Chapter 6

Type Systems for Impcore and μScheme

Contents

The languages of the preceding chapters, Impcore and μScheme, are *dynamically typed*, which is to say that such errors as applying a function to the wrong number of arguments, adding non-numbers, or applying `car` to symbols are not detected until run time. Dynamically typed languages are very flexible, but on any given execution, a program written in a dynamically typed language might surprise you; even a simple mistake like typing `cdr` when you meant `car` won't be detected until you find a test case that exercises the bad code. Even using `cdr` instead of `car` doesn't lead to a fault right away: `cdr` simply returns a list in a context where you were expecting an element instead. But if, for example, you then try to add 1 to the result of applying `cdr`, you are likely to get a run-time error message saying you tried to add 1 to a list value, which is not a number. This kind of error is called a *checked run-time error*. The idea of *static typing* is to detect such errors at compile time, without having to run the faulty code.

Before it is run, a program in a statically typed language undergoes a type analysis, which decides if it is OK to run the program. Different languages use different analyses, but all analyses spring from two basic approaches. In *type checking*, the programmer writes type annotations in the code, and these annotations determine the set of potential values that variables and parameters might have at run time. Type checking is used in such languages as Ada, Algol, C, C++, Java, Modula-3, and Pascal. In *type inference*, the programmer need not write explicit type annotations; instead, the rules of the language itself determine the potential values of variables and parameters. Type inference is used in such languages as Haskell, Hope, Miranda, OCaml, and Standard ML. In both approaches, a language-dependent *type system* determines what programs are accepted for code generation or interpretation.

A type system is good for much more than rejecting programs that might commit run-time errors; a type system plays an important role in documentation. The name and type of a function often provide enough information so a programmer can figure out what the function is supposed to do. The more expressive the type system, the better a type serves as documentation.

Language designers have many choices about how to use type systems, type checking, and type inference. One of the most important applications of type systems is to help guarantee that all meaningless operations (adding non-numbers, and so on) are detected. A programming language in which all meaningless operations are detected and reported is called *safe*. Although safety can be guaranteed by checking every operation at run time, as in Impcore, μScheme, and full Scheme, a good static type system performs most checks at compile time. Here are some examples of the kinds of guarantees that a simple static type system can provide:

- That numbers are added only to numbers
- That every function receives the correct number of arguments
- That only Booleans are used in `if` expressions

- That `car` and `cdr` are applied only to lists

When these properties are guaranteed, there is no need for the implementations of addition, function call, and `if` to check their operands at run time. More importantly, the potential of a meaningless operation is reported to the programmer right away, before the program is shipped to its users.

To guarantee safety, a type system must be crafted such that if a program is accepted by the type system, the rules of the type system guarantee that no meaningless operations can be executed. Technically the guarantee is provided by a *type-soundness theorem*, an idea so popular that it has its own slogan: "well-typed programs don't go wrong." It's a wonderful slogan, but it's also a cheat: well-typed programs don't go wrong only if you limit yourself to a narrow idea of what it means to go wrong. There are ways to go wrong—meaningless operations—which are hard to rule out using a static type system:

- A number is divided by zero.
- The `car` or `cdr` primitive is applied to an *empty* list.
- A reference to an element of an array falls outside the bounds of that array.

While many language designers would love to have type systems that rule out these kinds of meaningless operations, the type systems that have been proposed are complex and difficult to understand. The problem is a subject of ongoing research, and it is a difficult one; if you need a PhD to understand error messages from your type checker, then your type system is probably not ready for deployment. In practice, the safety of a language is guaranteed by a *combination* of a simple static type system and some run-time checks for errors like division by zero or array access out of bounds.

So much for safety. Static typing is useful even for *unsafe* languages: types still serve as documentation, and a type system can prevent many run-time errors, even if a few slip through. The ones that slip through are called *unchecked* run-time errors; the potential of an unchecked run-time error is what makes a language unsafe.

Why would you want an "unsafe" programming language? Here are two common reasons:

- You want to write programs that manipulate memory directly, like a device driver or a garbage collector. People who build systems would like such programs to be safe, but how best to make them safe—say, by a combination of a sophisticated type system and a formal proof of correctness—is a topic of ongoing research.
- You want a relatively simple type system and you don't want to pay overhead for run-time checks. For example, the C programming language has a simple type system that detects many errors, but if you cast a pointer from one type to another, as almost all C programs do, you're on your own.

If you're designing an unsafe language, the best practice is to isolate unsafe features—something the designers of Modula-3 have done especially well. For safe languages, however, maintaining flexibility is a real concern. Programmers always complain about overly restrictive type systems, using slang terms like "strong typing[1]" or "bondage-and-discipline language." One of the most common and most galling complaints about such languages is that they make it hard to reuse code. Pascal was notorious for making it impossible to write a function capable of sorting arrays of different *lengths*, for example. Even languages

[1] For weak minds.

that are popular today, however, can suffer from overly restrictive type systems. For example, to avoid using unsafe features in the C code in Chapters 2 and 3, we have duplicated procedures for manipulating different kinds of lists. The problem of reuse can be solved by using a *polymorphic* type system. Polymorphic type systems provide abstractions that can be parameterized by types; C++ templates are an example of such an abstraction.

The program that a type system rules out aren't exactly the programs we hope for. The discussion above focuses on programs that are faulty but are not rejected by the type system. But there are also programs which are *not* faulty and *are* rejected by the type system. When such a program is rejected, there is no appeal, so this kind of problem can be especially galling. We can find plenty of examples if we look at dynamically typed languages. For example, in full Scheme, it is perfectly reasonable to have a list whose elements are a heterogenous mix of strings and numbers. But in most static type systems, every element of a list must have the same type. As another example, Lua modules are represented by records, which don't have static types (Ierusalimschy 2003), so it is easy and common to add new operations to a module at run time. Statically typed languages are useful, but they are not always better.

To design a statically typed language, you will need many ideas and techniques about types. We present some of the most essential ideas, the ones that every designer needs. This chapter describes type systems and type checking. Unlike most chapters in this book, it presents not one language but two: Typed Impcore and Typed μScheme. Typed Impcore is relatively straightforward; it models the type systems of such restrictive languages as Pascal and C. The purpose of Typed Impcore is to introduce type systems and to serve as a bad example. Typed μScheme is more ambitious; although its design starts from μScheme, it ends up requiring so many type annotations that the result has a very different feel from μScheme. The purpose of Typed μScheme is to introduce polymorphism and to serve as an eye-opening example.

The power of Typed μScheme may be eye-opening, but the need for explicit type annotations makes Typed μScheme an unpleasant language to program in. In the next chapter (Chapter 7), we move from Typed μScheme to μML. The key change in the design is to restrict the type system. The resulting *Hindley-Milner type system* provides most of the power of polymorphism while also supporting type inference. Under Hindley-Milner, a programmer can benefit from polymorphism without writing types explicitly. The Hindley-Milner type system is such a "sweet spot" that it forms the core of many of today's innovative, statically typed languages, including Standard ML, Haskell, OCaml, and many others.

6.1 Typed Impcore: a statically typed imperative core

From the language designer's point of view, the role of a type system is to find errors that would otherwise be hard to detect.[2] In Impcore, the language implementation has few opportunities to detect errors. All values are integers, so it is impossible to use "the wrong kind" of value in any given context. The only error we can detect automatically is a violation of rule APPLYUSER, which requires that the number of formal parameters must equal the number of actual parameters in a function call.

[2]The language implementor may view the type system in a different role. In most implementations of languages, the type of a value or variable determines how the value is represented and where a variable can be stored. For example, an integer is best stored in an integer register, whereas a string may not even fit in a register.

To make Typed Impcore interesting enough to study, we add the following features:

- Values may be *integers*, *Booleans*, or *arrays*.

  We use a single representation (the one from Chapter 2) for both integers and Booleans. In serious implementations of real languages, it is quite common for two or more high-level types (e.g., integer and Boolean) to share a single low-level representation (e.g., machine word). We use a new representation for arrays.

- We require that every variable and expression have a *type* that is known at compile time. The types of some variables, such as the formal parameters of functions, are written down explicitly, much as they are in C and Java. We present precise, formal rules that show whether an expression has a type and what the type is.

- We show how to turn the type rules into the implementation of a *type checker*. The type checker permits a definition to be evaluated only if all the expressions in that item have types. The type checker we present in this chapter covers only integers and Booleans; extending the implementation to include arrays is left as Exercise 1.

- We discuss *type soundness* (Exercise 3), which means informally that in *every* execution, in *every* context, when an expression produces a value, that value is consistent with the expression's type.

Type soundness means that if a program passes the type checker, it is guaranteed not to suffer from type errors at run time. As noted above, the slogan for this property is "well-typed programs don't go wrong." The slogan is bit misleading: well-typed programs can misbehave in all sorts of ways, including silently producing wrong answers, because of problems the type system cannot detect. A more accurate, less catchy slogan would be "well-typed programs don't suffer from run-time type errors."

We present this material in two stages. We begin with the integer and Boolean parts of Typed Impcore, and then in Section 6.3, we focus on arrays.

6.1.1 Concrete syntax of Typed Impcore

The primary difference between the concrete syntax of Typed Impcore and the concrete syntax of Impcore is that in Typed Impcore we must declare the argument and result types of functions.

```
def       ::= exp
            | (use file-name)
            | (val variable-name exp)
            | (define type function-name (formals) exp)
exp       ::= value
            | variable-name
            | (set variable-name exp)
            | (if exp exp exp)
            | (while exp exp)
            | (begin {exp})
            | (function {exp})
formals   ::= {(type variable-name)}
type      ::= int | bool | unit | (array type)
value     ::= integer
function  ::= function-name | primitive
primitive ::= + | - | * | / | = | < | > | print
```

The types make Typed Impcore's syntax more cumbersome than Impcore's syntax.

234a ⟨*transcript* 234a⟩≡ 234b▷

```
-> (define int add1 ((int x)) (+ x 1))
add1
-> (add1 4)
5
-> (define int double ((int x)) (+ x x))
double
-> (double 4)
8
```

In Typed Impcore, an `if` expression requires a Boolean condition:

234b ⟨*transcript* 234a⟩+≡ ◁234a 249a▷

```
-> (if 1 77 88)
type error: Condition in if expression has type int, which should be bool
-> (if (= 0 0) 77 99)
77
```

6.1.2 The initial basis of Typed Impcore

These functions do the same work at run time as the corresponding functions in Chapter 2, but their definitions are different because they include explicit types for arguments and results. Because Typed Impcore has no Boolean literals, we write `(= 1 0)` for falsehood and `(= 0 0)` for truth.

234c ⟨*additions to the Typed Impcore initial basis* 234c⟩≡ 235a▷

```
(define bool and ((bool b) (bool c)) (if b c b))
(define bool or  ((bool b) (bool c)) (if b b c))
(define bool not ((bool b))          (if b (= 1 0) (= 0 0)))
```

The comparison functions accept integers and return Booleans.

235a ⟨*additions to the Typed Impcore initial basis* 234c⟩+≡ ◁234c 235b▷
```
(define bool <= ((int x) (int y)) (not (> x y)))
(define bool >= ((int x) (int y)) (not (< x y)))
(define bool != ((int x) (int y)) (not (= x y)))
```

As in Chapter 2, we define modulus in terms of division.

235b ⟨*additions to the Typed Impcore initial basis* 234c⟩+≡ ◁235a
```
(define int mod ((int m) (int n)) (- m (* n (/ m n))))
```

6.1.3 Abstract syntax, types, and values of Typed Impcore

As in Chapter 5, we define the abstract syntax of Typed Impcore by presenting the representations we use in our implementation. The type system is so simple that we use `ty` not only as the abstract syntax for types but also as the internal representation of types. The type `funty` represents the type of a function in Typed Impcore; there is no corresponding abstract syntax.

235c ⟨*types for Typed Impcore* 235c⟩≡ (247d) 235d▷
```
datatype ty    = INTTY | BOOLTY | UNITTY | ARRAYTY of ty
datatype funty = FUNTY of ty list * ty
```

Any representation of type `funty` describes *one* type; for example, the comparison functions all have a type that says "two integer arguments and a Boolean result." A language in which a name has at most one type is called *monomorphic*.

Appendix F defines a function used to print types.

235d ⟨*types for Typed Impcore* 235c⟩+≡ (247d) ◁235c 235e▷
⟨*printing types for Typed Impcore* 677a⟩
```
typeString : ty -> string
```

We define two functions that compare types for equality.

235e ⟨*types for Typed Impcore* 235c⟩+≡ (247d) ◁235d
```
fun eqType (INTTY,       INTTY)  = true
  | eqType (BOOLTY,      BOOLTY) = true
  | eqType (UNITTY,      UNITTY) = true
  | eqType (ARRAYTY t1, ARRAYTY t2) = eqType (t1, t2)
  | eqType (_,           _)      = false
and eqTypes ([],      [])      = true
  | eqTypes (t1::ts1, t2::ts2) = eqType (t1, t2) andalso eqTypes (ts1, ts2)
  | eqTypes (_,       _)       = false
```
```
eqType  : ty      * ty      -> bool
eqTypes : ty list * ty list -> bool
```

There are two kinds of value: `NUM`, which represents both integers and Booleans, and `ARRAY`, which represents arrays.

235f ⟨*abstract syntax and values for Typed Impcore* 235f⟩≡ (247d) 236a▷
```
datatype value = NUM   of int
               | ARRAY of value array
```

not 234c

The abstract syntax of expressions is more complicated than what you would expect for Impcore. The reason is that in Typed Impcore, the primitives `=` and `print` cannot be represented as functions, because they operate on values of more than one type: they are *polymorphic*. In a monomorphic language like Typed Impcore, polymorphic primitives require their own abstract syntax. Similarly, as described in Section 6.3.2, array operations require special abstract syntax.

236a ⟨*abstract syntax and values for Typed Impcore* 235f⟩+≡ (247d) ◁235f 236b▷

```
datatype exp   = LITERAL of value
               | VAR     of name
               | SET     of name * exp
               | IFX     of exp * exp * exp
               | WHILEX  of exp * exp
               | BEGIN   of exp list
               | EQ      of exp * exp
               | PRINT   of exp
               | APPLY   of name * exp list
               ⟨array extensions to Typed Impcore's abstract syntax 249c⟩
```

These changes are typical for this kind of design: in a monomorphic language, each polymorphic operation requires special abstract syntax, as opposed to just a built-in function. In C, for example, it is not possible to write a general function that can dereference a pointer of any type; instead, one uses the special `*` syntax.

In Typed Impcore, the abstract syntax for a function definition requires that the result type and the types of the formal parameters be identified explicitly. The concrete syntax puts types on the left, by analogy with C. The abstract syntax puts types on the right, as is customary in formal semantics and in Pascal-like and ML-like languages.

236b ⟨*abstract syntax and values for Typed Impcore* 235f⟩+≡ (247d) ◁236a 236c▷

```
type userfun = { formals : (name * ty) list, body : exp, returns : ty }
datatype def = VAL    of name * exp
             | EXP    of exp
             | DEFINE of name * userfun
             | USE    of name
```

In Typed Impcore, the function name in an application is not an ordinary value, so it can't stand for something of type `value` in our implementation. In Chapter 2, we represent Impcore functions using the C type `Fun`, but `fun` is a reserved word in ML, so we use the name `func` instead. As in our interpreter for μScheme, a primitive function is represented as an ML function.

236c ⟨*abstract syntax and values for Typed Impcore* 235f⟩+≡ (247d) ◁236b 236d▷

```
datatype func = USERDEF   of string list * exp
              | PRIMITIVE of value list -> value
```

type name 214
type ty 235c
type value 235f

There are no types in the representation of `func`; types are needed only during type checking, and the `func` representation is used at run time, after all types have been checked.

Just like types, values can be printed. The code appears in Appendix F.

236d ⟨*abstract syntax and values for Typed Impcore* 235f⟩+≡ (247d) ◁236c 250a▷

⟨*printing values for Typed Impcore* 677b⟩

```
valueString : value -> string
```

6.1.4 Type system for Typed Impcore

We've designed Typed Impcore's static type system to be somewhat more restrictive than Impcore's dynamic type system. For example, it is not legal to use an integer to control an `if` expression, because the type system distinguishes integers from Booleans. This restriction should not burden a programmer overmuch; if you have an integer `i` that you wish to treat as a Boolean, you can simply write `(!= i 0)`. Similarly, if you have a Boolean `b` that you wish to treat as an integer, you can write `(if b 1 0)`. Another restriction is that you cannot assign the result of a `while` loop to an integer variable. The motivation is that it seems like a strange thing to do, because a `while` loop produces no interesting result.

Typed Impcore's type system, like any type system, determines which expressions (also called *terms*) have types. The implementation accepts a definition only if its subterms have types; otherwise, the implementation rejects it. Just as we use a formal proof system to specify precisely which terms have values, we use another formal proof system to specify precisely which terms have types. Before getting into the details of the proof rules, however, we discuss the elements of the system.

The fundamental elements of the Typed Impcore type system are *simple types*, for which we write τ; three *basic types*, for which we write INT, BOOL, and UNIT; one *type constructor*, for which we write ARRAY; many *function types*, for which we write τ_f; and three *type environments*, for which we write Γ_ξ, Γ_ϕ, and Γ_ρ, and which give the types of global variables, functions, and formal parameters respectively.

A simple type is a basic type or the array constructor applied to a simple type.[3] Because each type in Typed Impcore requires special abstract syntax, we write the names using the SMALL CAPS font, which we conventionally use for abstract syntax.

$$\tau \quad ::= \text{INT} \mid \text{BOOL} \mid \text{UNIT} \mid \text{ARRAY}(\tau)$$

The type of a function has some arguments and a result.

$$\tau_f \quad ::= \tau_1 \times \cdots \times \tau_n \to \tau$$

Just as functions are not values, function types are not value types.

The purpose of the integer, Boolean, and array types should be obvious: they describe values that represent integers, Booleans, and arrays. The purpose of the unit type is not so obvious; a flippant way to explain it is that the purpose of the unit type is to be different from all the others. The problem that unit solves is this: what type should we give to the result of a while loop or print expression? These expressions are executed purely for their side effects; they don't produce interesting values. Most typed languages reserve a special type to represent the result of an operation that is executed only for side effects. In C, C++, and Java this type is called `void`, because the type is empty, i.e., there are no values of type `void`. In the functional language ML it is called type `unit`, because the `unit` type has exactly one value. (The value is not interesting because every expression of type `unit` is guaranteed to produce exactly that single value, unless it fails to terminate or it raises an exception.) The `unit` type is the more appropriate choice for Typed Impcore, since Typed Impcore is an expression-oriented language in which every evaluation that terminates produces a value.

[3] In Typed Impcore, the type of an array does not include the array's size.

6.1.5 Type rules for Typed Impcore

We write a type system using the same kind of formal rules we use to write operational semantics; only the forms of the judgments are different. The judgments in the operational semantics enable us to determine when an expression will be evaluated to produce a value. The judgments in the type system enable us to determine when an expression has a type. Exercise 3 explores the relationships between the judgments of the type system and the judgments of the operational semantics.

Type rules for expressions

The typing judgment for expressions is $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$, meaning that given type environments $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$, expression e has type τ. The rules of Typed Impcore's type system make it possible to conclude judgments of this form. The typing rules of Typed Impcore are deterministic, so every expression in a well-typed Typed Impcore program has exactly one type. In other words, given the environments $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ and the abstract-syntax tree e, there is at most one τ such that $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$. (For a proof of uniqueness see Exercise 4.) To find this τ, or to report an error if no such τ exists, is the job of a *type checker*.

Type rules for expressions

The type rule for literals is that all literals are integers. This rule is sound because there are no Boolean literals in Typed Impcore: the parser creates only integer literals.

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{literal}(v) : \textsc{int}} \qquad (\textsc{Literal})$$

The use of a variable is well typed if the variable is bound in the environment. As in Chapter 2, formals hide globals.

$$\frac{x \in \operatorname{dom} \Gamma_\rho}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{var}(x) : \Gamma_\rho(x)} \qquad (\textsc{FormalVar})$$

$$\frac{x \notin \operatorname{dom} \Gamma_\rho \qquad x \in \operatorname{dom} \Gamma_\xi}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{var}(x) : \Gamma_\xi(x)} \qquad (\textsc{GlobalVar})$$

An assignment is well typed if the type of the right-hand side matches the type of the variable being assigned to. As is our unhappy lot when working with Impcore, we must look for the variable in two environments.

$$\frac{x \in \operatorname{dom} \Gamma_\rho \qquad \Gamma_\rho(x) = \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{set}(x, e) : \tau} \qquad (\textsc{FormalAssign})$$

$$\frac{x \notin \operatorname{dom} \Gamma_\rho \qquad x \in \operatorname{dom} \Gamma_\xi \qquad \Gamma_\xi(x) = \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{set}(x, e) : \tau} \qquad (\textsc{GlobalAssign})$$

If the assignment is well typed, the type of the assignment is the type of the variable and the value assigned to it. We could have chosen to give the assignment expression type `unit`, but this choice would have ruled out such expressions as `(set x (set y 0))`. The rule we have chosen, which is copied from C, makes the language more expressive.

A conditional expression is well typed if the condition is Boolean and the two arms have the same type. In that case, the type of the conditional expression is the type of the arms.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{IF}(e_1, e_2, e_3) : \tau} \tag{IF}$$

The IF rule is the first rule that is different from the corresponding rules in the operational semantics. In the operational semantics, we have two rules: one corresponds to $e_1 \Downarrow \text{BOOL}(\texttt{\#t})$ and evaluates e_2, and one corresponds to $e_1 \Downarrow \text{BOOL}(\texttt{\#f})$ and evaluates e_3. The type system does not care about the *value* of e_1; it cares only about the *type* of e_1. The type system also checks the types of both e_2 and e_3 at one go.

A while loop is well typed if the condition is Boolean and the body is well typed. The τ that is the type of the body e_2 is not used in the conclusion of the rule, because we care only that the type τ exists, not what type it is.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{WHILE}(e_1, e_2) : \text{UNIT}} \tag{WHILE}$$

As explained above, a while loop does not produce a useful result, so we give it the unit type.

In the `while` expression, as in the `if` expression, we see a difference between the type system and the operational semantics. The type system does not need different rules for the cases of terminating and nonterminating loops. The structure of this WHILE rule ensures that during type checking every subexpression is type-checked exactly once, even though during *evaluation* every subexpression may well be executed a different, even unbounded number of times, For most reasonable type systems, then, type checking takes time linear in the size of the program. In practice, type checking is very fast indeed.

Because sequential composition is used to execute expressions for their side effects, we would be justified in requiring all expressions except the last to have type UNIT. Because it seems more flexible to allow expressions in a sequence to have any type, we do so.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_n : \tau_n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n} \tag{BEGIN}$$

The premises mentioning $e_1, \ldots, e_{n-1}$ are necessary because although we don't care what the types of $e_1, \ldots, e_{n-1}$ are, they still have to be well typed.

We give the empty BEGIN type UNIT.

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}() : \text{UNIT}} \tag{EMPTYBEGIN}$$

As in standard, untyped Impcore, there are no function values, only the names of functions. The functions have function types, which can be looked up in the function-type environment Γ_ϕ. The types of the arguments must match the types of the formal parameters. The type of the application is the result type of the function.

$$\frac{\Gamma_\phi(f) = \tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_i : \tau_i, \quad 1 \le i \le n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(f, e_1, \ldots, e_n) : \tau} \tag{APPLY}$$

As described so far, Typed Impcore is almost good enough to be a reasonable model for such languages as C and Pascal. Where the model breaks down is in *polymorphic operators*, i.e., operators that can be applied to more than one type. Most imperative languages have a handful of such operators, and Typed Impcore is no exception: its overloaded operator is equality. Luckily, equality is a fairly easy operator to specify; we want to be able to check any two values of the same type for equality.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(=, e_1, e_2) : \text{BOOL}} \quad \text{(APPLYEQ)}$$

The other polymorphic primitive is `print`, which can be applied to a value of any type.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(\texttt{print}, e) : \text{UNIT}} \quad \text{(APPLYPRINT)}$$

Type rules for definitions

By analogy with the operational semantics, the type rule for a definition may produce a new type environment. The judgment has the form $\langle t, \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma'_\phi \rangle$, which says that when item t is elaborated given type environments Γ_ξ and Γ_ϕ, the new environments are Γ'_ξ and Γ'_ϕ.

A VAL binding requires that the newly bound identifier take the type of its right-hand side.

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi\{x \mapsto \tau\}, \Gamma_\phi \rangle} \quad \text{(VAL)}$$

There are no constraints on x; x may be undefined before the VAL binding, or x may have an arbitrary type.

A top-level expression is syntactic sugar for a binding to `it`.

$$\frac{\langle \text{VAL}(\texttt{it}, e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma_\phi \rangle}{\langle \text{EXP}(e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma_\phi \rangle} \quad \text{(EXP)}$$

The rule for DEFINE updates the function environment. We require that whenever arguments $x_1, \ldots, x_n$ have the types $\tau_1, \ldots, \tau_n$ given in the function definition, the function's body e must be well typed and have type τ. In that case we give the function type $\tau_1 \times \cdots \times \tau_n \to \tau$. Because functions can be recursive, we put f into the type environment Γ_ϕ when type-checking e.

$$\frac{\begin{array}{c}\tau_1, \ldots, \tau_n \text{ are types} \\ \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \cdots \times \tau_n \to \tau\}, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau\end{array}}{\langle \text{DEFINE}(f, (\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e : \tau)), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \cdots \times \tau_n \to \tau\} \rangle} \quad \text{(DEFINE)}$$

To know that $\tau_1, \ldots, \tau_n$ are types, we need a type rule.

$$\frac{\tau \in \{\text{UNIT}, \text{INT}, \text{BOOL}\}}{\tau \text{ is a type}} \quad \text{(BASETYPES)}$$

6.2 A type-checking interpreter for Typed Impcore

Most of our interpreter for Typed Impcore is just what you would expect if you've looked at Chapter 5's interpreter for μScheme. There are a few differences because Typed Impcore is a typed language.

- In addition to abstract syntax for expressions and definitions, we also have abstract syntax for types, as shown in chunk ⟨*types for Typed Impcore* 235c⟩. Because the type system is so simple, the abstract syntax also serves as our internal representation of types.
- To ensure that every expression in a Typed Impcore program is well typed, we have added a type checker.
- We have added type checking to the top-level read-eval-print loop. It is now more properly called a read-check-eval-print loop.

Our type checker handles only integers, Booleans, and the unit type; array types are left for the Exercises.

6.2.1 Type checking

Given an expression e and a collection of type environments Γ_ξ, Γ_ϕ, and Γ_ρ, calling `typeof`$(e, \Gamma_\xi, \Gamma_\phi, \Gamma_\rho)$ returns a type τ such that $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$. If no such type exists, `typeof` raises the `TypeError` exception (or possibly `NotFound`). We use an auxiliary, nested function `ty`, which doesn't pass environments explicitly.

241a ⟨*type checking for Typed Impcore* 241a⟩≡ (247d) 244d▷

```
typeof  : exp * ty env * funty env * ty env -> ty
ty      : exp -> ty
```

```
exception TypeError of string
fun typeof (e, globals, functions, formals) =
  let ⟨function ty, to check the type of an expression, given Γξ, Γφ, Γρ 241b⟩
  in  ty e
  end
```

Just as we can derive an implementation of `eval` by examining the rules of operational semantics, we can derive an implementation of `ty` by examining the rules of the type system. To help show the connection between the type system and the type checker, we show the relevant rules before each case of the function `ty`.

All literals are integers.

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LITERAL}(v) : \text{INT}} \quad \text{(LITERAL)}$$

241b ⟨*function* ty, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩≡ (241a) 241c▷

```
fun ty (LITERAL v) = INTTY
```

To type a variable, we must try two environments.

$$\frac{x \in \operatorname{dom} \Gamma_\rho}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\rho(x)} \quad \text{(FORMALVAR)}$$

$$\frac{x \notin \operatorname{dom} \Gamma_\rho \qquad x \in \operatorname{dom} \Gamma_\xi}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\xi(x)} \quad \text{(GLOBALVAR)}$$

The code is shorter than the rules!

241c ⟨*function* ty, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩+≡ (241a) ◁241b 242a▷

```
    | ty (VAR x)     = (find (x, formals) handle NotFound _ => find (x, globals))
```

If the variable is not found in either Γ_ρ or Γ_ξ, the type checker raises the `NotFound` exception.

We also need two environments to check assignments.

$$\frac{x \in \mathrm{dom}\,\Gamma_\rho \qquad \Gamma_\rho(x) = \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{set}(x, e) : \tau} \qquad \text{(FormalAssign)}$$

$$\frac{x \notin \mathrm{dom}\,\Gamma_\rho \qquad x \in \mathrm{dom}\,\Gamma_\xi \qquad \Gamma_\xi(x) = \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{set}(x, e) : \tau} \qquad \text{(GlobalAssign)}$$

To implement these rules, we determine the types of the variable and the expression, then compare for equality. (The recursive call `ty (VAR x)` is a short cut that avoids duplicating code.) Giving an error message that is even remotely helpful requires as much code as checking the types.

242a ⟨*function* `ty`, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩+≡ (241a) ◁241c 242b▷

```
| ty (SET (x, e)) =
      let val tau_x = ty (VAR x)
          val tau_e = ty e
      in  if eqType (tau_x, tau_e) then
            tau_x
          else
            raise TypeError ("Set variable " ^ x ^ " of type " ^ typeString tau_x ^
                             " to value of type " ^ typeString tau_e)
      end
```

The premises of the IF rule require both branches to have the same type.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \textsc{bool} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{if}(e_1, e_2, e_3) : \tau} \qquad \text{(If)}$$

The implementation computes the types of the conditions and both branches. Again, most of the code is devoted to error messages.

242b ⟨*function* `ty`, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩+≡ (241a) ◁242a 243a▷

```
| ty (IFX (e1, e2, e3)) =
    let val tau1 = ty e1
        val tau2 = ty e2
        val tau3 = ty e3
    in  if eqType (tau1, BOOLTY) then
          if eqType (tau2, tau3) then
            tau2
          else
            raise TypeError ("In if expression, true branch has type " ^
                             typeString tau2 ^ " but false branch has type " ^
                             typeString tau3)
        else
          raise TypeError ("Condition in if expression has type " ^ typeString tau1 ^
                           ", which should be " ^ typeString BOOLTY)
    end
```


BOOLTY 235c
eqType 235e
IFX 236a
SET 236a
ty 241b
TypeError 241a
typeString 677a
VAR 236a

The checking of a `while` loop is very similar to that of an `if`, except that we don't care what the type of the body is.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{WHILE}(e_1, e_2) : \text{UNIT}} \quad \text{(WHILE)}$$

243a ⟨*function* ty, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩+≡ (241a) ◁242b 243b▷

```
| ty (WHILEX (e1, e2)) =
    let val tau1 = ty e1
        val tau2 = ty e2
    in  if eqType (tau1, BOOLTY) then
          UNITTY
        else
          raise TypeError ("Condition in while expression has type " ^
                           typeString tau1 ^ ", which should be " ^
                           typeString BOOLTY)
    end
```

The code for `begin` checks the types of all sub-expressions.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_n : \tau_n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n} \quad \text{(BEGIN)}$$

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}() : \text{UNIT}} \quad \text{(EMPTYBEGIN)}$$

We handle the EMPTYBEGIN rule by using `UNITTY` as the initial value of `tau` that is passed to `last`.

243b ⟨*function* ty, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩+≡ (241a) ◁243a 243c▷

```
| ty (BEGIN es) =
    let val bodytypes = map ty es
        fun last tau []     = tau
          | last tau (h::t) = last h t
    in  last UNITTY bodytypes
    end
```

Because the two polymorphic primitives have special rules, they require special code in the type checker.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(=, e_1, e_2) : \text{BOOL}} \quad \text{(APPLYEQ)}$$

Because functions in Typed Impcore are called by name, we can identify the equality primitive by name.

243c ⟨*function* ty, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩+≡ (241a) ◁243b 244a▷

```
| ty (EQ (e1, e2)) =
    let val (tau1, tau2) = (ty e1, ty e2)
    in  if eqType (tau1, tau2) then
          BOOLTY
        else
          raise TypeError ("Equality compares values of different types " ^
                           typeString tau1 ^ " and " ^ typeString tau2)
    end
```


BEGIN 236a
BOOLTY 235c
EQ 236a
eqType 235e
ty 241b
TypeError 241a
typeString 677a
UNITTY 235c
WHILEX 236a

The `print` primitive is similar.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{apply}(\texttt{print}, e) : \textsc{unit}} \qquad (\textsc{ApplyPrint})$$

We must check the type of the argument even though it doesn't affect the type of the result.

244a ⟨*function* ty, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩+≡ (241a) ◁243c 244b▷

```
| ty (PRINT e) = (ty e; UNITTY)
```

For the general case of function application, we look up the function in the appropriate environment. To issue a decent error message, we compare the types of the actual and formal parameters one by one.

$$\frac{\Gamma_\phi(f) = \tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_i : \tau_i, \quad 1 \le i \le n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{apply}(f, e_1, \ldots, e_n) : \tau} \qquad (\textsc{Apply})$$

244b ⟨*function* ty, *to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ 241b⟩+≡ (241a) ◁244a

```
parameterError : int * ty list * ty list -> 'a
```

```
| ty (APPLY (f, actuals)) =
    let val actualtypes                    = map ty actuals
        val FUNTY (formaltypes, resulttype) = find (f, functions)
        ⟨definition of parameterError 244c⟩
    in  if eqTypes (actualtypes, formaltypes) then
          resulttype
        else
          parameterError (1, actualtypes, formaltypes)
    end
```

244c ⟨*definition of* parameterError 244c⟩≡ (244b)

```
fun parameterError (n, atau::actuals, ftau::formals) =
      if eqType (atau, ftau) then
        parameterError (n+1, actuals, formals)
      else
        raise TypeError ("In call to " ^ f ^ ", parameter " ^
                         Int.toString n ^ " has type " ^ typeString atau ^
                         " where type " ^ typeString ftau ^ " is expected")
  | parameterError _ =
      raise TypeError ("Function " ^ f ^ " expects " ^
                       Int.toString (length formaltypes) ^ " parameters " ^
                       "but got " ^ Int.toString (length actualtypes))
```

APPLY	236a
eqType	235e
eqTypes	235e
find	214
functions	241a
FUNTY	235c
PRINT	236a
ty	241b
TypeError	241a
typeString	677a
UNITTY	235c

6.2.2 Type-checking definitions

The form of the typing judgment for a definition d is $\langle d, \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\phi, \Gamma'_\xi \rangle$. The process of type-checking a definition and extending the type environments is called *elaboration*.

244d ⟨*type checking for Typed Impcore* 241a⟩+≡ (247d) ◁241a

```
elabdef : def * ty env * funty env -> ty env * funty env
```

```
exception RuntimeError of string
fun elabdef (d, globals, functions) =
    case d
      of ⟨cases for elaborating definitions in Typed Impcore 245a⟩
```

Elaborating a `val` binding requires extending the global-variable environment.

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau}{\langle \textsc{val}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi\{x \mapsto \tau\}, \Gamma_\phi \rangle} \quad (\textsc{Val})$$

245a ⟨*cases for elaborating definitions in Typed Impcore* 245a⟩≡ (244d) 245b▷

```
VAL (x, e) => (bind (x, typeof (e, globals, functions, emptyEnv), globals), functions)
```

A top-level expression is syntactic sugar for a definition of `it`.

$$\frac{\langle \textsc{val}(\texttt{it}, e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma_\phi \rangle}{\langle \textsc{exp}(e), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma_\phi \rangle} \quad (\textsc{Exp})$$

245b ⟨*cases for elaborating definitions in Typed Impcore* 245a⟩+≡ (244d) ◁245a 245c▷

```
| EXP e => elabdef (VAL ("it", e), globals, functions)
```

A function definition requires significant manipulation of the function environment.

$$\frac{\begin{array}{c}\tau_1, \ldots, \tau_n \text{ are types} \\ \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \cdots \times \tau_n \to \tau\}, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau\end{array}}{\langle \textsc{define}(f, (\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e : \tau)), \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \cdots \times \tau_n \to \tau\} \rangle} \quad (\textsc{Define})$$

245c ⟨*cases for elaborating definitions in Typed Impcore* 245a⟩+≡ (244d) ◁245b 245d▷

```
| DEFINE (name, {returns, formals, body}) =>
    let val (fnames, ftys) = ListPair.unzip formals
        val functions' = bind (name, FUNTY (ftys, returns), functions)
        val tau = typeof (body, globals, functions', bindList (fnames, ftys, emptyEnv))
    in  if eqType (tau, returns) then
          (globals, functions')
        else
          raise TypeError ("Body of function has type " ^ typeString tau ^
                           ", which does not match declaration of " ^
                           typeString returns)
    end
```

A `use` directive should never get this far.

245d ⟨*cases for elaborating definitions in Typed Impcore* 245a⟩+≡ (244d) ◁245c

```
| USE _ => raise BugInTypeChecking "'use' reached the type checker"
```

6.2.3 Top-level processing: type checking and evaluation

In an interpreter for a dynamically typed language such as Impcore or μScheme, we can evaluate a top-level item as soon as it is parsed. In an interpreter for a statically typed language such as Typed Impcore, we require a type-checking step between parsing and evaluation. A simple way to introduce this step is to define a function `checkThenEval` which first elaborates a definition (type-checking all its subexpressions), then evaluates it. This function manipulates not only the top-level type environments Γ_ϕ and Γ_ξ but also the top-level value and function environments ϕ and ξ. We define type `env_bundle` to refer to this group of four environments. The value environment ξ is the only one that can be mutated during evaluation, so it is the only one that has a `ref` in its type.

246a ⟨*checking and evaluation for Typed Impcore* 246a⟩≡ (247d) 246b▷

```
checkThenEval : def * env_bundle * (string->unit) -> env_bundle
```

⟨*evaluation for Typed Impcore* 681a⟩

```
type env_bundle = ty env * funty env * value ref env * func env
fun checkThenEval (d, envs as (tglobals, tfuns, vglobals, vfuns), echo) =
  case d
    of USE filename => use readCheckEvalPrint timpcoreSyntax filename envs
     | _ =>
       let val (tglobals, tfuns) = elabdef (d, tglobals, tfuns)
           val (vglobals, vfuns) = evaldef (d, vglobals, vfuns, echo)
       in  (tglobals, tfuns, vglobals, vfuns)
       end
```

The distinction between "compile time," where we run the type checker `elabdef`, and "run time," where we run the evaluator `evaldef`, is sometimes called the *phase distinction*. The phase distinction is easy to overlook, especially when you're using an interactive interpreter or compiler, but the code shows the phase distinction is real.

The definition of the evaluation function `evaldef` appears in Appendix F.

6.2.4 The read-eval-print loop

As in μScheme, the read-eval-print loop is higher-order. There are two differences: the environment parameter's name is `envs` instead of `rho`, and we have handlers for new exceptions: `TypeError` and `BugInTypeChecking`.

⟨*checking and evaluation for Typed Impcore* 246a⟩+≡ (247d) ◁246a

```
heckEvalPrint : def stream * (string->unit) * (string->unit) -> env_bundle -> env_bundle
```

```
and readCheckEvalPrint (defs, echo, errmsg) envs =
  let fun processDef (def, envs) =
            let fun continue msg = (errmsg msg; envs)
            in  checkThenEval (def, envs, echo)
                handle IO.Io {name, ...} => continue ("I/O error: " ^ name)
                ⟨more read-eval-print handlers 247a⟩
            end
  in  streamFold processDef envs defs
  end
```


elabdef 244d
type env 214
evaldef 681b
type func 236c
type funty 235c
streamFold 650a
timpcoreSyntax 679b
type ty 235c
USE 236b
use 222a
type value 235f

The ⟨*more read-eval-print handlers* 247a⟩ are the same handlers used in μScheme. We also have handlers for two new exceptions. `TypeError` is raised not at parsing time, and not at evaluation time, but at type-checking time. `BugInTypeChecking` should never be raised.

247a ⟨*more read-eval-print handlers* 247a⟩≡ (246b 222b)
```
| TypeError         msg => continue ("type error: " ^ msg)
| BugInTypeChecking msg => continue ("bug in type checking: " ^ msg)
```

6.2.5 Initializing the interpreter

We're ready to put everything together into a working interpreter.

247b ⟨*initialization for Typed Impcore* 247b⟩≡ (247d) 247c▷
```
val initialEnvs =
  let fun addPrim ((name, prim, funty), (tfuns, vfuns)) =
        ( bind (name, funty, tfuns)
        , bind (name, PRIMITIVE prim, vfuns)
        )
      val (tfuns, vfuns)  = foldl addPrim (emptyEnv, emptyEnv)
                            (⟨primops for Typed Impcore :: 681f⟩ nil)
      val envs  = (emptyEnv, tfuns, emptyEnv, vfuns)
      val basis = ⟨ML representation of initial basis (automatically generated)⟩
      val defs  = reader timpcoreSyntax noPrompts ("initial basis", streamOfList basis)
  in  readCheckEvalPrint (defs, fn _ => (), fn _ => ()) envs
  end
```

The code for the primitives appears in Appendix F. It is similar to the code in Chapter 5, except that it supplies a type, not just a value, for each primitive.

The function `runInterpreter` takes one argument, which tells it whether to prompt.

247c ⟨*initialization for Typed Impcore* 247b⟩+≡ (247d) ◁247b
```
fun runInterpreter noisy =
  let fun writeln s = app print [s, "\n"]
      fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
      val prompts = if noisy then stdPrompts else noPrompts
      val defs =
        reader timpcoreSyntax prompts ("standard input", streamOfLines TextIO.stdIn)
  in  ignore (readCheckEvalPrint (defs, writeln, errorln) initialEnvs)
  end
```

6.2.6 Putting all the pieces together

We stitch together the parts of the implementation in this order:

247d ⟨*timpcore.sml* 247d⟩≡
```
⟨environments 214⟩
exception LeftAsExercise of string
⟨types for Typed Impcore 235c⟩
⟨lexical analysis 671a⟩
⟨abstract syntax and values for Typed Impcore 235f⟩
⟨parsing for Typed Impcore 678⟩
⟨implementation of use 222a⟩
⟨type checking for Typed Impcore 241a⟩
⟨checking and evaluation for Typed Impcore 246a⟩
⟨initialization for Typed Impcore 247b⟩
⟨command line 223d⟩
```

6.3 Extending Typed Impcore with arrays

Type systems are intimately connected with data structures, and a new data structure often requires a new type. As an example of how to use types in a language design, we extend Typed Impcore by adding arrays. Arrays are found in almost every programming language and are frequently used data structures in such languages as Fortran, Java, and Pascal. This section presents types, concrete syntax, abstract syntax, evaluation code, and type rules for arrays.

6.3.1 Types for arrays

The integer, Boolean, and UNIT types are useful but limited; programmers can't make up new representations out of these types alone. Arrays, by contrast, are unlimited; there is literally no limit to the number of types we can make with arrays. Arrays of integers, arrays of Booleans, and arrays of arrays of integers are just a few of the possibilities. Properly speaking, "array" is not a "type" at all—it is a *type constructor*, i.e., a thing you use to build types. Given any type τ, you can make an array of τ, and you have constructed a new type. In our abstract syntax, the array type constructor is represented by ARRAY (in the ML code, `ARRAYTY`, as shown in chunk 235c). In the concrete syntax, it is represented by `(array` *type*`)`, as shown on page 234. So for example, `(array int)` is the type of arrays of integers, and `(array (array bool))` is the type of two-dimensional arrays (matrices) of Booleans.

ARRAY is not the only type constructor in Typed Impcore. Even INT, UNIT, and BOOL can be viewed as type constructors, although they are boring type constructors because unlike ARRAY they can be used to build only one type apiece. Such type constructors are usually called *base types*, because they form the bases from which all other types in the language are built.

Whenever you add a new type to a language, whether it is a base type or a more interesting type constructor, you also add operations for values of that type. We now discuss operations for arrays.

6.3.2 Operations for arrays

To support arrays in Typed Impcore we need to be able to make an array, get at an array's elements, and find out how many elements an array has. We provide four operations:

`(array-make` $\mathtt{e}_1$ $\mathtt{e}_2$`)` — Evaluate $\mathtt{e}_1$ to get an integer i and $\mathtt{e}_2$ to get a value v. Return a new array containing i elements, each initialized to v. An `array-make` expression is analogous to a creator (Section 3.4.2) for the array type.

`(array-get` $\mathtt{e}_1$ $\mathtt{e}_2$`)` — Evaluate $\mathtt{e}_1$ to get an array a and $\mathtt{e}_2$ to get an integer i. Return the ith element of a, where the first element is element number 0. An `array-get` expression is analogous to an observer for the array type.

`(array-set` $\mathtt{e}_1$ $\mathtt{e}_2$ $\mathtt{e}_3$`)` — Evaluate $\mathtt{e}_1$ to get an array a, $\mathtt{e}_2$ to get an integer i, and $\mathtt{e}_3$ to get a value v. Modify a by storing v as its ith element. An `array-set` expression is analogous to a mutator for the array type.

`(array-length e)` — Evaluate `e` to get an array a. Return the number of elements in a. An `array-length` expression is analogous to an observer for the array type.

Here's an example in which we create and initialize a matrix, which in Typed Impcore is represented as an array of arrays.

249a ⟨*transcript* 234a⟩+≡ ◁234b 249b▷

```
-> (define (array (array int)) matrix-helper
     ; return square matrix of side length; a and i are for local use only
     ((int length) ((array (array int)) a) (int i))
    (begin
      (set a (array-make length (array-make 0 0)))
      (set i 0)
      (while (< i length)
         (begin
           (array-set a i (array-make length 0))
           (set i (+ i 1))))
      a))
-> (define (array (array int)) matrix ((int length))
     (matrix-helper length (array-make 0 (array-make 0 0)) 0))
```

249b ⟨*transcript* 234a⟩+≡ ◁249a

```
-> (val a (matrix 3))
[[0 0 0] [0 0 0] [0 0 0]]
-> (val i 0)
-> (val j 0)
-> (while (< i 3) (begin
     (set j 0)
     (while (< j 3) (begin
        (array-set (array-get a i) j (+ i j))
        (set j (+ j 1))))
     (set i (+ i 1))))
-> a
[[0 1 2] [1 2 3] [2 3 4]]
-> (val a.1 (array-get a 1))
[1 2 3]
-> (val a.1.1 (array-get a.1 1))
2
```

As required by the specification, the `array-get` operation is quite flexible. In the last example, the `array-get` operation takes an argument of type `(array int)` and returns a result of type `int`, but in the example before that, it takes an argument of type `(array (array int))` and returns a result of type `(array int)`. No mere function in Typed Impcore can do such things; this behavior is *polymorphic*. Typed Impcore is *monomorphic*, which means that each function can be used for arguments and results of one and only one type. This means we can't just put `array-get` into the initial basis for Typed Impcore; we have to make it a special operation in the abstract syntax. In fact, *all* of the array operations are polymorphic, so we have to add these expressions to the abstract syntax:

type exp 236a

249c ⟨*array extensions to Typed Impcore's abstract syntax* 249c⟩≡ (236a)

```
| AGET  of exp * exp
| ASET  of exp * exp * exp
| AMAKE of exp * exp
| ALEN  of exp
```

This example illustrates a general principle: *in a monomorphic language, polymorphic primitives require special abstract syntax.* This principle also applies to C and C++, for example, which denote array operations with special syntax involving square brackets.

Since the array operations are not ordinary functions but require special abstract syntax, we need rules that tell us how to type-check and evaluate that syntax. The type rules are covered in the next section. The evaluation *rules* are not particularly interesting, so they are omitted from this book, but the evaluation *code* is interesting enough to include here.

To interpret any array operation, we need to convert at least one argument from a `value` to an array or to an integer. If the program type checks, the conversion should always succeed; if a conversion fails, there is a bug in the type checker. The `toArray` and `toInt` functions do the job.

250a ⟨*abstract syntax and values for Typed Impcore* 235f⟩+≡ (247d) ◁236d

```
toArray : value -> value array
toInt   : value -> int
```

```
exception BugInTypeChecking of string
fun toArray (ARRAY a) = a
  | toArray _         = raise BugInTypeChecking "non-array value"
fun toInt   (NUM n)   = n
  | toInt _           = raise BugInTypeChecking "non-integer value"
```

Once we have `toArray` and `toInt`, interpreting the array operations is easy. Everything we need is in the `Array` module from ML's Standard Basis Library. The library includes run-time checks for bad subscripts or array sizes; we need these checks because Typed Impcore's type system is not powerful enough to preclude such errors.

250b ⟨*more alternatives for* `ev` *for Typed Impcore* 250b⟩≡ (680a)

```
ev : exp -> value
```

```
| ev (AGET (a, i)) =
    (Array.sub (toArray (ev a), toInt (ev i))
     handle Subscript => raise RuntimeError "Array subscript out of bounds")
| ev (ASET (e1, e2, e3)) =
    (let val (a, i, v) = (ev e1, ev e2, ev e3)
     in  Array.update (toArray a, toInt i, v);
         v
     end handle Subscript => raise RuntimeError "Array subscript out of bounds")
| ev (AMAKE (len, init)) =
    (ARRAY (Array.array (toInt (ev len), ev init))
     handle Size => raise RuntimeError "array length too large (or negative)")
| ev (ALEN a) = NUM (Array.length (toArray (ev a)))
```

When we build the interpreter from this book, this code becomes part of the evaluator in Appendix F.

6.3.3 Type rules for arrays

If you're a language designer, adding a new base type can be useful (think of strings, I/O streams, and floating-point numbers), but it's not very challenging intellectually—you add the new type, you add a bunch of operations to support it, and you're done. Adding a new type constructor is a lot more interesting because you get to add not only new operations but also new abstract syntax and new type rules.

When you add abstract syntax, the classification of operations in Section 3.4.2 applies. Abstract syntax should include creators and observers, as well as producers, mutators, or both. When you add rules, what should you include? First, you should say how to use the new type constructor to make new types. Rules that say how to do this are called *formation rules.* Second, you should say what expressions *create new values* that are described by the new type constructor. Rules that say how to do this are called *introduction rules.* An expression that appears in an introduction rule is analogous to a *creator* function as described in Section 3.4.2. Finally, you should say what expressions *use the values* that are described by the new type constructor. Rules that say how to do this are called *elimination rules.* An expression that appears in an elimination rule may be analogous to a mutator or observer function as described in Section 3.4.2.

Here is how it works for arrays. In Typed Impcore, you make an array type by supplying the type of the elements. The length of the array is *not* part of its type.[4]

$$\frac{\tau \text{ is a type}}{\textsc{array}(\tau) \text{ is a type}} \qquad (\textsc{ArrayFormation})$$

You make new arrays using `array-make`, so there is only one introduction rule for arrays.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \textsc{int} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{array-make}(e_1, e_2) : \textsc{array}(\tau)} \qquad (\textsc{ArrayMake})$$

Another way to identify the rule as an introduction rule for the array constructor is that the array constructor appears in the conclusion, not in the premise.

There are three elimination rules for arrays: one each for get, set, and length.

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \textsc{array}(\tau) \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \textsc{int}}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{array-get}(e_1, e_2) : \tau} \qquad (\textsc{ArrayGet})$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \textsc{array}(\tau) \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \textsc{int} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{array-set}(e_1, e_2, e_3) : \tau} \qquad (\textsc{ArraySet})$$

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \textsc{array}(\tau)}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \textsc{array-length}(e) : \textsc{int}} \qquad (\textsc{ArrayLength})$$

We don't show you how to turn these rules into code for the type checker; that problem is left for Exercise 1.

251 ⟨*function* `ty`*, to check the type of an expression, given* $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ **[[prototype]]** 251⟩≡

```
| ty (AGET (a, i))       = raise LeftAsExercise "AGET"
| ty (ASET (a, i, e))    = raise LeftAsExercise "ASET"
| ty (AMAKE (len, init)) = raise LeftAsExercise "AMAKE"
| ty (ALEN a)            = raise LeftAsExercise "ALEN"
```

[4]Because the length of an array is not part of its type, Typed Impcore requires a run-time safety check for every array access. There are powerful type systems in which the length of an array *is* part of its type (Xi and Pfenning 1998), but they are beyond the scope of this book.

6.4 Interlude: common type constructors you find around the house every day

In the transition from Impcore to Typed Impcore, we added arrays. Arrays make a nice example because they are simple and they appear in many languages. Most statically typed languages provide not only arrays but many other abstractions that are useful for describing structured data. Some abstractions have proven their worth over and over in different language designs. This section presents type rules for some of the most commonly used abstractions.

Functions If functions are first-class values, it makes sense for them to have first-class types. The function type constructor takes two arguments, and we write it as an infix $\rightarrow$. The introduction and elimination rules are simply λ-abstraction and function application. Just as in the functions for Typed Impcore, we require that the λ-abstraction give the types of formal parameters. To keep the rules simple, we describe only one-argument functions here.

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \rightarrow \tau_2 \text{ is a type}} \qquad \text{(ArrowFormation)}$$

$$\frac{\Gamma\{x \mapsto \tau\} \vdash e : \tau'}{\Gamma \vdash \text{LAMBDA}(x : \tau, e) : \tau \rightarrow \tau'} \qquad \text{(ArrowIntro)}$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{APPLY}(e_1, e_2) : \tau'} \qquad \text{(ArrowElim)}$$

Products A *product*, often called a *pair* or *tuple*, groups together values of different types. It corresponds to ML's "tuple" type, to C's "struct", and to Pascal's "record". We use an infix $\times$ to write the product type constructor. It comes with three pieces of abstract syntax: PAIR, FST, and SND.

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \times \tau_2 \text{ is a type}} \qquad \text{(PairFormation)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{PAIR}(e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{(PairIntro)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{FST}(e) : \tau_1} \qquad \text{(Fst)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{SND}(e) : \tau_2} \qquad \text{(Snd)}$$

Like array operations, the pair operations are polymorphic, i.e., they can work with pairs of any types.

We already know the pair operations, because Chapter 3 presents them under other names: `cons`, `car`, and `cdr`. As noted in that Chapter, their dynamic semantics is given by these algebraic laws:

$$\begin{aligned} \text{FST}(\text{PAIR}(v_1, v_2)) &= v_1 \\ \text{SND}(\text{PAIR}(v_1, v_2)) &= v_2 \end{aligned}$$

In addition to `cons`, `car`, and `cdr`, there are other ways to write the concrete syntax of pair operations. Often $\text{PAIR}(e_1, e_2)$ is written in concrete syntax as (e_1, e_2). Operations FST and SND may be written as functions `fst` and `snd`. In ML, these functions are called `#1` and `#2`; in mathematical notation, they are sometimes written π_1 and π_2. Postfix notation is also used, so, e.g., $\text{FST}(e)$ might be written as $e.1$.

It can be awkward to have to use FST and SND to get the elements of a pair. In our ML code, we use a more convenient idiom: pattern matching, which is a combination of elimination and binding. We can model the ML expression `let val (x, y) = e' in e end` by the abstract syntax $\text{LETPAIRIN}(x, y, e', e)$.

$$\frac{\Gamma \vdash e' : \tau_1 \times \tau_2 \qquad \Gamma\{x \mapsto \tau_1, y \mapsto \tau_2\} \vdash e : \tau}{\Gamma \vdash \text{LETPAIRIN}(x, y, e', e) : \tau} \quad \text{(LETPAIR)}$$

The pair rules can be generalized to give rules for tuples with any number of elements—even zero! The type of tuples with zero elements can serve as a UNIT type, since it has only one value: the empty tuple. The tuple rules can be further generalized to give a name to each element of a tuple, which makes it possible to refer to elements by name instead of by position (Exercise 6).

Sums Where a product provides an ordered collection of values of different types, a sum provides a *choice* among values of different types. Almost every language provides obvious support for products, but support for sums is sometimes absent and often hard to recognize. C's "union" types and Pascal's "variant records" are examples of sum types. ML's `datatype` is also a form of sum type. A value of type $\tau_1 + \tau_2$ is either a value of type τ_1 or a value of type τ_2, and you can always tell which.

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 + \tau_2 \text{ is a type}} \quad \text{(SUMFORMATION)}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_2 \text{ is a type}}{\Gamma \vdash \text{LEFT}_{\tau_2}(e) : \tau_1 + \tau_2} \quad \text{(LEFT)}$$

$$\frac{\Gamma \vdash e : \tau_2 \qquad \tau_1 \text{ is a type}}{\Gamma \vdash \text{RIGHT}_{\tau_1}(e) : \tau_1 + \tau_2} \quad \text{(RIGHT)}$$

Sum elimination uses `case`, which is called `switch` in some languages.[5] We don't try to invent a readable abstract syntax for the `case` expression; it's easier to use some ML-like concrete syntax.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma\{x_1 \mapsto \tau_1\} \vdash e_1 : \tau \qquad \Gamma\{x_2 \mapsto \tau_2\} \vdash e_2 : \tau}{\Gamma \vdash \textbf{case } e \textbf{ of } \text{LEFT}(x_1) \Rightarrow e_1 \mid \text{RIGHT}(x_2) \Rightarrow e_2 : \tau} \quad \text{(SUMELIMCASE)}$$

Just like products, sums can be generalized so that each alternative has a name (Exercise 7).

[5]The case statement was invented by Tony Hoare, who said it was one of the three things in his career he was most proud of. The other two? The Quicksort algorithm and the fact that he started and finished his career in industry.

The dynamic semantics of sums is given by the CASE rules of operational semantics:

$$\frac{\langle e, \rho, \sigma\rangle \Downarrow \langle \text{LEFT}(v_1), \sigma'\rangle \quad \ell_1 \notin \operatorname{dom}\sigma' \quad \langle e_1, \rho\{x_1 \mapsto \ell_1\}, \sigma'\{\ell_1 \mapsto v_1\}\rangle \Downarrow \langle v, \sigma''\rangle}{\langle \textbf{case } e \textbf{ of } \text{LEFT}(x_1) \Rightarrow e_1 \mid \text{RIGHT}(x_2) \Rightarrow e_2, \rho, \sigma\rangle \Downarrow \langle v, \sigma''\rangle} \quad \text{(CASELEFT)}$$

$$\frac{\langle e, \rho, \sigma\rangle \Downarrow \langle \text{RIGHT}(v_2), \sigma'\rangle \quad \ell_2 \notin \operatorname{dom}\sigma' \quad \langle e_2, \rho\{x_2 \mapsto \ell_2\}, \sigma'\{\ell_2 \mapsto v_2\}\rangle \Downarrow \langle v, \sigma''\rangle}{\langle \textbf{case } e \textbf{ of } \text{LEFT}(x_1) \Rightarrow e_1 \mid \text{RIGHT}(x_2) \Rightarrow e_2, \rho, \sigma\rangle \Downarrow \langle v, \sigma''\rangle} \quad \text{(CASERIGHT)}$$

If a language has first-class functions, there is an alternative to `case`: the `either` construct. The expression (`either` e f g) applies either f or g to the value "carried" inside e. It is equivalent to **case** e **of** LEFT$(x_1) \Rightarrow f(x_1) \mid$ RIGHT$(x_2) \Rightarrow g(x_2)$.

Products and sums There is a duality between products and sums. To introduce a product of type $\tau_1 \times \tau_2$, you have to have a value of each type. To introduce a sum of type $\tau_1 + \tau_2$, you have a choice of whether to start with a value of type τ_1 or one of type τ_2. To eliminate a product of type $\tau_1 \times \tau_2$, you have a choice of whether you want to extract a value of type τ_1 or one of type τ_2. To eliminate a sum of type $\tau_1 + \tau_2$, you have no choice; you must be prepared to deal with both a value of type τ_1 and one of type τ_2.

Mutable cells The types shown above are all *immutable*, meaning that once you create a value you can't change it. Since mutation is a valuable programming technique, essential for both imperative and object-oriented programming, it makes sense to have type-system support for mutable cells. Exercise 8 suggests that you design type rules for a type constructor that supports mutation. You can use ML's `ref` constructor, with its three operations `ref`, `!`, and `:=`, as a model.

6.5 Type soundness

If, with the aid of a type checker, you use rules like those given above to show that a program is well typed, what have you gained? The answer depends on the language designer. A good language designer uses a static type system to guarantee some property about the program, usually a safety property. In other words, if a program type checks at compile time, that should tell you something about the program's behavior at run time.

A serious professional will pick a safety property, design a type system around that property, and prove that "well-typed programs don't go wrong," i.e., that well-typed programs satisfy the safety property. The type systems used in this book can guarantee such properties as "the program never attempts to take `car` of an integer" or "a function is always called with the correct number of arguments." More advanced type systems can guarantee such properties as "no array access is ever out of bounds" or "no pointer reference ever points to memory that has been deallocated." Whatever the property of interest, if you can prove that a type system guarantees it, you have a *soundness* result.

To express a claim about soundness, we need to talk about the meaning of a type. Given a type τ, we write $[\![\tau]\!]$ for a set of values associated with τ. Here are some examples:

$$\begin{array}{lcl} [\![\texttt{int}]\!] & = & \{\text{NUMBER}(n) \mid n \text{ is an integer}\} \\ [\![\texttt{bool}]\!] & = & \{\text{BOOL}(\texttt{\#t}), \text{BOOL}(\texttt{\#f})\} \\ [\![\texttt{sym}]\!] & = & \{\text{SYMBOL}(s) \mid s \text{ is a string}\} \\ [\![\tau \to \tau']\!] & = & \text{a set of functions taking values in } [\![\tau]\!] \text{ to values in } [\![\tau']\!] \end{array}$$

We can now express a soundness claim that says, essentially, that an expression of type τ evaluates to a value in the set $[\![\tau]\!]$. There are two details: environments must match up, and the evaluation of the expression must terminate. We need environments to deal with such expressions as function application. We need to assume termination because a simple type system typically does not guarantee termination.

Given this machinery, here is an example of a soundness claim. If for any x in $\operatorname{dom}\Gamma$, $x \in \operatorname{dom}\rho \wedge \rho(x) \in [\![\Gamma(x)]\!]$, and if $\Gamma \vdash e : \tau$, and if $\langle \rho, e\rangle \Downarrow v$, then $v \in [\![\tau]\!]$. That is, if the environments make sense, and if expression e has type τ, and if the evaluation of e terminates, then it produces a value in $[\![\tau]\!]$. To simplify the statement of the claim, we've simplified the notation: there's only one type environment, and we don't worry about a store. A claim like this could be proved by a simultaneous induction on the structure of the proof of $\Gamma \vdash e : \tau$ and the proof of $\langle \rho, e\rangle \Downarrow v$.

If a type system is sound, most run-time errors are impossible. For example, if the claim above holds for a Scheme-like language, then if evaluating e does not attempt to divide by zero or take `car` or `cdr` of the empty list, and if evaluating e doesn't get into an infinite loop, then the evaluation of e completes successfully.

6.6 Polymorphic type systems and Typed μScheme

The type system of Typed Impcore is at once too complicated and not powerful enough. It is too complicated because of the multiple type environments Γ_ξ, Γ_ϕ, and Γ_ρ. It is not powerful enough because an operation that works with values of more than one type has to be built into Typed Impcore. A function defined by a user can operate only on values of a single type, which is to say it is *monomorphic*. For example, it is impossible to write a general-purpose array-reversal function. Limitations like these make Typed Impcore an accurate model of such languages as C and Pascal. Pascal's especially Draconian restrictions on general-purpose functions were partly responsible for its demise.[6]

[6] In Pascal, not only the element type but also the *length* is part of an array's type. This decision makes it impossible to write a general-purpose sort routine even for arrays of integers. In C, one can write many general-purpose functions because C provides a loophole: a value of any pointer type can be cast to and from `void *`. Using `void *`, for example, it *is* possible to write a general-purpose sort function, a general-purpose array-reversal function, and much more. The catch is that if you use `void *`, you give up type safety, and it is quite possible for your C program to fail mysteriously ("dump core"), as every experienced C programmer knows.

To restrict a function to a single type hurts programmers because many useful data constructs inherently work with multiple types: they are *polymorphic*. Polymorphic constructs include arrays, tables, pointers, lists, products, sums, and objects. Without such constructs, our favorite languages would be unusable. In a monomorphic language, a polymorphic construct can be supported only by means of special abstract syntax. As an example, C provides special syntax for sums ("unions"), products ("structs"), arrays, and pointers. The problem is that given polymorphic data types, programmers naturally tend to write polymorphic functions. For example, in μScheme, the `length` function on lists is polymorphic; it works no matter what the types of the elements. In a monomorphic language such as C, as discussed in Chapter 2, it is not possible to write a single, type-safe, polymorphic function to compute the length of a list.

Another problem with a monomorphic language is that every time you want a new type constructor you have to add new rules to the type system, and every time you do that you have to revisit your proof of type soundness. This problem makes it very difficult for a programmer to add a user-defined type constructor such as the `env` type constructor we use for environments in Chapter 5. At best, in a monomorphic language, a programmer can add a new type.

A language designer solves these problems by using a *polymorphic type system*. Such a type system makes it possible for a programmer to write polymorphic functions, and for anyone—a programmer or a language designer—to add a new type constructor. To experiment with polymorphism, we have developed a language called *Typed micro-Scheme*, or Typed μScheme for short. Typed μScheme uses the same *values* as μScheme, but it has simpler abstract syntax, and it has a static, polymorphic type system. We begin our study of Typed μScheme with a view of the concrete syntax. We continue by examining *kinds* and *quantified types*, which are the two ideas at the core of the type system. Kinds classify types in much the same way that types classify terms. Quantified types make it possible to implement polymorphic operations using ordinary functions instead of special abstract syntax. We finish our study by showing how these ideas are integrated into μScheme and realized in an interpreter.

6.6.1 Concrete syntax of Typed μScheme

Typed μScheme is much like μScheme, except as follows.

- Function definitions and `lambda` abstractions require explicit types for parameters.
- Function definitions require result types.
- Typed μScheme does not have `letrec`; the type annotations would be too much trouble.
- To reduce the burden of type annotation, Typed μScheme provides two forms of top-level binding: `val` and `val-rec`. The `val` binding is for non-recursive values and requires no type annotation. The `val-rec` binding is for recursive values and requires an explicit type annotation. These bindings are similar to the corresponding bindings in ML. The `val-rec` binding is useful for defining polymorphic, recursive functions; the explicit type makes it possible to type-check a recursive call.

The type system of Typed μScheme is more powerful than that of Typed Impcore.

- There are polymorphic types, which are written with `forall`.
- There are two new terms: `type-lambda`, which introduces a polymorphic type, and `@`, which eliminates a polymorphic type.
- There is no syntactic distinction between "type" and "type constructor;" both appear in the syntax as *type-exp*.
- There are no type constructors built into the language; in Typed μScheme, type constructors are part of the initial basis. The type constructor for functions is the only one that requires special-purpose abstract syntax (`lambda` abstraction and function application).

Here is the syntax:

def ::= (`val` *variable-name exp*)
| (`val-rec` *type-exp variable-name exp*)
| (`define` *type-exp function-name* (*formals*) *exp*)
| *exp*
| (`use` *file-name*)

exp ::= *literal*
| *variable-name*
| (`set` *variable-name exp*)
| (`if` *exp exp exp*)
| (`while` *exp exp*)
| (`begin` {*exp*})
| (*exp* {*exp*})
| (*let-keyword* ({(*variable-name exp*)}) *exp*)
| (`lambda` (*formals*) *exp*)
| (`type-lambda` (*type-formals*) *exp*)
| (`@` *exp* {*type-exp*})

let-keyword ::= `let` | `let*`

formals ::= {(*type-exp variable-name*)}

type-formals ::= {'*type-variable-name*}

type-exp ::= *type-constructor-name*
| '*type-variable-name*
| (`forall` ({'*type-variable-name*}) *type-exp*)
| (`function` ({*type-exp*}) *type-exp*)
| (*type-exp* {*type-exp*})

literal ::= *integer* | `#t` | `#f` | '*S-exp* | (`quote` *S-exp*)

S-exp ::= *literal* | *symbol-name* | ({*S-exp*})

integer ::= sequence of digits, possibly prefixed with a minus sign

**-name* ::= sequence of characters not an *integer* and not containing (,), ;, or whitespace

6.6.2 A replacement for type-formation rules: kinds

In a language with a fixed number of types or type constructors, we can use type-formation rules to explain all the types in the language. But each new type constructor requires a new rule. A better approach uses just a few rules to encompass arbitrarily many type constructors. The basis of this approach is to assign each type constructor a *kind*.

A kind shows how a type constructor may be used. For example, both `int` and `array` are type constructors of Typed Impcore, but the rules require that they be used differently. The `int` constructor is a kind of constructor that is a type all by itself; the `array` constructor is a kind of constructor that has to be applied to an element type in order to make another type. We write the first kind $*$ and pronounce it "type;" we write the second kind $* \Rightarrow *$ and pronounce it "type arrow type." To use a kind, we make a formal judgment that a type constructor has a particular kind. Formally, we write $\tau :: \kappa$ to say that type constructor τ has kind κ. For example, the judgment "$\tau :: *$" is equivalent to the judgment "τ is a type" we use in Typed Impcore.

The structure of kinds is simple; $*$ (type) is the basic kind, and we make other kinds using arrows. As a concrete notation, we write

$$\kappa \qquad ::= * \mid \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa$$

This syntax corresponds to the following rules for kinds.

$$\frac{}{* \text{ is a kind}} \qquad \text{(KindFormationType)}$$

$$\frac{\kappa_1, \ldots, \kappa_n \text{ are kinds} \qquad \kappa \text{ is a kind}}{\kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \text{ is a kind}} \qquad \text{(KindFormationArrow)}$$

In our ML code, we represent the abstract syntax of a kind using the datatype `kind`.

258a ⟨*types for Typed μScheme* 258a⟩≡ (275a) 260▷

```
datatype kind = TYPE                          (* kind of all types *)
              | ARROW of kind list * kind     (* kind of many constructors *)
```

How do we know which type constructors have which kinds? In Typed Impcore, we consult type rules. For example, the BaseTypes rule shows that `int` has kind $*$, and the ArrayFormation rule shows that `array` has kind $* \Rightarrow *$. But one of the problems we are trying to solve is that type rules are not easily extensible. In Typed μScheme, type constructors are not built in. Instead, we put the kind of each constructor in a *kind environment*. Here is an example environment that shows common type constructors and their kinds. For clarity, we write the binding in a kind environment using the :: symbol instead of $\mapsto$. (As noted above, the :: symbol is pronounced "has kind.")

$$\Delta_0 = \{\, \texttt{int} :: *,\ \texttt{bool} :: *,\ \texttt{unit} :: *,\ \texttt{pair} :: * \times * \Rightarrow *,\\ \texttt{sum} :: * \times * \Rightarrow *,\ \rightarrow :: * \times * \Rightarrow *,\ \texttt{array} :: * \Rightarrow *,\ \texttt{list} :: * \Rightarrow * \,\}$$

All by itself, this environment enables us to see how both `int` and `array` may be used. Even better, we can add new type constructors to a language just by adding them to the kind environment. Here are the type constructors that are built into Typed μScheme, except those that are left as Exercises.

258b ⟨*primitive type constructors for Typed μScheme* :: 258b⟩≡ (274b)

```
("int",      TYPE) ::
("bool",     TYPE) ::
("sym",      TYPE) ::
("unit",     TYPE) ::
("list",     ARROW ([TYPE],       TYPE)) ::
("function", ARROW ([TYPE, TYPE], TYPE)) ::
```

As these examples show, kinds classify types in much the same way that types classify expressions.

We use a kind environment in the general form of a kinding judgment, which is $\Delta \vdash \tau :: \kappa$. This judgment says that in kind environment Δ, type-level expression τ has kind κ. The kind environment Δ gives, among other things, the kind of each type constructor.

With kinds as background, we show how to form types. At minimum, we need two ways of forming types:

- We need a way of writing a type constructor. We write the abstract syntax as $\textsc{tycon}(\mu)$, where μ is the name of the constructor. We write the concrete syntax as the name of the constructor; for example, we write `int` for the type "integer."
- We need a way of applying a type to a list of other types. We write the abstract syntax as $\textsc{conapp}(\tau, [\tau_1, \ldots, \tau_n])$, where τ and $\tau_1, \ldots, \tau_n$ are type-level expressions. We write the concrete syntax just as we write function application; for example, we write `(list int)` for the type "list of integer."

The following two rules give the kinds of type-level expressions built with TYCON and CONAPP. The kind of each type constructor must be stored in the kind environment Δ.

$$\frac{\mu \in \operatorname{dom} \Delta}{\Delta \vdash \textsc{tycon}(\mu) :: \Delta(\mu)} \qquad (\textsc{KindIntroCon})$$

$$\frac{\Delta \vdash \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \qquad \Delta \vdash \tau_i :: \kappa_i, \quad 1 \leq i \leq n}{\Delta \vdash \textsc{conapp}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa} \qquad (\textsc{KindApp})$$

There is one special type constructor in Typed μScheme; the tuple constructor may be applied to any number of arguments, provided all are types. The result is a type.

$$\frac{\Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n}{\Delta \vdash \textsc{conapp}(\textsc{tycon}(\texttt{tuple}), [\tau_1, \ldots, \tau_n]) :: *} \qquad (\textsc{KindTuple})$$

In this book, we use special syntactic sugar for the `tuple` constructor; instead of $\textsc{conapp}(\textsc{tycon}(\texttt{tuple}), [\tau_1, \ldots, \tau_n])$, we write $\tau_1 \times \cdots \times \tau_n$. We also use syntactic sugar for function types. For a function of n arguments, we write $\tau_1 \times \cdots \times \tau_n \to \tau$. In the concrete syntax of Typed μScheme, we write `(function (`τ_1 ... τ_n`)` τ`)`.

The rules above suffice to replace the type-formation rules of Typed Impcore. To get to Typed μScheme, however, we need something more: quantified types.

6.6.3 The heart of polymorphism: quantified types

In μScheme, we can write the function `length`, which can be applied to any list, no matter what the types of its elements.

```
(define length (l)
  (if (null? l) 0 (+ 1 (length (cdr l)))))
```

The difficulty with defining such a function in a typed language is what types to use. Even if we were to add lists to Typed Impcore (see Exercise 5), we could not write a single version of `length`. We would have to write a version for each type:

```
(define int lengthI (((list int) l))
  (if (null? l) 0 (+ 1 (lengthI (cdr l)))))
(define int lengthB (((list bool) l))
  (if (null? l) 0 (+ 1 (lengthB (cdr l)))))
(define int lengthS (((list sym) l))
  (if (null? l) 0 (+ 1 (lengthS (cdr l)))))
```

Such duplication of code is a waste of effort; except for the types, the functions are identical. The problem is that using only Typed Impcore, we have no way to say that the behavior of `length` is independent of the type of the elements in the list. In a polymorphic type system, we solve this problem by introducing *type variables* and *quantified types.*

A type variable stands for an unknown type. A quantified type gives us permission to substitute *any* type for the unknown type. In this book, we write type variables using the Greek letters α, β, and γ; we write quantified types using $\forall$. For example, we can write a `length` function that has type $\forall\alpha.\alpha$ `list` $\to$ `int`. In Typed μScheme we would write this type as `(forall ('a) (function ((list 'a)) int))`. To use this length function on a list of integers, we strip off the $\forall$ and replace α with `int`. The type of the resulting function is `int list` $\to$ `int`, or in Typed μScheme, `(function ((list int)) int)`.

To support type variables and quantified types, we need suitable abstract syntax and kinding rules. As the abstract syntax, we use TYVAR and FORALL. The kinding rules take the place of type-formation rules.

The kinding rule for a type variable says that, just like a type constructor, a variable is looked up in the environment Δ.

$$\frac{\alpha \in \operatorname{dom}\Delta}{\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)} \qquad \text{(KINDINTROVAR)}$$

The kind of a quantified type is always $*$, and the FORALL quantifier can be used only over type-level expressions of kind $*$. Within the body of the FORALL, the quantified variables stand for types. We therefore introduce them into the kind environment with kind $*$.

$$\frac{\Delta\{\alpha_1 :: *, \ldots, \alpha_n :: *\} \vdash \tau :: *}{\Delta \vdash \text{FORALL}(\langle\alpha_1, \ldots, \alpha_n\rangle, \tau) :: *} \qquad \text{(KINDALL)}$$

In Typed μScheme, as in Standard ML, all type variables have kind $*$. But in some polymorphic type systems, type variables may have any kind. For example, the functional language Haskell uses a type system that allows type variables to have other kinds.

type name 214

To build types in Typed μScheme, we use not only type variables and the FORALL quantifier but also type constructors and constructor application. In our ML code, we represent the abstract syntax of types using the type `tyex`.

260 ⟨*types for Typed μScheme* 258a⟩+≡ (275a) ◁258a 263c▷

```
datatype tyex = TYCON  of name                (* type constructor *)
              | CONAPP of tyex * tyex list    (* type-level application *)
              | FORALL of name list * tyex
              | TYVAR  of name                (* type variable *)
```

Because not every combination of type constructors, variables, and `forall` is actually a type (e.g., `array array` is not a type), it may be best to call such phrases "type-level expressions." When we're not being careful, however, we refer to them simply as "types."

The abstract syntax defined by `tyex` corresponds to the *type-exp* nonterminal in the concrete syntax on page 257. As examples of this concrete syntax, here are the types of some polymorphic functions and values related to lists.

261a ⟨*transcript* 261a⟩≡ 261b▷

```
-> length
<procedure> : (forall ('a) (function ((list 'a)) int))
-> cons
<procedure> : (forall ('a) (function ('a (list 'a)) (list 'a)))
-> car
<procedure> : (forall ('a) (function ((list 'a)) 'a))
-> cdr
<procedure> : (forall ('a) (function ((list 'a)) (list 'a)))
-> '()
() : (forall ('a) (list 'a))
```

Polymorphism is not restricted to functions; even though it is not a function, the empty list has a polymorphic type.

The parenthesized-prefix syntax above is easy to parse and easy to understand, but it is not so easy to read. In the text, we also use an algebraic notation in which we write type constructors μ, type variables α, and types τ. We follow the ML postfix convention for application of type constructors, and we provide special notation for the function and tuple constructors:

$$\tau \quad ::= \mu \mid \alpha \mid (\tau_1, \ldots, \tau_n)\,\tau \mid \forall\alpha_1, \ldots, \alpha_n\,.\,\tau \mid \tau_1 \times \cdots \times \tau_n \mid \tau_1 \times \cdots \times \tau_n \to \tau$$

Using this notation, we write the types above as follows:

$$\begin{array}{ll} \texttt{length} & : \forall\alpha\,.\,\alpha\ \texttt{list} \to \texttt{int} \\ \texttt{cons} & : \forall\alpha\,.\,\alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list} \\ \texttt{car} & : \forall\alpha\,.\,\alpha\ \texttt{list} \to \alpha \\ \texttt{cdr} & : \forall\alpha\,.\,\alpha\ \texttt{list} \to \alpha\ \texttt{list} \\ \texttt{'()} & : \forall\alpha\,.\,\alpha\ \texttt{list} \end{array}$$

To use a value with a polymorphic type, we have to say what unknown type each type variable stands for. This process is called *instantiation.* To instantiate a polymorphic value, we use the `@` operator in the concrete syntax. We give the value to be instantiated, plus a type to be substituted for each type variable in the quantified type. For reasons that may be evident from the syntax, instantiation is sometimes also called *type application.* An instantiation or application may be pronounced with the word "at," as in "`length` at integer."

261b ⟨*transcript* 261a⟩+≡ ◁261a 262a▷

```
-> (val length-int (@ length int))
length-int : (function ((list int)) int)
-> (val cons-bool (@ cons bool))
cons-bool : (function (bool (list bool)) (list bool))
-> pair
<procedure> : (forall ('a 'b) (function ('a 'b) (pair 'a 'b)))
-> (val car-pair (@ car (pair sym int)))
car-pair : (function ((list (pair sym int))) (pair sym int))
-> (val cdr-sym (@ cdr sym))
cdr-sym : (function ((list sym)) (list sym))
-> (val empty-int (@ '() int))
() : (list int)
```

In each case, we get the type of the instantiated function by substituting each type parameter for the corresponding type variable in the `forall`.

Once instantiated, the polymorphic functions are monomorphic, and they can be applied to values.

262a ⟨*transcript* 261a⟩+≡ ◁261b

```
-> (length-int '(1 4 9 16 25))
5 : int
-> (cons-bool #t '(#f #f))
(#t #f #f) : (list bool)
-> (car-pair ((@ cons (pair sym int))
                ((@ pair sym int) 'Office 231)
                (@ '() (pair sym int))))
(Office . 231) : (pair sym int)
-> (cdr-sym '(a b c d))
(b c d) : (list sym)
```

As the `car-pair` example shows, there is no need to name instantiated values; they can be used directly.

The type-instantiation operator `@` enables a programmer to *use* a polymorphic value. But to enjoy the full power of polymorphism, a programmer must also be able to *create* a polymorphic value. For this purpose, Typed μScheme provides a *type-abstraction* operator: `type-lambda`. As an example, we use `type-lambda` to define the polymorphic functions `list1`, `list2`, and `list3`.

262b ⟨*additions to the Typed μScheme initial basis* 262b⟩≡ 262c▷

```
(val list1 (type-lambda ('a) (lambda (('a x)) ((@ cons 'a) x (@ '() 'a)))))
(val list2 (type-lambda ('a) (lambda (('a x) ('a y))
                                ((@ cons 'a) x ((@ list1 'a) y)))))
(val list3 (type-lambda ('a) (lambda (('a x) ('a y) ('a z))
                                ((@ cons 'a) x ((@ list2 'a) y z)))))
```

These functions are part of the initial basis of Typed μScheme.

Some of the higher-order functions of Chapter 3 are also easy to write using `type-lambda`. Here are the types of function composition, Currying, and uncurrying:

$$\begin{array}{ll} \texttt{o} & : \forall\alpha,\beta,\gamma\,.\,(\beta \to \gamma) \times (\alpha \to \beta) \to (\alpha \to \gamma) \\ \texttt{curry} & : \forall\alpha,\beta,\gamma\,.\,(\alpha \times \beta \to \gamma) \to (\alpha \to (\beta \to \gamma)) \\ \texttt{uncurry} & : \forall\alpha,\beta,\gamma\,.\,(\alpha \to (\beta \to \gamma)) \to (\alpha \times \beta \to \gamma) \end{array}$$

We use `type-lambda` to define these functions as well.

262c ⟨*additions to the Typed μScheme initial basis* 262b⟩+≡ ◁262b 263a▷

```
(val o (type-lambda ('a 'b 'c)
  (lambda (((function ('b) 'c) f)
           ((function ('a) 'b) g))
    (lambda (('a x)) (f (g x))))))

(val curry (type-lambda ('a 'b 'c)
   (lambda (((function ('a 'b) 'c) f))
      (lambda (('a x)) (lambda (('b y)) (f x y))))))

(val uncurry (type-lambda ('a 'b 'c)
   (lambda (((function ('a) (function ('b) 'c)) f))
      (lambda (('a x) ('b y)) ((f x) y)))))
```

The functions above are not recursive. To define a recursive function, we need to know the type of that function; otherwise there is no way to type-check a recursive call. Because `define` introduces a function with explicit argument and result types, we can figure out the type of the function being defined, so we can use `define` to define a recursive function. But we cannot define a *polymorphic* function with `define`, because `define` does not abstract over a type. To define a function that is both polymorphic and recursive, we need both type abstraction and recursive binding. We get type abstraction from `type-lambda`; to get recursive binding, we introduce a new top-level binding construct: `val-rec`. It is like `val` except that it requires an explicit type, and the right-hand side can refer to the value being defined. Its syntax is `(val-rec` *type name exp*`)`, where *exp* can refer to *name*, and both *name* and *exp* have type *type*. Its typical use is to define a recursive, polymorphic function, such as `length`.

263a ⟨*additions to the Typed μScheme initial basis* 262b⟩+≡ ◁262c 263b▷

```
(val-rec
  (forall ('a) (function ((list 'a)) int))  ; type of length
  length                                    ; name
  (type-lambda ('a)                         ; value : polymorphic function
     (lambda (((list 'a) l))
       (if ((@ null? 'a) l) 0
          (+ 1 ((@ length 'a) ((@ cdr 'a) l)))))))
```

As another example, here is an explicitly typed version of the reverse-append function.

263b ⟨*additions to the Typed μScheme initial basis* 262b⟩+≡ ◁263a

```
(val-rec
  (forall ('a) (function ((list 'a) (list 'a)) (list 'a)))
  revapp
  (type-lambda ('a)
     (lambda (((list 'a) xs)  ((list 'a) ys))
        (if ((@ null? 'a) xs)
        ys
        ((@ revapp 'a) ((@ cdr 'a) xs) ((@ cons 'a) ((@ car 'a) xs) ys))))))
```

As you can see, explicit types and polymorphism impose a heavy notational burden. Chapter 7 shows how to use type inference to lighten the load. Appendix G shows more of the initial basis of Typed μScheme, which provides more examples.

To summarize this section, the essential new idea in a polymorphic type system is the quantified type. It comes with its own special-purpose syntax: `forall` to form a quantified type, `type-lambda` to introduce a quantified type, and `@` to eliminate a quantified type. The rest of this chapter shows how a type system based on quantified types is combined with μScheme to produce Typed μScheme.

6.6.4 Manipulating types and kinds in an interpreter

Before we go on to abstract syntax, values, and type rules for Typed μScheme, we show some basic manipulations of kinds and types. Given a type-level expression, we need to know if it has a kind and if so, what its kind is. We also need to instantiate polymorphic types. Finally, to implement the type rules of Typed μScheme, we need to compare types for equality.

Throughout the interpreter, we print types using the `typeString` function from Appendix G.

263c ⟨*types for Typed μScheme* 258a⟩+≡ (275a) ◁260 267b▷
⟨*printing types for Typed μScheme* 683⟩

```
typeString : tyex -> string
```

Checking a kind

The function `kindof` implements the kinding judgment $\Delta \vdash \tau :: \kappa$, which says that given kind environment Δ, type-level expression τ has kind κ. Given Δ and τ, `kindof(τ, Δ)` returns a κ such that $\Delta \vdash \tau :: \kappa$, or if no such kind exists, it raises the exception `TypeError`.

264a ⟨*type checking for Typed μScheme* 264a⟩≡ (275a) 265c▷

```
fun kindof (tau, Delta) =
  let ⟨internal function kind 264b⟩
  in  kind tau
  end
```

```
kindof : tyex * kind env -> kind
kind   : tyex            -> kind
```

The internal function `kind` computes the kind of `tau`; the environment `Delta` is assumed. The implementation of `kind` is derived directly from the kinding rules. The derivation follows the same process as the derivation of a type checker from typing rules or the derivation of an interpreter from operational semantics. As usual, we repeat the rules to show the connection with the code.

A type variable is looked up in the environment.

$$\frac{\alpha \in \operatorname{dom}\Delta}{\Delta \vdash \textsc{tyvar}(\alpha) :: \Delta(\alpha)} \qquad (\textsc{KindIntroVar})$$

The parser guarantees that the name of a type variable begins with a quote mark, so it is distinct from any type constructor.

264b ⟨*internal function* `kind` 264b⟩≡ (264a) 264c▷

```
fun kind (TYVAR a) =
      (find (a, Delta)
       handle NotFound _ => raise TypeError ("unknown type variable " ^ a))
```

A type constructor is also looked up.

$$\frac{\mu \in \operatorname{dom}\Delta}{\Delta \vdash \textsc{tycon}(\mu) :: \Delta(\mu)} \qquad (\textsc{KindIntroCon})$$

264c ⟨*internal function* `kind` 264b⟩+≡ (264a) ◁264b 264d▷

```
  | kind (TYCON c) =
      (find (c, Delta)
       handle NotFound _ => raise TypeError ("unknown type constructor " ^ c))
```

The tuple type constructor may be used to combine any number of types.

$$\frac{\Delta \vdash \tau_i :: *, \qquad 1 \le i \le n}{\Delta \vdash \textsc{conapp}(\textsc{tycon}(\texttt{tuple}), [\tau_1, \ldots, \tau_n]) :: *} \qquad (\textsc{KindTuple})$$

⟨*internal function* `kind` 264b⟩+≡ (264a) ◁264c 265a▷

```
  | kind (CONAPP (TYCON "tuple", actuals)) =
      if List.all (fn tau => kind tau = TYPE) actuals then
          TYPE
      else
          raise TypeError "tuple formed from non-types"
```

The standard function `List.all` corresponds to μScheme's function `all?`.

CONAPP 260
find 214
NotFound 214
TYCON 260
TYPE 258a
TypeError 275a
TYVAR 260

Every type constructor other than `tuple` must be applied in a way that is consistent with its kind.

$$\frac{\Delta \vdash \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \qquad \Delta \vdash \tau_i :: \kappa_i, \quad 1 \le i \le n}{\Delta \vdash \textsc{conapp}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa} \quad (\textsc{KindApp})$$

265a ⟨*internal function* `kind` 264b⟩+≡ (264a) ◁264d 265b▷

```
| kind (CONAPP (tau, actuals)) =
    (case kind tau
       of ARROW (formals, result) =>
            if formals = map kind actuals then
                result
            else
                raise TypeError ("type constructor " ^ typeString tau ^
                                 " applied to the wrong arguments")
        | TYPE =>
            raise TypeError ("tried to apply type " ^ typeString tau ^
                             " as type constructor"))
```

A quantified type must always have kind $*$.

$$\frac{\Delta\{\alpha_1 :: *, \ldots, \alpha_n :: *\} \vdash \tau :: *}{\Delta \vdash \textsc{forall}(\langle \alpha_1, \ldots, \alpha_n \rangle, \tau) :: *} \quad (\textsc{KindAll})$$

The quantified variables $\alpha_1, \ldots, \alpha_n$ may be used in τ, so we insert them into Δ before checking the kind of τ.

265b ⟨*internal function* `kind` 264b⟩+≡ (264a) ◁265a

```
| kind (FORALL (alphas, tau)) =
    let val Delta' =
          foldl (fn (a, Delta) => bind (a, TYPE, Delta)) Delta alphas
    in  case kindof (tau, Delta')
          of TYPE    => TYPE
           | ARROW _ => raise TypeError "quantifed a non-nullary type constructor"
    end
```

A type-level expression used to describe a variable or parameter must have kind `TYPE`. The function `asType` ensures it.

265c ⟨*type checking for Typed μScheme* 264a⟩+≡ (275a) ◁264a 266a▷

```
fun asType (ty, Delta) =
  case kindof (ty, Delta)
    of TYPE    => ty
     | ARROW _ => raise TypeError ("used type constructor `" ^ typeString ty ^
                                   "' as a type")
```

`asType : tyex * kind env -> tyex`

ARROW	258a
bind	214
CONAPP	260
Delta	264a
FORALL	260
kind	264b
kindof	264a
TYPE	258a
TypeError	275a
typeString	683

Instantiating a polymorphic type

Our implementation of substitution is based on a special environment that tells what to substitute for each type variable. We don't substitute inside a `FORALL` for a variable bound by that `FORALL`. To arrange this behavior, we extend the special environment so that such a variable stands for itself.

The function `tysubst (τ, varenv)` changes type τ by substituting for type variables as specified by the environment `varenv`. Environment `varenv` does not necessarily substitute for every type variable in τ; some type variables may be left alone. The inner function `subst` walks the type, leaving `varenv` alone.

266a ⟨*type checking for Typed μScheme* 264a⟩+≡ (275a) ◁265c 266b▷

```
tysubst : tyex * tyex env -> tyex
subst   : tyex            -> tyex
```

```
fun tysubst (tau, varenv) =
  let fun subst (TYVAR a) = (find (a, varenv) handle NotFound _ => TYVAR a)
        | subst (TYCON c) = (TYCON c)
        | subst (CONAPP (tau, taus)) = CONAPP (subst tau, map subst taus)
        | subst (FORALL (alphas, tau)) =
            FORALL (alphas, tysubst (tau, bindList (alphas, map TYVAR alphas, varenv)))
  in  subst tau
  end
```

Instantiation is a matter of substituting for type variables. Most of the code is error checking. We must instantiate only at type-level expressions of kind type, we must instantiate with the right number of types, and we must instantiate only polymorphic types.

266b ⟨*type checking for Typed μScheme* 264a⟩+≡ (275a) ◁266a 267a▷

```
instantiate : tyex * tyex list * kind env -> tyex
List.find : ('a -> bool) -> 'a list -> 'a option
```

```
fun instantiate (FORALL (formals, tau), actuals, Delta) =
      (case List.find (fn t => kindof (t, Delta) <> TYPE) actuals
         of SOME t => raise TypeError ("instantiated at type constructor `" ^
                                       typeString t ^ "', which is not a type")
          | NONE =>
              (tysubst (tau, bindList (formals, actuals, emptyEnv))
               handle BindListLength =>
                 raise TypeError
                   "instantiated polymorphic term at wrong number of arguments"))
  | instantiate (tau, _, _) =
      raise TypeError ("tried to instantiate term of non-polymorphic type " ^
                       typeString tau)
```

The standard function `List.find` takes a predicate and searches a list for an element satisfying that predicate.

bindList 214
BindListLength 214
CONAPP 260
emptyEnv 214
find 214
FORALL 260
kindof 264a
NotFound 214
TYCON 260
TYPE 258a
TypeError 275a
typeString 683
TYVAR 260

Type equality

As in Typed Impcore, many rules of Typed μScheme require that two types be the same. We can't simply use built-in equality to check, however, because when we compare two quantified types, the *names* of the type variables should be irrelevant. For example, the two types $\forall\alpha\,.\,\alpha$ `list` $\rightarrow$ `int` and $\forall\beta\,.\,\beta$ `list` $\rightarrow$ `int` should be considered to be equal. We implement the check by substituting one set of names for the other.

267a ⟨*type checking for Typed μScheme* 264a⟩+≡ (275a) ◁266b 271▷

```
eqType  : tyex       * tyex       -> bool
eqTypes : tyex list * tyex list -> bool
```

```
fun eqType (TYCON c, TYCON c') = c = c'
  | eqType (CONAPP (tau, taus), CONAPP (tau', taus')) =
      eqType (tau, tau') andalso eqTypes (taus, taus')
  | eqType (FORALL (alphas, tau), FORALL (alphas', tau')) =
      (eqType (tau, tysubst (tau', bindList (alphas', map TYVAR alphas, emptyEnv)))
       handle BindListLength => false)
  | eqType (TYVAR a, TYVAR a') = a = a'
  | eqType _ = false
and eqTypes (t::taus, t'::taus') = eqType (t, t') andalso eqTypes (taus, taus')
  | eqTypes ([], []) = true
  | eqTypes _ = false
```

Standard type constructors of Typed μScheme

The code above gives representations of and operations on types, but it doesn't make it easy to write types. For example, inside the interpreter, the type of `cons` has to be written using this enormous phrase:

```
FORALL (["'a"],
          CONAPP (TYCON "function",
                  [CONAPP (TYCON "tuple",
                           [TYVAR "'a", CONAPP (TYCON "list", [TYVAR "'a"])]),
                   CONAPP (TYCON "list", [TYVAR "'a"])]))
```

To make it easier to define the primitive operations of Typed μScheme, we provide convenience functions.

267b ⟨*types for Typed μScheme* 258a⟩+≡ (275a) ◁263c

```
inttype   : tyex
booltype  : tyex
symtype   : tyex
unittype  : tyex
tyvarA    : tyex
tupletype : tyex list -> tyex
listtype  : tyex -> tyex
funtype   : tyex list * tyex -> tyex
```

```
val inttype  = TYCON "int"
val booltype = TYCON "bool"
val symtype  = TYCON "sym"
val unittype = TYCON "unit"
val tyvarA   = TYVAR "'a"
fun tupletype l = CONAPP (TYCON "tuple", l)
fun listtype ty = CONAPP (TYCON "list",[ty])
fun funtype (args, result) = CONAPP (TYCON "function", [tupletype args, result])
```

```
bindList        214
BindListLength
                214
CONAPP          260
emptyEnv        214
FORALL          260
NIL             268a
TYCON           260
tysubst         266a
TYVAR           260
```

Although in principle a value of type `unit` needs no run-time representation, in our interpreter, it's easier to provide one. The representation should be the same everywhere because the programmer has the right to compare two values of type `unit` for equality, and our implementation of equality compares representations. This, therefore, is a convenient place to define a common representation:

267c ⟨*checking and evaluation for Typed μScheme* 267c⟩≡ (275a) 273a▷

```
unitVal : value
```

```
val unitVal = NIL
```

We now have all the type machinery we need to implement Typed μScheme. We continue by presenting the abstract syntax, values, and type rules of Typed μScheme.

6.6.5 Abstract syntax, values, and evaluation of Typed μScheme

As with the concrete syntax, there are only minor differences between the abstract syntax of μScheme and the abstract syntax of Typed μScheme. We add two new expressions, TYLAMBDA and TYAPPLY, which introduce and eliminate polymorphic types. We drop LETREC. We also require that the parameters of functions have explicit types.

268a ⟨*abstract syntax and values for Typed μScheme* 268a⟩≡ (275a) 268b▷

```
datatype exp = LITERAL  of value
             | VAR      of name
             | SET      of name * exp
             | IFX      of exp * exp * exp
             | WHILEX   of exp * exp
             | BEGIN    of exp list
             | APPLY    of exp * exp list
             | LETX     of let_kind * (name * exp) list * exp
             | LAMBDA   of lambda_exp
             | TYLAMBDA of name list * exp
             | TYAPPLY  of exp * tyex list
and let_kind = LET | LETSTAR
and    value = NIL
             | BOOL      of bool
             | NUM       of int
             | SYM       of name
             | PAIR      of value * value
             | CLOSURE   of lambda_value * value ref env
             | PRIMITIVE of primitive
withtype primitive    = value list -> value (* raises RuntimeError *)
     and lambda_exp   = (name * tyex) list * exp
     and lambda_value = name          list * exp

exception RuntimeError of string
```

The values of Typed μScheme are the same as the values of μScheme; adding a type system doesn't require us to change any run-time representation.

The definitions of Typed μScheme are like those of Typed Impcore, with one addition. So we can define functions that are both recursive and polymorphic, we provide VALREC.

268b ⟨*abstract syntax and values for Typed μScheme* 268a⟩+≡ (275a) ◁268a

```
datatype def = VAL    of name * exp
             | VALREC of name * tyex * exp
             | EXP    of exp
             | DEFINE of name * tyex * lambda_exp
             | USE    of name
```

type env 214
type name 214
type tyex 260

In VALREC, DEFINE, and each parameter in LAMBDA, although the concrete syntax (page 257) puts the type on the left and the name on the right, the abstract syntax puts the name on the left and the type on the right. Why aren't they consistent? Because the concrete syntax emulates C code and the abstract syntax emulates common programming-language theory.

6.6.6 Type rules for Typed μScheme

The type rules for Typed μScheme are very similar to the rules for Typed Impcore. The important differences are these:

- There is only one type environment, Γ, mapping values to types.
- There is a kind environment, Δ, to keep track of the kinds of type variables and type constructors.
- There are no special-purpose typing rules associated with individual type constructors; type formation is handled by the general-purpose kinding rules, and the rest is handled by the general-purpose typing rules for type abstraction, instantiation, `lambda` abstraction, and application.

Type rules for expressions

The typing judgment for an expression is $\Delta, \Gamma \vdash e : \tau$, meaning that given kind environment Δ and type environment Γ, expression e has type τ.

The type of a literal depends on its value. There are quite a few rules, but they are what you would expect for homogeneous lists. There is one fine point: empty lists are polymorphic, but nonempty lists are monomorphic.

$$\frac{}{\Delta, \Gamma \vdash \textsc{literal}(\textsc{num}(n)) : \texttt{int}} \qquad \frac{}{\Delta, \Gamma \vdash \textsc{literal}(\textsc{bool}(n)) : \texttt{bool}} \qquad \frac{}{\Delta, \Gamma \vdash \textsc{literal}(\textsc{sym}(n)) : \texttt{sym}} \quad (\textsc{Literals})$$

$$\frac{}{\Delta, \Gamma \vdash \textsc{literal}(\textsc{nil}) : \forall\alpha\,.\,\alpha\ \texttt{list}} \qquad \frac{\Delta, \Gamma \vdash \textsc{literal}(v) : \tau}{\Delta, \Gamma \vdash \textsc{literal}(\textsc{pair}(v, \textsc{nil})) : \tau\ \texttt{list}} \quad (\textsc{ListLiterals1})$$

$$\frac{\Delta, \Gamma \vdash \textsc{literal}(v) : \tau \qquad \Delta, \Gamma \vdash \textsc{literal}(v') : \tau\ \texttt{list}}{\Delta, \Gamma \vdash \textsc{literal}(\textsc{pair}(v, v')) : \tau\ \texttt{list}} \quad (\textsc{ListLiterals2})$$

The use of a variable is well typed if the variable is bound in the environment.

$$\frac{x \in \operatorname{dom}\Gamma \qquad \Gamma(x) = \tau}{\Delta, \Gamma \vdash \textsc{var}(x) : \tau} \quad (\textsc{Var})$$

$$\frac{\Delta, \Gamma \vdash e : \tau \qquad x \in \operatorname{dom}\Gamma \qquad \Gamma(x) = \tau}{\Delta, \Gamma \vdash \textsc{set}(x, e) : \tau} \quad (\textsc{Set})$$

As in Typed Impcore, a `while` loop is well typed if the condition is Boolean and the body is well typed. The result has type `unit`.

$$\frac{\Delta, \Gamma \vdash e_1 : \texttt{bool} \qquad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash \textsc{while}(e_1, e_2) : \texttt{unit}} \quad (\textsc{While})$$

Also as in Typed Impcore, a conditional expression is well typed if the condition is Boolean and the two arms have the same type. In that case, the type of the conditional expression is the type of the arms.

$$\frac{\Delta, \Gamma \vdash e_1 : \texttt{bool} \qquad \Delta, \Gamma \vdash e_2 : \tau \qquad \Delta, \Gamma \vdash e_3 : \tau}{\Delta, \Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \tau} \quad (\textsc{If})$$

As in Typed Impcore, we allow expressions in a sequence to have any type.

$$\frac{\Delta,\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Delta,\Gamma \vdash e_n : \tau_n}{\Delta,\Gamma \vdash \text{BEGIN}(e_1,\ldots,e_n) : \tau_n} \qquad \text{(BEGIN)}$$

Also as in Typed Impcore, we give the empty BEGIN type `unit`.

$$\frac{}{\Delta,\Gamma \vdash \text{BEGIN}() : \texttt{unit}} \qquad \text{(EMPTYBEGIN)}$$

The rule for LET is straightforward; we compute the types of all the bound names, then use those types when checking the body.

$$\frac{\Delta,\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n \qquad \Delta,\Gamma\{x_1 \mapsto \tau_1,\ldots,x_n \mapsto \tau_n\} \vdash e : \tau}{\Delta,\Gamma \vdash \text{LET}(\langle x_1,e_1,\ldots,x_n,e_n\rangle,e) : \tau} \qquad \text{(LET)}$$

The rule applies equally well to the empty LET.

Typed μScheme does not have LETREC because the notation would be horrendous.

The rule for LETSTAR would be annoying to write down directly, because it involves a lot of bookkeeping for environments. Instead, we use syntactic sugar, rewriting LETSTAR as a nest of LETs.

$$\frac{\Delta,\Gamma \vdash \text{LET}(\langle x_1,e_1\rangle,\text{LETSTAR}(\langle x_2,e_2,\ldots,x_n,e_n\rangle,e)) : \tau \qquad n > 0}{\Delta,\Gamma \vdash \text{LETSTAR}(\langle x_1,e_1,\ldots,x_n,e_n\rangle,e) : \tau} \qquad \text{(LETSTAR)}$$

$$\frac{\Delta,\Gamma \vdash e : \tau}{\Delta,\Gamma \vdash \text{LETSTAR}(\langle\rangle,e) : \tau} \qquad \text{(EMPTYLETSTAR)}$$

To compute the type of a lambda abstraction we use the types of the arguments, which must be well-kinded with kind $*$. With these types added to the environment Γ, we can compute the type of the body, which then determines the type of the function.

$$\frac{\Delta \vdash \tau_i :: *, 1 \le i \le n \qquad \Delta,\Gamma\{x_1 \mapsto \tau_1,\ldots,x_n \mapsto \tau_n\} \vdash e : \tau}{\Delta,\Gamma \vdash \text{LAMBDA}(\langle x_1 : \tau_1,\ldots,x_n : \tau_n\rangle,e) : \tau_1 \times \cdots \times \tau_n \to \tau} \qquad \text{(LAMBDA)}$$

In an application, the function must have an arrow type. The types and number of actual parameters must match the types and number of formal parameters on the left of the arrow.

$$\frac{\Delta,\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n \qquad \Delta,\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau}{\Delta,\Gamma \vdash \text{APPLY}(e,e_1,\ldots,e_n) : \tau} \qquad \text{(APPLY)}$$

The interesting rules, which have no counterpart in Typed Impcore, are for type abstraction and application, which introduce and eliminate polymorphism. The TYLAMBDA introduces new type variables, which go into the kind environment Δ. Because type variables always stand for types, they have kind $*$.

$$\frac{\Delta\{\alpha_1 :: *,\ldots\alpha_n :: *\},\Gamma \vdash e : \tau}{\Delta,\Gamma \vdash \text{TYLAMBDA}(\alpha_1,\ldots,\alpha_n,e) : \forall\alpha_1,\ldots,\alpha_n.\tau} \qquad \text{(TYLAMBDA)}$$

To use a polymorphic value, one chooses the types with which to instantiate the type variables. The notation $[\alpha_1 \mapsto \tau_1,\ldots,\alpha_n \mapsto \tau_n]$ indicates the simultaneous substitution of τ_1 for α_1, τ_2 for α_2, and so on.

$$\frac{\Delta,\Gamma \vdash e : \forall\alpha_1,\ldots,\alpha_n.\tau \qquad \Delta \vdash \tau_i :: *, 1 \le i \le n}{\Delta,\Gamma \vdash \text{TYAPPLY}(e,\tau_1,\ldots,\tau_n) : \tau[\alpha_1 \mapsto \tau_1,\ldots,\alpha_n \mapsto \tau_n]} \qquad \text{(TYAPPLY)}$$

Type rules for definitions

Just as in the operational semantics, a definition can produce a new environment. Here it is a type environment, not a value environment. The process of type-checking the item and producing the new type environment is called *elaboration*. The relevant judgment has the form $\langle d, \Gamma \rangle \to \Gamma'$, which says that when definition d is elaborated in environment Γ, the new environment is Γ'.

A VAL binding uses the current environment. It is not recursive, so the name being bound is not visible during the elaboration of the right-hand side.

$$\frac{\Delta, \Gamma \vdash e : \tau}{\langle \text{VAL}(x, e), \Gamma \rangle \to \Gamma\{x \mapsto \tau\}} \qquad \text{(VAL)}$$

A VAL-REC binding, by contrast, is recursive, and it requires an explicit type. The explicit type must be the type of the right-hand side.

$$\frac{\Delta, \Gamma\{x \mapsto \tau\} \vdash e : \tau}{\langle \text{VAL-REC}(x, \tau, e), \Gamma \rangle \to \Gamma\{x \mapsto \tau\}} \qquad \text{(VALREC)}$$

The bound name x is visible during the elaboration of the right-hand side e, but for safety, x must not be *evaluated* within e until *after* e's value has been stored in x. In μScheme, the way to keep something from being evaluated is to protect it under a LAMBDA.[7] We therefore ought to check that if x is defined using `val-rec`, all uses of x are protected by LAMBDAs.[8]

271 ⟨*type checking for Typed μScheme* 264a⟩+≡ (275a) ◁267a

```
fun appearsUnprotectedIn (x, e) =
  let fun evaluatesX (LITERAL n) = false
        | evaluatesX (VAR x') = x' = x
        | evaluatesX (SET (_, e)) = evaluatesX e
        | evaluatesX (WHILEX (e1, e2)) = evaluatesX e1 orelse evaluatesX e2
        | evaluatesX (APPLY (f, actuals)) =
            evaluatesX f orelse List.exists evaluatesX actuals
        | evaluatesX (LETX (LETSTAR, [], body)) = evaluatesX body
        | evaluatesX (LETX (LETSTAR, (x', e') :: bs, body)) =
            evaluatesX e' orelse
            (x <> x' andalso evaluatesX (LETX (LETSTAR, bs, body)))
        | evaluatesX (LETX (LET, bs, body)) =
            List.exists (fn (_, e) => evaluatesX e) bs orelse
            (not (List.exists (fn (x', _) => x' = x) bs) andalso evaluatesX body)
        | evaluatesX (IFX (e1, e2, e3)) =
            evaluatesX e1 orelse evaluatesX e2 orelse evaluatesX e3
        | evaluatesX (BEGIN es) = List.exists evaluatesX es
        | evaluatesX (LAMBDA (formals, body)) = false
        | evaluatesX (TYAPPLY (e, args)) = evaluatesX e
        | evaluatesX (TYLAMBDA (alphas, e)) = evaluatesX e
  in  evaluatesX e
  end
```

APPLY	268a
BEGIN	268a
IFX	268a
LAMBDA	268a
LET	268a
LETSTAR	268a
LETX	268a
LITERAL	268a
SET	268a
TYAPPLY	268a
TYLAMBDA	268a
VAR	268a
WHILEX	268a

[7] In a lazy language like Haskell, a right-hand side is never evaluated until its value is needed, so a definition like `(val-rec int x x)` is legal, but evaluating x produces an infinite loop (sometimes called a "black hole.")

[8] This check isn't strong enough; if you are clever, you can find a way to subvert the type system and force Typed μScheme to issue the error message "bug in type checking."

If you do Exercise 13, be sure to test `not (appearsUnprotectedIn (x, e))` as a side condition. For details about exactly what happens during evaluation, see page 688 in Appendix G.

A top-level expression is syntactic sugar for a binding to `it`.

$$\frac{\langle \text{VAL}(\texttt{it}, e), \Gamma\rangle \to \Gamma'}{\langle \text{EXP}(e), \Gamma\rangle \to \Gamma'} \qquad \text{(EXP)}$$

A DEFINE is syntactic sugar for a suitable VAL-REC.

$$\frac{\langle \text{VAL-REC}(f, \tau_1 \times \cdots \times \tau_n \to \tau, \text{LAMBDA}(\langle x_1 : \tau_1, \ldots, x_n : \tau_n\rangle, e)), \Gamma\rangle \to \Gamma'}{\langle \text{DEFINE}(f, \tau, \langle x_1 : \tau_1, \ldots, x_n : \tau_n\rangle, e), \Gamma\rangle \to \Gamma'} \qquad \text{(DEFINE)}$$

Type checking

This book does not provide a type checker for Typed μScheme; its implementation is left as Exercise 13. Type checking requires a definition, a type environment, and a kind environment. Calling `elabdef`(t, Γ, Δ) should return a pair (Γ', s), where $\langle t, \Gamma\rangle \to \Gamma'$ and s is a string that represents the type of the thing defined.

272a ⟨*type checking for Typed μScheme* [[**prototype**]] 272a⟩≡

```
elabdef : def * tyex env * kind env -> tyex env * string
```

```
exception LeftAsExercise of string
fun elabdef _ = raise LeftAsExercise "elabdef"
```

6.6.7 The rest of an interpreter for Typed μScheme

Evaluation

Here is an appropriate place to dispose of the evaluation rules for Typed μScheme. The rules for expressions are exactly the same as the rules for μScheme; at run time, the types have no effect whatever. We require new rules for type abstraction and application, but the evaluator behaves exactly as if these constructs aren't there.

$$\frac{\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{TYAPPLY}(e, \tau_1, \ldots, \tau_n), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \qquad \text{(TYAPPLY)}$$

$$\frac{\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{TYLAMBDA}(\langle \alpha_1, \ldots, \alpha_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \qquad \text{(TYLAMBDA)}$$

ev 686c
TYAPPLY 268a
TYLAMBDA 268a

Most of the evaluator for Typed μScheme is just like the evaluator for μScheme in Chapter 5. The code for the two new cases acts as if `TYAPPLY` and `TYLAMBDA` aren't there.

272b ⟨*alternatives for* `ev` *for* `TYAPPLY` *and* `TYLAMBDA` 272b⟩≡ (686c)

```
| ev (TYAPPLY  (e, _)) = ev e
| ev (TYLAMBDA (_, e)) = ev e
```

The rest of the evaluator appears in Appendix G.

The rules for definitions are slightly different from those in μScheme; as described in Exercise 37 in Chapter 3, VAL must always create a new binding. Otherwise, we could subvert the type system by using VAL to change the type of an existing value. The VAL rule must use the old environment; VAL-REC uses the new one.

$$\frac{\ell \notin \operatorname{dom}\sigma \qquad \langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{VAL}(x, e), \rho, \sigma\rangle \to \langle \rho\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\}\rangle} \quad \text{(VAL)}$$

$$\frac{\ell \notin \operatorname{dom}\sigma \qquad \langle e, \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto \text{unspecified}\}\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{VAL-REC}(x, \tau, e), \rho, \sigma\rangle \to \langle \rho\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\}\rangle} \quad \text{(VAL-REC)}$$

The code that implements these rules is in Appendix G.

Processing definitions in two phases

As in Typed Impcore, we process a definition by first elaborating it (which includes running the type checker), then evaluating it. The elaborator produces a string that represents a type, and the evaluator produces a string that either is empty or represents a value. If the value string is nonempty, we print both strings.

273a ⟨*checking and evaluation for Typed μScheme* 267c⟩+≡ (275a) ◁267c 273b▷

```
checkThenEval : def * env_bundle * (string->unit) -> env_bundle
exception BugInTypeChecking of string
⟨evaluation for Typed μScheme 686c⟩
type env_bundle = kind env * tyex env * value ref env
fun checkThenEval (d, envs as (delta, gamma, rho), echo) =
  case d
    of USE filename => use readCheckEvalPrint tuschemeSyntax filename envs
     | _ =>
       let val (gamma, tystring)  = elabdef (d, gamma, delta)
           val (rho,   valstring) = evaldef (d, rho)
           val _ = if size valstring > 0 then echo (valstring ^ " : " ^ tystring)
                   else ()
       in  (delta, gamma, rho)
       end
```

elabdef 800a
type env 214
evaldef 688a
type kind 258a
streamFold 650a
tuschemeSyntax 686b
type tyex 260
USE 268b
use 222a
type value 268a

The read-eval-print loop

The read-eval-print loop is exactly as for Typed Impcore.

273b ⟨*checking and evaluation for Typed μScheme* 267c⟩+≡ (275a) ◁273a

```
readCheckEvalPrint : def stream * (string->unit) * (string->unit) -> env_bundle -> env_bundle
and readCheckEvalPrint (defs, echo, errmsg) envs =
  let fun processDef (def, envs) =
          let fun continue msg = (errmsg msg; envs)
          in  checkThenEval (def, envs, echo)
              handle IO.Io {name, ...} => continue ("I/O error: " ^ name)
              ⟨more read-eval-print handlers 274a⟩
          end
  in  streamFold processDef envs defs
  end
```

We have the same new handlers as in Typed Impcore.

274a ⟨*more read-eval-print handlers* 274a⟩≡ (273b 222b)

```
| TypeError         msg => continue ("type error: " ^ msg)
| BugInTypeChecking msg => continue ("bug in type checking: " ^ msg)
```

Initializing the interpreter

To put everything together into a working interpreter, we need an initial kind environment as well as a type environment and a value environment.

274b ⟨*initialization for Typed μScheme* 274b⟩≡ (275a) 274c▷

```
val initialEnvs =
  let fun addPrim ((name, prim, funty), (types, values)) =
        ( bind (name, funty, types)
        , bind (name, ref (PRIMITIVE prim), values)
        )
      val (types, values) = foldl addPrim (emptyEnv, emptyEnv)
                            ⟨primitive functions for Typed μScheme :: 689b⟩ nil)
      fun addVal ((name, v, ty), (types, values)) =
        ( bind (name, ty, types)
        , bind (name, ref v, values)
        )
      val (types, values) = foldl addVal (types, values)
                            ⟨primitives that aren't functions, for Typed μScheme :: 690b⟩ nil)
      fun addKind ((name, kind), kinds) = bind (name, kind, kinds)
      val kinds   = foldl addKind emptyEnv
                      ⟨primitive type constructors for Typed μScheme :: 258b⟩ nil)
      val envs    = (kinds, types, values)
      val basis   = ⟨ML representation of initial basis (automatically generated)⟩
      val defs  = reader tuschemeSyntax noPrompts ("initial basis", streamOfList basis)
  in  readCheckEvalPrint (defs, fn _ => (), fn _ => ()) envs
  end
```

kinds	: kind	env
types	: tyex	env
values	: value ref	env
envs	: env_bundle	

The code for the primitives appears in Appendix G. It is similar to the code in Chapter 5, except that it supplies a type, not just a value, for each primitive.

The function `runInterpreter` takes one argument, which tells it whether to prompt.

⟨*initialization for Typed μScheme* 274b⟩+≡ (275a) ◁274b

```
fun runInterpreter noisy =
  let fun writeln s = app print [s, "\n"]
      fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
      val prompts = if noisy then stdPrompts else noPrompts
      val defs =
        reader tuschemeSyntax prompts ("standard input", streamOfLines TextIO.stdIn)
  in  ignore (readCheckEvalPrint (defs, writeln, errorln) initialEnvs)
  end
```

bind 214
BugInTypeChecking 273a
continue 273b
emptyEnv 214
noPrompts 668d
PRIMITIVE 268a
readCheckEvalPrint 273b
reader 669
stdPrompts 668d
streamOfLines 648b
streamOfList 647c
tuschemeSyntax 686b
TypeError 275a

Putting all the pieces together

We stitch together the parts of the implementation in this order:

275a ⟨*tuscheme.sml* 275a⟩≡ (275c)

```
⟨environments 214⟩
exception TypeError of string
⟨definition of separate 218d⟩
⟨types for Typed μScheme 258a⟩
⟨lexical analysis 671a⟩
⟨abstract syntax and values for Typed μScheme 268a⟩
⟨values 216b⟩
⟨type checking for Typed μScheme 264a⟩
⟨parsing for Typed μScheme 684a⟩
⟨implementation of use 222a⟩
⟨checking and evaluation for Typed μScheme 267c⟩
⟨initialization for Typed μScheme 274b⟩
⟨command line 223d⟩
```

275b ⟨*command line* [[**type-equality**]] 275b⟩≡

```
fun errexit ss = ( TextIO.output (TextIO.stdErr, concat ss)
                 ; OS.Process.exit OS.Process.failure
                 )
fun fst (t, _) = t

fun more () = raise SyntaxError "incomplete type in argument"
fun typeOfString s = ty (toLazy more (read ("argument", ~1) s))
fun main ([t1, t2] : tyex list) =
    OS.Process.exit (if eqType (t1, t2) then OS.Process.success else OS.Process.failure)
  | main _      = errexit ["Usage: ", CommandLine.name (), " t1 t2\n"]
val _ = main (map (fst o typeOfString) (CommandLine.arguments ()))
    handle SyntaxError s => errexit ["Syntax error: ", s, "\n"]
```

275c ⟨*tuscheme-api.sml* 275c⟩≡

```
functor TUSchemeFn (TextIO : TEXT_IO) = struct
  fun print s = TextIO.output(TextIO.stdOut, s)
  ⟨tuscheme.sml 275a⟩
end
```

6.7 Type systems as they really are

Typed Impcore is a good model of a type system for a monomorphic language, but real languages are more complicated. As one example, most programming languages, especially monomorphic ones, use product types with named fields. These are often called "record" or "struct" types. The type rules are mostly straightforward; type checking involves matching names, not using positional notation.

Typed μScheme is not a good model for any widely used language. No serious language designer would use a polymorphic type system that requires explicit types everywhere, because it would force the programmer to write too much. In Typed μScheme, writing an explicit instantiation of every polymorphic value is especially burdensome. Typed μScheme is, however, a good model for an intermediate form to which a polymorphic source language can be translated. For example, the full version of Clu can be modeled using something like Typed μScheme; we hope to demonstrate this approach in a future revision of Chapter 9. Typed μScheme becomes an even better, more powerful model, if we permit `type-lambda` to quantify over a type variable of *any* kind, not just of kind ∗. This sort of parameterization is available in some versions of the functional language Haskell.

In this chapter, comparing types for equality is fairly straightforward: two types are equal if and only if they apply the same constructor to equal arguments. Quantified types are considered equal up to renaming of bound type variables, but that is a minor matter. In real languages, comparing types for equality is a more complicated business, primarily because of *generativity*. A type constructor is generative if it always creates a distinct type, different from any other type, regardless of the arguments. As an example, the product-type constructor in C, called `struct`, is generative. The sum-type constructor in ML, called `datatype`, is generative. Languages without generativity are sometimes said to compare types using *structural equivalence*; languages with generativity are said to use *name equivalence* or *occurrence equivalence*.

Many languages permit an operator to be used at more than one type, but not at infinitely many types. For example, the `+` operator in ML may be used at two types: `int * int -> int` and `real * real -> real`. Such an operator cannot be described with a quantified type; instead, it is said to be *overloaded*. As another example, the `+` operator in C is heavily overloaded; because of implicit type conversions, it may be used at many types. Most languages with overloaded operators have a fixed set of overloaded operators built in. Languages such as C++ and Ada permit a programmer to add a new definition, at a new type, of an existing overloaded operator. The functional language Haskell is even more powerful; it permits a programmer to introduce new overloaded operators. Overloading complicates a type system significantly.

Type systems used for research go far beyond what is presented in this book. In systems based on *dependent types*, type checking can be undecidable while still giving useful results in practice!

6.8 Glossary

Generativity If a language construct always creates a new type distinct from any other type, that construct is called *generative*. Examples of generative constructs include C's `struct` and ML's `datatype`.

Kind To explain how type constructors can be combined, we use *kinds*. They enable us to distinguish a nullary type constructor (base type) such as `int` from a unary type constructor such as `list`.

Monomorphic type system If every term, variable, and function in a language has at most one type, that language uses a *monomorphic* type system. For example, Pascal's type system is monomorphic. A type system may also be *polymorphic*.

Parametric polymorphism The form of polymorphism that uses type parameters and instantiation is called *parametric polymorphism*.

Polymorphic type system If a term, variable, or function in a language may be used at more than one type, that language uses a *polymorphic* type system. For example, ML's type system is polymorphic. A type system may also be *monomorphic.*

Polymorphism A language is polymorphic if it is possible to write programs that operate on values of more than one type. For example, Scheme and ML are polymorphic languages. A value is polymorphic if it can be used at more than one type; for example, the list `length` function is polymorphic because it can operate on many types of lists.

Polytype A type that can be instantiated in more than one way is called a *polytype.* In Typed μScheme, a polytype has the form $\forall\alpha_1, \ldots, \alpha_n.\tau$.

Instantiation Instantiation is the process of deciding at which type a polymorphic value should be used. In Typed μScheme, a polymorphic value is instantiated explicitly using the `@` operator. In ML, instantiation is implicit and is handled automatically by the language implementation.

Type Abstraction To create a polymorphic value in a system with parametric polymorphism, we use a `type-lambda` to introduce a type parameter. This operation is called *type abstraction.*

Type Application To instantiate a polymorphic value in a system with parametric polymorphism, we use an `@` operation to substitute a type introduce for a type parameter. This operation is called *type application.*

Type Checker In a typed language, the implementation must guarantee that a program obeys the rules of the type system. If the type system is *static*, this guarantee is provided by checking at compile time, and the relevant part of the implementation is called the *type checker.*

Type Constructor The basic unit with which types are built is called a *type constructor.* Type constructors come in various *kinds.* Some kinds, such as `int` and `bool` are types all by themselves; others, such as `list` and `array`, have to be applied to other types to make types.

Type System A language's *type system* encompasses both the set of types that can be expressed in the language and the rules that say what terms have what types.

6.9 Further reading

Pierce (2002) has written a wonderful textbook covering many aspects of typed programming languages. Cardelli (1997) presents an alternative view of type systems; his tutorial inspired some of the material in this chapter.

6.10 Exercises

6.10.1 Learning about monomorphic type systems

1. Finish the type checker for Typed Impcore so that it handles arrays. It is sufficient to give code for the four cases in code chunk 251.

2. Using an argument about the rules in the type system, prove that type checking for Typed Impcore always terminates.

3. Show that the type system for Typed Impcore is sound. In particular, show that exception `BugInTypeChecking` is never raised.

4. Prove that an expression in Typed Impcore has at most one type. That is, prove that given environments $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho$ and abstract-syntax tree e, there is at most one τ such that $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$.

5. Imagine you want to add lists to Typed Impcore using the same techniques we use for arrays. Devise new abstract syntax to support lists, and write appropriate type-formation, type-introduction, and type-elimination rules. The rules should look similar to the rules shown in Section 6.4, and to make it obvious which rules are formation rules, which rules are introduction rules, and which rules are elimination rules, the rules should be divided into three groups.

   Your abstract syntax should cover all the list primitives defined in Chapter 3: the empty list, test to see if a list is empty, `cons`, `car`, and `cdr`. Your abstract syntax may differ from the abstract syntax used in μScheme.

   Be sure your rules are deterministic: it should be possible to compute the type of an expression given only the syntax of the expression and the current type environment.

6. Give type rules for records with named fields.

7. Give type rules for sums with named variants.

8. Mutable cells can be represented by a type constructor `ref`. The appropriate operations are `ref`, `!`, and `:=`. The function `ref` is like the function `allocate` in Chapter 3; applying `ref` to a value v allocates a new mutable cell and initializes it to hold v. Applying `!` to a mutable cell returns the value contained in that cell. Applying `:=` to a mutable cell and a value replaces the contents of the cell with the value.

   Give type rules for a type constructor for mutable cells. (See also Exercise 19.)

9. Write a higher-order printing function of type `ty -> value -> string`. The function should use information about types to print values appropriately. In particular, even though all three have the same representation, Booleans should print as `#t` or `#f`, integers should print as integers, and values of type `unit` should print as `()`. Arrays should print appropriately according to the types of their elements. (Hint: define a printing function for each type constructor.)

6.10.2 Learning about Typed μScheme

10. Implement `foldl` and `foldr` in Typed μScheme.

    (a) Both functions should have the same polymorphic type. Give it.

    (b) Write an implementation of each function.

11. Implement `exists?` and `all?` in Typed μScheme.

    (a) Both functions should have the same polymorphic type. Give it.

    (b) Write an implementation of `exists?` using recursion.

    (c) Write an implementation of `all?` using your implementation of `exists?` and De Morgan's laws.

12. Use `val-rec` to define a pair of mutually recursive functions `odd?` and `even?`, which test the parity of integers. You will have to use `val-rec` to define a pair of type `(pair (function (int) bool) (function (int) bool))`, then use `fst` and `snd` to extract `odd?` and `even?` from the pair.

6.10.3 Learning about polymorphic type systems

13. Write a type checker for Typed μScheme. That is, implement `elabdef` in code chunk 272a. Although you could write this checker by cloning and modifying the type checker for Typed Impcore, you may have better luck building a checker from scratch by following the type rules for Typed μScheme.

    When you type-check VAL-REC(x, τ, e), be sure to check that x is not evaluated in e, as described on page 271.

    When you type-check literals, use the rules on page 269. Although these rules are incomplete, they should suffice for anything the parser can produce. If a literal `PRIMITIVE` or `CLOSURE` reaches your type checker, the impossible has happened, and your code should raise an appropriate exception.

14. Function `appearsUnprotectedIn` on page 271 is not conservative enough. Subvert your type checker from Exercise 14 by using `val-rec` to write a program that defines `x` and then evaluates `x` before the definition is complete.

    Hint: Try defining `x` to have type `int` and value `NIL`. Then add 1 to `x`.

15. Suppose we get sick and tired of writing `@` signs everywhere, so we decide to extend Typed μScheme by making PAIR, FST, and SND abstract syntax instead of functions.

    (a) What is the type of the following function?

    ```
    (type-lambda ('a) (type-lambda ('b)
       (lambda (((pair a b) p)) (pair (snd p) (fst p)))))
    ```

    (b) Using the type rules from the chapter, give a derivation tree proving the correctness of your answer to part 15a.

16. A great advantage of a polymorphic type system is that the language can be extended without touching the abstract syntax, the values, the type checker, or the evaluator. Without changing any of these parts of Typed μScheme, extend Typed μScheme with a `queue` type constructor and the polymorphic values `empty-queue`, `empty?`, `put`, `get-first`, and `get-rest`. (A more typical functional queue provides a single `get` operation which returns a pair containing both the first element in the queue and any remaining elements. If instead you write the two functions `get-first` and `get-rest`, you won't have to fool with pair types.)

    (a) What is the kind of the type constructor `queue`? Add it to the initial Δ for Typed μScheme.

    (b) What are the types of `empty-queue`, `empty?`, `put`, `get-first`, and `get-rest`?

    (c) Add them to the initial Γ and ρ of Typed μScheme. You will need to write implementations in ML.

17. *Without* changing the abstract syntax, values, type checker, or evaluator of Typed μScheme, extend Typed μScheme with the `pair` type constructor and the polymorphic functions `pair`, `fst`, and `snd`.

    (a) What is the kind of the type constructor `pair`? Add it to the initial Δ for Typed μScheme.

    (b) What are the types of `pair`, `fst`, and `snd`?

    (c) Add them to the initial Γ and ρ of Typed μScheme. As you add them to ρ, you can use the same implementations that we use for `cons`, `car`, and `cdr`.

18. *Without* changing the abstract syntax, values, type checker, or evaluator of Typed μScheme, extend Typed μScheme with the `sum` type constructor and the polymorphic functions `left`, `right`, and `either`.

    (a) What is the kind of the type constructor `sum`? Add it to the initial Δ for Typed μScheme.

    (b) What are the types of `left`, `right`, and `either`?

    (c) Page 252 gives algebraic laws for pair primitives in a monomorphic language. If the sum primitives were added to a monomorphic language, what would be the laws relating LEFT, RIGHT, and EITHER?

    (d) Since `left`, `right`, and `either` have the polymorphic types in part 18b, what are the laws relating them?

    (e) Add `left`, `right`, and `either` to the initial Γ and ρ of Typed μScheme. Try representing a value of sum type as a `PAIR` containing a tag and a value.

19. In Typed μScheme, it is not necessary to add any special abstract syntax to support mutable cells as in Exercise 8. Give the kind of the `ref` constructor and the types of the operations `ref`, `!`, and `:=`.

20. The `ref` constructor, as described in Exercise 19, leads to unsoundness: it is possible to subvert the type system. The key technique is to the `ref` operations at a *polymorphic* type. If you have a polymorphic mutable cell, you can instantiate the `ref` operation at one type and the `!` operation at a different type, enabling you to write a function that, for example, converts a Boolean to an integer.

    (a) Using polymorphic references, trigger the `BugInTypeChecking` exception by adding 1 to `#t`.

    (b) Write an *identity* function of type `bool` $\rightarrow$ `int`

    (c) Write a polymorphic function of type $\forall\alpha, \beta.\alpha \rightarrow \beta$.

    (d) Close this dreadful loophole in the system by making `ref` abstract syntax and permitting it to be instantiated only at monotypes.

21. Extend Typed μScheme by adding polymorphic structures with named fields.

    (a) Add a new kind of definition. It should have this concrete syntax:

    def ::= `(define-struct` *struct-name* $\{$*field-name*$\}$`)`

    The abstract syntax can be this:

    ```
    DEFINE_STRUCT of name * name list
    ```

    (b) Modify `elabdef` so that it returns a new kind environment Δ' as well as a new type environment Γ'.

    (c) The `define-struct` should extend Δ by adding a new type constructor *struct-name* with kind $*_1 \times \cdots \times *_n \Rightarrow *$, where n is the number of *field-name*s.

    (d) The `define-struct` should add a new function with the same name as *struct-name*. This function should take one argument for each field and build an instance of the structure.

    (e) For each field, the `define-struct` should add a function named by joining the *struct-name* and *field-name* with a dash. This function should extract the named field from the structure and return its value.

Here is an example.

281 ⟨*exercise transcript* 281⟩≡ 282▷

```
-> (define-struct assoc key val)
-> (val p ((@ assoc sym int) 'class 152))
p : (assoc sym int)
-> ((@ assoc-key sym int) p)
class : sym
-> ((@ assoc-val sym int) p)
152 : int
```

22. Extend Typed μScheme by more flexible structures with named fields, which may or may not be polymorphic.

    (a) Add a new kind of definition. It should have this concrete syntax:

    def ::= `(define-typed-struct (`*struct-name* { '*type-variable-name* }) ({ (*type field-name*) }))`

    The idea is that the type parameters are listed explicitly, and the type of each field is declared in advance. Because of the type variables following the structure name, however, the new type can still be polymorphic.

    The abstract syntax can be this:

    ```
    DEFINE_TYPED_STRUCT of name * name list * (tyex * name) list
    ```

    (b) Modify `elabdef` so that it returns a new kind environment Δ' as well as a new type environment Γ'.

    (c) The `define-typed-struct` should extend Δ by adding a new type constructor *struct-name* with kind $*_1 \times \cdots \times *_n \Rightarrow *$, where n is the number of *type-variable-names* in the definition.

    (d) The `define-typed-struct` should add a new function with the same name as *struct-name*. This function should take one argument for each field and build an instance of the structure.

    (e) For each field, the `define-typed-struct` should add a function named by joining the *struct-name* and *field-name* with a dash. This function should extract the named field from the structure and return its value.

    Here is an example.

282 ⟨*exercise transcript* 281⟩+≡ ◁281

```
-> (define-typed-struct (assoc 'a) ((sym key) ('a val)))
-> (val p ((@ assoc int) 'class 152))
p : (assoc int)
-> ((@ assoc-key int) p)
class : sym
-> ((@ assoc-val int) p)
152 : int
```

23. Extend Typed μScheme with type abbreviations and recursive algebraic types modeled on ML. For example:

```
(type (node a t) ((t left) (t right) (a label)))
(rectype (tree a)
   (LEAF a)
   (NODE (node a (tree a))))
```

This extension will require adding a `case` expression to Typed μScheme.