```
          (sched-k (Tdone)))))
(set! yielder
      (lambda ()
              (let/cc thread-k
                (begin
                  (set! thread-resumer thread-k)
                  (sched-k (Tsuspended))))))
(thread-resumer 'tres))))]))
```

If we now replace `scheduler-loop-0` with `scheduler-loop-1` and `thread-0` with `thread-1` and re-run our example program above, we get just the output we desire.

14.6.4 Better Primitives for Web Programming

Finally, to tie the knot back to where we began, let's return to `read-number`: observe that, if the language running the server program has `call/cc`, instead of having to CPS the entire program, we can simply capture the current continuation and save it in the hash table, leaving the program structure again intact.

15 Checking Program Invariants Statically: Types

As programs grow larger or more subtle, developers need tools to help them describe and validate statements about program *invariants*. Invariants, as the name suggests, are statements about program elements that are expected to always hold of those elements. For example, when we write `x : number` in our typed language, we mean that `x` will always hold a `number`, and that all parts of the program that depend on `x` can rely on this statement being enforced. As we will see, types are just one point in a spectrum of invariants we might wish to state, and static type checking—itself a diverse family of techniques—is also a point in a spectrum of methods we can use to enforce the invariants.

15.1 Types as Static Disciplines

In this chapter, we will focus especially on *static type checking*: that is, checking (declared) types before the program even executes. We have already experienced a form of this in our programs by virtue of using a typed programming language. We will explore some of the design space of types and their trade-offs. Finally, though static typing is an especially powerful and important form of invariant enforcement, we will also examine some other techniques that we have available.

Consider this program in our typed language:

```
(define (f [n : number]) : number
  (+ n 3))

(f "x")
```

We get a static type error before the program begins execution. The same program (without the type annotations) in ordinary Racket fails only at runtime:

```
(define (f n)
  (+ n 3))

(f "x")
```

Exercise

> How would you test the assertions that one fails before the program executes while the other fails during execution?

Now consider the following Racket program:

```
(define f n
  (+ n 3))
```

This too fails before program execution begins, with a parse error. Though we think of parsing as being somehow distinct from type-checking—usually because the type-checker assumes it has a parsed program to begin with—it can be useful to think of parsing as being simply the very simplest kind of type-checking: determining (typically) whether the program obeys a context-*free* syntax. Type-checking then asks

whether it obeys a context-*sensitive* (or richer) syntax. In short, type-checking is a generalization of parsing, in that both are concerned with *syntactic* methods for enforcing disciplines on programs.

15.2 A Classical View of Types

We will begin by introducing a traditional core language of types. Later, we will explore both extensions [REF] and variations [REF].

15.2.1 A Simple Type Checker

Before we can define a type checker, we have to fix two things: the syntax of our *typed* core language and, hand-in-hand with that, the syntax of types themselves.

To begin with, we'll return to our language with functions-as-values [REF] but before we added mutation and other complications (some of which we'll return to later). To this language we have to add type annotations. Conventionally, we don't impose type annotations on constants or on primitive operations such as addition; instead, we impose them on the boundaries of functions or methods. Over the course of this study we will explore why this is a good locus for annotations.

Given this decision, our typed core language becomes:

```
(define-type TyExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [appC (fun : TyExprC) (arg : TyExprC)]
  [plusC (l : TyExprC) (r : TyExprC)]
  [multC (l : TyExprC) (r : TyExprC)]
  [lamC (arg : symbol) (argT : Type) (retT : Type) (body : TyExprC)])
```

That is, every procedure is annotated with the type of argument it expects and type of argument it purports to produce.

Now we have to decide on a language of types. To do so, we follow the tradition that the types *abstract over the set of values*. In our language, we have two kinds of values:

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : TyExprC) (env : Env)])
```

It follows that we should have two kinds of types: one for numbers and the other for functions.

Even numeric types may not be straightforward: What information does a number type need to record? In most languages, there are actually *many* numeric types, and indeed there may not even be a single one that represents "numbers". However, we have ignored these gradations between numbers [REF], so it's sufficient for us to have just one. Having decided that, do we record additional information about *which* number? We could in principle, but we would soon run into decidability problems.

As for functions, we have more information: the type of expected argument, and the type of claimed result. We might as well record this information we have been given until and unless it has proven to not be useful. Combining these, we obtain the following abstract language of types:

```
(define-type Type
  [numT]
  [funT (arg : Type) (ret : Type)])
```

Now that we've fixed both the term and type structure of the language, let's make sure we agree on what constitute type errors in our language (and, by fiat, everything not a type error must pass the type checker). There are three obvious forms of type errors:

- One or both arguments of + is not a number, i.e., is not a `numT`.
- One or both arguments of * is not a number.
- The expression in the function position of an application is not a function, i.e., is not a `funT`.

Do Now!

Any more?

We're actually missing one:

- The expression in the function position of an application is a function but the type of the actual argument does not match the type of the formal argument expected by the function.

It seems clear all other programs in our language ought to type-check.

A natural starting signature for the type-checker would be that it is a procedure that consumes an expression and returns a boolean value indicating whether or not the expression type-checked. Because we know expressions contain identifiers, it soon becomes clear that we will want a *type environment*, which maps names to types, analogous to the value environment we have seen so far.

Exercise

Define the types and functions associated with type environments.

Thus, we might begin our program as follows:

<***tc-take-1***> **::=**

```
(define (tc [expr : TyExprC] [tenv : TyEnv]) : boolean
  (type-case TyExprC expr
    <tc-take-1-numC-case>
    <tc-take-1-idC-case>
    <tc-take-1-appC-case>))
```

As the abbreviated set of cases above suggests, this approach will not work out. We'll soon see why.

Let's begin with the easy case: numbers. Does a number type-check? Well, on its own, of course it does; it may be that the surrounding context is not expecting a number, but that error would be signaled elsewhere. Thus:

<***tc-take-1-numC-case***> **::=**

```
[numC (n) true]
```

Now let's handle identifiers. Is an identifier well-typed? Again, on its own it would appear to be, provided it is actually a bound identifier; it may not be what the context desires, but hopefully that too would be handled elsewhere. Thus we might write

<***tc-take-1-idC-case***> **::=**

```
[idC (n) (if (lookup n tenv)
             true
             (error 'tc "not a bound identifier"))]
```

This should make you a little uncomfortable: `lookup` already signals an error if an identifier isn't bound, so there's no need to repeat it (indeed, we will never get to this `error` invocation). But let's push on.

Now we tackle applications. We should type-check both the function part, to make sure it's a function, then ensure that the actual argument's type is consistent with what the function declares to be the type of its formal argument. For instance, the function could be a number and the application could itself be a function, or vice versa, and in either case we want to prevent such mis-applications.

How does the code look?

<***tc-take-1-appC-case***> **::=**

```
[appC (f a) (let ([ft (tc f tenv)])
              ...)]
```

The recursive call to `tc` can only tell us whether the function expression type-checks or not. If it does, how do we know what type it has? If we have an immediate function, we could reach into its syntax and pull out the argument (and return) types. But if we have a complex expression, we need some procedure that will *calculate* the resulting type of that expression. Of course, such a procedure could only provide a type if the expression is well-typed; otherwise it would not be able to provide a coherent answer. In other words, *our type "calculator" has type "checking" as a special case*! We should therefore strengthen the inductive invariant on `tc`: that it not only tells us whether an expression is typed but also what its type is. Indeed, by giving any type at all it confirms that the expression types, and otherwise it signals an error.

Let's now define this richer notion of a type "checker".

<***tc***> **::=**

```
(define (tc [expr : TyExprC] [tenv : TyEnv]) : Type
  (type-case TyExprC expr
```

<tc-numC-case>
<tc-idC-case>
<tc-plusC-case>
<tc-multC-case>
<tc-appC-case>
<tc-lamC-case>))

Now let's fill in the pieces. Numbers are easy: they have the numeric type.

<***tc-numC-case***> **::=**

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

Similarly, identifiers have whatever type the environment says they do (and if they aren't bound, this signals an error).

<***tc-idC-case***> **::=**

```
[idC (n) (lookup n tenv)]
```

Observe, so far, the similarity to and difference from interpreting: in the identifier case we did essentially the same thing (except we returned a type rather than an actual value), whereas in the numeric case we returned the abstract "number" rather than indicate which specific number it was.

Let's now examine addition. We must make sure both sub-expressions have numeric type; only if they do will the overall expression evaluate to a number itself.

<***tc-plusC-case***> **::=**

```
[plusC (l r) (let ([lt (tc l tenv)]
                   [rt (tc r tenv)])
               (if (and (equal? lt (numT))
                        (equal? rt (numT)))
                   (numT)
                   (error 'tc "+ not both numbers")))]
```

We've usually glossed over multiplication after considering addition, but now it will be instructive to handle it explicitly:

<***tc-multC-case***> **::=**

```
[multC (l r) (let ([lt (tc l tenv)]
                   [rt (tc r tenv)])
               (if (and (equal? lt (numT))
                        (equal? rt (numT)))
                   (numT)
                   (error 'tc "* not both numbers")))]
```

Do Now!

Did you see what's different?

That's right: practically *nothing*! (The `multC` instead of `plusC` in the `type-case`, and the slightly different error message, notwithstanding.) That's because, from the perspective of type-checking (in this type language), there is no difference between addition and multiplication, or indeed between *any* two functions that consume two numbers and return one.

Observe another difference between interpreting and type-checking. Both care that the arguments be numbers. The interpreter then returns a precise sum or product, but the type-checker is indifferent to the differences between them: therefore the expression that computes what it returns (`(numT)`) is a constant, and the same constant in both cases.

Finally, the two hard cases: application and funcions. We've already discussed what application must do: compute the value of the function and argument expressions; ensure the function expression has function type; and check that the argument expression is of compatible type. If all this holds up, then the type of the overall application is whatever type the function body would return (because the value that eventually returns at run-time is the result of evaluating the function's body).

<*tc-appC-case*> **::=**

```
[appC (f a) (let ([ft (tc f tenv)]
                  [at (tc a tenv)])
              (cond
                [(not (funT? ft))
                 (error 'tc "not a function")]
                [(not (equal? (funT-arg ft) at))
                 (error 'tc "app arg mismatch")]
                [else (funT-ret ft)]))]
```

That leaves function definitions. The function has a formal parameter, which is presumably used in the body; unless this is bound in the environment, the body most probably will not type-check properly. Thus we have to extend the type environment with the formal name bound to its type, and in that extended environment type-check the body. Whatever value this computes must be the same as the declared type of the body. If that is so, then the function itself has a function type from the type of the argument to the type of the body.

Exercise

Why do I say "most probably" above?

<*tc-lamC-case*> **::=**

```
[lamC (a argT retT b)
      (if (equal? (tc b (extend-ty-env (bind a argT) tenv)) retT)
          (funT argT retT)
          (error 'tc "lam type mismatch"))]
```

Observe another curious difference between the interpreter and type-checker. In the interpreter, application was responsible for evaluating the argument expression, extending the environment, and evaluating the body. Here, the application case does check

the argument expression, but leaves the environment alone, and simply returns the type of the body *without traversing it*. Instead, the body is actually traversed by the checker when checking a function *definition*, so this is the point at which the environment actually extends.

15.2.2 Type-Checking Conditionals

Suppose we extend the above language with conditionals. Even the humble `if` introduces several design decisions. We'll discuss two here, and return to one of them later [REF].

1. What should be the type of the test expression? In some languages it must evaluate to a boolean value, in which case we have to enrich the type language to include booleans (which would probably be a good idea anyway). In other languages it can be any value, and some values are considered "truthy" while others "falsy".

2. What should be the relationship between the then- and else-branches? In some languages they must be of the same type, so that there is a single, unambiguous type for the overall expression (which is that one type). In other languages the two branches can have distinct types, which greatly changes the design of the type-language and -checker, but also of the nature of the programming language itself.

Exercise

Add booleans to the type language. What does this entail at a minimum, and what else might be expected in a typical language?

Exercise

Add a type rule for conditionals, where the test expression is expected to evaluate to a boolean and both then- and else-branches must have the same type, which is the type of the overall expression.

15.2.3 Recursion in Code

Now that we've obtained a basic programming language, let's add recursion to it. We saw earlier [REF] that this could be done easily through desugaring. It'll prove to be a more complex story here.

A First Attempt at Typing Recursion

Let's now try to express a simple recursive function. The simplest is, of course, one that loops forever. Can we write an infinite loop with just functions? We already could simply with this program—

```
((lambda (x) (x x))
 (lambda (x) (x x)))
```

—which we know we can represent in our language with functions as values.

Exercise

> Why does this construct an infinite loop? What subtle dependency is it making about the nature of function calls?

Now that we have a typed language, and one that forces us to annotate all functions, let's annotate it. For simplicity, from now on we'll assume we're writing programs in a typed surface syntax, and that desugaring takes care of constructing core language terms.

Observe, first, that we have two identical terms being applied to each other. Historically, the overall term is called Ω (capital omega in Greek) and each of the identical sub-terms is called ω (lower-case omega in Greek). It is not a given that identical terms must have precisely the same type, because it depends on what invariants we want to assert of the context of use. In this case, however, observe that `x` binds to ω, so the second ω goes into both the first and second positions. As a result, typing one effectively types both.

Therefore, let's try to type ω; let's call this type γ. It's clearly a function type, and the function takes one argument, so it must be of the form ϕ `->` ψ. Now what is that argument? It's ω itself. That is, the type of the value going into ϕ is itself γ. Thus, the type of ω is γ, which is ϕ `->` ψ, which expands into `(`ϕ `->` ψ`)` `->` ψ, which further expands to `((`ϕ `->` ψ`)` `->` ψ`)` `->` ψ, and so on. In other words, this type cannot be written as any finite string!

Do Now!

> Did you notice the subtle but important leap we just made?

Program Termination

We observed that the obvious typing of Ω, which entails typing γ, seems to run into serious problems. From that, however, we jumped to the conclusion that this type cannot be written as any finite string, for which we've given only an intuition, not a proof. In fact, something even stranger is true: in the type system we've defined so far, *we cannot type* Ω *at all*!

This is a strong statement, but we can actually say something much stronger. The *typed* language we have so far has a property called *strong normalization*: every expression that has a type will terminate computation after a finite number of steps. In other words, this special (and peculiar) infinite loop program isn't the only one we can't type; we can't type *any* infinite loop (or even potential infinite loop). A rough intuition that might help is that any type—which must be a finite string—can have only a finite number of `->`'s in it, and each application discharges one, so we can perform only a finite number of applications.

If our language permitted only straight-line programs, this would be unsurprising. However, we have conditionals and even functions being passed around as values, and with those we can encode any datatype we want. Yet, we still get this guarantee! That makes this a somewhat astonishing result.

Exercise

Try to encode lists using functions in the untyped and then in the typed language. What do you see? And what does that tell you about the impact of this type system on the encoding?

This result also says something deeper. It shows that, contrary to what you may believe—that a type system only prevents a few buggy programs from running—a type system can *change the semantics* of a language. Whereas previously we could write an infinite loop in just one to two lines, now we cannot write one at all. It also shows that the type system can establish invariants not just about a particular program, but *about the language itself*. If we want to absolutely ensure that a program will terminate, we simply need to write it in this language and pass the type checker, and the guarantee is ours!

What possible use is a language in which all programs terminate? For general-purpose programming, none, of course. But in many specialized domains, it's a tremendously useful guarantee to have. For instance, suppose you are implementing a complex scheduling algorithm; you would like to know that your scheduler is guaranteed to terminate so that the tasks being scheduled will actually run. There are many other domains, too, where we would benefit from such a guarantee: a packet-filter in a router; a real-time event processor; a device initializer; a configuration file; the callbacks in single-threaded JavaScript; and even a compiler or linker. In each case, we have an almost unstated expectation that these programs will always terminate. And now we have a language that can offer this guarantee—something it is impossible to test for, no less!

These are not hypothetical examples. In the Standard ML language, the language for linking modules uses essentially this typed language for writing module linking specifications. This means developers can write quite sophisticated abstractions—they have functions-as-values, after all!—while still being guaranteed that linking will always terminate, producing a program.

Typing Recursion

What this says is, whereas before we were able to handle `rec` entirely through desugaring, now we must make it an explicit part of the typed language. For simplicity, we will consider a special case of `rec`—which nevertheless covers the common uses—whereby the recursive identifier is bound to a function. Thus, in the surface syntax, one might write

```
(rec (Σ num (n num)
        (if0 n
             0
             (n + (Σ (n + -1)))))
  (Σ 10))
```

for a summation function, where Σ is the name of the function, `n` its argument, and `num` the type consumed by and returned from the function. The expression `(Σ 10)` represents the use of this function to sum the number from `10` until `0`.

How do we type such an expression? Clearly, we must have `n` bound in the body of the function as we type it (but not of course, in the use of the function); this much we know from typing functions. But what about Σ? Obviously it must be bound in the type environment when checking the use (`(Σ 10)`), and its type must be `num -> num`. But it must *also* be bound, to the same type, when checking the body of the function. (Observe, too, that the type returned by the body must match its declared return type.)

Now we can see how to break the shackles of the finiteness of the type. It is certainly true that we can write only a finite number of `->`'s in types in the program source. However, this rule for typing recursion duplicates the `->` in the body that refers to itself, thereby ensuring that there is an inexhaustible supply of applications. It's our infinite quiver of arrows.

The code to implement this rule would be as follows. Assuming `f` is bound to the function's name, `aT` is the function's argument type and `rT` is its return type, `b` is the function's body, and `u` is the function's use:

<tc-lamC-case> **::=**

```
[recC (f a aT rT b u)
      (let ([extended-env
             (extend-ty-env (bind f (funT aT rT)) tenv)])
        (cond
          [(not (equal? rT (tc b
                               (extend-ty-env
                                (bind a aT)
                                extended-env))))
           (error 'tc "body return type not correct")]
          [else (tc u extended-env)]))]
```

15.2.4 Recursion in Data

We have seen how to type recursive programs, but this doesn't yet enable us to create recursive data. We already have one kind of recursive datum—the function type—but this is built-in. We haven't yet seen how developers can create their own recursive datatypes.

Recursive Datatype Definitions

When we speak of allowing programmers to create recursive data, we are actually talking about three different facilities at once:

- Creating a new type.
- Letting instances of the new type have one or more fields.
- Letting some of these fields refer to instances of the same type.

In fact, once we allow the third, we must allow one more:

- Allowing non-recursive base-cases for the type.

This confluence of design criteria leads to what is commonly called an *algebraic datatype*, such as the types supported by our typed language. For instance, consider the following definition of a binary tree of numbers:

Later [REF], we will discuss how types can be parameterized.

```
(define-type BTnum
  [BTmt]
  [BTnd (n : number) (l : BTnum) (r : BTnum)])
```

Observe that without a name for the new datatype, `BTnum`, we would not have been able to refer back ot it in `BTnd`. Similarly, without the ability to have more than one kind of `BTnum`, we would not have been able to define `BTmt`, and thus wouldn't have been able to terminate the recursion. Finally, of course, we need multiple fields (as in `BTnd`) to construct useful and interesting data. In other words, all three mechanisms are packaged together because they are most useful in conjunction. (However, some langauges do permit the definition of stand-alone structures. We will return to the impact of this design decision on the type system later [REF].)

This concludes our initial presentation of recursive types, but it has a fatal problem. We have not actually explained where this new type, `BTnum`, comes from. That is because we have had to pretend it is baked into our type-checker. However, it is simply impractical to keep changing our type-checker for each new recursive type definition—it would be like modifying our interpreter each time the program contains a recursive function! Instead, we need to find a way to make such definitions intrinsic to the type language. We will return to this problem later [REF].

This style of data definition is sometimes also known as a *sum of products*. "Product" refers to the way fields combine in one variant: for instance, the legal values of a `BTnd` are the cross-product of legal values in each of the fields, supplied to the `BTnd` constructor. The "sum" is the aggregate of all these variants: any given `BTnum` value is just one of these. (Think of "product" as being "and", and "sum" as being "or".)

Introduced Types

Now, what impact does a datatype definition have? First, it introduces a new type; then it uses this to define several constructors, predicates, and selectors. For instance, in the above example, it first introduces `BTnum`, then uses it to ascribe the following types:

```
BTmt : -> BTnum
BTnd : number * BTnum * BTnum -> BTnum
BTmt? : BTnum -> boolean
BTnd? : BTnum -> boolean
BTnd-n : BTnum -> number
BTnd-l : BTnum -> BTnum
BTnd-r : BTnum -> BTnum
```

Observe a few salient facts:

- Both the constructors create instances of `BTnum`, not something more refined. We will discuss this design tradeoff later [REF].
- Both predicates consume values of type `BTnum`, not "any". This is because the type system can already tell us what type a value is. Thus, we only need to distinguish between the variants of that one type.
- The selectors really only work on instances of the relevant variant—e.g., `BTnd-n` can work only on instances of `BTnd`, not on instances of `BTmt`—but we don't have a way to express this in the static type system for lack of a suitable static

type. Thus, applying these can only result in a dynamic error, not a static one caught by the type system.

There is more to say about recursive types, which we will return to shortly [REF].

Pattern-Matching and Desugaring

Once we observe that these are the types, the only thing left is to provide an account of pattern-matching. For instance, we can write the expression

```
(type-case BTnum t
  [BTmt () e1]
  [BTnd (nv lt rt) e2])
```

We have already seen [REF] that this can be written in terms of the functions defined above. We can simulate the binding done by this pattern-matcher using `let`:

```
(cond
  [(BTmt? t) e1]
  [(BTnd? t) (let ([nv (BTnd-n t)]
                   [lt (BTnd-l t)]
                   [rt (BTnd-r t)])
               e2)])
```

In short, this can be done by a macro, so pattern-matching does not need to be in the core language and can instead be delegated to desugaring. This, in turn, means that one language can have many different pattern-matching mechanisms.

Except, that's not quite true. Somehow, the macro that generates the code above in terms of `cond` needs to know that the three positional selectors for a `BTnd` are `BTnd-n`, `BTnd-l`, and `BTnd-r`, respectively. This information is explicit in the type definition but only implicitly present in the use of the pattern-matcher (that, indeed, being the point). Thus, somehow this information must be communicated from definition to use. Thus the macro expander needs something akin to the type environment to accomplish its task.

Observe, furthermore, that expressions such as `e1` and `e2` cannot be type-checked—indeed, cannot even be reliable identified as *expressions*—until macro expansion expands the use of `type-case`. Thus, expansion depends on the type environment, while type-checking depends on the result of expansion. In other words, the two are symbiotic and need to happen, not quite in "parallel", but rather in lock-step. Thus, building desugaring for a typed language, where the syntactic sugar makes assumptions about types, is a little more intricate than doing so for an untyped language.

15.2.5 Types, Time, and Space

It is evident that types already bestow a performance benefit in safe languages. That is because the checks that would have been performed at run-time—e.g., `+` checking that both its arguments are indeed numbers—are now performed statically. In a typed

language, an annotation like `: number` already answers the question of whether or not something is of a particular a type; there is nothing to ask at run-time. As a result, these type-level predicates can (and need to) disappear entirely, and with them any need to use them in programs.

This is at some cost to the developer, who must convince the static type system that their program does not induce type errors; due to the limitations of decidability, even programs that might have run without error might run afoul of the type system. Nevertheless, for programs that meet this requirement, types provide a notable execution time saving.

Now let's discuss space. Until now, the language run-time system has needed to store information attached to every value indicating what its type is. This is how it can implement type-level predicates such as `number?`, which may be used both by developers and by primitives. If those predicates disappear, so does the space needed to hold information to implement them. Thus, type-tags are no longer necessary.

They would, however, still be needed by the garbage collector, though other representations such as BIBOP can greatly reduce their space impact.

The type-like predicates still left are those for variants: `BTmt?` and `BTnd?`, in the example above. These must indeed be applied at run-time. For instance, as we have noted, selectors like `BTnd-n` must perform this check. Of course, some more optimizations are possible. Consider the code generated by desugaring the pattern-matcher: there is no need for the three selectors to implement this check, because control could only have gotten to them after `BTnd?` returned a true vlaue. Thus, the run-time system could provide just the desugaring level access to special *unsafe* primitives that do not perform the check, resulting in generated code such as this:

```
(cond
  [(BTmt? t) e1]
  [(BTnd? t) (let ([nv (BTnd-n/no-check t)]
                   [lt (BTnd-l/no-check t)]
                   [rt (BTnd-r/no-check t)])
               e2)])
```

The net result, however, is that the run-time representation must still store enough information to accurately answer these questions. However, previously it needed to use enough bits to record every possible type (and variant). Now, because the types have been statically segregated, for a type with no variants (e.g., there is only one kind of string), there is no need to store any variant information at all; that means the run-time system can use all available bits to store actual dynamic values.

In contrast, when variants are present, the run-time system must sacrifice bits to distinguish between the variants, but the number of *variants within a type* is obviously far smaller than the number of variants and types *across all types*. In the `BTnum` example above, there are only two variants, so the run-time system needs to use only one bit to record which variant of `BTnum` a value represents.

Observe, in particular, that the type system's segregation prevents confusion. If there are two different datatypes that each have two variants, in the untyped world all these four variants require distinct representations. In contrast, in the typed world these representations can overlap across types, because the static type system will ensure one type's variants are never confused for that the another. Thus, types have a genuine space

(saving representation) and time (eliminating run-time checks) performance benefit for programs.

15.2.6 Types and Mutation

We have now covered most of the basic features of our core language other than mutation. In some ways, types have a simple interaction with mutation, and this is because in a classical setting, they don't interact at all. Consider, for instance, the following untyped program:

```
(let ([x 10])
  (begin
    (set! x 5)
    (set! x "something")))
```

What is "the type" of `x`? It doesn't really have one: for some time it's a number, and *later* (note the temporal word) it's a string. We simply can't give it a type. In general, type checking is an *atemporal* activity: it is done once, before the program runs, and must hence be independent of the specific order in which programs execute. Keeping track of the precise values in the store is hence beyond the reach of a type-checker.

The example above is, of course, easy to statically understand, but we should never be mislead by simple examples. Suppose instead we had a program like

```
(let ([x 10])
  (if (even? (read-number "Enter a number"))
    (set! x 5)
    (set! x "something")))
```

Now it is literally impossible to reach any static conclusion about the type of `x` after the conditional finishes, because only at run-time can we know what the user might have entered.

To avoid this morass, traditional type checkers adopt a simple policy: types must be *invariant* across mutation. That is, a mutation operation—whether variable mutation or structure mutation—cannot change the type of the mutant. Thus, the above examples would not type in our type language so far. How much flexibility this gives the programmer is, however, a function of the type language. For instance, if we were to admit a more flexible type that stands for "number or string", then the examples above would type, but `x` would always have this, less precise, type, and all uses of `x` would have to contend with its reduced specificity, an issue we will return to later [REF].

In short, mutation is easy to account for in a traditional type system because its rule is simply that, while the value can change in ways below the level of specificity of the type system, the type cannot change. In the case of an operation like `set!` (or our core language's `setC`), this means the type of the assigned value must match that of the variable. In the case of structure mutation, such as boxes, it means the assigned value must match that the box's contained type.

15.2.7 The Central Theorem: Type Soundness

We have seen earlier [REF] that certain type languages can offer very strong theorems about their programs: for instance, that all programs in the language terminate. In general, of course, we cannot obtain such a guarantee (indeed, we added general recursion precisely to let ourselves write unbounded loops). However, a meaningful type system—indeed, anything to which we wish to bestow the noble title of a *type system*—ought to provide some kind of meaningful guarantee that all typed programs enjoy. This is the payoff for the programmer: by typing this program, she can be certain that certain bad things will certainly not happen. Short of this, we have just a bug-finder; while it may be useful, it is not a sufficient basis for building any higher-level tools (e.g., for obtaining security or privacy or robustness guarantees).

We have repeatedly used the term "type *system*". A type system is usually a combination of three components: a language of types, a set of type rules, and an algorithm that applies these rules to programs. By largely presenting our type rules embedded in a function, we have blurred the distinction between the second and third of these, but can still be thought of as intellectually distinct.

What theorem might we want of a type system? Remember that the type checker runs over the static program, before execution. In doing so, it is essentially making a *prediction* about the program's behavior: for instance, when it states that a particular complex term has type `num`, it is effectively predicting that when run, that term will produce a numeric value. How do we know this prediction is sound, i.e., that the type checker never lies? Every type system should be accompanied by a theorem that proves this.

There is a good reason to be suspicious of a type system, beyond general skepticism. There are many differences between the way a type checker and a program evaluator work:

- The type checker only sees program text, whereas the evaluator runs over actual stores.
- The type environment binds identifiers to types, whereas the evaluator's environment binds identifiers to values or locations.
- The type checker compresses (even infinite) sets of values into types, whereas the evaluator treats these distinctly.
- The type checker always terminates, whereas the evaluator might not.
- The type checker passes over the body of each expression only once, whereas the evaluator might pass over each body anywhere from zero to infinite times.

Thus, we should not assume that these will always correspond!

The central result we wish to have for a given type-system is called *soundness*. It says this. Suppose we are given an expression (or program) `e`. We type-check it and conclude that its type is `t`. When we run `e`, let us say we obtain the value `v`. Then `v` will also have type `t`.

The standard way of proving this theorem is to prove it in two parts, known as *progress* and *preservation*. Progress says that if a term passes the type-checker, it will be able to make a step of evaluation (unless it is already a value); preservation says that the result of this step will have the same type as the original. If we interleave these steps (first progress, then preservation; repeat), we can conclude that the final answer will indeed have the same type as the original, so the type system is indeed sound.

For instance, consider this expression: `(+ 5 (* 2 3))`. It has the type `num`. In a sound type system, progress offers a proof that, because this term types, and is not already a value, it can take a step of execution—which it clearly can. After one step the program reduces to `(+ 5 6)`. Sure enough, as preservation proves, this has the same type as the original: `num`. Progress again says this can take a step, producing `11`. Preservation again shows that this has the same type as the previous (intermediate) expressions: `num`. Now progress finds that we are at an answer, so there are no steps left to be taken, and our answer is of the same type as that given for the original expression.

However, this isn't the entire story. There are two caveats:

1. The program may not produce an answer at all; it might loop forever. In this case, the theorem strictly speaking does not apply. However, we can still observe that every intermediate term still has the same type, so the program is computing meaningfully even if it isn't producing a value.

2. Any rich enough language has properties that cannot be decided statically (and others that perhaps could be, but the language designer chose to put off until run-time). When one of these properties fails—e.g., the array index being within bounds—there is no meaningful type for the program. Thus, implicit in every type soundness theorem is some set of published, permitted exceptions or error conditions that may occur. The developer who uses a type system implicitly signs on to accepting this set.

As an example of the latter set, the user of a typical typed language acknowledges that vector dereference, list indexing, and so on may all yield exceptions.

The latter caveat looks like a cop-out. In fact, it is easy to forget that it is really a statement about what *cannot* happen at run-time: any exception not in this set will provably not be raised. Of course, in languages designed with static types in the first place, it is not clear (except by loose analogy) what these exceptions might be, because there would be no need to define them. But when we retrofit a type system onto an existing programming language—especially languages with only dynamic enforcement, such as Racket or Python—then there is already a well-defined set of exceptions, and the type-checker is explicitly stating that some set of those exceptions (such as "non-function found in application position" or "method not found") will simply never occur. This is therefore the payoff that the programmer receives in return for accepting the type system's syntactic restrictions.

15.3 Extensions to the Core

Now that we have a basic typed language, let's explore how we can extend it to obtain a more useful programming language.

15.3.1 Explicit Parametric Polymorphism

Which of these is the same?

- `List<String>`

- `List<String>`
- `(listof string)`

Actually, none of these is quite the same. But the first and third are very alike, because the first is in Java and the third in our typed language, whereas the second, in C++, is different. All clear? No? Good, read on!

Parameterized Types

The language we have been programming in already demonstrates the value of parametric polymorphism. For instance, the type of `map` is given as

```
(('a -> 'b) (listof 'a) -> (listof 'b))
```

which says that for all types `'a` and `'b`, `map` consumes a function that generates `'b` values from `'a` values, and a list of `'a` values, and generates the corresponding list of `'b` values. Here, `'a` and `'b` are not concrete types; rather, they are *type variables* (in our terminology, these should properly be called "type identifiers" because they don't change within the course of an instantiation; however, we will stick to the traditional terminology).

A different way to understand this is that there is actually an infinite family of `map` functions. For instance, there is a `map` that has this type:

```
((number -> string) (listof number) -> (listof string))
```

and another one of this type (nothing says the types have to be base types):

```
((number -> (number -> number)) (listof number) -> (listof (number -
> number)))
```

and yet another one of this type (nothing says `'a` and `'b` can't be the same):

```
((string -> string) (listof string) -> (listof string))
```

and so on. Because they have different types, they would need different names: `map_num_str`, `map_num_num->num`, `map_str_str`, and so on. But that would make them different functions, so we'd have to always refer to a specific `map` rather than each of the generic ones.

Obviously, it is impossible to load all these functions into our standard library: there's an infinite number of these! We'd rather have a way to obtain each of these functions on demand. Our naming convention offers a hint: it is as if `map` takes two *parameters*, which are *types*. Given the pair of types as arguments, we can then obtain a `map` that is customized to that particular type. This kind of *parameterization over types* is called *parametric polymorphism*.

Not to be confused with the "polymorphism" of objects, which we will discuss below [REF].

Making Parameters Explicit

In other words, we're effectively saying that `map` is actually a function of perhaps four arguments, two of them types and two of them actual values (a function and a list). In a language with explicit types, we might try to write

```
(define (map [a : ???] [b : ???] [f : (a -> b)] [l : (listof a)]) : (listof b)
  ...)
```

but this raises some questions. First, what goes in place of the `???`? These are the types of `a` and `b`. But if `a` and `b` are themselves going to be replaced with *types*, then what is the type of a type? Second, do we really want to be calling `map` with four arguments on every instantiation? Third, do we really mean to take the type parameters first before any actual values? The answers to these questions actually lead to a very rich space of polymorphic type systems, most of which we will *not* explore here.

I recommend reading Pierce's *Types and Programming Languages* for a modern, accessible introduction.

Observe that once we start parameterizing, more code than we expect ends up being parameterized. For instance, consider the type of the humble `cons`. Its type really is parametric over the type of values in the list (even though it doesn't actually depend on those values!—more on that in a bit [REF]) so every use of `cons` must be instantiated at the appropriate type. For that matter, even `empty` must be instantiated to create an empty list of the correct type! Of course, Java and C++ programmers are familiar with this pain.

Rank-1 Polymorphism

Instead, we will limit ourselves to one particularly useful and tractable point in this space, which is the type system of Standard ML, of the typed language of this book, of earlier versions of Haskell, roughly that of Java and C# with generics, and roughly that obtained using templates in C++. This language defines what is called *predicative*, *rank-1*, or *prenex* polymorphism. It answers the above questions thus: nothing, no, and yes. Let's explore this below.

We first divide the world of types into two groups. The first group consists of the type language we've used until, but extended to include type variables; these are called *monotypes*. The second group, known as *polytypes*, consists of parameterized types; these are conventionally written with a ∀ prefix, a list of type variables, and then a type expression that might use these variables. Thus, the type of `map` would be:

```
∀ a, b : (('a -> 'b) (listof 'a) -> (listof 'b))
```

Since "∀" is the logic symbol for "for all", you would read this as: "for all types `'a` and `'b`, the type of `map` is...".

In rank-1 polymorphism, the type variables can only be substituted with monotypes. (Furthermore, these can only be concrete types, because there would be nothing left to substitute any remaining type variables.) As a result, we obtain a clear separation between the type variable-parameters and regular parameters. We don't need to provide a "type annotation" for the type variables because we know precisely what kind of thing they can be. This produces a relatively clean language that still offers considerable expressive power.

Impredicative languages erase the distinction between monotypes and polytypes, so a type variable can be instantiated with another polymorphic type.

Observe that because type variables can only be replaced with monotypes, they are all independent of each other. As a result, all type parameters can be brought to the front of the parameter list. This is what enables us to write types in the form `∀ tv, ... : t` where the `tv` are type variables and `t` is a monotype (that might refer to those variables). This justifies not only the syntax but also the name "prenex". It will prove to also be useful in the implementation.

Interpreting Rank-1 Polymorphism as Desugaring

The simplest implementation of this feature is to view it as a form of desugaring: this is essentially the interpretation taken by C++. (Put differently, because C++ has a macro system in the form of templates, by a happy accident it obtains a form of rank-1 polymorphism through the use of templates.) For instance, imagine we had a new syntactic form, `define-poly`, which takes a name, a type variable, and a body. On every provision of a type to the name, it replaces the type variable with the given type in the body. Thus:

```
(define-poly (id t) (lambda ([x : t]) : t x))
```

defines an identity function by first defining `id` to be polymorphic: given a concrete type for `t`, it yields a procedure of one argument of type `(t -> t)` (where `t` is appropriately substituted). Thus we can instantiate `id` at many different types—

```
(define id_num (id number))
(define id_str (id string))
```

—thereby obtaining identity functions at each of those types: `(test (id_num 5) 5) (test (id_str "x") "x")` In contrast, expressions like `(id_num "x") (id_str 5)` will, as we would expect, *fail to type-check* (rather than fail at run-time).

In case you're curious, here's the implementation. For simplicity, we assume there is only one type parameter; this is easy to generalize using `...`. We will not only define a macro for `define-poly`, *it* will in turn define a macro:

```
(define-syntax define-poly
  (syntax-rules ()
    [(_ (name tyvar) body)
     (define-syntax (name stx)
       (syntax-case stx ()
         [(_ type)
          (with-syntax ([tyvar #'type])
            #'body)]))]))
```

Thus, given a definition such as

```
(define-poly (id t) (lambda ([x : t]) : t x))
```

the language creates a *macro* named `id`: the part that begins with `(define-syntax (name ...) ...)` (where, in this example, `name` is `id`). An instantiation of `id`, such

as `(id number)`, replaces `t` the type variable, `tyvar`, with the given type. To circumvent hygiene, we use `with-syntax` to force all uses of the type variable (`tyvar`) to actually be replaced with the given type. Thus, in effect,

```
(define id_num (id number))
```

turns into

```
(define id_num (lambda ([x : number]) : number x))
```

However, this approach has two important limitations.

1. Let's try to define a recursive polymorphic function, such as `filter`. Earlier we have said that we ought to instantiate every single polymorphic value (such as even `cons` and `empty`) with types, but to keep our code concise we'll rely on the fact that the underlying typed language already does this, and focus just on type parameters for `filter`. Here's the code:

   ```
   (define-poly (filter t)
     (lambda ([f : (t -> boolean)] [l : (listof t)]) : (listof t)
       (cond
         [(empty? l) empty]
         [(cons? l) (if (f (first l))
                        (cons (first l)
                              ((filter t) f (rest l)))
                        ((filter t) f (rest l)))])))
   ```

   Observe that at the recursive uses of `filter`, we must instantiate it with the appropriate type.

   This is a perfectly good definition. There's just one problem. When we try to use it—e.g.,

   ```
   (define filter_num (filter number))
   ```

   DrRacket does not terminate. Specifically, macro expansion does not terminate, because it is repeatedly trying to make new *copies of the code of* `filter`. If, in contrast, we write the function as follows, expansion terminates—

   ```
   (define-poly (filter2 t)
     (letrec ([fltr
               (lambda ([f : (t -> boolean)] [l : (listof t)]) : (listof t)
                 (cond
                   [(empty? l) empty]
                   [(cons? l) (if (f (first l))
                                  (cons (first l) (fltr f (rest l)))
                                  (fltr f (rest l)))])])
       fltr))
   ```

but this needlessly pushes pain onto the user. Indeed, some template expanders will cache previous values of expansion and avoid re-generating code when given the same parameters. (Racket cannot do this because, in general, the body of a macro can depend on mutable variables and values and even perform input-output, so Racket cannot guarantee that a given input will always generate the same output.)

2. Consider two instantiations of the identity function. We cannot compare `id_num` and `id_str` because they are of different types, but even if they are of the same type, they are not `eq?`:

   ```
   (test (eq? (id number) (id number)) #f)
   ```

   This is because each use of `id` creates a new copy of the body. Now even if the optimization we mentioned above were applied, so for the *same* type there is only one code body, there would still be different code bodies for different types—but even this is unnecessary! There's absolutely nothing in the body of `id`, for instance, that actually depends on the type of the argument. Indeed, the entire infinite family of `id` functions can share just one implementation. The simple desugaring strategy fails to provide this.

Indeed, C++ templates are notorious for creating code bloat; this is one of the reasons.

In other words, the desugaring based strategy, which is essentially an implementation by substitution, has largely the same problems we saw earlier with regards to substitution as an implementation of parameter instantiation. However, in other cases substitution also gives us a ground truth for what we expect as the program's behavior. The same will be true with polymorphism, as we will soon see [REF].

Observe that one virtue to the desugaring strategy is that it does not require our type checker to "know" about polymorphism. Rather, the core type language can continue to be monomorphic, and all the (rank-1) polymorphism is handled entirely through expansion. This offers a cheap strategy for adding polymorphism to a language, though—as C++ shows—it also introduces significant overheads.

Finally, though we have only focused on functions, the preceding discussions applies equally well to data structures.

Alternate Implementations

There are other implementation strategies that don't suffer from these problems. We won't go into them here, but the essence of at least some of them is the "caching" approach we sketched above. Because we can be certain that, for a given set of type parameters, we will always get the same typed body, we never need to instantiate a polymorphic function at the same type twice. This avoids the infinite loop. If we type-check the instantiated body once, we can avoid checking at other instantiations of the same type (because the body will not have changed). Furthermore, we do not need to retain the instantiated sources: once we have checked the expanded program, we can dispose of the expanded terms and retain just one copy at run-time. This avoids all the problems discussed in the pure desugaring strategy shown above, while retaining the benefits.

Actually, we are being a little too glib. One of the benefits of static types is that they enable us to pick more precise run-time representations. For instance, a static type can tell us whether we have a 32-bit or 64-bit number, or for that matter a 32-bit value or a 1-bit value (effectively, a boolean). A compiler can then generate specialized code for each representation, taking advantage of how the bits are laid out (for example, 32 booleans can be *packed* into a single 32-bit word). Thus, after type-checking at each used type, the polymorphic instantiator may keep track of all the special types at which a function or data structure was used, and provide this information to the compiler for code-generation. This will then result in several copies of the function, none of which are `eq?` with each other—but for good reason and, because their operations are truly different, rightly so.

Relational Parametricity

There's one last detail we must address regarding polymorphism.

We earlier said that a function like `cons` doesn't depend on the specific values of its arguments. This is also true of `map`, `filter`, and so on. When `map` and `filter` want to operate on individual elements, they take as a parameter another function which in turn is responsible for making decisions about how to treat the elements; `map` and `filter` themselves simply obey their parameter functions.

One way to "test" whether this is true is to substitute some different values in the argument list, and a correspondingly different parameter function. That is, imagine we have a relation between two sets of values; we convert the list elements according to the relation, and the parameter function as well. The question is, will the output from `map` and `filter` also be predictable by the relation? If, for some input, this was not true of the output of `map`, then it must be that `map` inspected the actual values and did something with that information. But in fact this won't happen for `map`, or indeed most of the standard polymorphic functions.

Functions that obey this relational rule are called *relationally parametric*. This is another very powerful property that types give us, because they tell us there is a strong limit on the kinds of operations such polymorphic functions can perform: essentially, that they can drop, duplicate, and rearrange elements, but not directly inspect and make decisions on them.

Read Wadler's *Theorems for Free!* and Reynolds's *Types, Abstraction and Parametric Polymorphism*.

At first this sounds very impressive (and it is!), but on inspection you might realize this doesn't square with your experience. In Java, for instance, a polymorphic method can still use `instanceof` to check which particular kind of value it obtained at run-time, and change its behavior accordingly. Such a method would indeed not be relationally parametric! Indeed, relational parametricity can equally be viewed as a statement of the weakness of the language: that it permits only a very limited set of operations. (You could still inspect the type—but not act upon what you learned, which makes the inspection pointless. Therefore, a run-time system that wants to simulate relational parametricity would have to remove operations like `instanceof` as well as various proxies to it: for instance, adding `1` to a value and catching exceptions would reveal whether the value is a number.) Nevertheless, it is a very elegant and surprising result, and shows the power of program reasoning possible with rich type systems.

On the Web, you will often find this property described as the inability of a function to inspect the argument—which is not quite right.

15.3.2 Type Inference

Writing polymorphic type instantiations everywhere can be an awfully frustrating process, as users of many versions of Java and C++ can attest. Imagine if in our programs, every single time we wrote `first` or `rest`, we had to also instantiate it at a type! The reason we have been able to avoid this fate is because our language implements *type inference*. This is what enables us to write the definition

```
(define (mapper f l)
  (cond
    [(empty? l) empty]
    [(cons? l) (cons (f (first l)) (mapper f (rest l)))]))
```

and have the programming environment *automatically* declare that

```
> mapper
- (('a -> 'b) (listof 'a) -> (listof 'b))
```

Not only is this the correct type, this is a very general type! The process of being able to derive such general types just from the program structure feels almost magical. Now let's look behind the curtain.

First, let's understand what type inference is doing. Some people mistakenly think of languages with inference as having no type declarations, with inference taking their place. This is confused at multiple levels. For one thing, even in languages with inference, programmers are free (and for documentation purposes, often encouraged—as you have been) to declare types. Furthemore, in the absence of such declarations, it is not quite clear what inference actually *means*.

Sometimes, inference is also undecidable and programmers have no choice but to declare some of the types. Finally, writing explicit annotations can greatly reduce indecipherable error messages.

Instead, it is better to think of the underlying language as being fully, explicitly typed—just like the polymorphic language we have just studied [REF]. We will simply say that we are free to leave the type annotations after the `:`'s blank, and assume some programming environment feature will fill them in for us. (And if we can go that far, we can drop the `:`'s and extra embellishments as well, and let them all be inserted automatically. Thus, inference becomes simply a user convenience for alleviating the burden of writing type annotations, but the language underneath is explicitly typed.

How do we even think about what inference does? Suppose we have an expression (or program) `e`, written in an explicitly typed language: i.e., `e` has type annotations everywhere they are required. Now suppose we erase all annotations in `e`, and use a procedure `infer` to deduce them back.

Do Now!

What property do we expect of `infer`?

We could demand many things of it. One might be that it produces precisely those annotations that `e` originally had. This is problematic for many reasons, not least that `e` might not even type-check, in which case how could `infer` possibly know what they were (and hence should be)? This might strike you as a pedantic trifle: after all, if `e` didn't type-check, how can erasing its annotations and filling them back in make it do so? Since neither program type-checks, who cares?

Do Now!

Is this reasoning correct?

Suppose `e` is

```
(lambda ([x : number]) : string x)
```

This procedure obviously fails to type-check. But if we erase the type annotations—obtaining

```
(lambda (x) x)
```

—we equally obviously obtain a typeable function! Therefore, a more reasonable demand might be that if the original `e` type-checks, then so must the version with annotations replaced by inferred types. This one-directional implication is useful in two ways:

1. It does not say what must happen if `e` fails to type-check, i.e., it does not preclude the type inference algorithm we have, which makes the faultily-typed identity function above typeable.

2. More importantly, it assures us that we lose nothing by employing type inference: no program that was previously typeable will now cease to be so. That means we can focus on using explicit annotations where we want to, but will not be forced to do so.

Of course, this only holds if inference is decidable.

We might also expect that both versions type to the same type, but that is not a given: the function

```
(lambda ([x : number]) : number x)
```

types to `(number -> number)`, whereas applying inference to it after erasing types yields a much more general type. Therefore, relating these types and giving a precise definition of type equality is not trivial, though we will briefly return to this issue later [REF].

With these preliminaries out of the way, we are now ready to delve into the mechanics of type inference. The most important thing to note is that our simple, recursive-descent type-checking algorithm [REF] will no longer work. That was possible because we already had annotations on all function boundaries, so we could descend into function bodies carrying information about those annotations in the type environment. Sans these annotations, it is not clear how to descend.

In fact, it is not clear that there is any particular direction that makes more sense than another. In a definition like `mapper` above, each fragment of code influences the other. For instance, applying `empty?`, `cons?`, `first`, and `rest` to `l` all point to its being a list. But a list of what? We can't tell from any of those operations. However, the fact that we apply `f` to each (or indeed, any) `first` element means the list members must be of a type that can be passed to `f`. Similarly, we know the output must be a list because of `cons` and `empty`. But what are its elements? They must be the return type of `f`. Finally, note something very subtle: when the argument list is empty, we return

`empty`, not `l` (which we know is bound to `empty` at that point). Using the former leaves the type of the return free to be any kind of list at all (constrained only by what `f` returns); using the latter would force it to be the same type as the argument list.

All this information is in the function. But how do we extract it systematically and in an algorithm that terminates and enjoys the property we have stated above? We do this in two steps. First we *generate constraints*, based on program terms, on what the types must be. Then we *solve constraints* to identify inconsistencies and join together constraints spread across the function body. Each step is relatively simple, but the combination creates magic.

Constraint Generation

Our goal, ultimately, is to find a type to fill into every type annotation position. It will prove to be just as well to find a type for every *expression*. A moment's thought will show that this is likely necessary anyway: for instance, how can we determine the type to put on a function without knowing the type of its body? It is also sufficient, in that if every expression has had its type calculated, this will include the ones that need annotations.

First, we must generate constraints to (later) solve. Constraint generation walks the program source, emitting appropriate constraints on each expression, and returns this set of constraints. It works by recursive descent mainly for simplicity; it really computes a *set* of constraints, so the order of traversal and generation really does not matter in principle—so we may as well pick recursive descent, which is easy—though for simplicity we will use a list to represent this set.

What are constraints? They are simply statements about the types of expressions. In addition, though the binding instances of variables are not expressions, we must calculate their types too (because a function requires both argument and return types). In general, what can we say about the type of an expression?

1. That it is related to the type of some identifier.
2. That it is related to the type of some other expression.
3. That it is a number.
4. That it is a function, whose domain and range types are presumably further constrained.

Thus, we define the following two datatypes:

```
(define-type Constraints
  [eqCon (lhs : Term) (rhs : Term)])

(define-type Term
  [tExp (e : ExprC)]
  [tVar (s : symbol)]
  [tNum]
  [tArrow (dom : Term) (rng : Term)])
```

Now we can define the process of generating constraints:
<***constr-gen***> **::=**

```
(define (cg [e : ExprC]) : (listof Constraints)
  (type-case ExprC e
    <constr-gen-numC-case>
    <constr-gen-idC-case>
    <constr-gen-plusC/multC-case>
    <constr-gen-appC-case>
    <constr-gen-lamC-case>))
```

When the expression is a number, all we can say is that we expect the type of the expression to be numeric:
<***constr-gen-numC-case***> **::=**

```
[numC (_) (list (eqCon (tExp e) (tNum)))]
```

This might sound trivial, but what we don't know is what other expectations are being made of this expression by those containing it. Thus, there is the possibility that some outer expression will contradict the assertion that this expression's type must be numeric, leading to a type error.

For an identifier, we simply say that the type of the expression is whatever we expect to be the type of that identifier:
<***constr-gen-idC-case***> **::=**

```
[idC (s) (list (eqCon (tExp e) (tVar s)))]
```

If the context limits its type, then this expression's type will automatically be limited, and must then be consistent with what its context expects of it.

Addition gives us our first look at a contextual constraint. For an addition expression, we must first make sure we generate (and return) constraints in the two sub-expressions, which might be complex. That done, what do we expect? That each of the sub-expressions be of numeric type. (If the form of one of the sub-expressions demands a type that is not numeric, this will lead to a type error.) Finally, we assert that the entire expression's type is itself numeric.

`append3` is just a three-argument version of `append`.

<***constr-gen-plusC/multC-case***> **::=**

```
[plusC (l r) (append3 (cg l)
                      (cg r)
                      (list (eqCon (tExp l) (tNum))
                            (eqCon (tExp r) (tNum))
                            (eqCon (tExp e) (tNum))))]
```

The case for `multC` is identical other than the variant name.

Now we get to the other two interesting cases, function declaration and application. In both cases, we must remember to generate and return constraints of the sub-expressions.

In a function definition, the type of the function is a function ("arrow") type, whose argument type is that of the formal parameter, and whose return type is that of the body:

<*constr-gen-lamC-case*> **::=**

```
[lamC (a b) (append (cg b)
                    (list (eqCon (tExp e) (tArrow (tVar a) (tExp b))))))]
```

Finally, we have applications. We cannot directly state a constraint on the type of the application. Rather, we can say that the function in the application position must consume arguments of the actual parameter expression's type, and return types of the application expression's type:

<*constr-gen-appC-case*> **::=**

```
[appC (f a) (append3 (cg f)
                     (cg a)
                     (list (eqCon (tExp f) (tArrow (tExp a) (tExp e))))))]
```

And that's it! We have finished generating constraints; now we just have to solve them.

Constraint Solving Using Unification

The process used to solve constraints is known as *unification*. A unifier is given a set of equations. Each equation maps a variable to a term, whose datatype is above. Note one subtle point: we actually have *two* kinds of variables. Both `tvar` and `tExp` are "variables", the former evidently so but the latter equally so because we need to solve for the types of these expressions. (An alternate formulation would introduce fresh type variables for each expression, but we would still need a way to identify which ones correspond to which expression, which `eq?` on the expressions already does automatically. Also, this would generate far larger constraint sets, making visual inspection daunting.)

For our purposes, the goal of unification is generate a *substitution*, or mapping from variables to terms that do not contain any variables. This should sound familiar: we have a set of simultaneous equations in which each variable is used linearly; such equations are solved using *Gaussian elimination*. In that context, we know that we can end up with systems that are both under- and over-constrained. The same thing can happen here, as we will soon see.

The unification algorithm works iteratively over the set of constraints. Because each constraint equation has two terms and each term can be one of four kinds, there are essentially sixteen cases to consider. Fortunately, we can cover all sixteen with fewer actual code cases.

The algorithm begins with the set of all constraints, and the empty substitution. Each constraint is considered once and removed from the set, so in principle the termination argument should be utterly simple, but it will prove to be only slightly more tricky in reality. As constraints are disposed, the substitution set tends to grow. When all constraints have been disposed, unification returns the final substitution set.

For a given constraint, the unifier examines the left-hand-side of the equation. If it is a variable, it is now ripe for elimination. The unifier adds the variable's right-hand-side to the substitution and, to truly eliminate it, replaces all occurrences of the variable in the substitution with the this right-hand-side. In practice this needs to be implemented efficiently; for instance, using a mutational representation of these variables can avoid having to search-and-replace all occurrences. However, in a setting where we might need to backtrack (as we will, in the presence of unification [REF]), the mutational implementation has its own disadvantages.

Do Now!

Did you notice the subtle error above?

The subtle error is this. We said that the unifier *eliminates* the variable by replacing all instances of it in the substitution. However, that assumes that the right-hand-side does not contain any instances of the same variable. Otherwise we have a circular definition, and it becomes impossible to perform this particular substitution. For this reason, unifiers include a *occurs check*: a check for whether the same variable occurs on both sides and, if it does, decline to unify.

Do Now!

Construct a term whose constraints would trigger the occurs check.

Do you remember ω?

Let us now consider the implementation of unification. It is traditional to denote the substitution by the Greek letter Θ.

```
(define-type-alias Subst (listof Substitution))
(define-type Substitution
  [sub [var : Term] [is : Term]])

(define (unify [cs : (listof Constraints)]) : Subst
  (unify/Θ cs empty))
```

Let's get the easy parts out of the way:

<***unify/***Θ> **::=**

```
(define (unify/Θ [cs : (listof Constraints)] [Θ : Subst]) : Subst
  (cond
    [(empty? cs) Θ]
    [(cons? cs)
     (let ([l (eqCon-lhs (first cs))]
           [r (eqCon-rhs (first cs))])
       (type-case Term l
         <unify/Θ-tVar-case>
         <unify/Θ-tExp-case>
         <unify/Θ-tNum-case>
         <unify/Θ-tArrow-case>))]))
```

Now we're ready for the heart of unification. We will depend on a function, `extend+replace`, with this signature: `(Term Term Subst -> Subst)`. We expect this to perform the occurs test and, if it fails (i.e., there is no circularity), extends the substituion and replaces all existing instances of the first term with the second in the substitution. Similarly, we will assume the existence of `lookup`: `(Term subst -> (optionof Term))`

Exercise

Define `extend+replace` and `lookup`.

If the left-hand of a constraint equation is a variable, we first look it up in the substitution. If it is present, we replace the current constraint with a new one; otherwise, we extend the substitution:

<*unify/Θ-tVar-case*> **::=**

```
[tVar (s) (type-case (optionof Term) (lookup l Θ)
            [some (bound)
                  (unify/Θ (cons (eqCon bound r)
                                 (rest cs))
                           Θ)]
            [none ()
                  (unify/Θ (rest cs)
                           (extend+replace l r Θ))])]
```

The same logic applies when it is an expression designator:

<*unify/Θ-tExp-case*> **::=**

```
[tExp (e) (type-case (optionof Term) (lookup l Θ)
            [some (bound)
                  (unify/Θ (cons (eqCon bound r)
                                 (rest cs))
                           Θ)]
            [none ()
                  (unify/Θ (rest cs)
                           (extend+replace l r Θ))])]
```

If it is a base type, such as a number, then we examine the right-hand side. There are four possibilities, for the four different kinds of terms:

- If it is a number, then we have an equation that claims that the type `num` is the same as the type `num`, which is patently true. We can therefore ignore this constraint—because it tells us nothing new—and move on to the remainder.

  You should, of course, question why such a constraint would have come about in the first place. Clearly, our constraint generator did not generate such constraints. However, a prior extension to the current substitution might have resulted in this situation. Indeed, in practice we will encounter several of these.

- If it is a function type, then we clearly have a type error, because numeric and function types are disjoint. Again, we would never have generated such a constraint directly, but it must have resulted from a prior substitution.
- It could have been one of the two variable kinds. However, we have carefully arranged our constraint generator to never put these on the right-hand-side. Furthermore, substitution will not introduce them on the right-hand-side, either. Therefore, these two cases cannot occur.

This results in the following code:

<***unify/Θ-tNum-case***> **::=**

```
[tNum () (type-case Term r
           [tNum () (unify/Θ (rest cs) Θ)]
           [else (error 'unify "number and something else")])]
```

Finally, we left with function types. Here the argument is almost exactly the same as for numeric types.

<***unify/Θ-tArrow-case***> **::=**

```
[tArrow (d r) (type-case Term r
                [tArrow (d2 r2)
                        (unify/Θ (cons (eqCon d d2)
                                       (cons (eqCon r r2)
                                             cs))
                                 Θ)]
                [else (error 'unify "arrow and something else")])]
```

Note that we do not always shrink the size of the constraint set, so a simple argument does not suffice for proving termination. Instead, we must make an argument based on the size of the constraint set, and on the size of the substitution (including the number of variables in it).

The algorithm above is very general in that it works for all sorts of type terms, not only numbers and functions. We have used numbers as a stand-in for all form of base types; functions, similarly, stand for all constructed types, such as `listof` and `vectorof`.

With this, we are done. Unification produces a substitution. We can now traverse the substitution and find the types of all the expressions in the program, then insert the type annotations accordingly. A theorem, which we will not prove here, dictates that the success of the above process implies that the program would have typed-checked, so we need not explicitly run the type-checker over this program.

Observe, however, that the nature of a type error has now changed dramatically. Previously, we had a recursive-descent algorithm that walked a expressions using a type environment. The bindings in the type environment were programmer-declared types, and could hence be taken as (intended) authoritative *specifications* of types. As a result, any mismatch was blamed on the expressions, and reporting type errors was simple (and easy to understand). Here, however, a type error is a *failure to notify*. The unification failure is based on events that occur at the confluence of two

smart algorithms—constraint generation and unification—and hence are not necessarily comprehensible to the programmer. In particular, the equational nature of these constraints means that the location reported for the error, and the location of the "true" error, could be quite far apart. As a result, producing better error messages remains an active research area.

In practice the algorithm will maintain metadata on which program source terms were involved and probably on the history of unification, to be able to trace errors back to the source program.

Finally, remember that the constraints may not precisely dictate the type of all variables. If the system of equations is *over*-constrained, then we get clashes, resulting in type errors. If instead the system is *under*-constrained, that means we don't have enough information to make definitive statements about all expressions. For instance, in the expression `(lambda (x) x)` we do not have enough constraints to indicate what the type of `x`, and hence of the entire expression, must be. This is not an error; it simply means that `x` is free to be *any* type at all. In other words, its type is "the type of `x` `->` the type of `x`" with no other constraints. The types of these underconstrained identifiers are presented as type variables, so the above expression's type might be reported as `('a -> 'a)`.

The unification algorithm actually has a wonderful property: it automatically computes the *most general types* for an expression, also known as *principal types*. That is, any actual type the expression can have can be obtained by instantiating the inferred type variables with actual types. This is a remarkable result: in another example of computers beating humans, it says that no human can generate a more general type than the above algorithm can!

Let-Polymorphism

Unfortunately, though these type variables are superficially similar to the polymorphism we had earlier [REF], they are not. Consider the following program:

```
(let ([id (lambda (x) x)])
  (if (id true)
      (id 5)
      (id 6)))
```

If we write it with explicit type annotations, it type-checks:

```
(if ((id boolean) true)
    ((id number) 5)
    ((id number) 6))
```

However, if we use type inference, it does not! That is because the `'a`'s in the type of `id` unify either with `boolean` or with `number`, depending on the order in which the constraints are processed. At that point `id` effectively becomes either a `(boolean -> boolean)` or `(number -> number)` function. At the use of `id` of the other type, then, we get a type error!

The reason for this is because the types we have inferred through unification are not actually *polymorphic*. This is important to remember: just because you type variables, you haven't seen polymorphism! The type variables could be unified at the next use,

at which point you end up with a mere monomorphic function. Rather, true polymorphism only obtains when you have true *instantiation* of type variables.

In languages with true polymorphism, then, constraint generation and unification are not enough. Instead, languages like ML, Haskell, and even our typed programming language, implement something colloquially called *let-polymorphism*. In this strategy, when a term with type variables is bound in a lexical context, the type is automatically promoted to be a quantified one. At each use, the term is effectively automatically instantiated.

There are many implementation strategies that will accomplish this. The most naive (and unsatisfying) is to merely *copy the code* of the bound identifier; thus, each use of `id` above gets its own copy of `(lambda (x) x)`, so each gets its own type variables. The first might get the type `('a -> 'a)`, the second `('b -> 'b)`, the third `('c -> 'c)`, and so on. None of these type variables clash, so we get the effect of polymorphism. Obviously, this not only increases program size, it also does not work in the presence of recursion. However, it gives us insight into a better solution: instead of copying the code, why not just copy the *type*? Thus at each use, we create a renamed copy of the inferred type: `id`'s `('a -> 'a)` becomes `('b -> 'b)` at the first use, and so on, thus achieving the same effect as copying code but without its burdens. Because all these strategies effectively mimic copying code, however, they only work within a lexical context.

15.3.3 Union Types

Suppose we want to construct a list of zoo animals, of which there are many kinds: armadillos, boa constrictors, and so on. Currently, we are forced to create a new datatype:

"In Texas, there ain't nothing in the middle of the road but a yellow line and dead armadillos."—Jim Hightower

```
(define-type Animal
  [armadillo (alive? : boolean)]
  [boa (length : number)])
```

and make a list of these: `(listof Animal)`. The type `Animal` therefore represents a "union" of `armadillo` and `boa`, except the only way to construct such unions is to make a new type every time: if we want to represent the union of animals and plants, we need

```
(define-type LivingThings
  [animal (a : Animal)]
  [plant (p : Plant)])
```

so an actual animal is now one extra "level" deep. These datatypes are called *tagged unions* or *discriminated unions*, because we must introduce explicit tags (or *discriminators*), such as `animal` and `plant`, to tell them apart. In turn, a structure can only reside inside a datatype declaration; we have had to create datatypes with just one variant, such as

```
(define-type Constraints
  [eqCon (lhs : Term) (rhs : Term)])
```

to hold the datatype, and everywhere we've had to use the type `Constraints` because `eqCon` is not itself a type, only a variant that can be distinguished at run-time.

Either way, the point of a union type is to represent a disjunction, or "or". A value's type is one of the types in the union. A value usually belongs to only one of the types in the union, though this is a function of precisely how the union types are defined, whether there are rules for normalizing them, and so on.

Structures as Types

A natural reaction to this might be, why not lift this restriction? Why not allow each structure to exist on its own, and define a type to be a union of some collection of structures? After all, in languages ranging from C to Racket, programmers can define stand-alone structures without having to wrap them in some other type with a tag constructor! For instance, in raw Racket, we can write

```
(struct armadillo (alive?))
(struct boa (length))
```

and a comment that says

```
;; An Animal is either
;; - (armadillo <boolean>)
;; - (boa <number>)
```

but without enforced static types, the comparison is messy. However, we can more directly compare with *Typed Racket*, a typed form of Racket that is built into DrRacket. Here is the same typed code:

```
#lang typed/racket

(struct: armadillo ([alive? : Boolean]))
(struct: boa ([length : Real])) ;; feet
```

We can now define functions that consume values of type `boa` without any reference to `armadillos`:

```
;; http://en.wikipedia.org/wiki/Boa_constrictor#Size_and_weight
(define: (big-one? [b : boa]) : Boolean
  (> (boa-length b) 8))
```

In fact, if we apply this function to any other type, including an armadillo—`(big-one? (armadillo true))`—we get a *static* error. This is because armadillos are no more related to boas than numbers or strings are.

Of course, we can still define a union of these types:

```
(define-type Animal (U armadillo boa))
```

and functions over it:

```
(define: (safe-to-transport? [a : Animal]) : Boolean
  (cond
    [(boa? a) (not (big-one? a))]
    [(armadillo? a) (armadillo-alive? a)]))
```

Whereas before we had *one type with two variants*, now we have *three types*. It just so happens that two of the types form a union of convenience to define a third.

Untagged Unions

It might appear that we still need to have discriminative tags, but we don't. In languages with union types, the effect of the `optionof` type constructor is often obtained by combining the intended return type with a disjoint one representing failure or `noneness`. For instance, here is the moral equivalent of `(optionof number)`:

```
(define-type MaybeNumber (U Number Boolean))
```

For that matter, `Boolean` may itself be a union of `True` and `False`, as it is in Typed Racket, so a more accurate simulation of the option type would be:

```
(define-type MaybeNumber (U Number False))
```

More generally, we could define

```
(struct: none ())
(define-type (Maybeof T) (U T none))
```

which would work for all types, because `none` is a new, distinct type that cannot be confused for any other. This gives us the same benefit as the `optionof` type, except the value we want is not buried one level deep inside a `some` structure, but is rather available immediately. For instance, consider `member`, which has this Typed Racket type:

```
(All (a) (a (Listof a) -> (U False (Listof a))))
```

If the element is not found, `member` returns `false`. Otherwise, it returns the list starting from the element onward (i.e., the first element of the list will be the desired element):

```
> (member 2 (list 1 2 3))
'(2 3)
```

To convert this to use `Maybeof`, we can write

```
(define: (t) (in-list? [e : t] [l : (Listof t)]) : (Maybeof (Listof t))
  (let ([v (member e l)])
    (if v
        v
        (none))))
```

which, if the element is not found, returns the value `(none)`, but if it is found, still returns a list

```
> (in-list? 2 (list 1 2 3))
'(2 3)
```

so that there is no need to remove the list from a `some` wrapper.

Discriminating Untagged Unions

It's one thing to put values into unions; we have to also consider how to take them out, in a well-typed manner. In our ML-like type system, we use a stylized notation—`type-case` in our language, pattern-matching in ML—to identify and pull apart the pieces. In particular, when we write

```
(define (safe-to-transport? [a : Animal]) : boolean
  (type-case Animal a
    [armadillo (a?) a?]
    [boa (l) (not (big-one? l))]))
```

the type of `a` remains the same in the entire expression. The identifiers `a?` and `l` are bound to a boolean and numeric value, respectively, and `big-one?` must now be written to consume those types, not `armadillo` and `boa`. Put in different terms, we cannot have a function `big-one?` that consumes `boa`s, because there is no such type.

In contrast, with union types, we do have the `boa` type. Therefore, we follow the principle that the act of asking predicates of a value *narrows the type*. For instance, in the `cond` case

```
[(boa? a) (not (big-one? a))]
```

though `a` begins as type `Animal`, after it passes the `boa?` test, the type checker is expected to narrow its type to just the `boa` branch, so that the application of `big-one?` is well-typed. In turn, in the rest of the conditional its type is *not* `boa`—in this case, that leaves only one possibility, `armadillo`. This puts greater pressure on the type-checker's ability to test and recognize certain patterns—known as *if-splitting*—without which it would be impossible to program with union types; but it can always default to recognizing just those patterns that the ML-like system would have recognized, such as pattern-matching or `type-case`.

Retrofitting Types

It is unsurprising that Typed Racket uses union types. They are especially useful when *retrofitting* types onto existing programming languages whose programs were not defined with an ML-like type discipline in mind, such as in scripting languages. A common principle of such retrofitted types is to statically catch as many dynamic exceptions as possible. Of course, the checker must ultimately reject some programs, and if it rejects too many programs that would have run without an error, developers are unlikely to adopt it. Because these programs were written without type-checking in mind, the type checker may therefore need to go to heroic lengths to accept what are considered reasonable idioms in the language.

Unless it implements an interesting idea called *soft typing*, which rejects no programs but provides information about points where the program would not have been typeable.

Consider the following JavaScript function:

```
var slice = function (arr, start, stop) {
  var result = [];
  for (var i = 0; i <= stop - start; i++) {
    result[i] = arr[start + i];
```

```
  }
  return result;
}
```

It consumes an array and two indices, and produces the sub-array between those indices. For instance, `slice([5, 7, 11, 13], 0, 2)` produces `[5, 7, 11]`.

In JavaScript, however, developers are free to leave out any or all trailing arguments to a function. Every elided argument is given a special value, `undefined`, and it is up to the function to cope with this. For instance, a typical implementation of `splice` would let the user drop the third argument; the following definition

```
var slice = function (arr, start, stop) {
  if (typeof stop == "undefined")
    stop = arr.length - 1;
  var result = [];
  for (var i = 0; i <= stop - start; i++) {
    result[i] = arr[start + i];
  }
  return result;
}
```

automatically returns the subarray until the end of the array: thus, `slice([5, 7, 11, 13], 2)` returns `[11, 13]`.

In Typed JavaScript, a programmer can explicitly indicate a function's willingness to accept fewer arguments by giving a parameter the type `U Undefined`, giving it the type

Built at Brown by Arjun Guha and others. See our Web site.

```
∀ t : (Array[t] * Int * (Int U Undefined) -> Array[t])
```

In principle, this means there is a potential type error at the expression `stop - start`, because `stop` may not be a number. However, the assignment to `stop` sets it to a numeric type precisely when it was elided by the user. In other words, in all control paths, `stop` will eventually have a numeric type before the subtraction occurs, so this function is well-typed. Of course, this requires the type-checker to be able to reason about both control-flow (through the conditional) and state (through the assignment) to ensure that this function is well-typed; but Typed JavaScript can, and can thus bless functions such as this.

Design Choices

In languages with union types, it is common to have

- Stand-alone structure types (often represented using classes), rather than datatypes with variants.
- Ad hoc collections of structures to represent particular types.
- The use of sentinel values to represent failure.

To convert programs written in this style to an ML-like type discipline would be extremely onerous. Therefore, many retrofitted type systems adopt union types to ease the process of typing.

Of the three properties above, the first seems morally neutral, but the other two warrant more discussion. We will address them in reverse order.

- Let's tackle sentinels first. In many cases, sentinels ought to be replaced with exceptions, but in many languages, exceptions can be very costly. Thus, developers prefer to make a distinction between truly exceptional situations—that ought not occur—and situations that are expected in the normal course of operation. Checking whether an element is in a list and failing to find it is clearly in the latter category (if we already knew the element was or wasn't present, there would be no need to run this predicate). In the latter case, using sentinels is reasonable.

  However, we must square this with the observation that failure to check for exceptional sentinel values is a common source of error—and indeed, security flaws—in C programs. This is easy to reconcile. In C, the sentinel is of the *same type* (or at least, effectively the same type) as the regular return value, and furthermore, there are no run-time checks. Therefore, the sentinel can be used as a legitimate value without a type error. As a result, a sentinel of `0` can be treated as an address into which to allocate data, thus potentially crashing the system. In contrast, our sentinel is of a truly new type that cannot be used in any computation. We can easily reason about this by observing that no existing functions in our language consume values of type `none`.

- Setting aside the use of "ad hoc", which is pejorative, are different groupings of a set of structures a good idea? In fact, such groupings occur even in programs using an ML-like discipline, when programmers want to carve different subuniverses of a larger one. For instance, ML programmers use a type like

  ```
  (define-type SExp
    [numSexp (n : number)]
    [strSexp (s : string)]
    [listSexp (l : (listof SExp))])
  ```

  to represent s-expressions. If a function now operates on just some subset of these terms—say just numbers and lists of numbers—they must create a fresh type, and convert values between the two types even though their underlying representations are essentially identical. As another example, consider the set of CPS expressions. This is clearly a subset of all possible expressions, but if we were to create a fresh datatype for it, we would not be able to use any existing programs that process expressions—such as the interpreter.

In other words, union types appear to be a reasonable variation on the ML-style type system we have seen earlier. However, even within union types there are design variations, and these have consequences. For instance, can the type system create new unions, or are user-defined (and named) unions permitted? That is, can an expression like this

```
(if (phase-of-the-moon)
    10
    true)
```

be allowed to type (to `(U Number Boolean)`), or is it a type error to introduce unions that have not previously been named and explicitly identified? Typed Racket provides the former: it will construct truly ad hoc unions. This is arguably better for importing existing code into a typed setting, because it is more flexible. However, it is less clear whether this is a good design for writing new code, because unions the programmer did not intend can occur and there is no way to prevent them. This offers an unexplored corner in the design space of programming languages.

15.3.4 Nominal Versus Structural Systems

In our initial type-checker, two types were considered equivalent if they had the same structure. In fact, we offered no mechanism for naming types at all, so it is not clear what alternative we had.

Now consider Typed Racket. A developer can write

```
(define-type NB1 (U Number Boolean))
(define-type NB2 (U Number Boolean))
```

followed by

```
(define: v : NB1 5)
```

Suppose the developer also defines the function

```
(define: (f [x : NB2]) : NB2 x)
```

and tries to apply `f` to `v`, i.e., `(f v)`: should this application type or not?

There are two perfectly reasonable interpretations. One is to say that `v` was declared to be of type `NB1`, which is a different *name* than `NB2`, and hence should be considered a different *type*, so the above application should result in an error. Such a system is called *nominal*, because the name of a type is paramount for determining type equality.

In contrast, another interpretation is that because the *structure* of `NB1` and `NB2` are identical, there is no way for a developer to write a program that behaves differently on values of these two types, so these two types should be considered identical. Such a type system is called *structural*, and would successfully type the above expression. (Typed Racket follows a structural discipline, again to reduce the burden of importing existing untyped code, which—in Racket—is usually written with a structural interpretation in mind. In fact, Typed Racket not only types `(f v)`, it prints the result as having type `NB1`, despite the return type annotation on `f`!)

The difference between nominal and structural typing is most commonly contentious in object-oriented languages, and we will return to this issue briefly later [REF]. However, the point of this section is to illustrate that these questions are not intrinsically about "objects". Any language that permits types to be named—as all must,

If you want to get especially careful, you would note that there is a difference between being considered the same and actually being the same. We won't go into this issue here, but consider the implication for a compiler writer choosing representations of values, especially in a language that allows run-time inspection of the static types of values.

for programmer sanity—must contend with this question: is naming merely a convenience, or are the choices of names intended to be meaningful? Choosing the former answer leads to structural typing, while choosing the latter leads down the nominal path.

15.3.5 Intersection Types

Since we've just explored union types, you must naturally wonder whether there are also *intersection* types. Indeed there are.

If a union type means that a value (of that type) belongs to one of the types in the union, an intersection type clearly means the value belongs to *all* the types in the intersection: a conjunction, or "and". This might seem strange: how can a value belong to more than one type?

As a concrete answer, consider *overloaded functions*. For instance, in some languages + operates on both numbers and strings; given two numbers it produces a number, and given two strings it produces a string. In such a language, what is the proper type for +? It is not `(number number -> number)` alone, because that would reject its use on strings. By the same reasoning, it is not `(string string -> string)` alone either. It is not even

```
(U (number number -> number)
   (string string -> string))
```

because + is not just one of these functions: it truly is both of them. We could ascribe the type

```
((number U string) (number U string) -> (number U string))
```

reflecting the fact that each argument, and the result, can be only one of these types, not both. Doing so, however, leads to a loss of precision.

Do Now!

In what way does this type lose precision?

Observe that with this type, the return type on *all* invocations is `(number U string)`. Thus, on every return we must distinguish beween numeric and string returns, or else we will get a type error. Thus, even though we know that if given two numeric arguments we will get a numeric result, this information is lost to the type system.

More subtly, this type permits each argument's type to be chosen independently of the other. Thus, according to this type, the invocation `(+ 3 "x")` is perfectly valid (and produces a value of type `(number U string)`). But of course the addition operation we have specified is not defined for these inputs at all!

Thus the proper type to ascribe this form of addition is

```
(∧ (number number -> number)
   (string string -> string))
```

where $\wedge$ should be reminiscent of the conjunction operator in logic. This permits invocation with two numbers or two strings, but nothing else. An invocation with two numbers has a numeric result type; one with two strings has a string result type; and nothing else. This corresponds precisely to our intended behavior for overloading (sometimes also called *ad hoc polymorphism*). Observe that this only handles a finite number of overloaded cases.

15.3.6 Recursive Types

Now that we've seen union types, it pays to return to our original recursive datatype formulation. If we accept the variants as type constructors, can we write the recursive type as a union over these? For instance, returning to `BTnum`, shouldn't we be able to describe it as equivalent to

```
((BTmt) U (BTnd number BTnum BTnum))
```

thereby showing that `BTmt` is a zero-ary constructor, and `BTnd` takes three parameters? Except, what are the types of those three parameters? In the type we've written above, `BTnum` is either built into the type language (which is unsatisfactory) or unbound. Perhaps we mean

```
BTnum = ((BTmt) U (BTnd number BTnum BTnum))
```

Except now we have an equation that has no obvious solution (remember ω?).

This situation should be familiar from recursion in values [REF]. Then, we invented a recursive function constructor (and showed its implementation) to circumvent this problem. We similarly need a recursive *type* constructor. This is conventionally called μ (the Greek letter "mu"). With it, we would write the above type as

```
μ BTnum : ((BTmt) U (BTnd number BTnum BTnum))
```

μ is a binding construct; it binds `BTnum` to the entire type written after it, including the recursive binding of `BTnum` itself. In practice, of course, this entire recursive type is the one we wish to call `BTnum`:

```
BTnum = μ BTnum : ((BTmt) U (BTnd number BTnum BTnum))
```

Though this looks like a circular definition, notice that the name `BTnum` on the right does not depend on the one to the left of the equation: i.e., we could rewrite this as

```
BTnum = μ T : ((BTmt) U (BTnd number T T))
```

In other words, this definition of `BTnum` truly can be thought of as syntactic sugar and replaced everywhere in the program without fear of infinite regress.

At a semantic level, there are usually two very different ways of thinking about the meaning of types bound by μ: they can be interpreted as *isorecursive* or *equirecursive*. The distinction between these is, however, subtle and beyond the scope of this chapter. It suffices to note that a recursive type can be treated as equivalent to its unfolding. For instance, if we define a numeric list type as

This material is covered especially well in Pierce's book.

```
NumL = μ T : ((MtL) U (ConsL number T))
```

then

```
  μ T : ((MtL) U (ConsL number T))
= (MtL) U (ConsL number (μ T : ((MtL) U (ConsL number T))))
= (MtL) U (ConsL number (MtL))
        U (ConsL number (ConsL number (μ T : ((MtL) U (ConsL number T)))))
```

and so on (iso- and equi-recursiveness differ in precisely what the notion of equality is: definitional equality or isomorphism). At each step we simply replace the `T` parameter with the entire type. As with value recursion, this means we can "get another" `ConsL` constructor upon demand. Put differently, the *type* of a list can be written as the union of zero or arbitrarily many elements; this is the same as the *type* that consists of zero, one, or arbitrarily many elements; and so on. Any lists of numbers fits all (and precisely) these types.

Observe that even with this informal understanding of μ, we can now provide a type to ω, and hence to Ω.

Exercise

Ascribe types to ω and Ω.

15.3.7 Subtyping

Imagine we have a typical binary tree definition; for simplicity, we'll assume that all the values are numbers. We will write this in Typed Racket to illustrate a point:

```
#lang typed/racket

(define-struct: mt ())
(define-struct: nd ([v : Number] [l : BT] [r : BT]))
(define-type BT (U mt nd))
```

Now consider some concrete tree values:

```
> (mt)
- : mt
#<mt>
> (nd 5 (mt) (mt))
- : nd
#<nd>
```

Observe that each structure constructor makes a value of its own type, not a value of type `BT`. But consider the expression `(nd 5 (mt) (mt))`: the definition of `nd` declares that the sub-trees must be of type `BT`, and yet we are able to successfully give it values of type `mt`.

Obviously, it is not coincidental that we have defined `BT` in terms of `mt` and `nd`. However, it does indicate that when type-checking, we cannot simply be checking for function equality, at least not as we have so far. Instead, we must be checking that one

type "fits into" the other. This notion of fitting is called *subtyping* (and the act of being fit, *subsumption*).

The essence of subtyping is to define a relation, usually denoted by `<:`, that relates pairs of types. We say `S <: T` if a value of type `S` can be given where a value of type `T` is expected: in other words, subtyping formalizes the notion of *substitutability* (i.e., anywhere a value of type `T` was expected, it can be replaced with—substituted by—a value of type `S`). When this holds, `S` is called the *subtype* and `T` the *supertype*. It is useful (and usually accurate) to take a subset interpretation: if the values of `S` are a subset of `T`, then an expression expecting `T` values will not be unpleasantly surprised to receive only `S` values.

Subtyping has a pervasive effect on the type system. We have to reexamine every kind of type and understand its interaction with subtyping. For base types, this is usually quite obvious: disjoint types like `number`, `string`, etc., are all unrelated to each other. (In languages where one base type is used to represent another—for instance, in some scripting languages numbers are merely strings written with a special syntax, and in other languages, booleans are merely numbers—there might be subtyping relationships even between base types, but these are not common.) However, we do have to consider how subtyping interacts with every single compound type constructor.

In fact, even our very diction about types has to change. Suppose we have an expression of type `T`. Normally, we would say that it produces values of type `T`. Now, we should be careful to say that it produces values of *up to* or *at most* `T`, because it may only produce values of a subtype of `T`. Thus every reference to a type should implicitly be cloaked in a reference to the potential for subtyping. To avoid pestering you I will refrain from doing this, but be wary that it is possible to make reasoning errors by not keeping this implicit interpretation in mind.

Unions

Let us see how unions interact with subtyping. Clearly, every sub-union is a subtype of the entire union. In our running example, clearly every `mt` value is also a `BT`; likewise for `nd`. Thus,

```
mt <: BT
nd <: BT
```

As a result, `(mt)` also has type `BT`, thus enabling the expression `(nd 5 (mt) (mt))` to itself type, and to have the type `nd`—and hence, also the type `BT`. In general,

```
S <: (S U T)
T <: (S U T)
```

(we write what seems to be the same rule twice just to make clear it doesn't matter which "side" of the union the subtype is on). This says that a value of `S` can be thought of as a value of `S U T`, because any expression of type `S U T` can indeed contain a value of type `S`.

Intersections

While we're at it, we should also briefly visit intersections. As you might imagine, intersections behave dually:

```
(S ∧ T) <: S
(S ∧ T) <: T
```

To convince yourself of this, take the subset interpretation: if a value is both `S` and `T`, then clearly it is either one of them.

Do Now!

Why are the following two *not* valid subsumptions?

1. `(S U T) <: S`
2. `T <: (S ∧ T)`

The first is not valid because a value of type `T` is a perfectly valid element of type `(S U T)`. For instance, a number is a member of type `(string U number)`. However, a number cannot be supplied where a value of type `string` is expected.

As for the second, in general, a value of type `T` is not also a value of type `S`. Any consumer of a `(S ∧ T)` value is expecting to be able to treat it as both a `T` and a `S`, and the latter is not justified. For instance, given our overloaded `+` from before, if `T` is `(number number -> number)`, then a function of this type will not know how to operate on strings.

Functions

We have also seen parametric datatypes. In this edition, exploring subtyping for them is left as an exercise for the reader.

We have seen one more constructor: functions. We must therefore determine the rules for subtyping when either type can be a function. Since we usually assume functions are disjoint from all other types, we therefore only need to consider when one function type is a subtype of another: i.e., when is

```
(S1 -> T1) <: (S2 -> T2)
```

? For convenience, let us call the type `(S1 -> T1)` as `f1`, and `(S2 -> T2)` as `f2`. The question then is, if an expression is expecting functions of the `f2` type, when can we safely give it functions with the `f1` type? It is easiest to think through this using the subset interpretation.

Consider a use of the `f2` type. It returns values of type `T2`. Thus, the context surrounding the function application is satisfied with values of type `T2`. Clearly, if `T1` is the same as `T2`, the use of `f2` would continue to type; similarly, if `T1` consists of a subset of `T2` values, it would still be fine. The only problem is if `T1` has more values than `T2`, because the context would then encounter unexpected values that would result in undefined behavior. In other words, we need that `T1 <: T2`. Observe that the "direction" of containment is the same as that for the entire function type; this is called *covariance* (both vary in the same direction). This is perhaps precisely what you expected.

By the same token, you might expect covariance in the argument position as well: namely, that `S1 <: S2`. This would be predictable, and wrong. Let's see why.

An application of a function with `f2` type is providing parameter values of type `S2`. Suppose we instead substitute the function with one of type `f1`. If we had that `S1 <: S2`, that would mean that the new function accepts only values of type`S1`—a strictly smaller set. That means there may be some inputs—specifically those in `S2` that are not in `S1`—that the application is free to provide on which the substituted function is not defined, again resulting in undefined behavior. To avoid this, we have to make the subsumption go in the other direction: the substituting function should accept at least as many inputs as the one it replaces. Thus we need `S2 <: S1`, and say the function position is *contravariant*: it goes against the direction of subtyping.

Putting together these two observations, we obtain a subtyping rule for functions (and hence also methods):

```
(S2 <: S1) and (T1 <: T2) => (S1 -> T1) <: (S2 -> T2)
```

Implementing Subtyping

Of course, these rules assume that we have modified the type-checker to respect subtyping. The essence of subtyping is a rule that says, if an expression `e` is of type `S`, and `S <: T`, then `e` also has type `T`. While this sounds intuitive, it is also immediately problematic for two reasons:

- Until now all of our type rules have been syntax-driven, which is what enabled us to write a recursive-descent type-checker. Now, however, we have a rule that applies to *all* expressions, so we can no longer be sure when to apply it.
- There could be many levels of subtyping. As a result, it is no longer obvious when to "stop" subtyping. In particular, whereas before type-checking was able to calculate the type of an expression, now we have many possible types for each expression; if we return the "wrong" one, we might get a type error (due to that not being the type expected by the context) even though there exists some other type that was the one expected by the context.

What these two issues point to is that the description of subtyping we are giving here is fundamentally *declarative*: we are saying what must be true, but not showing how to turn it into an algorithm. For each actual type language, there is a less or more interesting problem in turning this into *algorithmic subtyping*: an actual algorithm that realizes a type-checker (ideally one that types exactly those programs that would have typed under the declarative regime, i.e., one that is both sound and complete).

15.3.8 Object Types

As we've mentioned earlier, types for objects are typically riven into two camps: nominal and structural. Nominal types are familiar to most programmers through Java, so I won't say much about them here. Structural types for objects dictate that an object's type is itself a structured object, consisting of names of fields and their types. For instance, an object with two methods, `add1` and `sub1` [REF], would have the type

```
{add1 : (number -> number), sub1 : (number -> number)}
```

(For future reference, let's call this type `addsub`.) Type-checking would then follow along predictable lines: for field access we would simply ensure the field exists and would use its declared type for the dereference expression; for method invocation we would have to ensure not only that the member exists but that it has a function type. So far, so straightforward.

Object types become complicated for many reasons:

- Self-reference. What is the type of `self`? It must the same type as the object itself, since any operation that can be applied to the object from the "outside" can also be applied to it from the "inside" using `self`. This means object types are recursive types.
- Access controls: private, public, and other restrictions. These lead to a distinction in the type of an object from "outside" and "inside".
- Inheritance. Not only do we have to give a type to the parent object(s), what is visible along inheritance paths may, again, differ from what is visible from the "outside".
- The interplay between multiple-inheritance and subtyping.
- The relationship between classes and interfaces in languages like Java, which has a run-time cost.
- Mutation.
- Casts.
- Snakes on a plane.

Whole books are therefore devoted to this topic. Abadi and Carelli's *A Theory of Objects* is important but now somewhat dated. Bruce's *Foundations of Object-Oriented Languages: Types and Semantics* is more modern, and also offers more gentle exposition. Pierce covers all the necessary theory beautifully.

and so on. Some of these problems simplify in the presence of nominal types, because given a type's name we can determine everything about its behavior (the type declarations effectively become a dictionary through which the object's description can be looked up on demand), which is one argument in favor of nominal typing.

A full exposition of these issues will take much more room than we have here. For now, we will limit ourselves to one interesting question. Remember that we said subtyping forces us to consider every type constructor? The structural typing of objects introduces one more: the object type constructor. We therefore have to understand its interaction with subtyping.

Before we do, let's make sure we understand what an object type even means. Consider the type `addsub` above, which lists two methods. What objects can be given this type? Obviously, an object with just those two methods, with precisely those two types, is eligible. Equally obviously, an object with only one and not the other of those two methods, no matter what else it has, is not. But the phrase "no matter what else it has" is meant to be leading. What if an object represents an arithmetic package that also contains methods `+` and `*`, in addition to the above two (all of the appropriate type)? In that case we certainly have an object that can supply those two methods,

Note that Java's approach is not the only way to build a nominal type system. We have already argued that Java's class system needlessly restricts the expressive power of programmers [REF]; in turn, Java's nominal type system needlessly conflates types (which are interface descriptions) and implementations. It is, therefore, possible to have much better nominal type systems than Java's. Scala, for instance, takes significant steps in this direction.

so the arithmetic package certainly has type `addsub`. Its other methods are simply inaccessible using type `addsub`.

Let us write out the type of this package, in full, and call this type `as+*`:

```
{add1  : (number -> number),
 sub1  : (number -> number),
 +     : (number number -> number),
 *     : (number number -> number)}
```

What we have just argued is that an object of type `as+*` should also be allowed to claim the type `addsub`, which means it can be substituted in any context expecting a value of type `addsub`. In other words, we have just said that we want `as+* <: addsub`:

```
{add1  : (number -> number),           {add1 : (number -> number),
 sub1  : (number -> number),       <:   sub1 : (number -> number)}
 +     : (number number -> number),
 *     : (number number -> number)}
```

This may momentarily look confusing: we've said that subtyping follows set inclusion, so we would expect the smaller set on the left and the larger set on the right. Yet, it looks like we have a "larger type" (certainly in terms of character count) on the left and a "smaller type" on the right.

To understand why this is sound, it helps to develop the intuition that the "larger" the type, the fewer values it can have. Every object that has the four methods on the left clearly also has the two methods on the right. However, there are many objects that have the two methods on the right that fail to have all four on the left. If we think of a type as a constraint on acceptable value shapes, the "bigger" type imposes more constraints and hence admits fewer values. Thus, though the *types* may appear to be of the wrong sizes, everything is well because the sets of values they subscribe are of the expected sizes.

More generally, this says that by dropping fields from an object's type, we obtain a supertype. This is called *width subtyping*, because the subtype is "wider", and we move up the subtyping hierarchy by adjusting the object's "width". We see this even in the nominal world of Java: as we go up the inheritance chain a class has fewer and fewer methods and fields, until we reach `Object`, the supertype of all classes, which has the fewest. Thus for all class types `C` in Java, `C <: Object`.

Somewhat confusingly, the terms *narrowing* and *widening* are sometimes used, but with what some might consider the opposite meaning. To widen is to go from subtype to supertype, because it goes from a "narrower" (smaller) to a "wider" (bigger) set. These terms evolved independently, but unfortunately not consistently.

As you might expect, there is another important form of subtyping, which is *within* a given member. This simply says that any particular member can be subsumed to a supertype in its corresponding position. For obvious reasons, this form is called *depth subtyping*.

Exercise

> Construct two examples of depth subtyping. In one, give the field itself an object type, and use width subtyping to subtype that field. In the other, give the field a function type.

Java has limited depth subtyping, preferring types to be *invariant* down the object hierarchy because this is a safe option for conventional mutation.