Programming Languages

Chapters 24 – 29
Types, Type Judgments, Type Systems and Type Safety - Chapters 24-28
Type Inference – Chapter 30
Explicit Polymorphism (Generics) – Chapter 29
Introduction

(+ 3 (define add3
    (lambda (x) (+ x 3))))

Welcome to DrScheme, version 4.2.1 [3m].
Language: PLAI Scheme; memory limit: 256 megabytes.
. define: not allowed in an expression context in: (define add3
  (lambda (x) (+ x 3)))
>
(define mystery true)
(+ 3 (if mystery 5 (define add3
    (lambda (x) (+ x 3))))))

Welcome to DrScheme, version 4.2.1 [3m].
Language: PLAI Scheme; memory limit: 256 megabytes.
. define: not allowed in an expression context in: (define add3
  (lambda (x) (+ x 3)))
>
Introduction

A **type** is any **property** of a program that we can establish **without executing** the program. In particular, types capture the intuition that we would like to predict a program’s behavior without executing it.

... A **type** labels every expression in the language, recording what kind of value evaluating that expression will yield. That is, types describe invariants that hold for all executions of a program. They approximate this information in that they typically record only what kind of value the expression yields, not the precise value itself.

... Note that we are careful to refer to **valid** programs, but never **correct** ones. **Types** do not ensure the correctness of a program. They only guarantee that the program does not make certain kinds of errors.
Types, Type Judgments and Type Systems

A *type system* is a collection of *types*, the corresponding *type judgments* that ascribe types to expressions, and an *algorithm* for performing this ascription.
Type Judgments

A type system is a collection of rules, known formally as type judgments, which describe how to determine the type of an expression. There must be at least one type rule for every kind of syntactic construct so that, given a program, at least one type rule applies to every sub-term. Judgments are often recursive, determining an expression’s type from the types of its parts.

Examples:
- Any numeral n has type number - n : number
- Any function fun has type function –(define fun (i) b) : function

But what is the type of an identifier? Clearly, we need a type environment (a mapping from identifiers to types). It’s conventional to use $\Gamma$ (the upper-case Greek “gamma”) for the type environment. All type judgments will have the following form:

$\Gamma |- e : t$

where e is an expression and t is a type, which we read as “$\Gamma$ proves that e has type t”.

Thus,
- $\Gamma |- n : \text{number}$
- $\Gamma |- \{\text{fun } \{i\} \ b\} : \text{function}$
- $\Gamma |- i : \Gamma(i)$
## Type Judgments

<table>
<thead>
<tr>
<th>Type Judgment</th>
<th>Pseudo Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma</td>
<td>- \text{L : number} \quad \Gamma</td>
</tr>
<tr>
<td>( \Gamma</td>
<td>- (+ \text{L} \text{R}) : \text{number} )</td>
</tr>
<tr>
<td>( \Gamma</td>
<td>- \text{f : } (\tau_1 \rightarrow \tau_2) \quad \Gamma</td>
</tr>
<tr>
<td>( \Gamma</td>
<td>- (\text{f} \text{a}) : \tau_2 )</td>
</tr>
<tr>
<td>( \Gamma[i \leftarrow \tau_1]</td>
<td>- \text{b : } \tau_2 )</td>
</tr>
<tr>
<td>( \Gamma</td>
<td>- \text{fun } { \text{i : } \tau_1 } : \tau_2 \text{ b : } (\tau_1 \rightarrow \tau_2) )</td>
</tr>
</tbody>
</table>

### Note:

By the way, it would help to understand the status of terms like i and b and n in these judgments. They are “variable” in the sense that they will be replaced by some program term: for instance, \( \text{fun } \{ \text{i : } \tau_1 \} : \tau_2 \text{ b} \) may be instantiated to \( \text{fun } \{ \text{x : number} \} : \text{number} \text{x} \), with i replaced by x, and so forth. But they are not program variables; rather, they are variables that stand for program text (including program variables). They are therefore called metavariables.

### True in Hmmp?

Dr. Philip Cannata
Function Declaration and Application

There is an important relationship between the type judgments for function declaration and for application:
• When typing the function declaration, we assume the argument will have the right type and guarantee that the body, or result, will have the promised type.
• When typing a function application, we guarantee the argument has the type the function demands, and assume the result will have the type the function promises.

This interplay between assumptions and guarantees is quite crucial to typing functions. The two “sides” are carefully balanced against each other to avoid fallacious reasoning about program behavior. In addition, just as number does not specify which number will be used, a function type does not limit which of many functions will be used. If, for instance, the type of a function is (number -> number), the function could be either increment or decrement (or a lot else, besides). The type checker is able to reject misuse of any function that has this type without needing to know which actual function the programmer will use.
Type Judgments

\[
\begin{array}{c}
\Gamma |- c : \text{boolean} \quad \Gamma |- t : \tau \quad \Gamma |- e : \tau \\
\hline
\Gamma |- (\text{if } c t e) : \tau
\end{array}
\]

Welcome to DrScheme, version 4.2.1 [3m].
Language: PLAI Scheme; memory limit: 256 megabytes.
> (if true ((lambda (x) 'x) 5) ((lambda (x) x) 1))
'x
> (if false ((lambda (x) 'x) 5) ((lambda (x) x) 1))
1
>
Type Judgments

\[
\begin{array}{c}
\Gamma \vdash c : \text{boolean} \quad \Gamma \vdash t : \tau \quad \Gamma \vdash e : \tau \\
\hline
\Gamma \vdash (\text{if } c \text{ } t \text{ } e) : \tau
\end{array}
\]

Haskell

\$\text{haskell if.hs}\$

myif True x _ = x
myif False _ y = y

\text{myift} :: \text{Bool} \rightarrow a \rightarrow a \rightarrow a

myift True x _ = x
myift False _ y = y

\$\text{cat if.hs}\$

Haskell 98 mode: Restart with command line option -98 to enable extensions

Main> :t myif
myif :: Bool -> a -> a -> a
Main> myif True ((\ x -> 1) 5) ((\ x -> 1) 5)
1
Main> myif True ((\ x -> 1) 5) ((\ x -> "a") 5)
ERROR - Cannot infer instance
*** Instance : Num [Char]
*** Expression : myif True ((\x -> 1) 5) ((\x -> "a") 5)

Main> :t myift
myift :: Bool -> a -> a -> a
Main> myift True ((\ x -> 1) 5) ((\ x -> 1) 5)
1
Main> myift True ((\ x -> 1) 5) ((\ x -> "a") 5)
ERROR - Cannot infer instance
*** Instance : Num [Char]
*** Expression : myift True ((\x -> 1) 5) ((\x -> "a") 5)
Type Judgment Trees

For the expression

\{+ 2 \{+ 5 7\}\}

We stack type judgments for this term as follows:

\[
\frac{\emptyset-2: \text{number} \quad \emptyset-5: \text{number} \quad \emptyset-7: \text{number}}{\emptyset-\{+ 5 7\}: \text{number}} \quad \frac{\emptyset-\{+ 2 \{+ 5 7\}\}: \text{number}}{\emptyset-\{+ 2 \{+ 5 7\}\}: \text{number}}
\]

For the expression

\{\{\text{fun} \ \{x : \text{number}\} : \text{number} \}

\{+ x 3}\}

\text{5}\}

The type judgment tree looks as follows:

\[
\frac{\{x\leftarrow \text{number}\}-x: \text{number} \quad \emptyset-3: \text{number}}{\{x\leftarrow \text{number}\}-\{+ x 3\}: \text{number}} \quad \frac{\emptyset-\text{fun} \ \{x : \text{number}\} : \text{number}{\{+ x 3\}}:(\text{number}\rightarrow \text{number}) \quad \emptyset-5: \text{number}}{\emptyset-\{\{\text{fun} \ \{x : \text{number}\} : \text{number}{\{+ x 3\}}: \text{number}\}\}\text{5}}: \text{number}
\]
Type Judgment Trees

(+ 3 (define add3 (lambda (x) (+ x 3))))

\[ \Gamma |- 3 : \text{number} \]

\[ \frac{}{(+ 3 (define add3 (lambda (x) (+ x 3)))) : ??} \]

We cannot construct a legal type derivation tree for the original term. However, to flag a program as erroneous, we must prove that no type derivation tree can possibly exist for that term. But perhaps some sequence of judgments that we haven’t thought of exists that (a) is legal and (b) correctly ascribes a type to the term! To avoid this we may need to employ quite a sophisticated proof technique, even human knowledge (Cycorp ?). A computer program might not be so lucky, and in fact may get stuck endlessly trying judgments!

This is why a set of type judgments alone does not suffice: what we’re really interested in is a type system that includes an algorithm for type-checking. Sometimes a simple top-down, syntax-directed algorithm suffices (e.g., Hmm) for (a) determining the type of each expression, and (b) concluding that some expressions manifest type errors. As our type judgments get more sophisticated, we will need to develop more complex algorithms to continue producing tractable and useful type systems.
Type Judgment - Recursion

Attempts at recursion – Text Section 26.2

{with \{f \{\text{fun} \{i\} \{f \ i\}\}\} \{f \ 10\}\}

(let ((f (\text{lambda} (i) (f \ i)))) (f \ 10))

{with \{\omega \{\text{fun} \{x\} \{x \ x\}\}\} \{\omega \omega\}\}

(let ((\omega (\text{lambda} (x) (x \ x)))) (\omega \omega))

\begin{align*}
\text{letrec } & ((\text{factorial} \ (\text{lambda} \ (N) \ (\text{if} (= N 0) 1 (* N (\text{factorial} \ (- N 1))))))) \ (\text{factorial} \ 5)) \\
& \begin{array}{c}
\Gamma[i <- \tau_i] |- b : \tau \\
\Gamma[i <- \tau_i] |- v : \tau_i
\end{array} \\
& \Gamma|- (\text{letrec} \ (i : \tau_i \ v) \ b) : \tau
\end{align*}
Type Judgment - Recursion

(letrec ((factorial (lambda (N) (if (= N 0) 1 (* N (factorial (- N 1))))))) (factorial 5))

Γ[i <- τ_i] |- b : τ  Γ[i <- τ_i] |- v : τ_i

Γ|- ( letrec (i : τ_i v) b) : τ

Maybe JDBLisp could have something like

(factorial (num) num) :: function
What is being Demonstrated Here

Welcome to DrRacket, version 5.0.2 [3m].
Language: racket; memory limit: 128 MB.

> (+ 3 (if true 5 (define add3 (lambda (x)(+ x 3 )))))
  . define: not allowed in an expression context in: (define add3 (lambda (x) (+ x 3)))
> (+ 3 (if false 5 (define add3 (lambda (x)(+ x 3 )))))
  . define: not allowed in an expression context in: (define add3 (lambda (x) (+ x 3)))

> (if true ((lambda (x) 'x) 5) ((lambda (x) x) 5))
'x
> (if false ((lambda (x) 'x) 5) ((lambda (x) x) 5))
5

In Haskell
myif True x _ = x
myif False _ y = y

Main> myif True ((\ x -> 1) 5) ((\ x -> "a") 5)
ERROR - Cannot infer instance
*** Instance : Num [Char]
*** Expression : myif True ((\x -> 1) 5) ((\x -> "a") 5)

$ cat hfac.h
-- fac 0 = 1
fac n = n * fac (n-1)
-- fac 0 = "a"
-- fac n = n * fac (n-1)
Type Inference – Chapter 30

From my PLAI 1 Notes, page 11

\[
\begin{align*}
\text{s } f \ g \ x &= f \ x \ (g \ x) \\
\text{k } x \ y &= x \\
\text{b } f \ g \ x &= f \ (g \ x) \\
\text{c } f \ g \ x &= f \ x \ g \\
\text{y } f &= f \ (y \ f) \\
\text{cond } p \ f \ g \ x &= \text{if } p \ x \ \text{then } f \ x \ \text{else } g \ x \\
\text{cfac} &= y \ (\text{b } (\text{cond } ((==) \ 0) \ (\text{k } 1)) \ (\text{b } (s \ (*) \ (\text{c } b \ \text{pred})))) \\
\end{align*}
\]

: \text{t cfac} \\
\text{cfac} :: \text{Integer } \rightarrow \text{Integer}

Main> :t s \\
s :: (a -> b -> c) -> (a -> b) -> a -> c \\
Main> s (\ x \ y -> x+y) (\ x -> x+3) 4 11

Main> :t c \\
c :: (a -> b -> c) -> b -> a -> c \\
Main> c (\ x \ y -> x+(y^2)) 4 2 18

Combinators
Type Soundness – rewording of the first paragraph of Chapter 28

*Type soundness:* For all programs (expressions) \( pe \), if the type checker assigns \( pe \) the type \( \tau \), and if the program semantics cause \( pe \) to evaluate to a value \( v \) of type \( t \), then the type checker will also have assigned \( v \) the type \( \tau \).

(\text{first (list)})

Should this

- Return a value such as \(-1\).
- Diverge, i.e., go into an infinite loop.
- Raise an exception.
Type Soundness

*Type soundness*: For all programs (expressions) `pe`, if the type checker assigns `pe` the type `t`, and if the program semantics cause `pe` to evaluate to a value `v` of type `t`, then the type checker will also have assigned `v` the type `t`. Otherwise one of a well-defined set of exceptions should have been raised.

Why is type soundness not obvious? Consider the following simple program (the details of the numbers aren’t relevant):

```plaintext
{if0 (+ 1 2)
  {{fun {x : number} : number {+ 1 x}} 7}
  {{fun {x : number} : number {+ 1 (+ 2 x)}} 1}}
```

During execution, the program will explore only one branch of the conditional:

\[
\begin{align*}
&1,\emptyset \Rightarrow 1 & 2,\emptyset \Rightarrow 2 & \cdots \\
&\{+ 1 2\},\emptyset \Rightarrow 3 & \{{\text{fun ...} 1\}}},\emptyset \Rightarrow 4 \\
&\{\text{if0 } (+ 1 2) \{{\text{fun ...} 7\}} \{{\text{fun ...} 1\}}},\emptyset \Rightarrow 4
\end{align*}
\]

but the type checker must explore both:

\[
\begin{align*}
\emptyset \vdash 1:\text{number} & & \emptyset \vdash 2:\text{number} & & \cdots & & \cdots \\
\emptyset \vdash \{+ 1 2\}:\text{number} & & \emptyset \vdash \{{\text{fun ...} 7\}}:\text{number} & & \emptyset \vdash \{{\text{fun ...} 1\}}}:\text{number} \\
\emptyset \vdash \{\text{if0 } (+ 1 2) \{{\text{fun ...} 7\}} \{{\text{fun ...} 1\}}}:\text{number}
\end{align*}
\]
Type Safety and Strongly Typed

*Type safety* is the property that no primitive operation ever applies to values of the wrong type. By primitive operation we mean not only addition and so forth, but also procedure application. A safe language honors the abstraction boundaries it erects.

So what is “*Strong Typing*”? This appears to be a meaningless phrase, and people often use it in a nonsensical fashion. To some it seems to mean “The language has a type checker”. To others it means “The language is sound” (that is, the type checker and run-time system are related). To most, it seems to just mean, “A language like Pascal, C or Java, related in a way I can’t quite make precise”. If someone uses this phrase, be sure to ask them to define it for you. (For amusement, watch them squirm.)
Type Safety

<table>
<thead>
<tr>
<th>statically checked</th>
<th>not statically checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>type safe</td>
<td>ML, Java</td>
</tr>
<tr>
<td>type unsafe</td>
<td>C, C++ (mainly because it allows embedded C)</td>
</tr>
</tbody>
</table>

The important thing to remember is, due to the Halting Problem, some checks simply can never be performed statically; something must always be deferred to execution time. The trade-off in type design is to maximize the number of these decisions statically without overly restricting the power of the programmer.

The designers of different languages have divergent views on the powers a programmer should have.
Explicit Polymorphism (Generics) – Chapter 29

(define lengthNum
  (lambda (l : numlist) : number
    (cond
      [(numEmpty? l) 0]
      [(numCons? l) (add1 (lengthNum (numRest l)))])))

(define lengthSym
  (lambda (l : symlist) : number
    (cond
      [(symEmpty? l) 0]
      [(symCons? l) (add1 (lengthSym (symRest l)))])))
(define length
  (∀ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?<τ> l) 0]
        [(Cons?<τ> l) (add1 (length<τ> (Rest<τ> l)))]))>)

(length<num> (list 1 2 3))
(length<sym> (list 'a 'b 'c))

It is therefore clear that the type procedures must accept arguments and evaluate their bodies before the type checker even begins execution. By that time, if all the type applications are over, it suffices to use the type checker built earlier, since what remains is a language with no type variables remaining. We call the phase that performs these type applications the type elaborator.