*consistent*, but cannot blame one spot or the other without potentially misleading the programmer.

Instead, we must report all these locations and let the programmer decide where the error is based on their *unstated intent* (in the form of a type annotation).

Algorithmic Details

The details of this algorithm—called *Hindley-Milner* inference—are fascinating, and worked out in detail in both the first and second editions of the book, PLAI (Chapter 30 in the first edition and Chapter 15.3.2 in the second edition).

For several worked examples of both constraint generation and constraint solving, refer to the first edition.

The first edition has a more algorithmic presentation, while the second provides code (it may be useful to compare the two). The prose in the second is different from that in the first, so different readers may prefer one over the other.

Algebraic Datatypes

We have written numerous `define-type` definitions so far, e.g., for expressions. Now we will study this mechanism, which is increasingly found in many new programming languages, in more detail.

To simplify things, consider a simple plait data definition of a binary tree of numbers:

```
(define-type BT
  [mt]
  [node (v : Number) (l : BT) (r : BT)])
```

The `define-type` construct here is doing three different things, and it's worth teasing them apart:

1. Giving a *name* to a new type, `BT`.
2. Allowing the type to be defined by multiple cases or *variants* (`mt` and `node`).
3. Permitting a *recursive* definition (`BT` references `BT`).

Algebraic Datatypes

It's worth asking whether all these pieces of functionality really have to be bundled together, or whether they can be handled separately.

- While they can indeed be separated, they often end up working in concert, especially when it comes to recursive definitions, which are quite common.
- A recursive definition needs a name for creating the recursion; therefore, the third feature requires the first.
- Furthermore, a recursive definition often needs a non-recursive case to "bottom out"; this requires there to be more than one variant, using the second feature.

Putting the three together, therefore, makes a lot of sense.

Algebraic Datatypes

This construct is called an *algebraic datatype*, sometimes also known as a “sum of products”.

- That is because the variants are read as an “or”: a `BT` is an `mt` or a `node`.
- Each variant is an “and” of its fields: a node has a `v` and an `l` and an `r`.
- In Boolean algebra, “or” is analogous to a sum and “and” is analogous to a product.

Sometimes, you will also see this referred to as a *tagged union*. The word “union” is because we can conceptually think of a `BT` as a union of `mt`s and `node`s. The tag is the constructor. This term makes more sense once we compare it against “untagged” union types.

Generated Bindings

Now the question is, how do we type code that uses such a definition? First, let's take an inventory of all the definitions that this might create. It at least creates two `constructors`:

```
(mt : ( -> BT))
(node : (Number BT BT -> BT))
```

We have been starting our interpretation and type-checking with the empty environment,

- but there is no reason we need to,
- nor do we do so in practice:
- the primordial environment can contain all kinds of pre-defined values and their types.

Thus, we can imagine the `define-type` above adding the above two definitions to the initial type environment, enabling uses of `mt` and `node` to be type-checked.

Generated Bindings

This much is standard across various languages. But less commonly, in plait you get two more families of functions: `predicates` for distinguishing between the variants:

```
(mt? : (BT -> Boolean))
(node? : (BT -> Boolean))
```

and accessors for getting the values out of fields:

```
(node-v : (BT -> Number))
(node-l : (BT -> BT))
(node-r : (BT -> BT))
```

Static Type Safety

We should be troubled by the types of these accessors. They seem to indiscriminately try to pull out field values, *whether the variant has them or not*. For instance, we can write and type-check this program, which is appealing:

```
(size-correct : (BT -> Number))
(define (size-correct (t : BT))
  (if (mt? t)
      0
      (+ 1 (+ (size-correct (node-l t)) (size-correct (node-r t))))))

(test (size-correct (mt)) 0)
```

Static Type Safety

However, we can just as well type-check *this* program:

```
(define (size-wrong (t : BT))
    (+ 1 (+ (size-wrong (node-l t)) (size-wrong (node-r t)))))
```

This should not type-check because it has a clear type-error. The type of `size-wrong` is

```
(size-wrong : (BT -> Number))
```

so it is perfectly type-correct to write:

```
(size-wrong (mt))
```

But running this, of course, results in a run-time error, the very kind of error we might have hoped the type-checker would catch.

Pattern-Matching and Type-Checking

This kind of error cannot occur naturally in languages like OCaml and Haskell. Instead of exposing all these predicates and accessors, instances of an algebraic datatype are deconstructed using pattern-matching. Thus, the size computation would be written as (-pm stands for "pattern matching"):

```
(size-pm : (BT -> Number))

(define (size-pm t)
  (type-case BT t
    [(mt) 0]
    [(node v l r) (+ 1 (+ (size-pm l) (size-pm r)))]))
```

This might seem like a convenience—

- it certainly makes the code much more compact and perhaps also much more readable—
- but it's also doing something more.
- The pattern-matcher is effectively baked into the way programs are type-checked.

Pattern-Matching and Type-Checking

```
(size-pm : (BT -> Number))

(define (size-pm t)
  (type-case BT t
    [(mt) 0]
    [(node v l r) (+ 1 (+ (size-pm l) (size-pm r)))]))
```

That is, the above algebraic datatype definition effectively adds the following typing rule to the type checker:

```
Γ |- e : BT
Γ |- e1 : T
Γ[V <- Number, L <- BT, R <- BT] |- e2 : T
------------------------------------------
Γ |- (type-case BT e
       [(mt) e1]
       [(node V L R) e2]) : T
```

Pattern-Matching and Type-Checking

```
Γ |- e : BT
Γ |- e1 : T
Γ[V <- Number, L <- BT, R <- BT] |- e2 : T
------------------------------------------
Γ |- (type-case BT e
       [(mt) e1]
       [(node V L R) e2]) : T
```

- The first antecedent is clear: we have to confirm that the expression `e` evaluates to a `BT` before we pattern-match `BT` patterns against it.
- The second type-checks `e1` in the same environment as in the consequent because the `mt` variant does not add any local bindings.
- The type of this expression needs to be the same as the type from the other branch, due to how we're handling conditionals.
- Finally, to type-check `e2`, we have to extend the consequent's type environment with the bound variables; their types we can read off directly from the data *definition*.
- In short, the above typing rule can be defined automatically by desugaring.

Pattern-Matching and Type-Checking

Aside: Notice that there is also an assume-guarantee here: we type-check `e2` in an environment that *assumes* the annotated types; this is *guaranteed* by the `node` constructor.

In particular, observe what we *couldn't* do!

- We didn't have awkward selectors, like `node-v`, for which we had to come up with some type.
- By saying they consumed a `BT`, we had to let them statically consume any kind of `BT`, which caused a problem at run-time.
- Here, there is *no selector*: pattern-matching means we can only write pattern-variables in variants where the algebraic datatype definition permits it,
- and the variables automatically get the right type.

Thus, pattern-matching plays a crucial role in the *statically safe* handling of types.

Algebraic Datatypes and Space

Earlier, we've seen that types can save us both time and space. We have to be a little more nuanced when it comes to algebraic datatypes.

- The new type introduced by an algebraic datatype still enjoys the space saving.
- Because the type checker can tell a `BT` apart from every other type, at run-time we don't need to record that a value is a `BT`:
- it doesn't need a type-tag.
- However, we still need to tell apart the different *variants*:

the function `size-pm` effectively desugars into (`-ds` stands for "desugared"):

```
(define (size-pm-ds (t : BT))
  (cond
    [(mt? t) 0]
    [(node? t)
     (let ([v (node-v t)]
           [l (node-l t)]
           [r (node-r t)])
        (+ 1 (+ (size-pm-ds l) (size-pm-ds r))))]))
```

(We've introduced the `let` to bind the names introduced by the pattern.)

Algebraic Datatypes and Space

- What this shows is that at run-time, there are conditional checks that need to know what kind of `BT` is bound to `t` on this iteration.
- Therefore, we need just enough tagging to tell the variants apart.
- In practice, this means we need as many bits as the logarithm of the number of variants; since this number is usually small,
- this information can often be squeezed into other parts of the data representation.

Union Types and Retrofitted Types

Typed Racket is an instance of a *retrofitted* type system: adding a type system to a language that did not previously have types.

The original language, which does not have a static type system, is usually called *dynamic*.

There are now numerous retrofitted type systems: e.g.,

- TypeScript for JavaScript and
- Static Python for Python.
- There are even multiple retrofitted type systems for some languages: e.g., both TypeScript and Flow add types to JavaScript.

Union Types and Retrofitted Types

The goal of a retrofitted type system is to turn run-time errors into static type errors.

Due to the Halting Problem, we cannot precisely turn every single run-time error into a static one, so the designer of the type system must make some decisions about which errors matter more than others.

In addition, programmers have already written considerable code in many dynamic languages, so changes that require programmers to rewrite code significantly would not be adopted. Instead, as much as possible, type system designers need to accommodate *idiomatic type-safe programs*.

Union Types and Retrofitted Types

Algebraic datatypes present a good example. Typically, they have tended to not be found in dynamic languages.

Instead, these languages have some kind of structure definition mechanism

- (such as classes, or lightweight variants thereof, like Python's dataclasses).
- Therefore, the elegant typing that goes with algebraic datatypes and their pattern-matching does not apply.

Because it is not practical to force dynamic language programmers to wholesale change to this "new" (to that dynamic language) style of programming, type system designers must find the idioms they use (that happen to be type-safe) and try to bless them. We will look at some examples of this.

Union Types and Retrofitted Types

A good working example of a retrofitted typed language is Typed Racket, which adds types to Racket while trying to preserve idiomatic Racket programs. (This is in contrast to plait, which is also a typed form of Racket but does *not* try very hard to preserve Racket idioms. The accessors we saw earlier, for algebraic datatypes, are forgiving in what they accept, at the cost of static safety.)

You Get a Type! And You Get a Type! And You Get a Type!

Let's return to our non-statically-type-safe accessors in plait: e.g.,

```
(node-v : (BT -> Number))
```

In a way, it's not fair to blame the accessor: the fault is really with the constructor, because

```
(node : (Number BT BT -> BT))
```

Once the node constructor creates a `BT`, the information about `node`-ness is lost, and there's not much that the accessors can do. So perhaps the alternative is to *not* create a `BT`, but instead create a value of the `node` type.

You Get a Type! And You Get a Type! And You Get a Type!

So let's start over. This time, we'll use a different typed language, Typed Racket:

```
#lang typed/racket
```

In Typed Racket, we can create products, called structures, which define a new type:

```
(struct mt ())
```

This creates a constructor with the type we'd expect:

```
> mt
- : (-> mt)
#<procedure:mt>
```

You Get a Type! And You Get a Type! And You Get a Type!

It also creates a predicate, whose type is a bit different; previously we had a function that could only take a `BT`, because it didn't make sense to apply `mt?` to any other type. Now, however, there isn't even a concept of a `BT` (yet), so `mt?` will take values of any type:

```
> mt?
- : (-> Any Boolean : mt)
#<procedure:mt?>
```

(The additional text, `: mt`, is telling us when the Boolean is true; ignore this for now.)

Now let's try to define nodes. Here we run into a problem:

```
(struct node ([v : Number] [l :
```

Union Types

Now let's try to define nodes. Here we run into a problem:

```
(struct node ([v : Number] [l :
```

Oops—what do we write here?!? We have to also introduce a notion of a binary tree.

- But we already have two existing types, `mt` and (in progress) `node`.
- Therefore, we need a way to define a binary tree that has a sum that combines these two existing types.

This suggests that we have a way of describing a new type as a *union* of existing types:

```
(define-type-alias BT (U mt node))
```

Observe that in this case, there are no special constructors to distinguish between the two kinds of `BT`. Therefore, this is called an *untagged union*, in contrast to tagged unions.

Union Types

Now we can go back and complete our definition of node:

```
(struct node ([v : Number] [l : BT] [r : BT]))
```

Now let's look at what Typed Racket tells us are the types of node's constructor, predicate, and selectors:

```
> node
- : (-> Number BT BT node)
> node-v
- : (-> node Number)
> node-l
- : (-> node BT)
> node-r
- : (-> node BT)
```

Union Types

Using these definitions we can create trees: e.g.,

```
(define t1
  (node 5
        (node 3
              (node 1 (mt) (mt))
              (mt))
        (node 7
              (mt)
              (node 9 (mt) (mt)))))
```

But now let's try to write a program to compute its size:

```
(define (size-tr [t : BT]) : Number
  (cond
    [(mt? t) 0]
    [(node? t) (+ 1 (size-tr (node-l t)) (size-tr (node-r t)))]))
```

Union Types

But now let's try to write a program to compute its size:

```
(define (size-tr [t : BT]) : Number
  (cond
    [(mt? t) 0]
    [(node? t) (+ 1 (size-tr (node-l t)) (size-tr (node-r t)))]))
```

It is not clear at all that this program should type-check. Consider the expression `(node-l t)`. The type of `node-l` expects its argument to be of type `node`. However, all we know is that `t` is of type `BT`. Yet this program type-checks!

Union Types

The fact that this does type-check, however, should not fill us with too much joy. We saw how `size-wrong` type-checked, only to halt with an undesired run-time error. So what if we instead write its analog, which is this?

```
(define (size-tr-wrong [t : BT]) : Number
  (+ 1 (size (node-l t)) (size (node-r t))))
```

This program does **not** type-check! Instead, it gives us a type error of exactly the sort we would have expected: `node-l` and `node-r` both complain that they were expecting a `node` and were given a `BT`. So the wonder is not that `size-tr-wrong` has a type-error, but rather that `size-tr` does not!

Union Types

To understand why it type-checks, we have to go back to the types of the predicates:

```
> mt?
- : (-> Any Boolean : mt)
> node?
- : (-> Any Boolean : node)
```

Critically,

- the `: mt` and `: node` are Typed Racket's way of saying that the Boolean will be true only when the input is an `mt` or `node`, respectively.
- This crucial *refinement* information is picked up by the type-checker.
- In the right-hand-side of the `cond` clauses, it *narrows* the type of `t` to be `mt` and `node`, respectively.
- Thus, `(node-l t)` is type-checked in a type environment where the type of `t` is `node` and not `BT`.

Union Types

To test this theory, we can try another wrong program:

```
(define (size-tr-w2 [t : BT]) : Number
  (cond
    [(node? t) 0]
    [(mt? t) (+ 1 (size-tr-w2 (node-l t)) (size-tr-w2 (node-r t)))]))
```

Here, we have swapped the predicates. It is not only important that this version produces a type error, it is also instructive to understand why, by reading the type error.

- This explicitly says that the program expected a `node` (for instance, in `node-l`)
- and was given an `mt` (based on the `mt?`).

This confirms that Typed Racket is refining the types in branches based on predicates.

If-Splitting

To summarize, `size-tr` type-checks is because the type-checker is doing something special when it sees the pattern

```
(define (size-tr [t : BT]) : Number
  (cond
    [(mt? t) …]
    [(node? t) …]))
```

It knows that every `BT` is related to `mt` and `node` through the union.

- When it sees the predicate, it *narrows* the type from the full union to the branch of the union that the predicate has checked.
- Thus, in the `mt?` branch, it narrows the type of `t` from `BT` to `mt`;
- in the `node?` branch, similarly, it narrows the type of `t` to just `node`.

Now, `node-l`, say, gets confirmation that it is indeed processing a `node` value, and the program is statically type-safe.

If-Splitting

In the absence of those predicates, in `size-tr-wrong`, the type of `t` does not get narrowed, resulting in the error. In `size-tr-w2`, swapping the predicates also gives an error. Here is one more version:

```
(define (size-tr-else [t : BT]) : Number
  (cond
    [(mt? t) 0]
    [else (+ 1 (size-tr (node-l t)) (size-tr (node-r t)))]))
```

This program could go either way!

- It just so happens that it does type-check in typed/racket, because typed/racket is "smart" enough to determine that
  - there are only two kinds of `BT` and one has been excluded,
  - so in the `else` case, it must be the other kind.
- But one could also imagine a less clever checker that expects to see an explicit test of `node?` to be able to bless the second clause.

If-Splitting

In short, both the algebraic datatype and union type approaches need some special treatment of syntax by the type-checker to handle variants.

- In the former case it's through pattern-matching.
- The narrowing technique above is sometimes called `if-splitting`, because an `if` (which `cond` and other conditional constructs desugar to) "splits" the union.

You will sometimes also see the terms *occurrence typing* and *flow typing* to describe variants of the ideas in this chapter.

Aside: This idea was invented by Typed Racket by studying how programmers write code in Scheme and Racket programs. It has later proved to be relevant to many real-world retrofitted type systems.

Introducing Union Types

As we discussed when evaluating conditionals, union types can be useful to represent partial functions. There are several ways of handling them:

- https://dcic-world.org/2022-08-28/partial-domains.html

Using an option type avoids the need for ad-hoc type unions.

- If we have unions anyway, however, then we can give types to partial functions: e.g.,
  - `(V U Boolean)` in Racket or
  - `(V U None)` in Python, respectively, where `V` is the normal return type.
- Thus, Racket's `string->number` can be given the type `(Number U Boolean)`.

What we've just seen is that with if-splitting, we can eliminate union types. That then raises the possibility that we can also introduce union types!

Introducing Union Types

One way is of course by giving union types to built-in functions, as above. But what about in user programs?

- Previously we had rejected such a solution: if we introduced a union, we had no way to deal with it.
- Now we can safely introduce them in languages that have solutions for deconstructing them.

How do we introduce union types? Curiously, using the same construct that eliminates them! Observe that we no longer need both branches of a conditional to return the same type:

```
Γ |- C : Bool    Γ |- T : V    Γ |- E : W
-----------------------------------------
Γ |- (if C T E) : (U V W)
```

where our notation means "the union of the types represented by `V` and `W`".

How Many Unions?

When we wrote an algebraic datatype, the variants “belonged” to the new type. We had no mechanism for mixing-and-matching variants.

In contrast, with union types, a new type is a collection of existing types. There’s nothing that prevents those existing types from engaging in several different unions. For instance, we had

```
(define-type-alias BT (U mt node))
```

But we could also write, say,

```
(struct link ((v : Number) (r : LinkedList)))
```

and reusing `mt` to define

```
(define-type-alias LinkedList (U mt link))
```

How Many Unions?

Therefore, given an `mt`, what "is" it?

- Is it a `BT`?
- A `LinkedList`?
- It's all those, but it's also just an `mt`, which can participate in any number of unions.

This provides a degree of flexibility that we don't get with algebraic datatypes—

- since we can create ad-hoc unions of existing types—

but that also means it becomes harder to tell all the ways a value might be used, and also complicates inferring types

- (if we see an `mt` constructed, are we also constructing a `BT`? a `LinkedList`?).

The Hindley-Milner inference algorithm doesn't cover these cases, though it can be extended to do so.

Union Types and Space

Therefore, union types combined with if-splitting gives us an alternate approach of obtaining something akin to algebraic datatypes in our programming language.

However, we don't obtain the space benefits of the algebraic datatype definition.

- We created two distinct types; in principle, that's not a problem.
- However, to write programs, we needed to have predicates (`mt?` and `node?`) that took any value.
- Therefore, those predicates need type-tags on the values to be able to tell what kind of value they are looking at.

Observe that these are *type* tags, not *variant* tags, so the amount of space they need is proportional to the number of types in the whole program, not just the number of variants in that particular algebraic datatype definition.

If-Splitting with Control Flow

This pattern, of dispatching based on type-tests and values, is quite common in dynamic (or “scripting”) languages.

- These languages do not have a static type system,
- but they do have safe run-times, which attach type tags to values and provide predicates that can check them.
- Programmers then adopt programming patterns that take advantage of this.

If-Splitting with Control Flow

Aside: The term *dynamic* language seems to have no clear fixed definition.

- It means, at least, that the language doesn't have static types.
- Sometimes it's implicit that the language is nevertheless safe.
- But some people use it to mean that the language has features that let you do things like inspect or even modify the program as it's running (features like `eval`).

In this book I use it in the second sense: not-statically typed, but still safe.

Aside: What, then, is a "scripting" language?

- I use the term to mean a dynamic language that is also very liberal with its types:
- e.g., many operations are either overloaded and/or very forgiving of what a statically-typed language would consider an error.

Scripting languages tend to be dynamic in all three senses:

1. they do not have a static type-system,
2. they are safe,
3. and they tend to have rich features for introspection and even modification.

They are designed to maximize expressiveness and thus minimize just about any useful static analysis.

If-Splitting with Control Flow

For instance, here's an example from JavaScript, of a serialization function. A serializer takes a value of (almost) *any* type and converts it into a string to be stored or transmitted. (This version is adapted from version 1.6.1 of `Prototype.js`.)

```
function serialize(val) {
  switch (typeof val) {
    case "undefined":
    case "function":
      return false;
    case "boolean":
      return val ? "true" :
                   "false";
    case "number":
      return "" + val;
    case "string":
      return val;
  }
  if (val === null)
    { return "null"; }
```

```
  var fields = [ ];
  for (var p in val) {
    var v = serialize(val[p]);
    if (typeof v === "string") {
      fields.push(p + ": " + v);
    }
  }
  return "{ " +
         fields.join(", ") +
         " }";
}
```

If-Splitting with Control Flow

Now suppose we're trying to retrofit a type system onto JavaScript. We would need to type-check such programs. But before we even ask *how* to do it, we should know what answer to expect: i.e., is this program even type-safe?

The answer is quite subtle. It uses JavaScript's `typeof` operator to check the tags.

- For two kinds of values, it returns `false`
  - (that is, the type of this function is not `Any -> String`, it's actually `Any -> (String U Boolean)`,
  - where the `false` value is used to signal that the value can't be serialized—
  - observe that an actual `false` value is serialized to `"false"`).
- For Booleans, numbers, and strings, it translates them appropriately into strings.
- In all these cases, execution returns.
- (Note, however, that the code also exploits JavaScript's "fall-through" behavior in `switch`, so that `"undefined"` and `"function"` are treated the same without having to repeat code. The type-checker needs to understand this part of JavaScript semantics.)

If none of these cases apply, then execution falls through;

- we need to know enough JavaScript to know that this corresponds to the one other return from `typeof`, namely objects.
- Now the code splits between objects that are and aren't `null`.
- In the non-`null` case, it iterates through each field, serializing it in turn.

Therefore, this program is actually type-safe…but for very complicated reasons!

If-Splitting with Control Flow and State

Here's another program, taken from the Python 2.5.2 standard library:

```
def insort_right(a, x, lo=0, hi=None):
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        if x < a[mid]:
            hi = mid
        else:
            lo = mid+1
    a.insert(lo, x)
```

This function inserts an element (`x`) into an already-sorted list (`a`).

It also takes a low search interval index (`lo`), which defaults to `0`, and a high interval (`hi`), which defaults to `None`.

It inserts the element into the right place in the array.

If-Splitting with Control Flow and State

Now let's ask whether this is actually type-correct. Observe that `lo` and `hi` are used in several arithmetic operations. These are the ones we're most interested in.

If it helps, here's the code with type annotations in Static Python:

```
from typing import Optional

def insort_right(a, x, lo: int = 0, hi: Optional[int] = None):
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        if x < a[mid]:
            hi = mid
        else:
            lo = mid+1
    a.insert(lo, x)
```

(In Static Python, `Optional[T]` is an abbreviation for `(T U None)`. So the annotation on `hi` above allows the user to pass in either an `int` or `None`. What makes the last two arguments optional is (perhaps confusingly) not the type `Optional` but rather the fact that they have default values in the function header.)

If-Splitting with Control Flow and State

It's easier to see what's happening with `lo`: it's allowed to be optional;

- if the optional argument is provided, it must be an `int`; and
- if it's not provided, it has value `0`, which also has type `int`.

So its type is effectively `(int U int)`, which is just `int`, so all uses of `lo` as an `int` are fine.

But now consider the type of `hi`. It is also optional.

- If it is provided, it has to be an `int`, which would be fine. But
- if it's *not* provided, its value is `None`, which cannot be used in arithmetic.
- However, right at the top, the function checks whether it is `None` and, if so, *changes* it to the result of `len(a)`—which is an `int`.

Therefore, once the `if` is done, no matter which path the program takes, `hi` is an `int`. Thus, the program is actually type-safe.

If-Splitting with Control Flow and State

That's all well and good for us to reason about by hand. However, our job is to build a type-checker that will neither reject programs needlessly nor approve type-incorrect programs. This balance is very hard to maintain.

This represents the challenge retrofitted type system designers face:

- they must either reject idiomatic programs or
- add complexity to the type system to handle them.

If we reject the program, we reject many other programs like it, which are idiomatically found in many "scripting" languages.

- The result would be very safe, but also very useless—
- indeed, safe *because* it would be very useless—type-checker (a type-checker that rejects every program would be extremely safe…).

Instead, we need an even more complicated solution than what we have seen until now.

Aside: See this paper:

- https://cs.brown.edu/people/sk/Publications/Papers/Published/gsk-flow-typing-theory/

for how to type such programs.

The Price of Retrofitting

Retrofitting a type-system onto an existing untyped language clearly puts a heavy burden on the creator of the type system.

But it also puts a burden on developers.

- If the type system is to not reject a bunch of existing code, then it must be based on some heuristics about program structure.
- The more complex these heuristics grow (as we've seen hints of in this chapter), the stranger it will be when a program falls outside what they can handle.

You might argue that it was ever thus: when type-checking algebraic datatypes, too, we had to use pattern-matching to help the type-checker.

- The difference there is that the type-checker was around at program *construction* time, so we adhered to its rules from the very start;
- we didn't try to add types after the fact.

The problem arises when programmers are allowed to write code however they like, and the type-checker must retroactively try to bless them.

Types and Tags

Finally, we should clarify something important about the `typeof` operator in JavaScript, which is analogous to the `type` function in Python.

- When we impose a type system on JavaScript, we expect, say, the type (`Number -> String`) to be different from the type (`String -> Boolean`).
- Similarly, an object that contains only the fields `x` and `y` is very different from the object that contains only the method `draw`.

However, these nuances are lost on `typeof`, which is innocent to even the existence of any such type systems.

- Therefore, all those functions are lumped under one tag, `"function"`,
- and all those objects are similarly treated uniformly as one tag, `"object"` (and analogously in Python).

This is because their names are misleading: what they are reporting are not the *types* but rather the run-time *tags*.

Types and Tags

The difference between types and tags can grow arbitrarily big.

- After all, the number of types in a program can grow without bound, and so can their size (e.g., you can have a list of lists of arrays of functions from ...).
- But the set of tags is fixed in many languages, though in those that allow you to define new (data)classes, this set might grow.

Nevertheless, tags are meant to take up a fixed amount of space and be checked in a small constant amount of time.

Of course, this difference is not inherently problematic. After all, even in statically-typed languages with algebraic datatypes, we still need space to track variants, which requires a kind of (intra-type) tag.

The issue is rather with the choice of *name*: that `typeof` and `type` do not, actually, return “types”. A more accurate name for them would be something like `tagof`, leaving the term “type” free for actual static type systems.

Nominal Types, Structural Types, and Subtyping

Let's go back to

```
(define-type BT
  [mt]
  [node (v : Number) (l : BT) (r : BT)])
```

and ask how we could have represented this in Java.

Do Now: Represent this in Java!

How did you do it? Did you create a single class with `null` for the empty case?

Exercise: Why is that solution not object-oriented?

Algebraic Datatypes Encoded With Nominal Types

We'll take a different approach. Observe from the datatype definition that we have two constructors and one type that represents their union. We can encode this in Java as:

```
abstract class BT {
  abstract public int size();
}

class node extends BT {
  int v;
  BT l, r;
  node(int v, BT l, BT r) {
    this.v = v;
    this.l = l;
    this.r = r;
  }
  public int size() {
    return 1 + this.l.size() + this.r.size();
  }
}
```

```
class mt extends BT {
  public int size() {
    return 0;
  }
}

class Main {
  public static void main(String[] args) {
    BT t = new node(5,
      new node(3, new mt(), new mt()),
      new mt());
    System.out.println(t.size());
  }
}
```

Algebraic Datatypes Encoded With Nominal Types

How is the “if-splitting” addressed here? It’s done in a hidden way, through dynamic dispatch.

- When we invoke a method, Java makes sure we run the right method:
- there are actually two concrete `size` methods, and the run-time picks the right one.
- Once that choice is made, the class in which the method resides automatically determines what is bound.

Thus, the `size` in `node` can safely use `this.l` and `this.r`, and the type-checker knows that those fields exist.

This is, then, similar to, yet different from, our two prior solutions: using algebraic datatypes and union types.

The solutions are structurally different, but they are all similar in that some *syntactic* pattern must be used to make the program statically type-able.

- With algebraic datatypes, it was pattern-matching;
- with union types, it was if-splitting;
- in Java, it’s the splitting of the code into separate methods.

Algebraic Datatypes Encoded With Nominal Types

The algebraic datatype and Java solutions are even more connected than we might imagine.

- With algebraic datatypes, we fixed the set of variants;
  - but we were free to add new functions *without having to edit existing code*.
- In Java, we fix the set of behaviors (above, one method),
  - but can add new variants without having to edit existing code.

Therefore, neither has an inherent advantage over the other, and one's strengths are the other's weakness. How to do *both* at once is the essence of the Expression Problem. See also the concrete examples and approaches given in these two papers, one focusing on a Java-based approach and another function-centric.

Nominal Types

The type system in Java is representative of an entire class of languages. These have *nominal* types, which means the *name* of a class matters. ("Nominal" comes from the Latin *nomen*, or name.) It's easiest to explain with an example.

Above we have the following class:

```
class mt extends BT {
  public int size() {
    return 0;
  }
}
```

Let's now suppose we create another class that is identical in every respect but its name:

```
class empty extends BT {
  public int size() {
    return 0;
  }
}
```

Nominal Types

Let's say we have a method that takes `mt` objects:

```
class Main {
  static int m(mt o) {
    return o.size();
  }
  public static void main(String[] args) {
    System.out.println(m(new mt()));
  }
}
```

Nominal Types

But observe that empty is a perfectly good substitute for mt: it too has a size method, which too takes no arguments, and it too returns an int (in fact, the very same value). Therefore, we try:

```
class Main {
  static int m(mt o) {
    return o.size();
  }
  public static void main(String[] args) {
    System.out.println(m(new empty()));
  }
}
```

But Java rejects this. That's because it expects an object that was constructed by the actual class mt, not just one that "looks like" it. That is, what matters is which actual (named) class, not what *structure* of class, created the value.

Structural Types

In contrast, we can imagine a different type system:

- one where the type of each of the above classes is not its name
- but rather a description of what fields and methods it has:
- i.e., it's structure, or its "services".

For instance, we might have:

```
mt : {size : ( -> int)}
node : {size : ( -> int)}
```

That is, each of these is a collection of names (one name, to be precise), which is a method that takes no parameters and returns an `int`.

Whenever two types are the same, objects of one can be used where objects of the other kind are expected. Indeed, it is unsurprising that both kinds of trees have the same type, because programs that process one will invariably also need to process the other because trees are a union of these two types.

Structural Types

Similarly, we also have

```
empty : {size : ( -> int)}
```

The above m method might be written as:

```
static int m(o : {size : (-> int)}) {
  return o.size();
}
```

That is, it only indicates what shape of object it expects, and doesn't indicate which constructor should have made it.

- This is called *structural* typing,
- though the Internet appears to have decided to call this “duck” typing
- (though it's hard to be clear: there is no actual theory of duck typing to compare against well-defined theories of structural typing: Abadi and Cardelli represent a classical viewpoint, and here's an extension for modern “scripting” languages).

Nominal Subtyping

We've been writing a bit gingerly about Java above: because we know that the `m` method will accept not only `mt`'s but also anything that is a sub-class of `mt`. Let's explore this further.

To simplify things, let's make some basic classes:

```
class A { String who = "A"; }
class B extends A { String who = "B"; }
class C extends A { String who = "C"; }
class D { String who = "D"; }
```

We'll also create a shell "runner":

```
class Main {
  public static void main(String[] args) {
    System.out.println((true ? _____ : _____).who);
  }
}
```

Nominal Subtyping

```
class A { String who = "A"; }
class B extends A { String who = "B"; }
class C extends A { String who = "C"; }
class D { String who = "D"; }
```

We'll try filling in different values for the blanks and seeing what output we get:

```
System.out.println((true ? new B() : new B()).who);
```

Unsurprisingly, this prints “B”. What about:

```
System.out.println((true ? new B() : new A()).who);
```

You might expect this to also print “B”, because that's the value that we created. However, it actually prints “A”!

Let's see a few more examples:

```
System.out.println((true ? new B() : new C()).who);
```

Will this print “B”? No, in fact, this also prints “A”!

Nominal Subtyping

```
class A { String who = "A"; }
class B extends A { String who = "B"; }
class C extends A { String who = "C"; }
class D { String who = "D"; }
```

How about:

```
System.out.println((true ? new B() : new D()).who)
System.out.println((true ? new B() : 3).who)
```

Both of these produce a static error. It's instructive to read the error message: in both cases they reference `Object`.

- In the former case, it's because there is nothing else common to `B` and `D`.
- But in the latter case, the primitive value `3` was effectively converted into an object—`new Integer(3)`—and those two object types were compared.

Nominal Subtyping

What is happening in the type system that causes this error? The cause is documented here:

https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.25.3

Specifically, the document says:
The type of the conditional expression is the result of applying capture conversion (§5.1.10) to lub(T1, T2).

where "lub" stands for "least upper bound": the "lowest" class "above" all the given ones. This type is determined *statically*. That is, the type rule is essentially:

```
Γ |- C : Bool    Γ |- T : V    Γ |- E : W    X = lub(V, W)
----------------------------------------------------------
Γ |- (if C T E) : X
```

Contrast this to the other rules we've seen for conditionals!

- The first type rule we saw was the most rigid, but produced the most usable values (because there was no ambiguity).
- The second type rule, for union types, was less rigid, but as a result the output type could have a union that needed to be split.
- This type rule is even less rigid (in terms of what the two branches produce), but the result could be as general as `Object`, with which we can do almost nothing.

Subtyping

The general principle here is called *subtyping*:

- we say that type `X` is a subtype of `Y`,
- written `X <: Y` (read the `<:` like a "less than" or "contained"),
- whenever `X` can be used wherever a `Y` was expected:
- i.e., `X` can *safely* be *substituted* for `Y`.

Java chose to make sub-*classes* into sub-*types*.

- Not all object-oriented languages do this,
- and indeed many consider it to be a mistake,
- but that's the design Java has.

Therefore, a sub-class is expected to offer at least as many services as its super-class; and hence, it can be substituted where a super-class is expected. The lub computation above finds the *most specific* common super-type.

This is an account of how subtyping works for *nominal* systems. This has the virtue of being fairly easy to understand. We can also define subtyping for *structural* systems, but that is rather more complex: some parts are easy to follow, other parts are a bit more tricky (but essential to obtain a sound type system). For a detailed explanation, with an illustrative example, see section 33.6.1 of PAPL.

From Scripts to Programs

As dynamic language programs grow, they become increasingly hard to maintain.

- Programmers use types to define interfaces, communicate expectations of behavior, document, and so on, and
- in their absence, we need several ad hoc tools.
- Put differently, we want “scripts” to grow up and become “programs”.

Thus, one of the most visible trends in programming languages over the past ten years is dynamic languages adding a static counterpart.

- In principle, this is as simple as adding a type-system to an existing language.
- As we’ve already seen when discussing retrofitted types, however, such a type system needs to take into account the idiomatic style of programming in the language;
- otherwise it would report as erroneous too many programs that are actually type-correct,
- and this high false-positive rate would make people not use the type system at all.

Therefore, we discussed some patterns of code that need to be supported.

From Scripts to Programs

Another major obstacle to adoption is that people often have a large amount of code lying around, and

- it is simply impractical to convert all of it to a typed language in one go.
- In fact, some of it may not even be typeable by most reasonable type systems:
- e.g., the `eval` construct, which takes a *dynamic* string (e.g., one that may be constructed on-the-fly during program execution) and runs it.
- By definition, we statically do not know what this string is; without knowing it, we can't possibly type it statically.

From Scripts to Programs

In short, there are two reasons why we cannot expect the whole program to make an instant transition from untyped to typed:

- The program is too large, and programmers have other things to do with their time.
- Some parts of the program may not even be typeable.

(You don't need `eval` to make things hard to type. Many dynamic constructs that look at program behavior and modify it have the same flavor. They enter the language because it's dynamic and doesn't have to worry about a static type discipline, and then create an obstacle for later typing.)

Despite this, type systems have been built for many real-world dynamic languages.

- These type systems exhibit a property called *gradual typing*:
- as the name suggests, you add types "gradually" to the program, hopefully making it more-and-more typed.
- What started out as an academic idea in the Scheme community (two papers in 2006 introduced gradual typing for Scheme) is now widely used in industry.

Micro Versus Macro

In gradual typing, we are going to add annotations to programs and then type-check the program. Within this broad principle, there are two schools of thought.

In “micro” gradual typing we can add annotations to any subset of the variables of the language. We saw this earlier in the Static Python example:

```
def insort_right(a, x, lo: int = 0, hi: Optional[int] = None):
```

Here, we have annotations on `lo` and `hi`, but not on `a` and `x`.

- Ergonomically, this is very convenient for the programmer: use annotations for the parts you care about, and not for the parts you don’t.
- Unfortunately, this comes at a cost: there is now a much more complex language where any parts of a program can be static and any other parts dynamic,
- and they can freely commingle in the same body of code,
- even in a single expression or line (e.g., from the same example: `hi = len(a)`).

Micro Versus Macro

The type system needs to somehow deal with constructs it cannot meaningfully type (like `eval`).

- Also, previously we had a clean and simple soundness result for the typed program;
- now it is rather unclear what soundness means.
- In turn, that means that programmers may put a lot of effort into annotations, but without a clear guarantee of what they are getting in return. (A large body of literature now tries to make sense of this.)

In contrast, there is another approach, often called “macro” gradual typing.

Micro Versus Macro

In the macro approach, there are *two languages*: the typed and the dynamic one.

- They are expected to be very similar—so similar that they have the same run-time system and can freely share values—so we'll refer to them as "sibling" languages.
- However, they may not have the same constructs (e.g., the typed language would not contain eval).
- Instead of freely mixing code between typed and untyped, we only have to figure out what happens when values travel *between* the languages, not within each one.

The expectation is that the programmer will gradually migrate part of their codebase from the dynamic to the typed language, typically a function at a time. Each language can import code from the other, but when importing into typed code, the programmer must specify a type for the imported code.

A canonical example of this approach is Typed Racket. Because Typed Racket is one of the oldest and most developed gradually typed languages (technically, it's the *combination* of Racket and Typed Racket that is gradually typed—Typed Racket itself is fully typed), and also offers some of the most interesting perspective on what happens when values travel between languages, we will use that as our exemplar for study.

Typed Racket at Work

In what follows, it's critical to pay attention to the exact details of error messages!

First, let's write the following function in `#lang racket` and test it out:

```
(define (g s)
  (+ 1 (or (string->number s) 0)))
> (g "5")
6
> (g "hi")
1
```

This is as we would expect: `"5"` represents a valid number and `"hi"` does not, but

```
> (g 5)
string->number: contract violation
  expected: string?
  given: 5
```

because 5 isn't a string at all.

Typed Racket at Work

Now let's define it in Typed Racket:

```
#lang typed/racket

(define (f [s : String]) : Number
  (+ 1 (or (string->number s) 0)))
```

The type-checker confirms that this program is well-typed.

Exercise: As a test, try

```
(define (f [s : String]) : Number
  (+ 1 (string->number s)))
```

and see what happens.

Typed Racket at Work

Now suppose we export this function from Typed Racket:

```
(provide f)
```

and import it into the Racket module:

```
(require "typed.rkt") ;; or whatever filename you've chosen
```

Let's try the same three tests. Predictably, two of them work the same:

```
> (f "5")
6
> (f "hi")
1
```

Typed Racket at Work

The third still produces an error, but a rather different kind of error:

```
> (f 5)
f: contract violation
  expected: string?
  given: 5
  in: the 1st argument of
      (-> string? any)
  contract from: typed.rkt
  blaming: untyped.rkt
   (assuming the contract is correct)
  at: typed.rkt:5:9
```

Here's what is happening. When we export `f` from Typed Racket, we don't just export the function in its raw form. Rather, Typed Racket wraps the function in *contracts* that "protect" it in a dynamic setting.

Typed Racket at Work

Thus, it is as if the function that was exported was

```
(define (wrapped-f s)
  (if (string? s)
      (let ([b (+ 1 (or (string->number s) 0))])
        (if (number? b)
            b
            (error 'contract "returned value was not a Number")))
      (error 'contract "provided value was not a String")))
```

(with suitably different error messages).

- Notice that `wrapped-f` behaves exactly like our imported `f` does:
- the error when given 5 is from a contract check,
- rather than from an internal operation.

Typed Racket at Work

Observe also that this wrapped version is quite easy to produce in a completely mechanical way, i.e., through desugaring:

```
(define (f [s : String]) : Number
  (+ 1 (or (string->number s) 0)))
```

became

```
(define (wrapped-f s)
  (if (string? s)
      (let ([b (+ 1 (or (string->number s) 0))])
        (if (number? b)
            b
            (error 'contract "returned value was not a Number")))
      (error 'contract "provided value was not a String")))
```

Exercise: Why do we bind `b` to the result of the body? Why not use the body expression directly?

Typed Racket at Work

The point of this wrapping is to put the type annotations to work in a dynamic setting. Essentially,

- the programmer who has put the effort to add annotations and get the program through the type-checker
- gets assurance that their function will not be abused
- through checks that are early and more informative than an internal error
- (that may not even occur, depending on the inputs, leaving the error to lurk!).

Here is a more interesting example. We define the following typed function:

```
(define (h [i : (-> String Number)]) : Number
  (+ (i "5") 1))
```

Here is its Racket counterpart:

```
(define (j i)
  (+ (i "5") 1))
```

Typed Racket at Work

```
(define (h [i : (-> String Number)]) : Number
  (+ (i "5") 1))

(define (j i)
  (+ (i "5") 1))
```

Now let's assume we are trying to use both of these from Racket. We first define a function that produces strings from strings, i.e., one that does *not* match the function expected by either `h` or `j`:

```
(define (str-dbl s) (string-append s s))
```

Now watch what happens when we run `(j str-dbl)` and `(h str-dbl)`.

- Both produce a run-time error, but very different ones.
- The former (which is entirely in Racket) gives an error at `+`: the "doubled" string is produced and makes it as far as `+`, which reports a violation.
- In contrast, in the latter case, the doubled string is produced but, when it tries to return from `(i "5")`, the type `(-> String Number)` has been turned into a contract, which halts execution saying that there is a *contract* violation!

Typed Racket at Work

Aside: To get a sense of Racket's contract system, see Contracts in the Racket Guide.

Exercise: Another interesting static-dynamic language combination is Racket with plait.

- plait does not try very much to accommodate Racket idioms,
- though it does to some extent: recall the predicates and accessors in algebraic dataytypes, though at the cost of static type safety.
- Largely, however, plait is trying to implement the Standard ML type language.
- Nevertheless, because plait lives in the context of Racket, its values can be exported and used from Racket. Try the above examples in plait!