Chapter 10

Typelang: A Language with Types

In this chapter, you will learn about types, type system, type checking, type inference and related concepts. The code for the interpreter discussed in this chapter is available in the directory `typelang`. If you have programmed in languages like C, C++, Java, C#, etc, you have already enjoyed the benefits of types in your programs. This chapter presents a systematic study of the foundation behind types and related facilities.

10.1 Why Types?

To motivate, let us consider the procedure `f` below.

```
(let ((f (lambda (x) (x 2))))
  ...
)
```

For this program, it is natural to ask *does the procedure f always run correctly?* After some deduction we may conclude that the answer is: no. The procedure `f` may not always run correctly due to one or more of the following reasons.

- its argument `x` may *not* be a procedure.
- `x` may *not* take 1 argument.
- `x`'s first argument may *not* be a numeric value.

For example, a programmer may inadvertantly write the following call expression

```
(f 2)
```

in the body of the let expression causing a runtime error. Runtime errors are better than the alternative (incorrect output), because they help us identify when a program has failed to perform as desired, but they can still lead to user dissatisfaction and in some cases critical failures[1]. Ideally we would like to detect and prevent as many runtime errors as we can.

In this setting, if we wanted to prevent all such runtime errors in usage of the procedure `f`, we can't just look at the procedure `f` by itself and understand what it is doing. Rather we must analyze every usage of `f` to figure out if `f` will work correctly. For example, in the listing below we might consider `f` to be incorrect, or the usage of `f` to be incorrect, or the entire program to be incorrect (depending on our personal preference).

```
(let ((f (lambda (x) (x 2))))
  (f 2)
)
```

What is basically wrong about this program?

1. The procedure expects data from its clients to satisfy certain contract. *Code that calls the procedure is a client of the procedure.*

2. The contract between clients and the procedure is missing, or at least implicit. Since the contract is implicit we cannot reliably answer: *who is to blame for the entire program going wrong?*

3. The contract of the procedure is not satisfied by its clients.

[1]The reader is encouraged to think about the impact of runtime errors in software that goes in systems like the pacemaker, radiation therapy machines, rockets, etc...

1. Leveson, Nancy G., and Turner, Clark S., "*An Investigation of the Therac-25 Accidents*," IEEE Computer, July 1993.

2. Lions, J. L. *et al.*, "*ARIANE 5: Flight 501 Failure, A Report by the Inquiry Board*, July 1996. Excerpts: *The internal SRI* software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.*

10.2 Kinds of Specifications

We will discuss the notion of contracts in full generality in chapter 11, but at the moment let us focus on contracts for procedures. What do we mean by contract of the procedure `f`? Our meaning is identical to the use of the phrase *contract between two parties* in non-programming settings. Main purpose of a procedure's contract in a program is to:

1. allow to divide the labor (what is this procedure's responsibility and what is client's responsibility),
2. allow to understand a procedure by looking at the procedure code and its contract, and
3. to allow blame assignment (when is client to blame/procedure to blame).

For our example, two parties are (1) procedure `f`, and (2) a client of the procedure `f`. We are concerned about two aspects:

1. The procedure's expectations from the client, and
2. its promise to the client.

What are the expectations of the procedure `f` in our example from any of its clients?

1. Clients must pass a procedure as argument
2. Furthermore, that procedure must be applicable to a number.

What is the promise of the procedure `f` to clients?

> If invoked properly using a procedure `x` as argument it will return a value that is the value returned by the procedure `x` passed as argument when invoked with `2` as a parameter.

This is still not completely concrete, but we have made progress.

In the previous example, if the programmer who wrote the code for `f` had written the code of the procedure as follows (added a comment) then they can clearly blame the client of `f` for not calling the procedure properly.

```
(let
  (
    (f
      (lambda (x /* x is a procedure that takes 1 numeric argument.*/)
        (x 2)
      )
    )
  )
  (f 2)
)
```

The newly added comment is a kind of *informal contract* between the client and the procedure. You will often hear the terms “contract” and “specification” used interchangeably. Specifications help eliminate unsafe computation, e.g. runtime errors that we discussed previously, but they also help elucidate what is desired of a computation. There are several kinds of specification.

1. Ultra-lightweight specifications: *Types*. These will be the subject of much of the discussion in this chapter. Most types are stated by the programmer writing program annotations and associating them with pieces of code that produce values.

2. Lightweight specifications: allow mixing mathematical and programmatic specifications. For example, in Eiffel language, JML for Java, Spec# for C#, CodeContracts for C#, etc... JML, for instance, is a behavioral interface specification language.

3. Fully mathematical specification: precise mathematical description of programs. For example, Z language and its object-oriented extension Z++, Object-Z, the Alloy language, etc...

Typically expressiveness of specifications goes up when we go from types to fully mathematical specification languages, i.e. more complex properties of programs can be represented. The cost is related, in that the cost of specifying programs also goes up from types to fully mathematical specification languages. In this chapter, we will focus on types (chapter 11 discusses specification languages in greater detail).

10.3 Types

In a nutshell, types divide values manipulated by programs into kinds[2]. For example, all values of numeric kind, boolean kind, string kind, etc. When written out explicitly as annotations, types can also be thought of as lightweight specification of the contracts between producers and consumers of values in a program. When we say that "a certain value has type `T`" we implicitly state that any operation that is considered valid for type `T` is also valid for that value. When we say that "certain expression has type `T`" we also implicitly state that all values produced by the expression would have the type `T`.

This division of program values into types provides several new opportunities

- *abstraction.* Instead of thinking in terms of concrete values, we can think in terms of types, which hides concrete details of values.
- *performance.* The explicit division of values into types allow language implementations to utilize type information in selecting proper routines to handle a set of values.
- *documentation.* When used as program annotations types also serve as excellent source of documentation[3]. For example, writing `x : num` (read as `x` has type `num`) is much more descriptive compared to writing just `x`.
- *verification.* Types can also be utilized to declare certain programs as illegal without observing concrete runtime errors in those programs. For example, we can declare the program `(+ n s)` as illegal by knowing that `n:num`, `s:String`, and that `+` accepts two numeric values.

[2] There is much controversy surrounding definitions of types, so we would not venture into providing yet another definition of types, but encourage you to look at some previous attempts.

- Benjamin Pierce, "*Types and Programming Languages*," The MIT Press (February 1, 2002).
- John C. Reynolds, "*Fable on Types*," 1983.

[3]Although they still need to supplemented by proper source code comments, and other more detailed documentation.

In the rest of this chapter, we will build support for types. We will create an extension of the Reflang programming language of chapter 7. We will call this extension *Typelang*. We will build Typelang as an explicitly-typed language, i.e. types will appear as syntactic annotations in user programs.

10.4 Adding Type Annotations

The core set of features in Typelang are presented in figure 10.1. In previous chapters, we have omitted the discussion of the full grammar, but for Typelang since some of the top-level elements will also change, we will review the full grammar to observe all syntactic changes.

program	::=	*definedecl* exp?*	*Programs*
definedecl	::=	(**define** *identifier* : *T exp*)	*Declarations*
exp	::=		*Expressions*
	|	*varexp*	*Variable expression*
	|	*numexp*	*Number constant*
	|	*addexp*	*Addition*
	|	*subexp*	*Subtraction*
	|	*multexp*	*Multiplication*
	|	*divexp*	*Division*
	|	*letexp*	*Let binding*
	|	*lambdaexp*	*Function creation*
	|	*callexp*	*Function Call*
	|	*letrecexp*	*Letrec*
	|	*refexp*	*Reference*
	|	*derefexp*	*Dereference*
	|	*assignexp*	*Assignment*
	|	*freeexp*	*Free*

Figure 10.1: Syntax of core elements of Typelang (omits some expressions presented in figure 10.7)

First change appears in the syntax of the define declarations, where a new non-terminal *T* is added. This non-terminal is defined in figure 10.2.

According to the grammar in figure 10.2 there are four kinds of types: unit types, number types, boolean types, function types, and reference types. The unit, number, and boolean types are *base types*, i.e. their definition does not consist of other types. The function and reference types are recursively-defined types, i.e. their definition makes use of other types.

T	::=		*Types*
	|	`unit`	*Unit Type*
	|	`num`	*Number Type*
	|	`bool`	*Boolean Type*
	|	`(` T^* `->` T `)`	*Function Type*
	|	`Ref` T	*Reference Type*

Figure 10.2: Basic types in Typelang

The following listing shows some example define declarations that make use of each of these types. These examples also introduce new syntax for other expressions, but at the moment let us focus on the syntax of define declarations and new types.

```
$ (define pi : num 3.14159265359)
$ (define r : Ref num (ref : num 2))
$ (define u : unit (free (ref : num 2)))
$ (define iden : (num -> num) (lambda (x : num) x))
```

First line says that the programmer's intent is to define `pi` as a variable of type `num`. Similarly, second line says that `r` is a variable of type `Ref num`, i.e. reference to a number, third line says that `u` is a variable of `unit` type, and fourth line says that `iden` is a variable of function type `(num->num)`, i.e. a function that accepts a number as argument and returns a number as result.

Given these definitions, we can clearly observe the difference between the earlier define form `(define pi 3.14159265359)` and this new form. For instance, in the new form the annotation `num` on the first line acts as a contract between the consumers of this definition `pi` and the expression to its right (`3.14159265359`). If consumers of `pi` expect this variable to have a value other than `num`, they are to blame. If expression in the define declaration provides a non-numeric value for `pi`, that expression is to blame. For example, in the following listing `id` is declared to be a variable of function type `(num->num)`, i.e. a function that accepts a number as argument and returns a number as result, but it is being assigned a function value that is of function type `((num->num)->(num->num))` and therefore the lambda expression is incorrect.

```
$ (define id : (num -> num) (lambda (x : (num -> num)) x))
```

Following listing shows other examples of incorrect programs.

```
$ (define fi : num "Hello")
$ (define f : Ref num (ref : bool #f))
$ (define t : unit (free (ref : bool #t)))
```

Since Typelang extends Reflang, it has all the standard expressions as shown in figure 10.1. Among these expressions, the syntax for `varexp`, `numexp`, `addexp`, `subexp`, `multexp`, and `divexp` does not change to include type annotations. The syntax for `numexp`, `addexp`, `subexp`, `multexp`, and `divexp` doesn't change because it is clear from the intended semantics of these expressions that they produce numeric values, therefore, additional type annotations would be superfluous. On the other hand, syntax for `varexp` does not include a type annotation because the variable expression, by itself cannot offer any guarantees about its valuation.

10.5 Checking Types

As mentioned in section 10.3 one of the important benefits of types is in verifying programs — essentially identifying those programs that this particular notions of types believes could lead to runtime errors.

Typechecking and Type-System

The process of verifying a program using types is often referred to as *type-checking*. This process is aided by a logical system of rules referred to as the *type-system*. Since types can be thought of as lightweight contracts between producers and consumers of values, the rules of a type system specify how that contract works for different language features. Similar to legal contracts, the system of rules must cover all eventualities.

Given a type-system, a program is said to be *well-typed* if it does not report any *type error*. An *ill-typed* TypeLang program is a program that fails typechecking. Intuitively, a type error is a situation where one of the rules of the type-system cannot be satisfied by the program.

In this section, we will gradually build a type-system for Typelang. The goal of this type-system will be prevent certain kinds of dynamic errors from showing up at runtime, rather we will detect and remove such errors during typechecking.

Logical Rules

A type-system is collection of rules. Informally, like any logic system there are two kinds of rules: those that unequivocally assert a fact, and those that imply a fact if some other assertion holds.

In the rest of this discussion, we will use the following notation to represent first kind of rules.

(Fact A)
$$A$$

We will use the following notation to represent second kind of rules.

(B if A)
$$\frac{A}{B}$$

This form is read as: if the assertion above the line will hold, then it implies that the assertion below the line will hold.

Sometimes we will also use the following notation to represent second kind of rules.

(C if A and B)
$$\frac{A \qquad B}{C}$$

This form is read as: if the assertions A and B above the line will hold, then it implies that the assertion below the line will hold.

10.6 Typechecking Rules for Constants

We will now begin developing the type-system of Typelang. Since the type-system must be capable of handling any arbitrary Typelang program, it must support each expression in the language. For example, the type-system of Typelang could unequivocally assert that all numeric values (constants) have type `num`. In the following n is a `NumVal`.

$$\begin{array}{c}(\textsc{Num})\\ n : \texttt{num}\end{array}$$

This assertion says that all numbers will be thought of as having type `num`. Recall that (Num) represents the name of the rule, and the term before ':' can be an expression, value, or program and the term after ':' is the type of the expression, value or program.

The notation *term* : T should be read as *term* has type T.

Similarly, the type-system of Typelang could unequivocally assert that all boolean values (constants) have type `bool`. In the following b is a `BoolVal`.

$$\begin{array}{c}(\textsc{Num})\\ b : \texttt{bool}\end{array}$$

We could have easily enumerated the set of boolean values in the definition above. We have other kinds of constants e.g. Strings, and later sections e.g. section 10.13 will enhance the type-system being developed here to add support for those constants. At the moment, let us focus on this smaller set for ease of presentation.

10.7 Typechecking Rules for Atomic Expressions

Next, we will introduce typechecking rules for atomic expressions, i.e. those expressions that do not have subexpressions. In the subset of Typelang that we are considering at the moment, only such expression is a numeric expression (*NumExp n*). The type-system of Typelang asserts that all numeric expressions (NumExp) have type `num`. In the following n is a `NumVal`.

$$\begin{array}{c}(\textsc{NumExp})\\ (\textit{NumExp}\ n) : \texttt{num}\end{array}$$

This assertion says that the type `num` is a contract between producer, which in this case is an expression of kind `NumExp`, and consumers of that expression. This contract says that the producer always promises to produce values of type `num`, and that any consumer cannot expect any other value besides a `num`.

In the case of (*NumExp n*), it is easy to see that the producer expression will never violate the contract. Also, notice that the assertion that (*NumExp n*) has type `num` isn't dependent on any other conditions (besides the fact that n is a `NumVal` that is already checked by the parser).

10.8 Typechecking Rules for Compound Expressions

The typechecking rules that we have seen so far are unconditional assertions. Now, consider an addition expression that adds the value of arbitrary subexpressions. Can we say that this addition expression will always produce values of type `num`? Clearly we will not be able to assert that without relying on the subexpressions. We might be able to make a different *conditional* assertion: if subexpressions of the addition expression always produce values of type `num`, then the addition expression will produce a value of type `num`. The typechecking rule below models this intent.

$$\text{(A}\textsc{dd}\text{E}\textsc{xp}\text{)}\quad \frac{e_i : \texttt{num}, \forall i \in 1..n}{(\mathit{AddExp}\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}$$

The rule above can be read as: if subexpressions e_0 to e_n have type `num`, then the expression (*AddExp* $e_0, e_1, \ldots, e_n$) will have type `num` also. This typechecking rule establishes a contract between producers of values in this context (expressions $e_0, e_1, \ldots, e_n$) and the consumer of these values (the addition expression). It also clearly states the conditions under which the addition expression is going to produce a numerical value. Notice that the rule does not mention situations where expressions $e_0, e_1, \ldots, e_n$ might produce a dynamic error. Such situations are implicitly covered. If expressions $e_0, e_1, \ldots, e_n$ fail to produce a numerical value the addition expression provides no guarantees.

We can similarly state rules for multiplication, subtraction, and division expressions.

$$\text{(M{\scriptsize ULT}E{\scriptsize XP})}\quad \frac{e_i : \texttt{num}, \forall i \in 1..n}{(\textit{MultExp}\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}$$

The typechecking rule for multiplication can also be read similarly, e.g. if subexpressions e_0 to e_n have type `num`, then the multiplication expression $(\textit{MultExp}\ e_0, e_1, \ldots, e_n)$ will have type `num` also. The rule for subtraction and division are similarly stated below.

$$\text{(S{\scriptsize UB}E{\scriptsize XP})}\quad \frac{e_i : \texttt{num}, \forall i \in 1..n}{(\textit{SubExp}\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}$$

$$\text{(D{\scriptsize IV}E{\scriptsize XP})}\quad \frac{e_i : \texttt{num}, \forall i \in 1..n}{(\textit{DivExp}\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}$$

The rule for division deserves special mention because division expressions can produce divide-by-zero errors. The rule above does not account for divide-by-zero errors. This is an example of a situation where the type-system being developed is insufficient to detect and remove certain classes of errors, e.g. the divide-by-zero errors. The reader is encouraged to think about mechanisms to check whether a subexpression of the division expression can evaluate to zero without evaluating the subexpression.

Exercise

10.8.1. *[Eliminate Simple Divide-By-Zero Errors]* For some expressions such as `(/ x 0)`, where `0` appears as an immediate subexpression it is easy to check and eliminate divide-by-zero errors. Enhance the typechecking rule for the division expression above so that the type-system is able to detect and remove such errors, where `0` is an immediate subexpression of the division expression.

10.9 Types for Variables

The typechecking rules that we have seen so far are about expressions that do not directly define and use variables. For our very first language, `Arithlang` these rules would work well, but all of our later languages support variable definition and usage, so it is important to enhance these rules to account for that.

What should be the type of a variable expression `x`? What should be a typechecking rule for a variable expression?

In the context of Varlang, we asked a similar question. What should be the value of a variable expression `x`? The answer was: it depends on what the surrounding context of `x` says. We then modeled that surrounding context as a data type *environment*. An environment is a map that provides an operation to look up the value of a variable. We saw both functional and object-oriented implementation of environments in previous chapters. Going forward, we will refer to the environments in which values are stored as "*value environments*" to distinguish them from another kind of environments that we will introduce shortly.

For types we can use a similar strategy. The type of a variable expression `x` is dependent on the surrounding context of `x`. The surrounding context is modeled as a data type *type environment*. A type environment is a map that provides an operation to look up the type of a variable. The definition below models this intent.

$$\texttt{get(tenv, v')} = \begin{cases} Error & tenv = \texttt{(EmptyEnv)} \\ t & tenv = \texttt{(ExtendEnv v t tenv')} \\ & \texttt{and } v = v' \\ \texttt{get(tenv', v')} & \text{Otherwise.} \end{cases}$$

Here, `v`, `v'` $\in$ `Identifier`, the set of identifier, `t` $\in$ `T`, the set of types in the Typelang language, and `tenv`, `tenv'` $\in$ `TEnv`, the set of type environments. As before, take the notation `(EmptyEnv)` to mean a type environment constructed using a constructor of type `EmptyEnv`, and take `EmptyEnv` to mean all such elements, i.e. the entire set. Similarly, `(ExtendEnv v t tenv)` is a type environment constructed using a constructor of type `ExtendEnv`, with `var`, `t`, `tenv` being values used to construct this type environment.

This definition provides a single operation `get` to lookup the type of a variable from a type environment. The definition says that looking any variable in an empty type environment leads to error. Furthermore, it says that looking up a variable in an extended type environment constructed using `var`, `t`, `tenv` is the type `t` if variable being searched is the same as `var`, otherwise it is the same as looking up the variable in `tenv`.

With this type environment, we are now ready to state the rule for typechecking a variable usage.

$$\text{(VarExp)}\quad \frac{get(tenv, var) = t}{tenv \vdash (\mathit{VarExp}\ var) : t}$$

The rule above makes use of a new notation $tenv \vdash e : t$. This should be read as assuming the type environment `tenv`, the expression `e` has type `t`. The overall rule above says that in the context of the type environment `tenv` the variable expression with variable `var` has type `t`, if looking up name `var` in the type environment `tenv` gives the result `t`.

Since our previous typechecking rules did not make use of the type environment, we will also need to enhance them accordingly. These enhanced versions are presented below.

$$\text{(Num)}\quad tenv \vdash n : \texttt{num}$$

$$\text{(Num)}\quad tenv \vdash b : \texttt{bool}$$

$$\text{(NumExp)}\quad tenv \vdash (\mathit{NumExp}\ n) : \texttt{num}$$

For the three rules above, the type environment doesn't play a major role because values produced (and thus types) of these expressions are not dependent on the context.

$$\text{(AddExp)}\quad \frac{tenv \vdash e_i : \texttt{num}, \forall i \in 1..n}{tenv \vdash (\mathit{AddExp}\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}$$

For the addition expression, only noteworthy aspect of the typechecking rule is that the type environment used to perform typechecking of subexpressions is the same as that of the addition expression. In other words, variables and their types stored in the type environment are not affected by the addition expression[4].

$$\text{(MultExp)}\quad \frac{tenv \vdash e_i : \texttt{num}, \forall i \in 1..n}{tenv \vdash (\textit{MultExp}\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}$$

$$\text{(SubExp)}\quad \frac{tenv \vdash e_i : \texttt{num}, \forall i \in 1..n}{tenv \vdash (\textit{SubExp}\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}$$

$$\text{(DivExp)}\quad \frac{tenv \vdash e_i : \texttt{num}, \forall i \in 1..n}{tenv \vdash (\textit{DivExp}\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}$$

The typechecking rules for multiplication, subtraction, and division are also similar and pass around the typing environment to rules that typecheck their subexpressions without inserting or removing any variables.

Type Annotations for Let Expressions

TypeLang requires a programmer to specify types of identifiers of a let expression, as shown in the syntax below. In a later section we will look at variations in which some of these types can be omitted. In a let expression each declared identifier has a type as shown in figure 10.3.

letexp	::=	(**let** ((*identifier* : *T* exp)$^+$) *exp*)	*Let expression*

Figure 10.3: Syntax of the Let expression in Typelang

To illustrate, consider the following let expression that declares a variable `x` of type number with value 2.

```
(let
  ((x : num 2))
  x
)
```

[4]The reader is encouraged to consider whether the type environment would remain the same while checking subexpressions of e_i.

The type annotation `: num` establishes a contract between producers and consumers of the value `x`. In this case, the consumer isn't necessarily dependent on that contract.

Similarly, in the following listing the let expression declares two variables `x` and `y` both of type `num` with values `2` and `5` thereby establishing a contract between producers of those values and their consumer (that is the addition expression). The consumer expression in this case relies on this contract to show that `(+ x y)` will produce a numeric value (by applying the (AddExp) typechecking rule described above).

```
(let
  ((x : num 2)
   (y : num 5))
  (+ x y)
)
```

However, a variation of the previous let expression shown in the listing below fails to typecheck.

```
(let
  ((x : num 2)
   (y : bool #t))
  (+ x y)
)
```

Clearly, an addition expression cannot add a number and a boolean. More formally, this let expression fails to typecheck because the let expression can only guarantee to the addition expression that `x` will be a `num` and `y` will be a `bool`, but the typechecking rule for the addition expression expects that the type of both `x` and `y` to be `num`. This mismatch results in a type error thrown by the (AddExp) rule.

To summarize, just like the define form, in a let expression the type annotation on identifier such as `num` in the previous two examples act as a contract between the consumers of this variable definition (the body of the let expression) and the expression to its right. If consumers of `x` expect this variable to have a value other than `num`, they are to blame. If expressions in the let expression provides a non-numeric value for `x`, that expression is to blame. The type-system can detect and prevent these runtime errors.

Typechecking Rules for Let Expressions

Checking a let expression can be thought of as a two step process.

1. Check whether variables are being assigned a compatible value. Following code has an example error that would be caught during this check.

   ```
   (let
     ((x : num #t))
        ...
   )
   ```

   The type annotation claims that `x` is a `num` and therefore assigning a value of `bool` is illegal.

2. Assuming that the variables have their declared type, checking whether the usage of those variables in the body of the let expression is compatible with their declared types. Following code presents an example error that would be caught during this check.

   ```
   (let
     ((x : num 2)
      (y : bool #t))
       (+ x y)
   )
   ```

The typechecking rule for let expression stated below includes both of these checks.

(LetExp)

$$\frac{\begin{array}{c} tenv \vdash e_i : t_i, \forall i \in 0..n \\ tenv_n = (ExtendEnv\ var_n\ t_n\ tenv_{n-1})\ \ldots \\ tenv_0 = (ExtendEnv\ var_0\ t_0\ tenv) \\ tenv_n \vdash e_{body} : t \end{array}}{tenv \vdash (LetExp\ var_0\ \ldots\ var_n\ \ t_0\ \ldots\ t_n\ \ e_0\ \ldots\ e_n\ \ e_{body}) : t}$$

The rule above also makes use of a new notation $tenv \vdash e : t$. Recall that this notation should be read as: assuming the type environment `tenv`, the expression `e` has type `t`. Informally the rule says that a let expression that is defining n variables ($var_0 \ldots var_n$), giving them types ($t_0 \ldots t_n$), and assigning them values of certain expressions ($e_0 \ldots e_n$) has type t if

- each value expression (e_i) has the same type as the declared type (t_i) of corresponding variable (var_i), and
- in the context of a new type environment that extends the type environment *tenv* with new bindings from variables (var_i) to their types (t_i), the body of the let expression has type t.

10.10 Types for Function and Calls

Function definition and calls are two language features where the idea of *types are contract* is more explicit. The type for a function is a contract between the body of the function (consumer of parameter values and producer of the result value) and the callers of the function (producers of parameter values and consumer of the result value). The duality in this notion of contracts naturally leads to two components of a function type: `(` T^* `->` T `)`.

Type Annotations for Lambda and Call Expressions

TypeLang requires a programmer to specify the type of the argument in a lambda expression. The type of the argument in a lambda expression is specified after `':'` in its declaration, as shown in the syntax below.

lambdaexp	::=	(**lambda** ({*identifier* : *T*}*) *exp*)	*Lambda*

Figure 10.4: Syntax of the Lambda expression in Typelang

A lambda expression must be of a function type, as specified in the syntax in figure 10.2. A function type specifies the types of function's arguments as well as return type of the function. In a function type ($\mathtt{t}_a$`->`$\mathtt{t}_r$), $\mathtt{t}_a$ is the argument type and $\mathtt{t}_r$ is the return type. The cardinality of arguments of a lambda expression must match the cardinality of the argument types in the function type. In other words, each arguments of a lambda expression must have a corresponding type in the function's type.

To illustrate, consider the following lambda expression declares a function with three arguments `x`, `y` and `z` and returns their sum.

```
(lambda
  (
    x : num      //Argument 1
    y : num      //Argument 2
    z : num      //Argument 3
  )
  (+ x (+ y z))
)
```

The type for this lambda expression is (*num num num* $->$ *num*) which specifies types of arguments `x`, `y` and `z` as numeric type `num` and specifies the function's return type as `num` as well. This lambda expression type checks in TypeLang.

As another example, the following call expression declares the same function above and then calls it by passing integer parameters `1`, `2` and `3` for arguments `x`,`y` and `z`.

```
(
  (lambda
    (
      x : num      //Argument 1
      y : num      //Argument 2
      z : num      //Argument 3
    )
    (+ x (+ y z))
  )
  1 2 3
)
```

To typecheck a call expression we have to typecheck the function being called, the number and types of actual parameters, and check the correspondence between the types of the formal parameters and the corresponding type of the actual parameter. The type for the lambda expression in this call expression is (*num num num* $->$ *num*), and the function is being invoked on three actual parameters each with `num` type. Since the number of type of actual parameters and formal parameters agree, this call expression type checks in TypeLang, and the type of the overall call expression is the return type of function (`num`). However, a variation of the call expression shown below would not type check because `#t` is of type bool and not of type number the function expects for `z`.

```
(
  (lambda
    (
      x : num        //Argument 1
      y : num        //Argument 2
      z : num        //Argument 3
    )
    (+ x (+ y z))
  )
  1 2 #t
)
```

More formally, this call expression is *ill-typed* because the type for the lambda expression in this call expression is $(num\ num\ num -> \ num)$, and the function is being invoked on three actual parameters with type `num`, `num`, and `bool`.

Typechecking Rules for Lambda and Call Expressions

The rule for checking lambda expressions and calls are similar to that for checking the let expresssion.

(LambdaExp)

$$\frac{\begin{array}{c} tenv_n = (ExtendEnv\ var_n\ t_n\ tenv_{n-1}) \ldots \\ tenv_0 = (ExtendEnv\ var_0\ t_0\ tenv) \\ tenv_n \vdash e_{body} : t \end{array}}{tenv \vdash (LambdaExp\ var_0\ \ldots\ var_n\ \ t_0\ \ldots\ t_n\ \ e_{body}) : (t_0\ \ldots\ t_n -> t)}$$

The rule above also makes use of a new notation $tenv \vdash e : t$. Recall that this notation should be read as: assuming the type environment `tenv`, the expression `e` has type `t`. Informally the rule says that a lambda expression that is defining a function with n formal parameters $(var_0\ \ldots\ var_n)$, and giving them types $(t_0\ \ldots\ t_n)$ has type $t_0\ \ldots\ t_n -> t$ if

- in the context of a new type environment that extends the type environment *tenv* with new bindings from variables (var_i) to their types (t_i), the body of the lambda expression has type t.

The rule for the call expression is the consumer of a function type.

$$\frac{\begin{array}{c}\text{(CALLEXP)}\\ tenv \vdash e_f : (t_0 \ \dots \ t_n -> t) \\ tenv \vdash e_i : t_i, \forall i \in 0..n\end{array}}{tenv \vdash (\mathit{CallExp}\ e_f\ e_0\ \dots\ e_n) : t}$$

Informally the rule says that a call expression that is calling a function with n formal parameters of types ($t_0 \ \dots \ t_n$) and return type (t) — all encoded in the function type $t_0 \ \dots \ t_n -> t$ — typechecks if

- each expression e_i whose value is an actual parameter has the corresponding type t_i.

Notice that in this rule we are not creating a new type environment as the type of the function (e_f) and actual parameters are computed in the same scope.

10.11 Types for Reference Expressions

Type annotations for Reference-related Expressions

TypeLang requires a ref expression to specify the type of content of the memory location it refers to, as shown in the following syntax.

refexp	::=	(**ref** : *T exp*)	*Ref expression*

Figure 10.5: Syntax of the reference expression in Typelang

To illustrate, following expression allocates a memory location of type `num` with value 2.

```
(ref : num 2)
```

Similarly, following reference expression allocates two memory locations: first of type `num` and second of type `Ref num`.

```
(ref : Ref num
  (ref : num 2)
)
```

The type `Ref num` is different from types like `num` and `bool` that we have seen so far, and similar to the function types in that it can help construct other types e.g. `Ref num` that is reference to a number, `Ref Ref num` that is reference to a reference to a number, and so on.

As another example, the following expression declares `r` as a reference to a reference with value number `5` and evaluation of the program returns `5`.

```
(let
  (
    (r : Ref Ref num (ref : Ref num (ref : num 5)))
  )
  (deref (deref r))
)
```

10.12 Types for Recursive Functions

Type annotations for Letrec Expression

Similar to a let expression, TypeLang requires a programmer to specify types of variables declared in a letrec expression, as shown in the syntax below.

$$\textit{letrecexp} \quad ::= \quad (\textbf{letrec}\ ((\textit{identifier}\!:\!T\ \text{exp})^{+})\ \textit{exp}) \qquad \textit{Letrec expression}$$

Figure 10.6: Syntax of the Letrec expression in Typelang

For example, the following letrec expression, declares `isEven` and `isOdd` variables to be of function type `(num -> boolean)`. `isEven` and `isOdd` are functions that take a number parameter and return a boolean.

```
(letrec
  (
    (isEven : (num -> bool)
      (lambda (n : num)
        (if (= 0 n) #t (isOdd (- n 1)))
      )
    )
    (isOdd : (num -> bool)
      (lambda (n : num)
        ( if (= 0 n) #f (isEven (- n 1)))
      )
    )
  )
  (isOdd 11)
)
```

10.13 Types for other Expressions

We now discuss types for additional expressions in Typelang shown in figure 10.7.

exp	::=	...	*Expressions*
	|	*strconst*	*String constant*
	|	*boolconst*	*Boolean constant*
	|	*lessexp*	*Less*
	|	*equalexp*	*Equal*
	|	*greaterexp*	*Greater*
	|	*ifexp*	*Conditional*
	|	*carexp*	*Car*
	|	*cdrexp*	*Cdr*
	|	*consexp*	*Cons*
	|	*listexp*	*List constructor*
	|	*nullexp*	*Null*

Figure 10.7: Extended Syntax of Typelang (includes expression omitted in figure 10.1)

To be able to give types to these expressions we also add some new kinds of types as shown in figure 10.8.

Types for List-related Expression

TypeLang requires a list expression to specify type of its elements, as shown in the following syntax. All elements of the list must have the same specified type.

type	::=	...	*Types*
	|	`String`	*String Type*
	|	`( T , T )`	*Pair Type*
	|	`List < T >`	*List Type*

Figure 10.8: Extended types in Typelang

```
listexp : '(' ' list ' ':' T exp* ')' ;
```

To illustrate, the following expression

```
( list : num 1 2 3)
```

constructs a list with elements `1`, `2` and `3` of type number; the list expression type checks in TypeLang. However, the following expression

```
( list : num 1 2 #t)
```

does not type check because the elements of the list are supposed to be of type `number` and `#t` is of type boolean. Similarly, the expression

```
(null? (cons 1 2))
```

does not type check because `null?` requires a parameter of type `list` while

```
(cons 1 2)
```

constructs a pair of pair type `(num, num)`.

As another example, the expression

```
( list : List<num> (list : num 2))
```

type checks and constructs a list in which its element (list : num 2) are list of numbers. However, the variation

```
( list : List<num> (list : bool 2))
```

of this expression does not type check because in

```
( list : bool 2) number 2 is not a boolean.
```

As another example, consider the following let expression

```
(let
  ((l: List<List<num>> (list : List<num> (list : num 2)))) (car l)
)
```

declares variable `l` as a list of lists of numbers and returns the first element of `l`.

10.14 Further Reading

- See: Abrial, Jean-Raymond; Schuman, Stephen A; Meyer, Bertrand (1980), "*A Specification Language*", in Macnaghten, AM; McKeag, RM, On the Construction of Programs, Cambridge University Press, ISBN 0-521-23090-X for introduction to the Z specification language. This work discusses basic ideas behind Z.
- See: Spivey, John Michael (1992), "*The Z Notation: A reference manual*", International Series in Computer Science (2nd ed.). Prentice Hall. for detailed introduction to the Z specification language.
- See Pierce's book "Types in Programming Languages" for a more detailed exposition of typed programming languages.