Lambda-Calculus

Michael Lewis

Summary The λ-calculus is a collection of formal systems that describe how functions or operators can be combined to form other operators.

First introduced in the 1930s by the American mathematician and logician Alonzo Church, it is a functional notion of computation. In the late 1930s both he and his PhD student Alan Turing, independently published papers to show that a general solution to the well-known *Entscheidungsproblem* [1] is impossible. Church used his λ-calculus and Turing, what we now term the Turing machine. These are equivalent models of computation[2].

The λ-calculus is Turing complete, *i.e.* can be used to simulate any single-taped Turing machine. It is the foundation of functional programming.

Contents

[1] The decision problem for first-order theories: Is there an *algorithm* which, given a set of axioms and a mathematical proposition, decides whether it is or is not provable from the axioms.

[2] Church-Turing thesis.

1 Introduction

Consider a simple expression '$x + y$'. We may define a function f (of one variable x) to represent this expression as $f(x) = x + y$ or $f : x \mapsto x + y$.
In the λ-notation this becomes $\lambda x.x + y$. So the application of the function $f(0)$ is:

$$f(0) = 0 + y, \quad (\lambda.x.x + y)(0) = 0 + y$$

The syntax may appear clumsy but it is intended to represent higher-order functions [3].

The notation can be extended to functions of more than one variable:

$$h(x, y) = x + y, \quad h = \lambda xy.x + y$$

To avoid the need for special notation for functions of several variables, we can use functions whose values are not numbers but other functions. For example:

$$\begin{aligned} h &= \lambda xy.x + y \\ h^* &= \lambda x.(\lambda y.x + y) \\ h^*(a) &= \lambda y.a + y, \text{ for each number } a. \end{aligned}$$

For each pair of numbers a and b, we have:

$$\begin{aligned} (h^*(a))(b) &= (\lambda y.a + y)(b) \\ &= a + b \\ &= h(a, b) \end{aligned}$$

Using h^* instead of h, we are translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument. This technique is called *currying*[4].

2 Pure Lambda Calculus

2.1 λ-terms

In the simplest form, everything in λ-calculus is an expression e or a variable x, for which we can define a simple grammar rule as follows:

$$e \to \; x \mid \lambda\, x.e \mid ee$$

[3] Examples can be understood easily by translating to the more usual $f(x)$ syntax when necessary.

[4] First introduced by Moses Shönfinkel and further developed by Haskell Curry.

More formally, the set of λ-terms is defined inductively as follows

(a) all **variables**, such as x are λ-terms.

(b) if M and N are any λ-terms, then (MN) is a λ-term called an **application**. It may be interpreted as applying M to the argument N.

(c) if M is any λ-term and x is any variable, then $(\lambda x.M)$ is a λ-term, called an **abstraction**.

2.1.1 Notation

Capital letters are used to donate arbitrary λ terms, with lower case letters denoting variables. In order to declutter the syntax a number of rules are applied:

(a) applications are left associative: $MNP \equiv ((MN)P)$.

(b) the body of an abstraction extends to the right as far as possible: $\lambda x.MN \equiv \lambda x.(MN)$.

(c) abstractions may be abbreviated, *e.g.* $\lambda x.\lambda y.\lambda z.N$ can be written as $\lambda xyz.N$.

Examples

If x, y, z are any distinct variables, the following are λ-terms (note, we can add brackets to aid readability)

(a) $(\lambda x.(xy))$

(b) $(x(\lambda x.(\lambda x.x)))$

(c) $((\lambda y.y)(\lambda x.(xy)))$

(d) $(\lambda x.(yz))$

Note, item (d) shows a term of form $(\lambda x.M)$ such that x does not occur in M. This is called a *vacuous abstraction.* These represent constant functions (functions whose output is the same for all inputs).

2.2 Free and bound variables

Given λ-terms P and Q, then P *occurs in* Q, (or P *is a subterm of* Q, or Q *contains* P) can be defined as

(a) P occurs in P

(b) if P occurs in M or N, then P occurs in MN

(c) if P occurs in M or $P \equiv x$, then P occurs in $\lambda x.M$

The meaning of '*an occurrence of P in Q*' is intuitively clear. For a particular occurrence of $\lambda x.M$ in a term P, the occurrence of M is called the *scope* of the occurrence of λx on the left. For example, let

$$P \equiv (\lambda y.yx(\lambda x.y(\lambda y.z)x))vw$$

The scope of the leftmost λy in P is $yx(\lambda x.y(\lambda y.z)x)$, the scope of λx is $y(\lambda y.z)x$, and that of the rightmost λy is z.

An occurrence of a variable x in a term P is called

(a) *bound* if it is in the scope of λx in P,

(b) *bound and binding*, iff it is the x in λx,

(c) *free* otherwise.

If x has at least one binding occurrence in P, we call x *a bound variable of* P. If x has at least one free occurrence in P, we call x a *free variable of* P. The set of all free variables of P is called $FV(P)$. A *closed term* is a term without any free variables.

Examples

(a) $\lambda y.x\ xy$ (*i.e.* $f(y) = xy$) then y is the bound variable and x is free.

(b) $\lambda x.y(\lambda x.z\ x)$, x is bound by the second abstraction.

2.3 Semantics

There are three kinds of term reductions within the λ-calculus:

(a) β-reduction, applying functions to their arguments. That is, we substitute the bound variable by the argument in the body of the abstraction.

(b) α-conversion *(or α-renaming)*, changing bound variables. Used to avoid name collisions.

(c) η-conversion, dealing with *extensionality*[5].

2.3.1 β-reduction

For any M, N, x, define $[N/x]M$ to be the result of substituting N for every free occurrence of x in M. For example (note, the $+$ operator is not part of the pure λ-calculus, though may be represented, it is included in this example for clarity),

$$(\lambda x.2x + y)(7) \Rightarrow 2 \times 7 + y$$

A reducible expression, or *redex*, is any expression to which the β-reduction rule can be *immediately* applied. For example, $\lambda x.(\lambda y.y)z$ is not a redex, but the nested expression $(\lambda y.y)z$ is. The *location* of a redex relative to another is the point at which it begins in a left-to-right scan of the text.

Some expressions may have more than one redex and it can be important to chose which redex to reduce first, generally we use *normal order* reduction. There are a number of strategies, these include:

(a) Normal-order reduction: Choose the left-most redex first.

[5]In λ-calculus, this means determining if two terms are observably equal.

(b) Applicative-order reduction: Choose the right-most redex first.

(c) Haskell evaluation: Choose the left-most redex first, only if it is not contained within the body of a lambda abstraction.

An expression with no redex is said to be in *normal form.* Note, some expressions have no terminating reduction *e.g.*

$$(\lambda x.xx)(\lambda x.xx) \Rightarrow (\lambda x.xx)(\lambda x.xx) \Rightarrow \ldots$$

There is a correlation between β reduction and parameter passing.

(a) Call by name, is the same as normal order reduction except that no redex in a lambda expression that lies within an abstraction is reduced.

(b) Call by value, is the same as applicative order reduction except that no redex in a lambda expression that lies within an abstraction is reduced.

2.3.2 α-conversion

Consider the following reduction:

$$(\lambda x.(\lambda y.x))y \Rightarrow \lambda y.y$$

The outer y is different from the inner y bound by the lambda. The solution is to *rename* the bound variable. For example,

$$\begin{aligned}&(\lambda x.\lambda y.xy)y\\ &\equiv (\lambda x.\lambda z.\ xz)y\\ &= \lambda z.yz\end{aligned}$$

Formally, let a term P contain an occurrence of $\lambda x.M$, and let $y \notin FV(M)$. The act of replacing this $\lambda x.M$ by $\lambda y.[y/x]M$ is called an α-conversion in P. Iff P can be changed to Q by a finite series of changes of bound variables, we say P is congruent to Q, or P α-converts to Q, or $P \equiv_\alpha Q$.
Example,

$$\begin{aligned}&\lambda xy.x(xy)\\ &\equiv \lambda x.(\lambda y.x(xy))\\ &\equiv_\alpha \lambda x.(\lambda v.x(xv))\\ &\equiv_\alpha \lambda u.(\lambda v.u(uv))\\ &\equiv \lambda uv.u(uv).\end{aligned}$$

2.3.3 η-conversion

η-conversion converts between $\lambda x.(fx)$ and f whenever x does not appear free in f. Example, $(\lambda y.\lambda x.yx)$ and $(y.y)$ are *observationally* equivalent.

3 Church Encoding

Church encoding is a means of representing data and operators in the λ-calculus.

3.1 Church Booleans

The Church encoding of true and false are functions of two parameters, where *true* choses the first parameter and *false* choses the second parameter. That is,

$$\begin{aligned}\text{true} &\equiv \lambda a.\lambda b.a\\ \text{false} &\equiv \lambda a.\lambda b.b\end{aligned}$$

A *predicate* (a function that returns a Boolean value) can now act as an *if-then-else* clause:

$$\text{predicate}\ \ x\ \ \text{then-clause}\ \ \text{else-clause}$$

as *true* and *false* choose the first or second parameter respectively. We can define logic operators similarly,

$$\begin{aligned}\text{and} &= \lambda p.\lambda q.p\ q\ p\\ \text{or} &= \lambda p.\lambda q.p\ p\ q\end{aligned}$$

The syntax can appear cumbersome, to recap,

$$\begin{aligned}&\lambda a.\lambda b.a\\ &\equiv \lambda ab.a\\ &\equiv f(a,b) = a\end{aligned}$$

Similarly, for the and operator,

$$\begin{aligned}&\lambda p.\lambda q.p\ q\ p\\ &\equiv \lambda pq\ pqp\\ &\equiv f(p,q) = p(q,p)\end{aligned}$$

For example, given $true = \lambda a.\lambda b.a$,

$$\begin{aligned}&\text{and}(true, false)\\ &= \lambda p.\lambda q.p\ q\ p\ \ \text{true}\ \ \text{false}\\ &= \text{true}(false, true)\\ &= false\end{aligned}$$

Church Booleans represented in Scala,

```
type Bool[T] = T => T => T
def True[T]: Bool[T] = a => b => a
def False[T]: Bool[T] = a => b => b
def and[T]: (Bool[T]) => (Bool[T]) => Bool[T]
        = p => q => a => b => p(q(a)(b))(p(a)(b))
def or[T]: (Bool[T]) => (Bool[T]) => Bool[T]
        = p => q => a => b => p(p(a)(b))(q(a)(b))

// To illustrate, can map built-in Boolean to Bool.
def bool: Bool[Boolean]
        => Boolean = a => a(true)(false)

bool(True) //> res0: Boolean = true
bool(False) //> res1: Boolean = false
bool(and(True)(False)) //> res2: Boolean = false
bool(and(False)(True)) //> res3: Boolean = false
bool(and(True)(True)) //> res4: Boolean = true
bool(or(False)(True)) //> res5: Boolean = true
bool(or(True)(False)) //> res6: Boolean = true
```

3.2 Church Numerals

Church numerals are the representations of $\mathbb{N}$ under Church encoding. Given a successor function S, which adds one, and *zero* (as follows), the natural numbers can be defined as:

$$\begin{aligned} 1 &= (S\ 0) \\ 2 &= (S\ 1) \equiv (S\ (S\ 0)) \\ 3 &= (S\ 2) \equiv (S\ (S\ (S\ 0))) \\ 4 &= \ldots \end{aligned}$$

A natural number n will be that n number of successors of zero. Alternatively, it may be expressed as the number of times some function f encapsulates its argument. As λ-expressions:

$$\begin{aligned} 0 &= \lambda f.\lambda x.x \\ 1 &= \lambda f.\lambda x.(f\ x) \\ 2 &= \lambda f.\lambda x.(f\ f(\ x)) \\ &\vdots \\ n &= \lambda f.\lambda x.f^n x \end{aligned}$$

We can define the successor function S as,

$$S \equiv \lambda n.\lambda f.\lambda x.f(nfx)$$

Using $f^{(m+n)}(x) = f^m(f^n(x))$ for an addition operator plus and $f^{(m\times n)}(x) = (f^n)^m(x)$ for a multiplication operator mult, we can define these operators as follows:

$$\begin{aligned} \text{plus} &= \lambda m.\lambda n.\lambda f.\lambda mf(nfx) \\ \text{mult} &= \lambda m.\lambda n.\lambda m(nf) \end{aligned}$$

Subtraction is more difficult and this was a problem until Kleene[6] found a way to represent a predecessor operator using pairs.

The encoding of the pair (two-tuple) type is as follows,

$$\begin{aligned} \text{pair} &= \lambda x.\lambda y.\lambda z.zxy \\ \text{first} &= \lambda p.\lambda p(\lambda x.\lambda y.x) \\ \text{second} &= \lambda p.\lambda p(\lambda x.\lambda y.y) \end{aligned}$$

For example,

$$\begin{aligned} \text{first (pair}\ \ a\ b) &= (\lambda p.p\ (\lambda x.\lambda y.x))\ ((\lambda x.\lambda y.\lambda z.z\ x\ y)\ a\ b) \\ &= (\lambda p.p\ (\lambda x.\lambda y.x))\ (\lambda z.z\ a\ b) \\ &= (\lambda z.z\ a\ b)\ (\lambda x.\lambda y.x) \\ &= (\lambda x.\lambda y.x)\ a\ b \\ &= a \end{aligned}$$

We can now define predecessor and subtraction operators. First we define a predecessor operator pred as:

$$\begin{aligned} f &= \lambda p.(\text{second}\ \ p)(\text{pair (first}\ \ p) false) \\ &\quad (\text{pair}\ (S\ (\text{first}\ \ p)\ false)) \\ \text{pc}_0 &= \text{pair}(\lambda f.\lambda x.x)\,\text{true} \\ \text{pred} &= \lambda n.\,\text{first}(n\ f\ pc_0) \end{aligned}$$

Given a way to define a predecessor, the minus operator is simply,

$$\text{minus} \equiv \lambda m.\lambda n.(n\ \ \text{pred})m$$

[6]Stephen Kleene a student of Church, who thought of the solution whilst at the dentist!

Church numerals represented in Scala,

```
type Num[T] = (T => T) => T => T
def succ[T]: Num[T] =>  Num[T]
= n => f => x => f(n(f)(x))
//> succ: [T]=> numerals.Num[T]
//                  => numerals.Num[T]

def zero[T]: Num[T] = f => x => x
//> zero: [T]=> numerals.Num[T]

def one[T]: Num[T] = f => x => f(x)
//> one: [T]=> numerals.Num[T]

def two[T]: Num[T] = f => x => f(f(x))
//> two: [T]=> numerals.Num[T]

def plus[T]: (Num[T]) => (Num[T]) => Num[T]
         = m => n => f => x => m(f)(n(f)(x))
//> plus: [T]=> numerals.Num[T]
// => (numerals.Num[T] => numerals.Num[T])

def toScalaInt(g: Num[Int]): Int
        = g((x: Int) => x + 1)(0)
//> toScalaInt: (g: numerals.Num[Int])Int

toScalaInt(plus(one)(two)) //> res0: Int = 3
```

4 Recursion

4.1 Y-Combinator

To achieve recursion in the λ-calculus a fixed point function [7] is required. That is, we need the ability to pass a function to itself as an argument. To do this, Haskell Curry defined what is called the Y combinator,

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

β-reduction of this gives,

$$\begin{aligned} Y\ g &= \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)) \\ &= (\lambda x.g(x\ x))(\lambda x.g(x\ x)) \\ &= g((\lambda x.g(x\ x))(\lambda x.g(x\ x))) \\ &= g(Y\ g) \end{aligned}$$

Repeatedly applying Y we have,

$$Y\ g = g(Y\ g) = g(g(Y\ g) = g(\ldots g(Y\ g)\ldots)$$

To illustrate this, the factorial function fact may be defined as:

$$\text{fact}\ \ n = \text{if}\ n = 0\ \text{then}\ 1\ \text{else}\ n * \text{fact}\ \ (n-1)$$

[7] A fixed point of a function f is a value that does not change under the application of the function f.

The Y combinator can be used to handle the recursion anonymously.
As represented in Scala,

```
// Y combinator
def Y[T](f: (T => T) => (T => T)): (T => T)
       = f(Y(f))(_:T)
       //> Y: [T](f: (T => T) => (T => T))T => T

// Mathematical definition of factorial.
def fn = { f: (Int => Int) => n: Int =>
       if (n <= 0) 1 else n * f(n - 1) }
//> fn: => (Int => Int) => (Int => Int)

Y(fn)(6) //> res0: Int = 720
```

5 Applied Lambda Calculus

The applied λ-calculus is an extension that introduces a constant terms, given by the syntax:

$$e \to x \mid \lambda x.e \mid ee \mid c$$

A *particular* applied λ-calculus is obtained by choosing a particular set of constants. These may be data values, functions, or functions on functions *etc*.

For example,

$$\begin{aligned} c \to\ & \mathbb{N} \mid \text{plus} \mid \text{Eq} \mid \text{isZero} \mid \text{true} \mid \text{false} \\ & \mid \text{if}\ e_0\ \text{then}\ e_1\ \text{else}\ e_2 \\ & \mid \text{let}\ x = e_2\ \text{in}\ e_1 \end{aligned}$$

Given the definitions for the *Church Numerals* we can define some simple expressions as follows:

$$\begin{aligned} \text{IsZero} &= \lambda n.n(\lambda x.\,\text{false})\,\text{true} \\ \text{LEq} &= \lambda m.\lambda n.\,\text{IsZero}(\text{minus}\,m\ n) \end{aligned}$$

Using, $x = y \equiv (x \leq y \wedge y \leq x)$,

$$\text{Eq} = \lambda m.\lambda n.\,\text{and}(\text{Leq}\,n\ m)$$

The let statement, in the simple form above, is just syntactic sugar,

$$\text{let}\,x = y\,\text{in}\,z \equiv (\lambda x.z)y$$

The let expression leads onto the idea of a typed λ-calculus. We have two questions:

(a) Given some expression e and some type τ does e have type τ? More generally, is e well typed? This is called type checking.

(b) Given an expression e is there a type τ such that e has type τ? This is called type inference.

6 Simply Typed Lambda-Calculus

6.1 Syntax

A typing environment represents the association between variables names and data types.

An environment Γ can be thought of as a set of ordered pairs $x{:}\tau$, where x is a variable name and τ[8] is its type. An expression e of type τ is written $e{:}\tau$. More formally, it is a partial function[9], $\Gamma{:}\mathit{Variables} \rightharpoonup \mathit{Types}$.

The syntax for inference is the same as used in propositional logic, *i.e.*

$$\frac{\Gamma_1 \vdash e_1{:}\tau_1 \quad \cdots \quad \Gamma_n \vdash e_n{:}\tau_n}{\Gamma \vdash e{:}\tau}$$

Which can be read as, if expression e_i has type τ_i in environment Γ_i, $\forall i = 1 \ldots n$, then the expression e will have an environment Γ and type τ.

The term syntax from the applied λ-calculus is updated to include the type information,

$$e \to x \mid \lambda x{:}\tau.e \mid ee \mid c$$

Similar we can define a type syntax, for example, for types t,

$$t \to \text{Int} \;\mid\; \text{Bool} \mid t_1 \to t_2 \mid t_1 \times\; t_2$$

Intuitively, if we have a function, we would expect it to have type $t_1 \to t_2$ and a tuple to have type $t_1 \times t_2$. Additionally, we have a *signature* Σ that assigns types to *constants*, *e.g.* $\Sigma(\text{plus}) \to \text{Int} \times \text{Int} \to \text{Int}$.

6.2 Semantics

We can categorise the semantics as a set of intuitive rules. These type substitution rules are used to build the type inferencing algorithm described later,

(a) [Var]

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma}$$

This is a simple tautology, given x has a type σ in environment Γ then you can infer that the type of x is σ, by defintion.

(b) [App]

$$\frac{\Gamma \vdash e_0{:}\tau \to \tau' \quad \Gamma \vdash e_1{:}\tau}{\Gamma \vdash e_0 e_1{:}\tau'}$$

If we have an expression (e_0) of type $\tau \to \tau'$ and an expression e_1 of type τ, then we can infer that the application of e_0 to e_1 has type τ'.

[8]σ is also often used to denote a type.

[9]We do not need to assign a type to every variable, only variables found in expressions.

(c) [Abs]

$$\frac{\Gamma, x{:}\tau \vdash e{:}\tau'}{\Gamma \vdash \lambda x.e{:}\tau \to \tau'}$$

If we can know x has type τ and can infer an expression e is of type τ', then a function of x returning expression e is of type $\tau - \tau'$.

(d) [Let]

$$\frac{\Gamma \vdash e_0{:}\sigma \quad \Gamma, x{:}\sigma \vdash e_1{:}\tau}{\Gamma \vdash \text{let}\, x = e_0 \,\text{in}\, e_1{:}\tau}$$

If we know e_0 has type σ and given x has type σ, we could infer that e_1 has type τ; then we can conclude that the result of letting $x = e_0$, and substituting it into e_1 has type τ.

6.3 Normalisation

Given the above semantics, the simply typed λ-calculus is strongly normalising. That is, well-typed terms will always reduce to a λ abstraction.

This is because recursion is not possible by the typing rules.

Theorem. If e is a recursion free λ-expression and e is closed (no free variables), if $\Gamma \vdash e{:}t$ then e is strongly normalising.

The term $(\lambda x.xx)(\lambda x.xx)$ for example, has no normal form and therefore can never be well typed.

Since the λ-calculus is strongly normalising, it is *decidable* whether or not a simply typed λ-calculus program halts[10], and therefore (unlike untyped λ-calculus) is not Turing complete.

Sometimes, the term simply typed λ-calculus ($\lambda^{\to}$ is used to refer to the simply typed λ-calculus together with a fixed point operator (Y combinator) and recursive let. This is not strictly correct.

7 Polymorphic Lambda-Calculus

7.1 Motivation

Under $\lambda^{\to}$ some reasonable programs fail to be well typed. Consider,

$$\lambda x.\,\text{if IsZero}\,7\,\text{then false else}\,4$$

The expression is not well typed but it is strongly normalising (the 'then' branch is not reachable). In the expression,

$$\text{let}\, f = \lambda x.x\,\text{in}(f\,\text{true}, f0)$$

We can see that there are two possible simple types for the expression,

[10]A simply typed λ-calculus program always halts.

(a) f true implies f has type $Bool \to Bool$,

(b) $f0$ implies type $Int \to Int$.

This is not allowed under simple typing.

What we would like to say is, let f have type $t \to t, \forall t$. That is, $f{:}\forall \alpha.\alpha \to \alpha$. This is called a type scheme. For example, we may want to *substitute* say Int or Bool for α. This makes α a quantified type, hence the term polymorphic.

7.2 Syntax

In terms of syntax, this is an extension to the types used for $\lambda^{\to}$, as follows,

$$\begin{aligned} t &\to \text{Int} \mid \text{Bool} \mid t_1 \to t_2 \mid t_1 \times\ t_2 \mid \alpha \\ T &\to t \mid \forall \alpha.T \end{aligned}$$

Here the types T can be either a type or a quantified type. In the rules above α is a type variable and $\forall \alpha.T$ a type scheme.

7.3 Semantics

The following type substitution rules are added to those defined for t$\lambda^{\to}$,

(a) [Inst]

$$\frac{\Gamma \vdash e{:}\sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e{:}\sigma}$$

Intuitively, we can see than the type Int $\to$ Int is an instantiation (or specialisation) of the polymorphic type $\forall \alpha.\alpha \to \alpha$, where we substitute Int for α.

More generally, if e has type σ' and σ' is a subtype of σ, then we can infer that e has type σ.

(b) [Generalization]

$$\frac{\Gamma \vdash e{:}\sigma \quad \alpha \ni \text{free}(\Gamma)}{\Gamma \vdash e{:}\forall \alpha.\sigma}$$

If some variable α is not free in the environment Γ and we can infer that expression e has type σ, then we can infer that e has type type σ independently of what α turns out to be, thus e has polymorphic type $\forall \alpha.\sigma$.

8 Polymorphic Type Inference

8.1 Motivation

For type inference we need to solve the following problem: Given a type environment Γ and expression e, is there a substitution σ and type t such that, $\sigma(e) \vdash e{:}t$?
An algorithm should find σ and t, iff they exist.

The idea behind the algorithm is to construct a type derivation tree and solve equations over type expressions. Intuitively, if we have some expression $e_1 e_2$, with types $e_2{:}t_2$, $e_1{:}t_1$, then $t_1 = t_2 \to t$, the question is what is the type of t?
That is given an $e_1 e_2$, we introduce a new type variable a, $t_1 = t_2 \to a$ and solve for a. In other words, find a type substitution for a.

Given $t = t'$ we need to find a substitution σ such that $\sigma(t) = \sigma(t')$. In fact, we want the most general substitution that satisfies this.

Definition. If σ, τ are type substitutions, then σ is more general than τ is there is a type substitution σ' such that, $\tau = \sigma'\sigma$.

For example, given type substitutions,

$$\begin{aligned} \sigma &= [t \mapsto (u \to u)] \\ \tau &= [t \mapsto ((v \times v) \to (v \times v))] \end{aligned}$$

There exists a substitution σ' such that $\tau = \sigma'\sigma$, where

$$\sigma' = [u \mapsto v \times v]$$

Definition. A type equation $s = t$ can be solved if there is a type substitution σ such that $\sigma(s) = \sigma(t)$. We call such a σ a *unifier* of s and t.

A most general unifier for s and t is a unifier such that it is more general than every other unifier for s and t. Sometimes, there are no unifiers, consider the type expressions,

$$\begin{aligned} s &= u_1 \times u_2 \\ t &= u_1 \to u_2 \end{aligned}$$

These have no unifier. There is no way of assigning types to u_1 and u_2 such that these two type expressions are identical.

8.2 Hindley-Milner and Algorithm W

Hildnley-Milner [11] is a type system that implements a polymorphic type system for the *lambda*-calculus, using the type rules described and an algorithm for type inferencing called *Algorithm W*.

[11] J. Roger Hindley and Robin Milner.

The W algorithm is a function that takes a type environment and expression $W(\Gamma, e)$ and returns a substitution and type (σ, t) such that $\sigma(\Gamma) \vdash e{:}t$, *i.e.* the application of the reduction implies e has type t. First assume there is an identity substitution *id* that returns type t for input type t.

First assume there is an algorithm unify, such that unify(s, t) returns the most general unifier of s and t if it exists.

We can define a set of tests for the type of some expression e within environment Γ; $W(\Gamma, e)$ as follows,

```
$W(\Gamma, e):$
$e = \mathrm{Int},$
    $\mathrm{if}\ e = 0\ \mathrm{return}(\mathrm{id}, \mathrm{Int})$
    $\mathrm{if}\ e = \mathrm{S}(e_1):$
        $\mathrm{if}\ W(\Gamma, e) = (\sigma, \mathrm{Int})\ \mathrm{return}(\sigma, \mathrm{Int})$
        $\mathrm{else\ fail}$

$e = \mathrm{Bool}, W(\Gamma, e):$
    $\mathrm{if}\ e = \mathrm{True\ return}(\mathrm{id}, \mathit{Bool})$
    $\mathrm{if}\ e = \mathrm{False\ return}(\mathrm{id}, \mathit{Bool})$
    $\mathrm{otherwise\ fail}$

$e = e_1 \times e_2, W(\Gamma, e):$
    $\mathrm{if}\ W(\Gamma, e_1) = (\sigma_1, t_1),\ W(\Gamma, e_2) = (\sigma_2, t_2)$
    $\mathrm{return}(\sigma_2\sigma_1, t_1 \times t_2)$
    $\mathrm{otherwise\ fail}$

$e = x\ (\mathrm{variable}), W(\Gamma, e):$
    $\mathrm{if}\ x \in \mathrm{dom}(\Gamma)\ \mathrm{and}\ \Gamma(x) = \forall a_1 \ldots a_k{:}s$
    $\mathrm{return}(\mathrm{id}, s)$
    $\mathrm{otherwise}\ x \notin \mathrm{dom}(\Gamma)\ \mathrm{fail}$

$e = e_1 e_2\ (\mathrm{application}), W(\Gamma, e):$
   Suppose:
      $W(\Gamma, e_1) = (\sigma_1, t_1)$
      $W(\Gamma, e_2) = (\sigma_2, t_2)$
   Let $a$ be a new type variable.
   If both calls succeed,
   $\sigma = \mathrm{unify}(\sigma_2(t_1), t_2 \to a)$
    $\mathrm{return}(\sigma\sigma_2\sigma_1, \sigma a)$
    $\mathrm{otherwise\ fail}$

$e = \lambda x.e_1\ (\mathrm{abstraction}), W(\Gamma, e):$
   Let $a$ be a new type variable.
    $\mathrm{if}\ W(\Gamma, x{:}a, e_1) = (\sigma, t)$
    return $(\sigma, \sigma(a) \mapsto t)$

$e = \mathrm{let}\ f = e_1\ \mathrm{in}\ e_2$
   Suppose:
      $W(\Gamma, e_1) = (\sigma_1, s_1)$
      $W(\Gamma', e_2) = (\sigma_2, s_2), \mathrm{where},$
      $\Gamma' = \sigma_1(\Gamma)[f \mapsto (\sigma_q(\Gamma), s_1)]$
       $\mathrm{return}(\sigma_2\sigma_1, s_2)$
       $\mathrm{otherwise\ fail}$
```