Chapter 9

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 enjoyed the benefits of types in your programs. This chapter presents a systematic study of the foundation behind types and related facilities.

9.1 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.

Rajan PL DG (DRAFT). Not for Distribution. COM S 342, Spring 2015

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 not explicit.

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.*

9.2 Kinds of Specifications

We will discuss the notion of contracts in full generality in chapter **??**, 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*

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

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 **??** discusses specification languages in greater detail).

9.3 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 **??**. 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.

9.4 Adding Type Annotations

The core set of features in Typelang are presented in figure 9.1. 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⟩ : ⟨type⟩ ⟨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 9.1: Syntax of core elements of Typelang (omits some expressions presented in figure 9.6)

First change appears in the syntax of the define declarations, where a new non-terminal ⟨type⟩ is added. This non-terminal is defined in figure 9.2.

According to the grammar in figure 9.2 there are four kinds of types: unit types, number types, function types, and reference types. The unit

and number 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.

⟨type⟩	::=		*Types*
	|	`unit`	*Unit Type*
	|	`num`	*Number Type*
	|	`(` ⟨type⟩* `->` ⟨type⟩ `)`	*Function Type*
	|	`Ref` ⟨type⟩	*Reference Type*

Figure 9.2: 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 id : (num -> num) (lambda (x : (num -> 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 `id` 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.

Since Typelang extends Reflang, it has all the standard expressions as shown in figure 9.1. 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.

Types for Let Expressions

TypeLang requires a programmer to specify types of identifiers of a let expression, as shown in the syntax below. In a let expression each declared identifier has a type as shown in figure 9.3.

⟨letexp⟩ ::= (**let** ((⟨identifier⟩:⟨type⟩ exp)$^+$) ⟨exp⟩) *Let expression*

Figure 9.3: Syntax of the Let expression in Typelang

To illustrate, the following let expression

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

declares a variable `x` of type number with value `2`. Similarly, the following let expression

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

declares two variables `x` and `y` both of type integers with values `2` and `5`. However, the variation

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

fails to typecheck because an addition expression cannot add a number and a boolean.

Just like the define form, in a let expression the type annotation on identifier such as `num` in previous two examples acts 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.

Types for Function and Calls

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

⟨lambdaexp⟩ ::= (**lambda** (⟨identifier⟩*:⟨type⟩) ⟨exp⟩) *Lambda*

Figure 9.4: Syntax of the Lambda expression in Typelang

A lambda expression must be of a function type, as specified in the syntax in figure 9.2. A function type specifies the types of function's arguments as well as return type of the function. In a function type `(type`$_a$`->type`$_r$`)`, `type`$_a$ is the argument type and `type`$_r$ 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, the following lambda expression

```
(lambda
  (
    x y z :         //Arguments
    (num num num -> num) //Function type
  )
  (+ x (+ y z))
)
```

declares a function with three arguments `x`, `y` and `z` and returns their sum. The type for this function 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 when evaluated, however, the following lambda expression

```
(lambda (x y z : (num num num -> bool))
  (+ x (+ y z))
)
```

does not type check. This is because addition of integer parameters `x`,`y` and `z` produces a number and not a boolean.

As another example, the following call expression

```
(
  (lambda (x y z : (num num num -> num))
    (+ x (+ y z))
  )
  1 2 3
)
```

declares the same function above and then calls it by passing integer parameters `1`, `2` and `3` for arguments `x`,`y` and `z`. This call expression type checks in TypeLang as well, however, its variation

```
(
  (lambda (x y z : (num num num -> num))
    (+ x (+ y z))
  )
  1 2 #t
)
```

would not type check because `#t` is of type bool and not of type number the function expects for `z`.

Types 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.

```
letrecexp : '(' Letrec '(' ( '(' Identifier ':' type exp ')' )+ ')' exp
    ')' ;
```

⟨letrecexp⟩ ::= (**letrec** ((⟨identifier⟩:⟨type⟩ exp)$^+$) ⟨exp⟩) *Letrec expression*

Figure 9.5: 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 -> bool))
        ( if (= 0 n) #t (isOdd (- n 1)))
      )
    )
    (isOdd : (num -> bool)
      (lambda (n : (num -> bool))
        ( if (= 0 n) #f (isEven (- n 1)))
      )
    )
  )
  (isOdd 11)
)
```

Types 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' ':' type exp ')' ;
```

To illustrate,

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

allocates a memory location of type number with value 2. Similarly, following reference expression

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

allocates a memory location of type to a reference which its content is 2.

As another example,

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

```
  (deref (deref r))
)
```

declares `r` as a reference to a reference with value number `5` and evaluation of the program returns `5`.

9.5 Types for other Expressions

We now discuss types for additional expressions in Typelang shown in figure 9.6.

⟨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 9.6: Extended Syntax of Typelang (includes expression omitted in figure 9.1)

To be able to give types to these expressions we also add some new kinds of types as shown in figure 9.7.

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

Figure 9.7: Extended types in Typelang

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.

```
listexp : '(' ' list ' ':' type 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`.

9.6 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.