

# Chapter 1

## Untyped Lambda Calculus

### 1.1 Introduction

In the mid 1960s, Landin observed that a complex programming language can be understood in terms of a tiny *core language* lambda calculus capturing the essential mechanisms of programming together with a collection of convenient *derived forms* whose behavior is understood by translating them to the core. Landin's core language was the lambda calculus, a formal system in which all computations are reduced to the basic operations of function definition and application. Since the 60's, the lambda calculus has seen widespread use in the specification of programming language features, language design and implementation, and the study of type systems. Its importance arises from the fact that it can be viewed simultaneously as a simple programming language in which computations can be described and as a mathematical object about which rigorous statements can be proved.

### 1.2 Syntax

$$\begin{array}{ll} e ::= & x \quad \text{(variable)} \\ & | \lambda x. e \quad \text{(function abstraction)} \\ & | e_1 e_2 \quad \text{(function application)} \\ & | (e) \quad \text{(bracketed expression)} \end{array}$$

The function application is left associative (e.g.,  $e_1 e_2 e_3 \equiv (e_1 e_2) e_3$ ). The scope of  $\lambda$  expands as far to the right as possible (e.g.,  $\lambda x. x \lambda y. x y \equiv \lambda x. (x \lambda y. (xy))$ ). A variable  $x$  is said to be free in a  $\lambda$ -term  $M$  when  $x$  appears at some position in  $M$  where it is not bound by an enclosing lambda abstraction on  $x$ . A  $\lambda$ -term with no free variables is called a closed  $\lambda$ - term.

Let us see some example functions in  $\lambda$ -calculus.

1.  $\lambda x. x$  - Identity function
2.  $\lambda x. \lambda y. x$  - Selection function: function that takes two arguments  $x$  and  $y$  and returns first argument  $x$
3.  $\lambda f. \lambda x. f(f x)$  - A higher order function that takes a function  $f$  and a value  $x$  as arguments and applies  $f$  on  $x$  twice

Intuitively it is clear that the expression  $\lambda x. x$  and  $\lambda y. y$  describe exactly the same function - the function that, given any argument  $N$  returns  $N$ . The fact that we use  $x$  in one case and  $y$  in the other is of no consequence. This principle is captured by the rule of *renaming of bound variables* (often called  $\alpha$ -renaming). This states that we may freely replace the bound variable  $x$  by another variable  $y$  in a lambda abstraction  $\lambda x. M$  as long as  $y$  is not among the free variables of  $M$ . The side condition is needed because, for example we do not want to consider  $\lambda x. y$  and  $\lambda y. y$  to be the same function). Formally,  $\lambda x. M \equiv \lambda y. [x \mapsto y]M$ .

Taking the above argument further, De Bruijn gave a way to remove all the variables from a closed  $\lambda$ -term. Let us define *De Bruijn index* of a variable to be the number of  $\lambda$ 's that separate the occurrence from the binder. If the variable occurrences are replaced with their corresponding De Bruijn indices, then we say the  $\lambda$ -term is in *De Bruijn notation*. For example,  $\lambda x. \lambda y. x y \equiv \lambda. \lambda. 1 0$ ,  $\lambda x. \lambda x. x \equiv \lambda. \lambda. 0$ ,  $\lambda z. \lambda y. y \equiv \lambda. \lambda. 0$ . Note that, if two closed  $\lambda$ -terms are  $\alpha$ -renaming equivalent, then they have the same De Bruijn notation.

A closed  $\lambda$ -term is also called *combinator*. Some of the standard combinators are

$$I = \lambda x. x$$

$$K = \lambda x. \lambda y. x$$

$$S = \lambda f. \lambda g. \lambda x. f x (g x)$$

$$D = \lambda x. x x$$

$$Y = \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))$$

Any combinator can be equivalently expressed as one which uses only  $S, K$  and  $I$ .

## 1.3 Semantics

In its pure form, the lambda calculus has no built-in constants or operators - no numbers, arithmetic operations, loops, records, sequencing, I/O etc. The sole means by which expressions *compute* is the application of functions to arguments, which is captured formally by  $\beta$ -**reduction** rule

$$(\lambda x. e_1) e_2 \rightarrow_{\beta} (x \mapsto e_2) e_1$$

This rule says that an expression can be *reduced* by replacing some sub-expression of the form  $(\lambda x. M)N$ , called a *redex*, by the result of substituting the argument  $N$  for the bound variable  $x$  in the body  $M$ . For example,  $(\lambda x. x y) (u v)$  reduces to  $u v y$ , while  $(\lambda x. \lambda y. x) z w$  reduces to  $(\lambda y. z) w$ , which reduces to  $z$ . We write  $(\lambda x. \lambda y. x) zw \rightarrow_{\beta}^* z$  to show that the first expression reduces to the second by some sequence of steps of reduction. For the sake of simplicity, we use  $\rightarrow$  instead of  $\rightarrow_{\beta}$  in the sequel.

The notion of reduction gives rise to a natural definition of what it means for two expressions to be the same *modulo reduction*.  $M$  and  $N$  are  $\beta$ -*convertible* written  $M =_{\beta} N$ , if they are identical, or if one reduces to the other, or if they are each convertible to some third expression  $L$ . The relation  $=_{\beta}$  is reflexive, symmetric and transitive closure of  $\rightarrow$ . The problem of checking whether  $e =_{\beta} e'$  is undecidable. Again, we omit  $\beta$  for the ease of reading.

To make the notions of  $\beta$ -reduction and conversion completely precise, we must define *substitution* formally. Substituting an expression  $N$  for  $x$  in an expression consisting only of  $x$  itself yields  $N$ . Substituting  $N$  for  $x$  in an expression consisting only of a different variable  $z$  yields  $z$ . To substitute  $N$  for  $x$  in an expression consisting an application  $M L$ , we substitute in  $M$  and  $L$  separately. To substitute  $N$  for  $x$  in an expression consisting a lambda-abstraction  $\lambda z. M$ , we substitute in the body  $M$ : however, we do this only when  $z$  is not the same as  $x$  and  $z$  is not one of the free variables of  $N$ . The first part of this condition ensures that we do not allow nonsensical reductions like  $(\lambda x. \lambda x. x) y \rightarrow \lambda x. y$ , where  $x$  is replaced by  $y$  even though it is actually bound by an inner abstraction, not the one being reduced. The second prevents a similar kind of mistake where free variables of  $N$  are *captured* by abstractions inside a term being substituted into, resulting in reductions like  $(\lambda x. \lambda y. x) y \rightarrow \lambda y. y$ . Thus, strictly speaking, the substitution  $[x \mapsto N]M$  is undefined for some values of  $N, x$ , and  $M$ . But it is always possible to change the names of bound variables in  $N$  and/or  $M$  so that the substitution makes sense.

A lambda-expression containing more than one redex can be reduced in more than one way, leading, in general, to different results. For example,

$K I (K True False)$  reduces in one step to either  $I$  or  $K I True$  (remember the combinators  $K$  and  $I$  defined earlier). Here, we can further reduce the second result, yielding  $I$  again: but we might worry that there could be some  $M$  such that  $M$  could be reduced to either  $N_1$  or  $N_2$  in such a way that  $N_1$  and  $N_2$  could never be *brought back together* by reducing them further. The following theorem says that the reduction in lambda-calculus is *confluent* i.e.,

**Theorem[Church-Rosser ]** If  $M$  reduces to two different expressions.  $N_1$  and  $N_2$ , then these further reduce to some common expression  $L$ . (In symbols: if  $M \rightarrow^* N_1$  and  $M \rightarrow^* N_2$ , then there is some  $L$  such that  $N_1 \rightarrow^* L$  and  $N_2 \rightarrow^* L$ )

The above property is called diamond property. From the theorem above, we understand that  $\rightarrow^*$  satisfies diamond property. However, note that the diamond property is not satisfied by  $\rightarrow$  (one step reduction).

A closed term without redexes is said to be in normal form. One implication of Church-Rosser theorem is that, we will not find more than one normal form, independent of the reduction strategy. However, some reduction strategies might fail to find a normal form. For example  $(\lambda x. y)((\lambda y. y y)(\lambda y. y y))$ . It reduces to  $(\lambda x. y)((\lambda y. y y)(\lambda y. y y)) \rightarrow (\lambda x. y)((\lambda y. y y)(\lambda y. y y)) \rightarrow \dots$ .

A *reduction strategy* is a rule specifying which redexes should be reduced first. There are several common reduction strategies. The *normal-order* reduction strategy picks the leftmost outermost redex and reduces it. It does not reduce the expression inside a lambda enclosure. A result about normal-order reduction strategy is that if  $e$  has normal form  $e'$ , then normal order reduction will reduce  $e$  to  $e'$ . A *call by name* reduction strategy does not evaluate the argument of the function first, it executes in a lazy fashion, only when it is needed. It does not reduce the expression inside a lambda enclosure. This strategy is demand driven as the expressions are not evaluated unless it is needed. A *call by value* reduction strategy does evaluate the argument of a function first. It does not reduce the expression inside a lambda enclosure.

## 1.4 Programming in $\lambda$ -calculus

Lambda calculus is expressive enough to encode turing machines. Let us try and encode some of the basic language features.

$$True := \lambda t. \lambda f. t$$

$$False := \lambda t. \lambda f. t$$

Both of these expressions are combinators. This means that they are inert with respect to substitutions. We can define a combinator *If* using the above combinators with the property that *If L then M else N* reduces to *M* when *L* is *True* and reduces to *N* when *L* is *False*.

$$\text{If } l \ m \ n := \lambda l. \lambda m. \lambda n. l \ m \ n$$

The *If* combinator does not actually do much: *If L M N* means just *L M N*. In effect, the boolean value *L* itself is the conditional: it takes two arguments and chooses the first (if it is *True*) or the second (if it is *False*). For example, the expression *If True M N* reduces as follows:

$$\begin{aligned} \text{If True M N} &= (\lambda l. \lambda m. \lambda n. l \ m \ n) \ \text{True} \ M \ N \\ &\rightarrow (\lambda m. \lambda n. \text{True } m \ n) \ M \ N \\ &\rightarrow (\lambda n. \text{True } M \ n) \ N \\ &\rightarrow \text{True } M \ N \\ &= (\lambda t. \lambda f. t) \ M \ N \\ &\rightarrow (\lambda f. M) \ N \\ &\rightarrow M \end{aligned}$$

The encoding of numbers as lambda-expressions is only slightly more intricate. Define the *Church Numerals*  $C_0, C_1, C_2 \dots$  as follows:

$$\begin{aligned} C_0 &= \lambda z. \lambda s. z \\ C_1 &= \lambda z. \lambda s. s \ z \\ C_2 &= \lambda z. \lambda s. s(s \ z) \\ &\vdots \\ C_n &= \lambda z. \lambda s. s^n(z) \end{aligned}$$

That is, each number  $n$  is represented by a combinator  $C_n$  that takes two arguments,  $z$  and  $s$  (*zero* and *successor*), and applies  $n$  copies of  $s$  to  $z$ . As with booleans, this encoding makes numbers into active entities: the number  $n$  is represented by a function that does something  $n$  times - a kind of active unary numeral.

We can define some common arithmetic operations on Church numerals as follows:

$$\begin{aligned} \text{Plus} &= \lambda m. \lambda n. \lambda z. \lambda s. m \ (n \ z \ s) \ s \\ \text{Times} &= \lambda m. \lambda n. m \ C_0 \ (\text{Plus } n) \end{aligned}$$

Here, *Plus* is a combinator that takes two Church numerals,  $m$  and  $n$ , as arguments, and yields another Church numeral - i.e., a function that accepts arguments  $z$  and  $s$ , applies  $s$  iterated  $n$  times to  $z$ , and then applies  $s$  iterated  $m$  more times to the result. It is easy to check, for example, that

*Plus*  $C_2 C_5 = C_7$ . The definition of *Times* uses another trick: since *Plus* takes its arguments one at a time, applying it to just one argument  $n$  yields the function that adds  $n$  to whatever argument it is given. Passing this function as the second argument to  $m$  and zero as the first argument means “apply the function that adds  $n$  to its argument, iterated  $m$  times, to zero,” i.e., “add together  $m$  copies of  $n$ ”.

To test whether a Church numeral is zero, we must give it a pair of arguments  $Z$  and  $S$  such that applying  $S$  to  $Z$  one or more times yields *False*, while not applying it at all yields *True*. Clearly, we can take  $Z$  to be just *True*. As for  $S$ , we use a function that throws away its argument and always returns *False*.

$$IsZero = \lambda m. m \ True \ (\lambda x. \ False)$$

Other common datatypes like lists, trees, arrays, and variant records can be encoded using similar techniques. An important operator is the *Fix* operator, which can be used to define recursive functions.

$$Fix = \lambda f. (\lambda x. f \ (\lambda y. x \ x \ y))(\lambda x. f \ (\lambda y. x \ x \ y))$$

Note that it has the property that  $Fix \ F \ \rightarrow \ F \ (Fix \ F)$ . Now, suppose we want to write a recursive function definition of the form  $F = \langle body \ containing \ F \rangle$ . The intention is that the recursive definition should be “unrolled” at the point where it occurs: for example, the definition of *Factorial* would intuitively be written

$$\begin{aligned} & \text{if } n = 0 \ \text{then } 1 \\ & \text{else } n \cdot \quad \text{if } n - 1 = 0 \ \text{then } 1 \\ & \qquad \qquad \text{else } n - 1 \cdot \quad \text{if } n - 2 = 0 \ \text{then } 1 \\ & \qquad \qquad \qquad \text{else } n - 1 \cdot \dots \end{aligned}$$

This effect can be achieved by defining  $G = \lambda f. \langle bodycontaining \ f \rangle$  and  $F = Fix \ G$ , since then

$$\begin{aligned} F &= Fix \ G \\ &= G(Fix \ G) \\ &= \langle body \ containing \ (Fix \ G) \rangle \\ &= \langle body \ containing \ \langle body \ containing \ (Fix \ G) \rangle \rangle \\ &\vdots \end{aligned}$$

For example, if we define the factorial function by

$$\begin{aligned} Fact &= \lambda fact. \lambda n. \text{If } (IsZero \ n) \ C_1 \ (Times \ n \ (fact \ (Pred \ n))) \\ Factorial &= Fix \ Fact \end{aligned}$$

then applying *Factorial* to the Church numeral for 2 leads to the following calculation.

$$\begin{aligned}
\text{Factorial } C_2 &= Y \text{ Fact } C_2 \\
&=_{\beta} \text{Fact } (Y \text{ Fact}) C_2 \\
&=_{\beta} (\lambda \text{fact}. \lambda n. \text{If } (\text{IsZero } n) C_1 (\text{Times } n (\text{fact } (\text{Pred } n)))) (Y \text{ Fact}) C_2 \\
&=_{\beta} (\lambda n. \text{If } (\text{IsZero } n) C_1 (\text{Times } n (Y \text{ Fact } (\text{Pred } n)))) C_2 \\
&=_{\beta} \text{If } (\text{IsZero } C_2) C_1 (\text{Times } C_2 (Y \text{ Fact } (\text{Pred } C_2))) \\
&=_{\beta} \text{If } \text{False } C_1 (\text{Times } C_2 (Y \text{ Fact } C_1)) \\
&=_{\beta} \text{Times } C_2 (Y \text{ Fact } C_1) \\
&= \text{Times } C_2 (\text{Factorial } C_1)
\end{aligned}$$

## 1.5 Exercises

1. Can all closed  $\lambda$ -terms be reduced to normal forms? Justify.
2. Reduce  $(\lambda y. (\lambda x. x) y)((\lambda u. u)(\lambda v. v))$  using *call by name* and *call by value* strategies.
3. Encode *Pred* and *Subtract* function in  $\lambda$ -calculus.

# Chapter 2

## Simply typed Lambda Calculus

### 2.1 Introduction

Simply typed lambda calculus is a refinement of untyped lambda calculus, whose only type is  $\rightarrow$  (function type). This makes it canonical and a simplest example of typed lambda calculus.

There are  $\lambda$  expressions which cannot be reduced further and they are not values either (a valid function) for example,  $x (\lambda x. x)$ . Such a term is called a *stuck* term. Stuck terms correspond to meaningless or erroneous programs. We would therefore like to be able to tell, without actually evaluating a term, that its evaluation will definitely *not* get stuck. Towards this objective we introduce typing to the lambda calculus.

### 2.2 Typed Arithmetic Expressions

Let us consider a simpler language of arithmetic expressions before introducing typing to the lambda calculus. The syntax is

$t ::=$	terms:
$true$	constant true
$false$	constant false
$if\ t\ then\ t\ else\ t$	conditional
$0$	constant zero
$succ\ t$	successor
$pred\ t$	predecessor
$iszero\ t$	zero test

Evaluating a term can either result in a *value* or else get stuck at some stage.

The values are defined to be

$v ::=$	$true$	values:
	$false$	true value
	$nv$	false value
		numeric value
$nv ::=$	$0$	numeric values:
	$succ\ nv$	zero value
		successor value

The operational semantics is given as

$if\ true\ then\ t_2\ else\ t_3 \rightarrow t_2$	(E-IFTRUE)
$if\ false\ then\ t_2\ else\ t_3 \rightarrow t_3$	(E-IFFALSE)
$\frac{t_1 \rightarrow t'_1}{if\ t_1\ then\ t_2\ else\ t_3 \rightarrow if\ t'_1\ then\ t_2\ else\ t_3}$	(E-IF)
$\frac{t \rightarrow t'}{succ\ t \rightarrow succ\ t'}$	(E-SUCC)
$\frac{t \rightarrow t'}{pred\ t \rightarrow pred\ t'}$	(E-PRED)
$pred\ 0 \rightarrow 0$	(E-PREDZ)
$pred\ (succ\ nv) \rightarrow nv$	(E-PREDSUCC)
$iszero\ 0 \rightarrow true$	(E-ISZEROZERO)
$iszero\ (succ\ nv) \rightarrow false$	(E-ISZEROSUCC)
$\frac{t \rightarrow t'}{iszero\ t \rightarrow iszero\ t'}$	(E-ISZERO)

Some of the examples of bad programs are

- $succ\ true$
- $iszero\ false$
- $if\ (succ\ zero)\ then\ true\ else\ false$

The goal is to come up with some static verification technique to rule out such bad programs. We introduce *types* to that effect. We need to be able to distinguish between terms whose result will be a numerical value (since these are the only ones that should appear as arguments to *pred*, *succ* and *iszero* and terms whose result will be a boolean (since only these should appear as the guard of a conditional). We introduce two types: *Nat* and *Bool*, for classifying terms in this way.

$$T ::= Bool \mid Nat$$

Saying that a term  $t$  has type  $T$  ( $t : T$ ) means that  $t$  evaluates to a value of the appropriate form. For example, the term *if true then 0 else succ 0* has type *Nat*. Formally the typing rules are

$$\begin{array}{l}
\text{true} : \text{Bool} \qquad\qquad\qquad (\text{T-TRUE}) \\
\text{false} : \text{Bool} \qquad\qquad\qquad (\text{T-FALSE}) \\
\frac{t_1 : \text{Bool}, t_2 : T, t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \qquad (\text{T-IF}) \\
0 : \text{Nat} \qquad\qquad\qquad (\text{T-ZERO}) \\
\frac{t : \text{Nat}}{\text{succ } t : \text{Nat}} \qquad\qquad\qquad (\text{T-SUCC}) \\
\frac{t : \text{Nat}}{\text{pred } t : \text{Nat}} \qquad\qquad\qquad (\text{T-PRED}) \\
\frac{t : \text{Nat}}{\text{iszero } t : \text{Bool}} \qquad\qquad\qquad (\text{T-ISZERO})
\end{array}$$

A term  $t$  is said to be *well-typed* if there is some type  $T$  such that  $t : T$  can be derived from the inference rules above.

The most basic property of a type system is *safety* (also called soundness): well-typed terms do not get stuck. We show this in two steps, commonly known as the *progress* and *preservation* theorems.

**Progress** A well-typed term is not stuck (either it is a value or it can take a step according to the operational semantics)

**Preservation** If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

These properties together tell us that a well-typed term can never reach a stuck state. We will use the following lemma for the proof progress theorem.

**Lemma 1** (Canonical Forms). *1. If  $v$  is a value of type *Bool*, then  $v$  is either *true* or *false*.*

*2. If  $v$  is a value of type *Nat*, then  $v$  is a numeric value.*

**Theorem 1** (Progress). *Suppose  $t$  is well-typed term (i.e.,  $t : T$  for some  $T$ ). Then either  $t$  is a value or else there is some  $t'$  with  $t \rightarrow t'$ .*

We prove a stronger version of the Preservation theorem i.e,

**Theorem 2** (Preservation). *If  $t : T$  and  $t \rightarrow t'$ , then  $t' : T$*

All the proofs use induction on length of the derivation. Refer the book (given in the reference) for the proofs.

## 2.3 Typed Lambda Calculus

We construct a type system for a language combining booleans with the primitives of the pure lambda calculus. In order to assign a type to an abstraction  $\lambda x. e$ , we need to calculate what will happen when the abstraction is applied to some argument. The next question that arises is: how do we know what type of arguments to expect? There are two possible responses: either we can simply annotate the  $\lambda$ -abstraction with the intended type of its arguments, or else we can analyze the body of the abstraction to see how the argument is used and try to deduce what type it should have. For now, we choose the first alternative. Instead of just  $\lambda x. t$ , we write  $\lambda x : T. e$ , where the annotation on the bound variable tells us to assume that the argument will be of type  $T$ . Formally, the syntax is

$e ::=$	$x$	(variable)
	$\lambda x : T. e$	(function abstraction)
	$e_1 e_2$	(function application)
	$true$	(true value)
	$false$	(false value)
	$if\ e_1\ then\ e_2\ else\ e_3$	(conditional)

The values are *true*, *false* and  $\lambda x : T. e$ . The types are defined by the following grammar.

$$T ::= Bool$$

$$| T \rightarrow T$$

The operational semantics is same as the untyped one but for the annotation of a type in the argument of the function abstraction. Once we know the type of the argument to the abstraction, it is clear that the type of the function's result will be just the type of the body  $e$ , where occurrences of  $x$  in  $e$  are assumed to denote terms of type  $T_1$ . This intuition is captured by the following typing rule:

$$\frac{x : T_1 \vdash e : T_2}{(\lambda x : T_1. e) : T_1 \rightarrow T_2}$$

Since terms may contain nested  $\lambda$ -abstractions, we need to talk about several such assumptions. This changes the typing relation from a two-place relation,  $t : T$ , to a three-place relation,  $\Gamma \vdash e : T$ , where  $\Gamma$  is a set of assumptions about the types of the free variables in  $e$ . Formally, the typing rules are

$$\begin{array}{l}
\text{true} : \text{Bool} \qquad\qquad\qquad (\text{T-TRUE}) \\
\text{false} : \text{Bool} \qquad\qquad\qquad (\text{T-FALSE}) \\
\frac{e_1 : \text{Bool}, e_2 : T, e_3 : T}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \qquad\qquad\qquad (\text{T-IF}) \\
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad\qquad\qquad (\text{T-VAR}) \\
\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1. e : T_1 \rightarrow T_2} \qquad\qquad\qquad (\text{T-ABS}) \\
\frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2, \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 e_2 : T_2} \qquad\qquad\qquad (\text{T-APP})
\end{array}$$

Refer the book for the proof of type-safety theorem.

## 2.4 Exercises

1. Prove that each well-typed term in typed arithmetic language has at-most one type and if it has a type, there is exactly one derivation for it (*Uniqueness of Types* theorem).
2. Is the type system for language of arithmetic expressions complete. Justify.
3. Prove *Uniqueness of Types* theorem for simply typed lambda calculus.

# Chapter 3

## Type Inference

### 3.1 Introduction

The typechecking algorithms for the calculi we have seen so far all depend on explicit type annotations - in particular, they require that lambda abstractions be annotated with their argument types. In this lecture, we develop a more powerful *type inference* algorithm, capable of calculating a type for a term in which some or all of these annotations are left unspecified. Related algorithms lie at the heart of languages like ML and Haskell.

### 3.2 Constraint Based Typing

The approach is to introduce *type variables* and perform type derivation with type variables and collect the *constraints*. Solve the constraints and infer the type. There can be many solutions to the constraints. We illustrate the approach to get the *most general* solution.

Let us consider an example  $\lambda f. \lambda a. f(f(a))$ . Let us annotate with type variables:  $(\lambda f : X. \lambda a : Y. f(f(a))) : Z$ . One of the solutions is  $X \mapsto Bool \rightarrow Bool, Y \mapsto Bool, Z \mapsto (Bool \rightarrow Bool) \rightarrow Bool \rightarrow Bool$ . However the solution  $X \mapsto (Y \rightarrow Y), Z \mapsto (Y \rightarrow Y) \rightarrow Y \rightarrow Y$  is more general than the first solution. These solutions are called *type substitutions*.

Formally, a type substitution  $\sigma$  is a finite mapping from type variables to types. Application of a substitution to a type is defined as

$$\begin{aligned}\sigma(X) &= \begin{cases} T & \text{if } (X \mapsto T) \in \sigma \\ X & \text{if } X \text{ is not in the domain of } \sigma \end{cases} \\ \sigma(Bool) &= Bool \\ \sigma(T_1 \rightarrow T_2) &= \sigma T_1 \rightarrow \sigma T_2\end{aligned}$$

The composition of two substitutions  $\sigma$  and  $\gamma$  is defined as

$$\sigma \cdot \gamma = \begin{cases} X \mapsto \sigma(T) & \text{for each } (X \mapsto T) \in \gamma \\ X \mapsto T & \text{for each } (X \mapsto T) \in \sigma \text{ with } X \notin \text{dom}(\gamma) \end{cases}$$

The substitution  $\sigma$  is said to be *more general* than  $\sigma'$  (written as  $\sigma \sqsubseteq \sigma'$ ) if  $\sigma' = \gamma \cdot \sigma$  for some substitution  $\gamma$ . A substitution more general than all substitutions (solutions) is called the *most general* substitution.

The constraint typing relation is denoted by  $\Gamma \vdash t : T |_{\chi} C$ . Informally, it can be read as “term  $t$  has type  $T$  under assumptions  $\Gamma$  whenever constraints  $C$  are satisfied by the type variables in  $T$ .”. The  $\chi$  subscript is used to track the type variables introduced in each subderivation and make sure that the fresh variables created in different subderivations are actually distinct.

The constraint typing rules are

$$\begin{array}{c} \Gamma \vdash \text{true} : \text{Bool} |_{\emptyset} \emptyset \quad \text{(CT-TRUE)} \\ \Gamma \vdash \text{false} : \text{Bool} |_{\emptyset} \emptyset \quad \text{(CT-FALSE)} \\ \Gamma \vdash e_1 : T_1 |_{\chi_1} C_1 \quad \Gamma \vdash e_2 : T_2 |_{\chi_2} C_2 \\ \hline \Gamma \vdash e_3 : T_3 |_{\chi_3} C_3 \quad \chi_1, \chi_2, \chi_3 \text{ mutually disjoint} \\ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T_2 |_{\chi_1 \cup \chi_2 \cup \chi_3} \\ C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Bool}, T_2 = T_3\} \quad \text{(CT-IF)} \\ \hline x : T \in \Gamma \\ \hline \Gamma \vdash x : T |_{\emptyset} \emptyset \quad \text{(CT-VAR)} \\ \hline \Gamma, x : T_1 \vdash e : T_2 |_{\chi} C \\ \hline \Gamma \vdash \lambda x : T_1. e : T_2 |_{\chi} C \quad \text{(CT-ABS)} \\ \hline \Gamma \vdash e_1 : T_1 |_{\chi_1} C_1 \quad \Gamma \vdash e_2 : T_2 |_{\chi_2} C_2 \\ \chi_1 \cap \chi_2 = \chi_1 \cap FV(T_2) = \chi_2 \cap FV(T_1) = \emptyset \\ X \notin \chi_1, \chi_2, T_1, T_2, C_1, C_2, \Gamma \\ \hline \Gamma \vdash e_1 e_2 : X |_{\chi_1 \cup \chi_2 \cup \{X\}} C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow X\} \quad \text{(CT-APP)} \end{array}$$

Consider the example  $\lambda x. \lambda y. \lambda z. (xz)(yz)$ . Let us annotate with type variables  $\lambda x : X. \lambda y : Y. \lambda z : Z. (xz)(yz) : S$ . We derive the constraints on these type variables as shown below. (The set of variables is left out for the ease of reading).

$$\frac{\frac{x : X, z : Z \vdash x z : A | \{X = Z \rightarrow A\} \quad y : Y, z : Z \vdash y z : B | \{Y = Z \rightarrow B\}}{x : X, y : Y, z : Z \vdash (x z) (y z) : S | \{X = Z \rightarrow A, Y = Z \rightarrow B, A = B \rightarrow S\}}}{\vdash \lambda x : X. \lambda y : Y. \lambda z : Z. (x z) (y z) : S | \{X = Z \rightarrow A, Y = Z \rightarrow B, A = B \rightarrow S\}}$$

### 3.3 Unification

To calculate solutions to constraint sets, we use the idea, due to Hindley (1969) and Milner (1978), of using unification (Robinson, 1971) to check that the set of solutions is nonempty and, if so, to find a most general solution, in the sense that all solutions can be generated straightforwardly from this one.

---

**Algorithm 1:** Unification Algorithm

---

**Input:**  $C$ , a finite set of constraints  
**Output:** most general type substitution  
 $\text{unify}(C) =$   
**if**  $C = \emptyset$  **then**  
     $\emptyset$   
**else**  
    **let**  $\{S = T\} \cup C' = C$  **in**  
    **if**  $S = T$  **then**  
         $\text{unify}(C')$   
    **else if**  $S = X$  *and*  $X \notin FV(T)$  **then**  
         $\text{unify}([X \mapsto T]C') \cdot [X \mapsto T]$   
    **else if**  $T = X$  *and*  $X \notin FV(S)$  **then**  
         $\text{unify}([X \mapsto S]C') \cdot [X \mapsto S]$   
    **else if**  $S = S_1 \rightarrow S_2$  *and*  $T = T_1 \rightarrow T_2$  **then**  
         $\text{unify}(C' \cup \{S_1 = T_1, S_2 = T_2\})$   
    **else**  
        FAIL

---

The time complexity of this algorithm is polynomial in the number of constraints. The algorithm can be made efficient by maintaining equivalence classes and using *union-find*.

Refer to the book for the proofs of the soundness of constraint based typing and the correctness of unification procedure that it produces the most general solution.

# Chapter 4

## Extending STLC to model Programming language features

### 4.1 Introduction

The simply typed lambda-calculus (STLC) has enough structure to make its theoretical properties interesting, but it is not yet much of a programming language. In this chapter, we begin to close the gap with more familiar languages by introducing a number of familiar features. Note that the type safety is preserved in all of these additions.

### 4.2 Unit Type and Sequencing

A *unit* type is a singleton type interpreted in the simplest possible way. Even in purely functional language, the type *Unit* is not completely without interest, but its main application is in languages with side effects, such as assignments to reference cells. In such languages, it is often the side effect, not the result, of an expression that we care about. *Unit* is an appropriate result type for such expressions. In languages with side effects, it is often useful to evaluate two or more expressions in *sequence*. The sequencing notation  $t_1; t_2$  has the effect of evaluating  $t_1$ , throwing away its trivial result, and going on to evaluate  $t_2$ . We need to add the following things to STLC to handle unit type and sequencing.

<b>Syntax</b>	$e ::= \text{unit} \mid e_1; e_2$
<b>Values</b>	$v ::= \text{unit}$
<b>Types</b>	$T ::= \text{Unit}$
<b>Semantics</b>	$\frac{e_1 \rightarrow e'_1}{e_1; e_2 \rightarrow e'_1; e_2}$ $\text{unit}; e_1 \rightarrow e_1$
<b>Typing Rules</b>	$\vdash \text{unit} : \text{Unit}$ $\frac{\Gamma \vdash e_1 : \text{Unit} \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1; e_2 : T_2}$

### 4.3 Let binding

When writing a complex expression, it is often useful - both for avoiding repetition and for increasing readability - to give names to some of its subexpressions. Most languages provide one or more ways of doing this. In ML, for example, we write  $\text{let } x = t_1 \text{ in } t_2$  to mean “evaluate the expression  $t_1$  and bind the name  $x$  to the resulting value, while evaluating  $t_2$ .”

Our let-binder follows ML’s in choosing a call-by-value evaluation order, where the let-bound term must be fully evaluated before evaluation of the let-body can begin. The formal additions to STLC are

<b>Syntax</b>	$e ::= \text{let } x = e_1 \text{ in } e_2$
<b>Semantics</b>	$\text{let } x = v \text{ in } e \rightarrow [x \mapsto v]e$ $\frac{e_1 \rightarrow e'_1}{\text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e'_1 \text{ in } e_2}$
<b>Typing Rules</b>	$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}$

### 4.4 Records and Variants

We introduce *records* and *variants* to STLC. Records and Variants are an aggregation of several items of possibly different types. The difference between records and variants is that in variants exactly one of the fields can be accessed.

Some examples are

**Record**

*let*  $x = \{real = 5, imag = 6\}$  *in*  
*square*( $x.real * x.real + x.imag * x.imag$ )

**Variant**

*Addr* =  $\langle physical : PhysicalAddr,$   
 $virtual : VirtualAddr \rangle$   
*let*  $a = \langle physical = pa \rangle$  *as* *Addr*;  
*getname* =  $\lambda a : Addr.$   
*case*  $a$  *of*  
 $\langle physical = x \rangle \Rightarrow x.firstlast$   
 $|\langle virtual = y \rangle \Rightarrow y.name$

The formal additions to STLC for records are

**Syntax**

$$e ::= \{l_i = e_i^{i \in 1 \dots n}\}$$
**Values**

$$v ::= \{l_i = v_i^{i \in 1 \dots n}\}$$
**Types**

$$\{l_i : T_i^{i \in 1 \dots n}\}$$
**Semantics**

$$\{l_i = v_i^{i \in 1 \dots n}\}.l_j \rightarrow v_j$$

$$\frac{e_1 \rightarrow e'_1}{e_1.l \rightarrow e'_1.l}$$

$$e_j \rightarrow e'_j$$

$$\frac{\{l_i = e_i^{i \in 1 \dots j-1}, l_j = e_j, l_i = e_i^{i \in j+1 \dots n}\} \rightarrow \{l_i = e_i^{i \in 1 \dots j-1}, l_j = e'_j, l_i = e_i^{i \in j+1 \dots n}\}}$$
**Typing Rules**

$$\frac{\text{for each } i \Gamma \vdash e_i : T_i}{\Gamma \vdash \{l_i = e_i^{i \in 1 \dots n}\} : \{l_i : T_i^{i \in 1 \dots n}\}}$$

$$\frac{\Gamma \vdash e : \{l_i : T_i^{i \in 1 \dots n}\}}{\Gamma \vdash e.l_j : T_j}$$

The formal additions to STLC for variants are

**Syntax**  
**Types**  
**Semantics**

$$e ::= \langle l = e \rangle \text{ as } T \mid \text{case } e \text{ of } \langle l_i = x_i \rangle \Rightarrow T_i^{i \in 1 \dots n}$$

$$T ::= \langle l_i : T_i^{i \in 1 \dots n} \rangle$$

$$\frac{\text{case } (\langle l_j = v_j \rangle \text{ as } T) \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{i \in 1 \dots n} \rightarrow [x_j \mapsto v_j] t_j}{e \rightarrow e'}$$

$$\frac{\text{case } e \text{ of } \langle l_i = x_i \rangle \Rightarrow e_i^{i \in 1 \dots n} \rightarrow \text{case } e' \text{ of } \langle l_i = x_i \rangle \Rightarrow e_i^{i \in 1 \dots n}}{e \rightarrow e'}$$

$$\frac{}{\langle l_i = e \rangle \text{ as } T \rightarrow \langle l_i = e' \rangle \text{ as } T}$$

**Typing Rules**

$$\frac{\Gamma \vdash e_j : T_j}{\Gamma \vdash \langle l_j = e_j \rangle \text{ as } \langle l_i : T_i^{i \in 1 \dots n} \rangle : \langle l_i : T_i^{i \in 1 \dots n} \rangle}$$

$$\frac{\Gamma \vdash e : \langle l_i : T_i^{i \in 1 \dots n} \rangle \text{ for each } i, \Gamma, x_i : T_i \vdash e_i : T}{\Gamma \vdash \text{case } e \text{ of } \langle l_i = x_i \rangle \Rightarrow e_i^{i \in 1 \dots n} : T}$$

## 4.5 Subtyping

Unlike the features we have studied up to now, which could be formulated more or less orthogonally to each other, *subtyping* is a cross-cutting extension, interacting with most other language features in non-trivial ways. Subtyping is characteristically found in object-oriented languages and is often considered an essential feature of the object-oriented style. We do not explore this connection in great detail but study subtyping with just functions and records.

Consider an example:  $(\lambda r : \{x : \text{Nat}\}. r.x) \{x = 0, y = 1\}$ . In this example, the type of the argument expected is  $\{x : \text{Nat}\}$ , but we are passing an argument of type  $\{x : \text{Nat}, y : \text{Nat}\}$ . The typing rules, in general, will reject this term as being bad. The idea of subtyping is to make these kind of programs usable.

A type  $S$  is a subtype of  $T$ , written  $S <: T$ , means that, a term of type  $S$  can be safely used wherever a term of type  $T$  is expected. Formally,

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

The formal additions to STLC are

**Syntax**

$$T ::= Top$$

**Typing Rules**

$$\begin{array}{c}
S <: S \\
\hline
S <: U \quad U <: T \\
\hline
S <: T \\
S <: Top \\
\hline
T_1 <: S_1 \quad S_2 <: T_2 \\
\hline
S_1 \rightarrow S_2 <: T_1 \rightarrow T_2 \\
\hline
\Gamma \vdash e : S \quad S <: T \\
\hline
\Gamma \vdash e : T \\
\hline
\{l_i : T_i^{i \in 1 \dots n+k}\} <: \{l_i : T_i^{i \in 1 \dots n}\} \\
\text{for each } i \quad S_i <: T_i \\
\hline
\{l_i : S_i^{i \in 1 \dots n}\} <: \{l_i : T_i^{i \in 1 \dots n}\} \\
\{k_j : S_j^{j \in 1 \dots n}\} \text{ is a permutation of } \{l_i : T_i^{i \in 1 \dots n}\} \\
\hline
\{k_j : S_j^{i \in 1 \dots n}\} <: \{l_i : T_i^{i \in 1 \dots n}\}
\end{array}$$

## 4.6 Pointers or References

Nearly every programming language provides some form of assignment operation that changes the contents of a previously allocated piece of storage. We can have a variable  $x$  whose value is the number 5, or a variable  $y$  whose value is a reference (or pointer) to a mutable cell whose current contents is 5, and the difference is visible to the programmer. We can add  $x$  to another number, but not assign to it. We can use  $y$  directly to assign a new value to the cell that it points to (by writing  $y := 84$ ), but we cannot use it directly as an argument to plus. Instead, we must explicitly dereference it, writing  $!y$  to obtain its current contents.

The basic operations on references are allocation, dereferencing, and assignment. To allocate a reference, we use the *ref* operator, providing an initial value for the new cell  $r = \text{ref } 5$ ; Here  $r$  is of type *Ref Nat*. To read the value from the cell  $r$ , we use  $!r$ . To change the value stored in the cell, we use the assignment operator  $r := 7$ . The result of this expression is *unit* : *Unit*. A simple example is

$$\begin{array}{l}
\text{let } x = \text{ref } 5 \text{ in } x := !x + 1 \\
!x
\end{array}$$

The formal additions to typing rules are

$$\frac{\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{ref } t_1 : \text{Ref } T_1}}{\Gamma \vdash t_1 : \text{Ref } T_1}}{\frac{\Gamma \vdash !t_1 : T_1}{\Gamma \vdash t_1 : \text{Ref } T_1} \quad \Gamma \vdash t_2 : T_1}}{\Gamma \vdash t_1 := t_2 : \text{Unit}}$$

A more subtle aspect of the treatment of references appears when we consider how to formalize their operational semantics. One way to see why is to ask, “What should be the *values* of type *Ref T*?” The crucial observation that we need to take into account is that evaluating a *ref* operator should allocate some storage and the result of the operation should be a reference to this storage.

What, then, is a reference?

The run-time store in most programming language implementations is essentially just a big array of bytes. The run-time system keeps track of which parts of this array are currently in use. We can think of the store as an array of values, rather than an array of bytes, abstracting away from the different sizes of the run-time representations of different values. Furthermore, we can abstract away from the fact that references (i.e., indexes into this array) are numbers. We take references to be elements of some uninterpreted set  $L$  of store locations, and take the store to be simply a partial function from locations  $l$  to values. We use the metavariable  $\mu$  to range over stores. A reference, then, is a location - an abstract index into the store. The statement  $e|\mu \rightarrow e'|\mu'$  means that a term  $e$  evaluates to  $e'$  changing memory from  $e$  to  $e'$  as a side effect.

The formal additions to STLC are

<b>Syntax</b>	$e ::= \text{ref } e \mid e_1 := e_2 \mid !e$
<b>Values</b>	$v ::= l$
<b>Types</b>	$T ::= \text{Ref } T$
<b>Semantics</b>	$\frac{\frac{\frac{e_1 \mid \mu \rightarrow e'_1 \mid \mu'}{!e_1 \mid \mu \rightarrow !e'_1 \mid \mu'}}{\mu(l) = v}}{!l \mid \mu \rightarrow v \mid \mu}}{e_1 \mid \mu \rightarrow e'_1 \mid \mu'}$ $\frac{e_1 := e_2 \mid \mu \rightarrow e'_1 := e_2 \mid \mu'}{e_2 \mid \mu \rightarrow e'_2 \mid \mu'}$ $\frac{v := e_2 \mid \mu \rightarrow v := e_2 \mid \mu'}{l := v_2 \mid \mu \rightarrow \text{unit} \mid [l \mapsto v_2] \mu}$ $\frac{e_1 \mid \mu \rightarrow e'_1 \mid \mu'}{\text{ref } e_1 \mid \mu \rightarrow \text{ref } e'_1 \mid \mu'}$ $\frac{l \notin \text{dom}(\mu)}{\text{ref } v \mid \mu \rightarrow l \mid (\mu, l \mapsto v)}$

We have not yet described the typing rules. To find the type of a location  $l$ , we look up the current contents of  $l$  in the store and calculate the type  $T_1$  of its contents.

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \text{Ref } T_1}$$

The type of the location is  $\text{Ref } T_1$ . In effect, by making the type of a term depend on the store, we have changed the typing relation from a three place relation (between contexts, terms and types) to a four place relation (between contexts, *stores*, terms and types). Our rule for typing rules now has the form

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T_1}{\Gamma \mid \mu \vdash l : \text{Ref } T_1}$$

However, there are two problems with this rule. First, typechecking is rather inefficient, since calculating the type of a location  $l$  involves calculating the type of the current contents  $v$  of  $l$ . If  $l$  appears many times in a term  $t$ , we will re-calculate the type of  $v$  many times in the course of constructing a typing derivation for  $t$ . Worse, if  $v$  itself contains locations, then we will have to recalculate their types each time they appear. For example, if the store contains

$$\begin{aligned}
& (l_1 \mapsto \lambda x : \text{Nat. } 999, \\
& l_2 \mapsto \lambda x : \text{Nat. } (!l_1)x, \\
& l_3 \mapsto \lambda x : \text{Nat. } (!l_2)x, \\
& l_4 \mapsto \lambda x : \text{Nat. } (!l_3)x, \\
& l_5 \mapsto \lambda x : \text{Nat. } (!l_4)x),
\end{aligned}$$

then calculating the type of  $l_5$  involves calculating those of  $l_4, l_3, l_2$  and  $l_1$ .

Second, the proposed typing rule for locations may not allow us to derive anything at all, if the store contains a cycle. For example, there is no finite typing derivation for the location  $l_2$  with respect to the store in the example

$$\begin{aligned}
& (l_1 \mapsto \lambda x : \text{Nat. } (!l_2)x, \\
& l_2 \mapsto \lambda x : \text{Nat. } (!l_1)x),
\end{aligned}$$

since calculating a type for  $l_2$  requires finding the type of  $l_1$ , which in turn involves  $l_2$ , etc. Cyclic reference structures do arise in practice (e.g., they can be used for building doubly linked lists), and we would like our type system to be able to deal with them.

Both of these problems arise from the fact that our proposed typing rule for locations requires us to recalculate the type of a location every time we mention it in a term. But this, intuitively, should not be necessary. After all, when a location is first created, we know the type of the initial value that we are storing into it. Moreover, although we may later store other values into this location, those other values will always have the same type as the initial one. These intended types can be collected together as a *store typing* - a finite function mapping locations to types. We'll use  $\Sigma$  to range over such functions. Now formally the typing rules are

$$\begin{array}{c}
\frac{\Sigma(l) = T_1}{\Gamma|\Sigma \vdash l : \text{Ref } T_1} \\
\frac{\Gamma|\Sigma \vdash e_1 : T_1}{\Gamma|\Sigma \vdash \text{ref } e_1 : \text{Ref } T_1} \\
\frac{\Gamma|\Sigma \vdash e_1 : \text{Ref } T_1}{\Gamma|\Sigma \vdash !e_1 : T_1} \\
\frac{\Gamma|\Sigma \vdash e_1 : \text{Ref } T \quad \Gamma|\Sigma \vdash e_2 : T}{\Gamma|\Sigma \vdash e_1 := e_2 : \text{Unit}}
\end{array}$$

## 4.7 Interaction: References and Subtyping

As we extend our simple calculus with subtyping toward a full-blown programming language, each new feature must be examined carefully to see how it interacts with subtyping. For example, the following typing rule is unsound

$$\frac{S <: T}{\text{Ref } S <: \text{Ref } T}$$

To see this more clearly, consider the following example.

```

let x = ref {a : 5, b : 10} in
let bar = λy : ref {a : Nat}. y := {a : 15} in
bar x;
x.b

```

Though the above program is correct as per the type system, the value of  $x.b$  cannot be calculated and hence this typing rule is unsound. The correct typing rule however is

$$\frac{S <: T \quad T <: S}{\text{Ref } S <: \text{Ref } T}$$

## 4.8 Let Polymorphism

Consider the following lambda terms.

```

let double = λf. λa. f(f(a)) in
let a = double(λx : Nat. succ(succ(x))) 1 in
let b = double(λx : Bool. not x) false in
...

```

Recall the typing rule for *let*

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}$$

Naturally this rule is not enough. Milner introduced a new typing rule to handle let polymorphisms.

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash [x \mapsto e_1]e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}$$

# Chapter 5

## Type Based Program Analysis - Pointer analysis

### 5.1 Introduction

The goal of this lecture is to show that the type systems can be used to specify and implement various program analyses. In this lecture, we will apply the framework of types, typing rules, typing judgements and type inference for non-standard types.

The problem we address is : given a C/C#/Java program, can we find for each pointer in the program, a conservative approximation to the set of all locations it can point to? There are various ways to address this problem: context sensitive, flow sensitive, field sensitive, object sensitive. However in this lecture we focus on an analysis which is flow, context, field and object insensitive. In the next section, we discuss Steensgaard's approach. In Section 3, Anderson's approach is discussed.

### 5.2 Steensgaard's Analysis

The lecture is the simplified version of the Steensgard's paper. Readers are requested to go through the paper "Points-to Analysis in Almost Linear Time". Since the Steensgaard approach is flow insensitive, the control structures of the program are irrelevant. We will also assume that the program does not have any procedures for the moment. It will be added later. Let us consider a syntax which is capable of handling pointers.

$$\begin{aligned}
S ::= & \quad x = y \\
& \quad | \quad x = \&y \\
& \quad | \quad *x = y \\
& \quad | \quad x = *y \\
& \quad | \quad x = \text{allocate}(n)
\end{aligned}$$

The syntax for computing the addresses of variables and for pointers are borrowed from the C programming language. All variables are assumed to have unique names. The  $\text{allocate}(n)$  expression dynamically allocates a block of memory of size  $n$ .

For the purposes of performing the points-to analysis, we define a non-standard set of types describing the store. The types have nothing to do with the types normally used in typed imperative languages (e.g, integer, float etc). We use types to model how storage is used in a program at runtime (a storage model). Each type describes a set of locations as well as the possible runtime contents of those locations. The non standard types used by our points-to analysis can be described by

$$\alpha = \perp \mid \text{ref}(\alpha)$$

Consider the following example.

```

a = &x;
b = &y;
if p then
  y = &z;
else
  y = &x;
c = &y;

```

The idea of the analysis is simple. We initialize the type of each variable to point to a separate location. For each assignment we unify the types on either side of the assignment. The following table shows the initial, intermediate and final types of each of the variables in the example.

Initial Types	After first 3 assignments	Finally
$a : \alpha_1 = \perp$	$a : \alpha_1 = \text{ref}(\alpha_4)$	$a : \alpha_1 = \text{ref}(\alpha_{4,6})$
$b : \alpha_2 = \perp$	$b : \alpha_2 = \text{ref}(\alpha_5)$	$b : \alpha_2 = \text{ref}(\alpha_5)$
$c : \alpha_3 = \perp$	$c : \perp$	$c : \alpha_3 = \text{ref}(\alpha_5)$
$x : \alpha_4 = \perp$	$x : \perp$	$x, z : \alpha_{4,6} = \perp$
$y : \alpha_5 = \perp$	$y : \alpha_5 = \text{ref}(\alpha_6)$	$y : \alpha_5 = \text{ref}(\alpha_{4,6})$
$z : \alpha_6 = \perp$	$z : \perp$	
$p : \alpha_7 = \perp$	$p : \perp$	$p : \alpha_7 = \perp$

**Typing Rules** The typing rules specify when a program is well-typed. A well-typed program is one for which the static storage graph indicated by the types is a safe (conservative) description of all possible dynamic (runtime) storage configurations.

$$\frac{A \vdash x : \alpha \quad A \vdash y : \alpha}{A \vdash WT(x = y)}$$

$$\frac{A \vdash x : ref(\alpha) \quad A \vdash y : \alpha}{A \vdash WT(*x = y)}$$

$$\frac{A \vdash x : ref(-)}{A \vdash WT(x = allocate(n))}$$

$$\frac{A \vdash x : ref(\alpha) \quad A \vdash y : \alpha}{A \vdash WT(x = \&y)}$$

$$\frac{A \vdash x : \alpha \quad A \vdash y : ref(\alpha)}{A \vdash WT(x = *y)}$$

The task of performing a points-to analysis has now been reduced to the task of inferring a typing environment ( $A$ ) under which a program is well-typed. More precisely, the typing environment we seek is the minimal solution to the well-typedness problem, i.e., each location type variable in the typing environment describes as few locations as possible.

The basic principle of the implementation is that we start with the assumption that all variables are described by different types (type variables) and then proceed to merge types as necessary to ensure well-typedness of different parts of the program. Merging is made fast by using fast union/find data structures. The efficient implementation runs in a time almost linear to the number of the variables in the program. The asymptotic time complexity is given by  $O(N * \alpha(N, N))$ , where  $\alpha$  is the inverse ackerman function and  $\alpha(N, N)$  is the cost of one union-find operation. Atmost  $N$  such operations are required.

Consider the following example.

`a = 4;`

$x = a;$   
 $y = a;$

The typing rules mentioned above requires that  $x, y$  and  $a$  be of the same type i.e, pointing same locations. However, these variables are not pointers. Our typing rules unnecessarily impose restrictions on these variables also. The actual typing rule for assignments in Steensgaard's paper is as follows.

$$\begin{array}{c}
 A \vdash x : ref(\alpha_1) \\
 A \vdash y : ref(\alpha_2) \\
 \alpha_2 \preceq \alpha_1 \\
 \hline
 A \vdash WT(x = y)
 \end{array}$$

where  $t_1 \preceq t_2 := (t_1 = \perp \vee t_1 = t_2)$

This rule avoids non-pointer variables for typing rules.

### 5.3 Anderson's Analysis

In Anderson's approach, the types are defined to be

$$\begin{array}{l}
 Locs = l_0 | l_1 \dots \\
 \alpha = Locs \mid ref(\{\alpha_1, \alpha_2, \dots\})
 \end{array}$$

#### Typing Rules

$$\frac{A \vdash x : \alpha \quad A \vdash y : \beta \quad PT(\beta) \subseteq PT(\alpha)}{A \vdash WT(x = y)}$$

$$\frac{A \vdash x : ref(X) \quad A \vdash y : \beta \quad \forall \alpha \in X. PT(\beta) \subseteq PT(\alpha)}{A \vdash WT(*x = y)}$$

$$\frac{A \vdash x : ref(X) \quad l_i \in X}{A \vdash WT(x = allocate_{l_i}(n))}$$

$$\frac{A \vdash x : ref(X) \quad A \vdash y : \beta \quad \beta \in X}{A \vdash WT(x = \&y)}$$

$$\frac{A \vdash x : \alpha \quad A \vdash y : ref(Y) \quad \forall \beta \in Y. PT(\beta) \subseteq PT(\alpha)}{A \vdash WT(x = *y)}$$

Let us revisit the earlier example with Anderson's approach.

Initial Types	After first 3 assignments	Finally
$a : \alpha_1 = ref(\{\})$	$a : \alpha_1 = ref(\{\alpha_4\})$	$a : \alpha_1 = ref(\{\alpha_4\})$
$b : \alpha_2 = ref(\{\})$	$b : \alpha_2 = ref(\{\alpha_5\})$	$b : \alpha_2 = ref(\{\alpha_5\})$
$c : \alpha_3 = ref(\{\})$	$c : \alpha_3 = ref(\{\})$	$c : \alpha_3 = ref(\{\alpha_5\})$
$x : \alpha_4 = ref(\{\})$	$x : \alpha_4 = ref(\{\})$	$x : \alpha_4 = ref(\{\})$
$y : \alpha_5 = ref(\{\})$	$y : \alpha_5 = ref(\{\alpha_6\})$	$y : \alpha_5 = ref(\{\alpha_4, \alpha_6\})$
$z : \alpha_6 = ref(\{\})$	$z : \alpha_6 = ref(\{\})$	$z : \alpha_6 = ref(\{\})$
$p : \alpha_7 = ref(\{\})$	$p : \alpha_7 = ref(\{\})$	$p : \alpha_7 = ref(\{\})$

We can observe that Anderson's algorithm is more accurate than Steensgaard's algorithm. However the time complexity of Anderson's algorithm is  $O(N^3)$ .

# References

1. “Foundational Calculi for Programming Languages”, Benjamin C. Pierce.
2. “Types and Programming Languages”, Benjamin C. Pierce