Unofficial Types Notes

Student written notes for the type theory course,
based on the 2021 lectures

J. O'Connor

jo429

THIS IS NOT AN OFFICIAL DOCUMENT!
This was written by students to collate and reinforce their own understanding and should not be used as a substitute for the official documents. There may be missing content, things may be written incorrectly or misunderstood and entire points may be missed.

Thanks to the following people for proofreading this document:

R. Laine

K. Druciarek

Contents

Types

Adding types to a programming language's design seems like an obvious choice[1] to make; we can add some guarantees to the format and uses of data, which can be checked at compile time via Type Checking, allowing for Type Safety to be guaranteed before the program is even run. Type systems are often also used as a form of documentation, where in the names of types may dictate the function of the data stored within instances of the types. This leads naturally to Object Oriented Programming, where the types not only represent the format and guarantees on the data stored within the type, but also the functions that are allowed to be called on the data. Type systems also lead to faster code, as the compilers for type safe languages can make strong assumptions about the data stored in a memory location based on the type of the variable that the memory location corresponds to.

However, type systems lead a double life. Once we begin to formalise some of the above notions of 'Type Safety' and 'Typing judgements', we begin to see a direct parallel between **Types**, and **Logic & Proof**.

Semantics of Programming Languages Revision

In IB Semantics of Programming Languages we saw how we can define a grammar of a language, then an Operational Semantics and Typing Judgement for terms within this grammar. In II Types we start with an understanding of these concepts.

For example, we may have a simple language of Booleans and Integers. Its grammar may look like the following:

$$e ::= true \mid false \mid n \mid e_1 \leq e_2 \mid e_1 + e_2 \mid e_1 \wedge e_2 \mid \neg e$$

From this grammar, we can begin to build terms:

$$3 + 4 \leq 5$$
$$(3 + 4 \leq 7) \wedge (7 \leq 3 + 4)$$

Excellent! However, we can also build some terms that don't make sense:

$$4 \wedge true$$
$$false + 7$$

The obvious thing to do is to modify our grammar to only allow for valid terms. In this language, we can do this by splitting the terms into three rules:

[1] except to Lisp programmers

$$e_1 ::= n \mid e_1 + e_2$$
$$e_2 ::= true \mid false \mid e_1 \leq e_1 \mid e_2 \wedge e_2 \mid \neg e_2$$
$$e ::= e_1 \mid e_2$$

By doing this, we have actually introduced a basic notion of types. We can see that e_1 is all of the expressions with type *number* and e_2 is all of the expressions with type *boolean*.

Unfortunately, then, we have managed to entangle our concepts of types and expressions. To disentangle these ideas, we introduce *typing judgements* on expressions, where a judgement only exists if an expression is typed correctly, or *well-typed*. This means that we can have

Returning to our original language of Booleans and Integers:

$$e ::= true \mid false \mid n \mid e_1 \leq e_2 \mid e_1 + e_2 \mid e_1 \wedge e_2 \mid \neg e$$

We can add judgement rules for every expression with a type as follows:

$$\frac{}{n : \mathbb{N}}\ \text{Num} \qquad \frac{}{true : bool}\ \text{True} \qquad \frac{}{false : bool}\ \text{False}$$

$$\frac{e : bool}{\neg e : bool}\ \text{Neg} \qquad \frac{e : \mathbb{N} \qquad e' : \mathbb{N}}{e + e' : \mathbb{N}}\ \text{Plus}$$

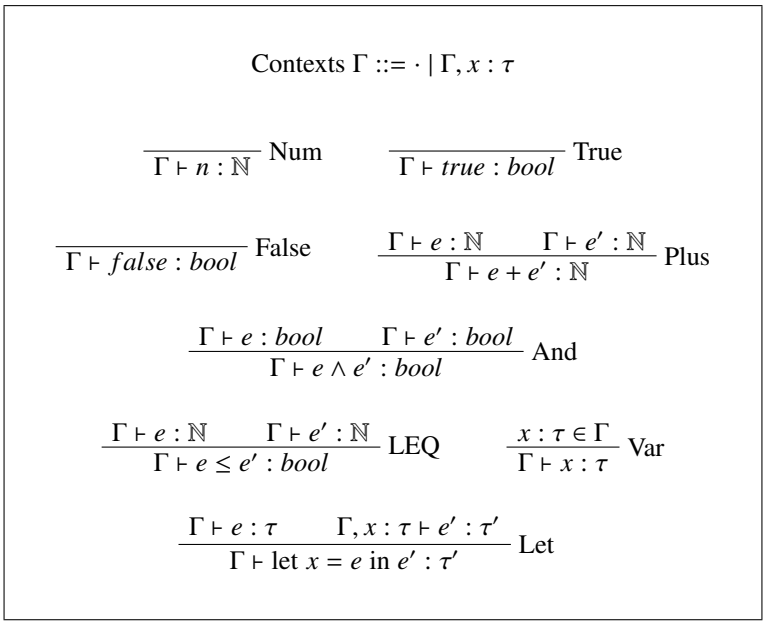
$$\frac{e : bool \qquad e' : bool}{e \wedge e' : bool}\ \text{And} \qquad \frac{e : \mathbb{N} \qquad e' : \mathbb{N}}{e \leq e' : bool}\ \text{LEQ}$$

Note that we have not yet defined how these expressions actually behave when we step through their execution. Typing is a process done on expressions before they are run, and does not change the path that the expression takes when it is executed. Typing judgements simply tell us whether an expression is *well-typed*, and we can show that well-typed expressions within some languages have nice properties like **Termination**.

But now comes the question of variables. For example, the following statement should only be valid if the variable x stores a number:

$$(x + x) \leq 10$$

To do this we add a context which holds the type of every variable in the expression. This context propagates through the proof tree as we build it from the bottom up, allowing us to see the types of variables within the local context in which they occupy.



Structural Properties and Substitution

We have introduced variables into our language, so we should introduce a notion of substitution as well:

$$
\begin{aligned}
[e/x]true &= true \\
[e/x]false &= false \\
[e/x]n &= n \\
[e/x](e_1 + e_2) &= [e/x]e_1 + [e/x]e_2 \\
[e/x](e_1 \leq e_2) &= [e/x]e_1 \leq [e/x]e_2 \\
[e/x](e_1 \wedge e_2) &= [e/x]e_1 \wedge [e/x]e_2 \\
[e/x]x &= e \\
[e/x]z &= z \\
[e/x](\text{let } z = e_1 \text{ in } e_2) &= \text{let } z = [e/x]e_1 \text{ in } [e/x]e_2 \\
&\quad (\text{assuming } z \notin dom(e))
\end{aligned}
$$

These rules are akin to β-reduction in Lambda Calculus, and should be rules that you are very familiar with. Note that we did not need to define these substitution rules when defining the sequents for types in the previous section - we only use substitution when evaluating an expression, or when describing properties that hold for well-typed expressions.

There are three properties that we like to have hold for any type system. These are as follows:

1. (**Weakening**)

   If a term typechecks in a context, then it will still typecheck in a bigger context.

   $\Gamma, \Gamma' \vdash e : \tau \implies \Gamma, x : \tau'', \Gamma' \vdash e : \tau$

2. (**Exchange**)

   If a term typechecks in a context, then it will still typecheck after reordering the variables in the context.

   $\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e : \tau \implies \Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e : \tau$

3. (**Substitution**)

   Subsituting a type-correct term for a variable will preserve type correctness.

   $(\Gamma \vdash e : \tau) \wedge (\Gamma, x : \tau \vdash e' : \tau') \implies \Gamma \vdash [e/x]e' : \tau'$

These properties are all proven by structural induction in the lectures. That is, by proving that each property holds for each possible structure of expression independently, we prove that all expressions must have these properties.

Operational Semantics

We have a language and type system. We therefore have an idea for what programs are valid within our language and also what the type of the value calculated will be. How do we say what value a program computes? With an *Operational Semantics*, a two-place relation on terms $e \rightsquigarrow e'$, pronounced as "e steps to e prime".

Values $v ::= n \mid true \mid false$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \wedge e_2 \rightsquigarrow e_1' \wedge e_2}\ \text{AndCong} \qquad \frac{}{true \wedge e \rightsquigarrow e}\ \text{AndTrue}$$

$$\frac{}{false \wedge e \rightsquigarrow false}\ \text{AndFalse}$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \leq e_2 \rightsquigarrow e_1' \leq e_2}\ \text{LEQCong1} \qquad \frac{e \rightsquigarrow e'}{v \leq e \rightsquigarrow v \leq e'}\ \text{LEQCong2}$$

$$\frac{n_1 \leq n_2}{n_1 \leq n_2 \rightsquigarrow true}\ \text{LEQTrue} \qquad \frac{n_1 > n_2}{n_1 \leq n_2 \rightsquigarrow false}\ \text{LEQFalse}$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 + e_2 \rightsquigarrow e_1' + e_2}\ \text{AddCong1} \qquad \frac{e \rightsquigarrow e'}{v + e \rightsquigarrow v + e'}\ \text{AddCong2}$$

$$\frac{n_1 + n_2 = n_3}{n_1 + n_2 \rightsquigarrow n_3}\ \text{AddStep}$$

$$\frac{e_1 \rightsquigarrow e_1'}{\text{let } z = e_1 \text{ in } e_2 \rightsquigarrow \text{let } z = e_1' \text{ in } e_2}\ \text{LetCong}$$

$$\frac{}{\text{let } z = v \text{ in } e \rightsquigarrow [v/z]e}\ \text{LetStep}$$

A reduction sequence is a sequence of transitions $e_1 \rightsquigarrow e_2, e_2 \rightsquigarrow e_3, ..., e_{n-1} \rightsquigarrow e_n \implies e_1 \rightsquigarrow^* e_n$ A term e is stuck if it is not a value, and there is no e' such that $e \rightsquigarrow e'$

Stuck terms are erroneous programs with no defined behaviour, so we like to avoid them. Therefore we have two properties that we like to have hold:

1. (**Progress**)

   Well-typed programs are not stuck: they can always take a step of progress (or are done).

   $\cdot \vdash e : \tau \implies (e \in v) \vee (\exists e'.e \rightsquigarrow e')$

2. (**Preservation**)

   If a well-typed program takes a step, it will stay well-typed.

   $(\cdot \vdash e : \tau) \wedge (e \rightsquigarrow e') \implies \cdot \vdash e' : \tau$

All five of these key properties of **Weakening**, **Exchange**, **Substitution**, **Progress** and **Preservation** are collectively known as *Type Safety*

These properties are also proven by structural induction in the lectures.

Typed Lambda Calculus

The above language is fine, but it has a lot of fluff with numbers and addition and so on. We've seen that we can represent computability with Lambda Calculus, which doesn't come with any fancy numbers or booleans. We can add types to lambda calculus exactly as you'd expect:

$$
\begin{aligned}
X &::= 1 | X \times Y \mid 0 \mid X + Y \mid X \to Y \\
e &::= x | \langle\rangle \mid \langle e, e' \rangle \mid \text{fst } e \mid \text{snd } e \mid \text{abort } e \\
&\quad\; | Le \mid Re \mid \text{case}(e, Lx \to e', Ry \to e'') \\
&\quad\; | \lambda x : X.e \mid ee' \\
\Gamma &::= \cdot | \Gamma, x : X
\end{aligned}
$$

Where the only 'base' types are unit $\langle\rangle$ with type 1 and the empty type $\perp$ that cannot be constructed with type 0. See the next section to understand why the empty type is in our system.

Note that we have inherently restricted the expressions that we can form with this new typed calculus. This was actually our aim - we want to prevent the formation of expressions that do not have type safety. However our notation of type safety at this point is restrictive, and there are some expressions within lambda calculus which terminate but cannot be expressed here. For example, you cannot create the Ackermann function within STLC, but we could within untyped Lambda Calculus.

Also of interest is that we've added product and sum types to our simply typed lambda calculus, which weren't present in the original lambda calculus. This is because in LC we can represent these ideas purely through functions and function applications, however our type system is restrictive and prevents us from encoding these structures (See Polymorphic Lambda Calculus later on for more detail). Therefore we add product and sum expressions, just to preserve some of the computability of Lambda Calculus. Other typed lambda caluli we can add other combinations of these expressions, but our STLC has function applications, products, sums and the empty type $\perp$.

The typing derivations are as follows:

$$\frac{}{\Gamma \vdash \langle\rangle : 1}\ 1\mathrm{I} \qquad \frac{\Gamma \vdash e : X \qquad \Gamma \vdash e' : Y}{\Gamma \vdash \langle e, e' \rangle : X \times Y}\ \times\mathrm{I}$$

$$\frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \mathrm{fst}\ e : X}\ \times\mathrm{E}_1 \qquad \frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \mathrm{snd}\ e : Y}\ \times\mathrm{E}_2$$

$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X}\ \mathrm{HYP} \qquad \frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x : X.e : X \to Y}\ \to\mathrm{I}$$

$$\frac{\Gamma \vdash e : X \to Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash ee' : Y}\ \to\mathrm{E} \qquad \frac{\Gamma \vdash e : X}{\Gamma \vdash Le : X + Y}\ +\mathrm{I}_1$$

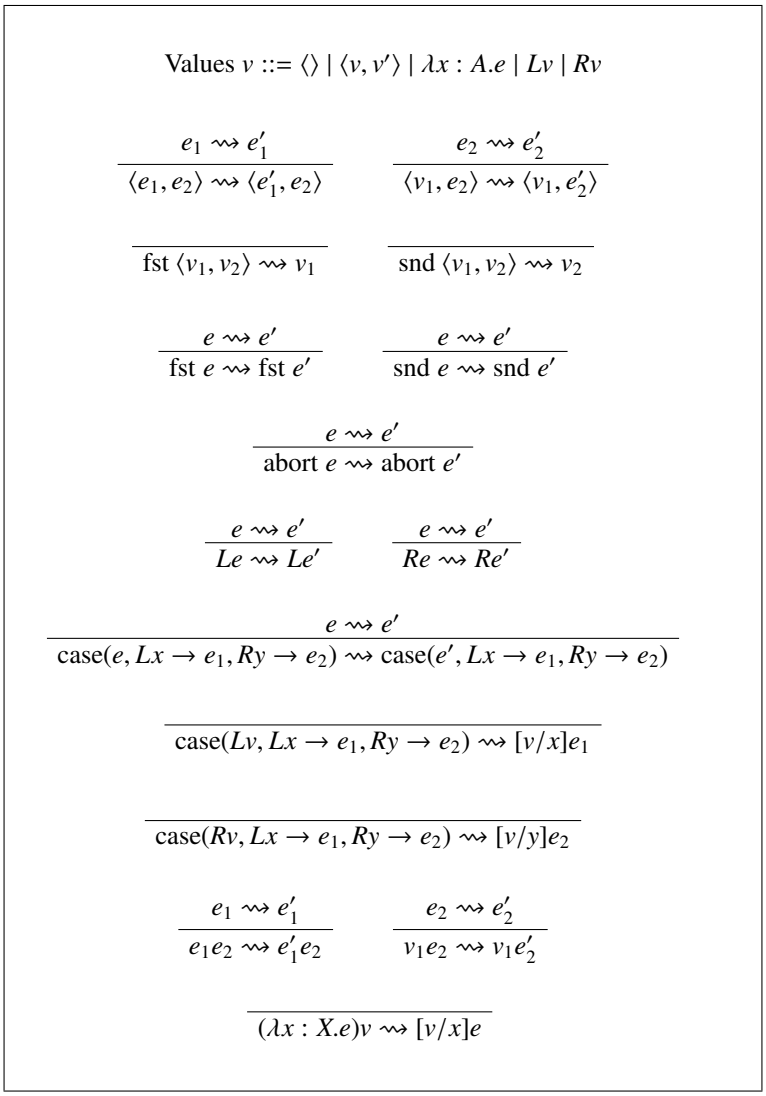
$$\frac{\Gamma \vdash e : Y}{\Gamma \vdash Re : X + Y}\ +\mathrm{I}_2$$

$$\frac{\Gamma \vdash e : X + Y \qquad \Gamma, x : X \vdash e' : Z \qquad \Gamma, y : Y \vdash e'' : Z}{\Gamma \vdash \mathrm{case}(e, Lx \to e', Ry \to e'') : Z}\ +\mathrm{E}$$

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \mathrm{abort}\ e : Z}\ 0\mathrm{E}$$

Note that we cannot build any expression of type 0, but if we could then we could abort it to create any other type we want. We will see that this is akin to being able to prove anything from falsehood.

We then add the operational semantics:



The five key properties of **Weakening**, **Exchange**, **Substitution**, **Progress** and **Preservation** all hold for these rules.

The Curry-Howard Correspondence

Imagine that we have a program that has type $A \to B$. Then we know that if we have an instance of the type A then we can feed it in to the program and get an instance of the type B.

This looks suspiciously like Modus Ponens. That is, if we assume that having an instance of a type is equivalent to proving that the type is 'true', then we have derived the following rule:

$$\frac{\text{A} \qquad \text{A} \to \text{B}}{\text{B}}$$

This shows a mapping between a function of type $A \to B$ and implication $A' \implies B'$ where A' and B' are the prepositions that the types A and B correspond to.

We can also see that, if having an instance of a type is equivalent to proving that the corresponding preposition is 'true', then the type that corresponds to falsehood must be the type that has no instance. We call this type $\bot$

Continuing in this line of reasoning, we can imagine Conjunction as product types since we must have evidence for both elements to generate the product, and Disjunction as Unions since we only need evidence of a single of the elements to generate the union.

It is also useful to establish the concept of a normal form for a proof. For example, take the following expression:

$$\lambda x : A.\lambda y : A \to B.yx$$

This expression has the type $A \to (A \to B) \to B$

We could also consider a slightly more complicated function:

$$\lambda x : A.\lambda y : A \to B.((\lambda c : A \to B.c)y)((\lambda b : A.b)x)$$

This expression also has the type $A \to (A \to B) \to B$

Therefore both expressions are an instance of the type corresponding to Modus Ponens, so both functions can be interpreted as proofs for the same thing.

However if you study the expressions for a bit you'll notice that there is some fluff in the second expression that will be evaluated away immediately: $(\lambda b : A.b)x \rightsquigarrow [x/b]b = x$. In fact, the second expression will reduce down to the first.

You'll also notice that the first expression is a value. That is, it cannot be reduced any further. Therefore we can assert that the the first expression is a normal form, and proof normalisation is equivalent to stepping the expression to a value.

This gives us the following table:

Logic	Programming
Formulas	Types
Proofs	Programs
Truth	Unit
Falsehood	Empty Type
Conjugation	Products
Disjunction	Unions
Implication	Functions
Normal Form	Value
Proof Normalization	Evaluation
Normalization Strategy	Evaluation Order

We can construct an expression of Negation using our idea of Falsehood, of $\bot$. Since a type being false corresponds to there being no object that inhabits the type, then proving that a statement is false is equivalent to proving that if the statement were true, we would be able to inhabit $\bot$.

Therefore the logical statement $\neg A$ equates to the type $A \to \bot$.

Interestingly, we find that we cannot prove some statements within this system that we would normally take as valid. For example, we cannot construct a function of the following type:

$$((A \to \bot) \to \bot) \to A$$

Which means that, within this system of proving statements by finding instances of types, we cannot prove that $\neg\neg A \implies A$ (Double Negation Elimination). We also find that we cannot prove $A \vee \neg A$ (Law of Excluded Middle).

Overall, the set of logical statements that we are able to prove with the Curry-Howard Correspondence does not contain every logical statement that we can prove to be true in Classical Logic. This system of logic is instead called Intuitionistic Propositional Logic, and is in fact equivalent to Classical Logic, just without DNE as an axiom.

Not Halting and Falsehood

We said before that there should be no way to generate an instance of $\bot$, as it should equate to falsehood and since any instance of a type equates to the type being true, then $\bot$ should have no instances. If we were able to generate a proof (i.e an example program) which suggests from its type that we can generate a term of type $\bot$ then our logic is inconsistent.

Since we know that there is no value of type $\bot$, and also that all well-typed programs in STLC must be a value or progress, we know that the only hope that we have to create a term that types to $\bot$ would be a term that loops forever and does not halt.

However if we try to create a function which loops forever (and so types to $\bot$) we find that we cannot type it. For example the infinite looping function Ω:

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

We know that Ω loops forever, but what would its type be? Well if we try to start figuring it out, we can see that for each half of Ω, the parameter x must have a type. This type must be a function, since x is applied to something, and the parameter of the function must be the type of x. So we get that $x : A \to B$ and also that $A = A \to B$. This infinite type is not allowed within STLC so Ω is not a well-typed expression.

This is valid intuition as to why STLC cannot loop forever (and so we cannot build expressions that have type $\bot$), but for a proof we need to be more rigorous.

A Proof of Termination

So far in this course we have used structural induction to prove most of the properties that we want to hold. Unfortunately, as seen in the slides, proving termination using structural induction is not possible. This is because the result of function application may itself be a function application, and so we cannot use the inductive hypothesis to assume that the result of function application terminates since we are in the process of proving if function application terminates!

Instead we first build a collection of sets of terms named *Halt*, where the following hold:

- $Halt_0 = \emptyset$

  i.e. for all e, $e \notin Halt_0$
- $e \in Halt_1$ when e halts
- $e \in Halt_{X \to Y}$ when:
  - $e \in Halt_1$ (i.e. e halts)
  - $\forall e' \in Halt_X . (ee') \in Halt_Y$

We can therefore read a set *Halt* in the following way:

- $Halt_1$ is the set of expressions that halt.
- $Halt_{1 \to 1}$ is the set of expressions that halt when applied to an expression that also halts.
- $Halt_{(1 \to 1) \to 1}$ is the set of expressions that halt when applied to an expression e' that halts when applied to an expression that halts.
- $Halt_{1 \to (1 \to 1)}$ is the set of expressions that, when applied to an expression that halts, result in an expression which preserves halting.
- $Halt_{(1 \to 1) \to (1 \to 1)}$ is the set of expressions that, when applied to an expression which preserves halting, result in an expression which preserves halting.

and so on.

So essentially for all expressions in a language to always halt, all expressions e in that language must be in $Halt_1$, though this is not a property that we will attempt to prove, since we only really care whether $\perp$ can be formed, which can be proven in a different way using the fundamental lemma.

Closure Lemma

Before we prove the *fundamental lemma* we first prove the *closure lemma*. That is:

$$e \rightsquigarrow e' \implies (e' \in Halt_X \iff e \in Halt_X)$$

Or, in english, if some property of halting or preserving halting is held by some expression e then any expression that steps to e and any expression that e steps to must also preserve that property of halting or preserving halting.

We prove the statement by induction on X and this can be seen in the lecture slides.

Fundamental Lemma

The fundamental lemma is as follows:

$$\begin{gathered} x_1 : X_1, ..., x_n : X_n \vdash e : Z \\ \wedge \forall i \in 1...n.(\cdot \vdash v_i : X_i) \wedge (v_i \in Halt_{X_i}) \\ \implies [v_1/x_1, ..., v_n/x_n]e \in Halt_Z \end{gathered}$$

That is, if we have an expression e which has type Z and free variables x_1 to x_n with types X_1 to X_n, and we can show that there are values v_1 to v_n with types X_1 to X_n that all preserve halting as described by their respective type X_i, then if we substitute all of these values in to the variables in e we get an expression that preserves halting as described by the type Z.

Or, in English, haltingness is invarient under the $\rightsquigarrow$ relation.

This is interesting because we get the halting property for e seemingly from nowhere. The clever part is that when we apply an expression to a value, we add the value to the set of substitutions and so we can refer back to the inductive hypothesis (step 8 for $Case \rightarrow l$)

And from this we can prove **Consistency**:

$$\text{There are no terms } \cdot \vdash e : 0$$

1. Assume $\cdot \vdash e : 0$
2. $e \in Halt_0$ by Fundamental lemma
3. $Halt_0 = \emptyset$ by definition

which gives a contradiction.

The Halting Problem

Since every closed program reduces to a value, and there are no values of empty type, there are no programs of the empty type. But the only programs of the empty type are the ones that do not halt. So have we avoided the halting problem?

$$e \text{ well-formed} \wedge \cdot \vdash e : \tau \implies e \text{ Halts}$$
$$e \in \text{Simply Typed Lambda Calculus} \implies e \text{ Halts}$$

The thing to notice is that this isn't a bi-implication! There are programs within LC that do halt but are not accepted by STLC, e.g. *ack*. So then how can we make STLC stronger?

Loops

We know from Foudations of Computer Science that while loops can be represented using unbounded recursion. However unfortunately adding unbounded recursion runs us straight into the issue that we were trying to avoid above; we can construct terms that loop forever and so typecheck to 0. For example:

$$\cdot \vdash (fun_{1\to 0} f\, x.\, f\, x)\langle\rangle : 0$$

However we do know that recursion with a base case that will be hit, or a for loop with a bounded upper bound, will always terminate as long as the code being run within that loop also terminates. This leads us to adding bounded recursion, and inventing Gödel's T.

Gödel's T

Gödel's T begins as STLC, but we add integers and bounded iteration over those integers. This gives us the following grammar:

$$
\begin{aligned}
X ::= {} & 1 \mid X \times Y \mid 0 \mid X + Y \mid X \to Y \mid \mathbb{N} \\
e ::= {} & x \mid \langle\rangle \mid \langle e, e' \rangle \mid \text{fst } e \mid \text{snd } e \mid \text{abort } e \\
& \mid Le \mid Re \mid \text{case}(e, Lx \to e', Ry \to e'') \\
& \mid \lambda x : X.e \mid ee' \\
& \mid z \mid s(e) \mid \text{iter}(e, z \to e', s(x) \to e'') \\
\Gamma ::= {} & \cdot \mid \Gamma, x : X
\end{aligned}
$$

And the following rules **in addition to the ones within STLC**:

$$\frac{}{\Gamma \vdash z : \mathbb{N}}\ \mathbb{N}I_z \qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash s(e) : \mathbb{N}}\ \mathbb{N}I_s$$

$$\frac{\Gamma \vdash e_0 : \mathbb{N} \qquad \Gamma \vdash e_1 : X \qquad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \text{iter}(e_0, z \to e_1, s(x) \to e_2) : X}\ \mathbb{N}E$$

$$\frac{e_0 \rightsquigarrow e_0'}{\text{iter}(e_0, z \to e_1, s(x) \to e_2) \rightsquigarrow \text{iter}(e_0', z \to e_1, s(x) \to e_2)}$$

$$\frac{}{\text{iter}(z, z \to e_1, s(x) \to e_2) \rightsquigarrow e_1}$$

$$\frac{}{\text{iter}(s(v), z \to e_1, s(x) \to e_2) \rightsquigarrow [\text{iter}(v, z \to e_1, s(x) \to e_2)/x]e_2}$$

We can see that the above language is at least as powerful as primitive recursion, however it is a little bit more powerful than that. It can't be as powerful as partial recursion since:

1. we have still preserved our property of every well-typed expression halting
2. being as powerful as partial recursion would mean the ability to compute all computable functions

Therefore being as powerful as partial recursion would violate The Halting Problem. But it sits somewhere in-between. For example, *ack* is computable with Gödel's T but not by primitive recursion.

More Data Structures

We have just added integers within Gödel's T, but we might want to go further and add some more useful structures like lists.

The naive approach is to add these structures to our language through sequents:

$$\frac{}{\Gamma \vdash [] : \text{list } X}\ \text{ListNil}$$

$$\frac{\Gamma \vdash e : X \qquad \Gamma \vdash e' : \text{list } X}{\Gamma \vdash e :: e' : \text{list } X}\ \text{ListCons}$$

$$\frac{\Gamma \vdash e_0 : \text{list } X \qquad \Gamma \vdash e_1 : Z \qquad \Gamma, x : X, r : Z \vdash e_2 : Z}{\Gamma \vdash \text{fold}(e_0, [] \to e_1, x :: r \to e_2) : Z}\ \text{ListFold}$$

$$\frac{e_0 \rightsquigarrow e_0'}{e_0 :: e_1 \rightsquigarrow e_0' :: e_1} \qquad \frac{e_1 \rightsquigarrow e_1'}{v_0 :: e_1 \rightsquigarrow v_0 :: e_1'}$$

$$\frac{e_0 \rightsquigarrow e_0'}{\text{fold}(e_0, [] \to e_1, x :: r \to e_2) \rightsquigarrow \text{fold}(e_0', [] \to e_1, x :: r \to e_2)}$$

$$\frac{}{\text{fold}([], [] \to e_1, x :: r \to e_2) \rightsquigarrow e_1}$$

$$\frac{R \triangleq \text{fold}(v', [] \to e_1, x :: r \to e_2)}{\text{fold}(v :: v', [] \to e_1, x :: r \to e_2) \rightsquigarrow [v/x, R/r]e_2}$$

However the biggest issue here is that now when we define functions over these lists within this Typed Lambda Calculus we must define the type of the list within the function signature. For example, consider the map function:

$$\lambda f : A \to B.\lambda xs : \text{List } A.$$
$$\text{fold}(xs, [] \to [], x :: r \to (f\,x) :: r)$$

We must define the specific A and B that this map function will use at the time of definition, even though we can run the function with any different map f.

The solution is polymorphism.

Polymorphic Lambda Calculus (AKA System F)

To add polymorphism, we need to extend our type representation to be able to represent type 'variables, or types that have not yet been filled in. At this stage, we only want to think about polymorphism that accepts all types, i.e. universal quantification over types. This is because universal quantification is the one that solves our polymorphic map issue as above; we don't care about the type of the variables in our list as long as our function to apply to all of the elements matches the list elements then the map function will work. Later we will see the concept of existential quantification, but it serves a different purpose.

$$\text{Types } A ::= \alpha \mid A \to B \mid \forall \alpha.A$$

You might notice that we have removed both the unit and empty types. We will show later that we can represent data using polymorphics without base data types like unit.

We also need to extend our terms definition since we need to be able to now also abstract over types. We do this by a sort of lambda abstraction, just like we do in regular basic lambda calculus over variables. These lambda abstractions, written with a big lambda Λ, take a type rather than a value to substitute in.

$$\text{Terms } e ::= x \mid \lambda x : A.e \mid ee' \mid \Lambda \alpha.e \mid eA$$

This gives us some power to represent the map function that we wanted:

$$\begin{aligned} &map : \forall \alpha.\forall \beta.(\alpha \to \beta) \to \text{list } \alpha \to \text{list } \beta \\ &map = \Lambda \alpha.\Lambda \beta.\lambda f : \alpha \to \beta.\lambda xs : \text{List } \alpha. \\ &\quad \text{fold}(xs, [] \to [], x :: r \to (fx) :: r) \end{aligned}$$

Well-formedness of types for these expressions is more tricky since we need to keep track of what type variables are currently being abstracted over - a type is not valid if it contains α while α is not currently abstracted over.

We introduce a type context Θ that holds all of the currently abstracted over variables. From this set we can deduce if a type is well-formed. We can think of this adding an extra step: it is no longer good enough to type an expression $e : T$, we must also ensure that $\Theta \vdash T$ type.

The rules for checking Θ are as follows:

$$\text{Type Contexts } \Theta ::= \cdot \mid \Theta, \alpha$$
$$\text{Term Contexts } \Gamma ::= \cdot \mid \Gamma, x : A$$

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}} \qquad \frac{\Theta \vdash A \text{ type} \qquad \Theta \vdash B \text{ type}}{\Theta \vdash A \to B \text{ type}}$$

$$\frac{\Theta, \alpha \vdash A \text{ type}}{\Theta \vdash \forall \alpha . A \text{ type}} \qquad \frac{x : A \in \Gamma}{\Theta; \Gamma \vdash x : A}$$

$$\frac{\Theta \vdash A \text{ type} \qquad \Theta; \Gamma, x : A \vdash e : B}{\Theta; \Gamma \vdash \lambda x : A . e : A \to B}$$

$$\frac{\Theta; \Gamma \vdash e : A \to B \qquad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e e' : B} \qquad \frac{\Theta, \alpha; \Gamma \vdash e : B}{\Theta; \Gamma \vdash \Lambda \alpha . e : \forall \alpha . B}$$

$$\frac{\Theta; \Gamma \vdash e : \forall \alpha . B \qquad \Theta \vdash A \text{ type}}{\Theta; \Gamma \vdash e A : [A/\alpha] B}$$

Note the presence of substitution in the typing rules! In regular Lambda Calculus we substitute in the operational semantics when λ terms are applied to values, whereas in PLC we substitute in the typing rules when the Λ terms are applied to types.

For these rules, we'd like to ensure that **Weakening**, **Exchange** and **Substitution** apply to the type well-formedness Θ rules:

1. (**Type Weakening**)
   $\Theta, \Theta' \vdash A \text{ type} \implies \Theta, \beta, \Theta' \vdash A \text{ type}$
2. (**Type Exchange**)
   $\Theta, \beta, \gamma, \Theta' \vdash A \text{ type} \implies \Theta, \gamma, \beta, \Theta' \vdash A \text{ type}$
3. (**Type Substitution**)
   $(\Theta \vdash A \text{ type}) \wedge (\Theta, \alpha \vdash B \text{ type}) \implies \Theta \vdash [A/\alpha] B \text{ type}$

These all follow essentially the same format as **Weakening**, **Exchange** and **Substitution** for the Γ well-typedness equivalents, both in the way they are presented above and the inductive proofs.

We can lift these up a level and say that an entire context is well formed if each type in the context is well formed under Θ:

$$\frac{}{\Theta \vdash \cdot \text{ ctx}} \qquad \frac{\Theta \vdash \Gamma \text{ ctx} \qquad \Theta \vdash \tau \text{ type}}{\Theta \vdash \Gamma, \tau \text{ ctx}}$$

A well-formed context has the three key properties:

1. (**Context Weakening**)
   $\Theta, \Theta' \vdash \Gamma \text{ ctx} \implies \Theta, \beta, \Theta' \vdash \Gamma \text{ ctx}$
2. (**Context Exchange**)
   $\Theta, \beta, \gamma, \Theta' \vdash \Gamma \text{ ctx} \implies \Theta, \gamma, \beta, \Theta' \vdash \Gamma \text{ ctx}$
3. (**Context Substitution**)
   $(\Theta \vdash A \text{ type}) \wedge (\Theta, \alpha \vdash \Gamma \text{ ctx}) \implies \Theta \vdash [A/\alpha]\Gamma \text{ ctx}$

To prove these properties is trivial by induction on the size of Γ

We also have a property of **Regularity**:

$$(\Theta \vdash \Gamma \text{ ctx}) \wedge (\Theta; \Gamma \vdash e : A) \implies \Theta \vdash A \text{ type}$$

In English, this just says if typechecking succeeds, and the context is well-formed, then we found a well-formed type.

We also still want to prove **Weakening**, **Exchange** and **Substitution** for terms, not just types and contexts. This gives us 6 more iterations, since we can modify Θ or Γ within the Antecedent:

1. (**Type Weakening of Terms**)
   $(\Theta, \Theta' \vdash \Gamma \text{ ctx}) \wedge (\Theta, \Theta'; \Gamma \vdash e : A) \implies \Theta, \alpha, \Theta'; \Gamma \vdash e : A$
2. (**Type Exchange of Terms**)
   $(\Theta, \alpha, \beta, \Theta' \vdash \Gamma \text{ ctx}) \wedge (\Theta, \alpha, \beta, \Theta'; \Gamma \vdash e : A) \implies \Theta, \beta, \alpha, \Theta'; \Gamma \vdash e : A$
3. (**Type Substitution of Terms**)
   $(\Theta, \alpha \vdash \Gamma \text{ ctx}) \wedge (\Theta \vdash A \text{ type}) \wedge (\Theta, \alpha; \Gamma \vdash e : B) \implies \Theta; [A/\alpha]\Gamma \vdash [A/\alpha]e : [A/\alpha]B$

4. (**Weakening of Terms**)
   $(\Theta \vdash \Gamma, \Gamma' \text{ ctx}) \wedge (\Theta \vdash B \text{ type}) \wedge (\Theta; \Gamma, \Gamma' \vdash e : A) \implies \Theta; \Gamma, y : B, \Gamma' \vdash e : A$
5. (**Exchange of Terms**)
   $(\Theta \vdash \Gamma, y : B, z : C, \Gamma' \text{ ctx}) \wedge (\Theta; \Gamma, y : B, z : C, \Gamma' \vdash e : A) \implies \Theta; \Gamma, z : C, y : B, \Gamma' \vdash e : A$
6. (**Substitution of Terms**)
   $(\Theta \vdash \Gamma, x : A \text{ ctx}) \wedge (\Theta; \Gamma \vdash e : A) \wedge (\Theta; \Gamma, x : A \vdash e' : B) \implies \Theta; \Gamma \vdash [e/x]e' : B$

This brings us to a total of 4 variants of **Weakening**, **Exchange** and **Substitution**, plus a single **Regularity** rule. To prove some of the rules we have to assume well-formedness conditions, but the proofs are all otherwise similar to STLC.

And then the operational semantics:

$$\text{Values } v ::= \lambda x : A.e \mid \Lambda\alpha.e$$

$$\frac{e_0 \rightsquigarrow e_0'}{e_0 e_1 \rightsquigarrow e_0' e_1}\ \text{CongFun} \qquad \frac{e_1 \rightsquigarrow e_1'}{v_0 e_1 \rightsquigarrow v_0 e_1'}\ \text{CongFunArg}$$

$$\frac{}{(\lambda x : X.e)v \rightsquigarrow [v/x]e}\ \text{FunEval}$$

$$\frac{e \rightsquigarrow e'}{eA \rightsquigarrow e'A}\ \text{CongForAll} \qquad \frac{}{(\Lambda\alpha.e)A \rightsquigarrow [A/\alpha]e}\ \text{ForAllEval}$$

The first three sequents are ripped straight from STLC, and the last two are added just to deal with our new type lambdas. I find it interesting that we only need these five operational semantics rules, rather than the fifteen in STLC.

And as per usual we want the two type safety rules:

1. (**Progress**)
   $\cdot; \cdot \vdash e : \tau \implies (e \in v) \vee (\exists e'.e \rightsquigarrow e')$
2. (**Preservation**)
   $(\cdot; \cdot \vdash e : \tau) \wedge (e \rightsquigarrow e') \implies \cdot; \cdot \vdash e' : \tau$

But where did the data go?

Data and System F

Discovered in 1941 by Alonzo Church, the idea follows the following observations:

1. Data is used to make choices
2. Based on the choice, you perform different results
3. So we can encode data as functions which take different possible results, and return the right one

For an example, take a boolean. The only reason that a boolean is useful is if we can run an if statement on it. There is no use being able to store a boolean if we can't print it out, run a branching statement or modify other data based on the boolean's value.

We can imagine then that a boolean and a function which either executes the first branch or the second branch are equivalent. That is, if the only way that we can use a boolean is to either do a first thing or a second thing, then we may as well just encode the boolean as doing either the first or the second thing.

This gives us a type for a boolean of $\forall\alpha.\alpha \to \alpha \to \alpha$. Or in English, if you give me two α values, the boolean will either give the first value or the second value.

We can then encode true as $\Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.x$, getting the first value, and false as $\Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.y$, getting the second value.

We also get the if statement for free:

$$\text{if } e \text{ then } e' \text{ else } e'' : X \implies eXe'e''$$

Another way to see this is from a functional point of view. In functional languages we define new types as tagged unions. These tagged unions can then be acted on using a match statement. Since the only way that a tagged union can be interrogated is this match statement, we can combine the two and encode the tagged union *as* the match statement, that takes a set of values and gives back the value that the tagged union contains.

```
type bool = True \; | \; False;

let encode (b: bool) (t: 'a) (f: 'a) =
    match b with
    | True -> t
    | False -> f;;

let true: 'a -> 'a -> 'a = encode True;;
let false: 'a -> 'a -> 'a = encode False;;
```

We see that *true* and *false* both have type $\forall\alpha.\alpha \to \alpha \to \alpha$ as expected.

When we have a tagged union that contains some data, we must make sure that the encoded data is managed. For example, imagine a tagged union that holds either X or Y

```
type union = L of x \; | \; R of y;
```

Then we need to manage where these values go when passed to the match function. The answer is to enforce that the values passed to the match function are functions that take X or Y:

```
let encode (u: union) (l: x -> 'a) (r: y -> 'a) =
    match u with
    | L(v) -> l v
    | R(v) -> r v;;

let left: (x -> 'a) -> (y -> 'a) -> 'a = encode (L x1);;
let right: (x -> 'a) -> (y -> 'a) -> 'a = encode (R y1);;
```

And from these types we can read off our encodings of unions within PLC:

$$\begin{aligned} X + Y &\implies \forall\alpha.(X \to \alpha) \to (Y \to \alpha) \to \alpha \\ Le &\implies \Lambda\alpha.\lambda f : X \to \alpha.\lambda g : Y \to \alpha.fe \\ Re &\implies \Lambda\alpha.\lambda f : X \to \alpha.\lambda g : Y \to \alpha.ge \end{aligned}$$

The Case statement is encoded with a bit more fluff but follows the same idea:

$$\text{case}(e, Lx \to e1, Ry \to e2) : Z \implies eZ(\lambda x : X \to Z.e1)(\lambda y : Y \to Z.e2)$$

We can encode product types in the same way, except far easier since we don't actually have a choice in the types stored:

```
type prod = Some of (x * y);
```

Which gives us the following:

$$\begin{aligned} X \times Y &\implies \forall\alpha.(X \to Y \to \alpha) \to \alpha \\ \langle e, e' \rangle &\implies \Lambda\alpha.\lambda k : X \to Y \to \alpha.kee' \\ \text{fst } e &\implies eX(\lambda x : X.\lambda y : Y.x) \\ \text{snd } e &\implies eY(\lambda x : X.\lambda y : Y.y) \end{aligned}$$

Where this becomes more complicated is when the data that we add to the tagged union is recursive. For example, imagine a representation of integers. Then we might have the following type:

```
type int = Zero \; | \; Succ of int;
```

Then the match statement for the Succ branch must take the variable stored, meaning the Succ value must be a function that takes an int:

```
let encode (b: int) (z: 'a) (s: int -> 'a) =
    match b with
    | Zero -> z
    | Succ(i) -> s i;;
```

This doesn't really help us to encode int since we still need to know the encoding of an int within our definition for the type of s.

The trick is to leverage that recursion into the function as follows:

```
let encode (b: int) (z: 'a) (s: 'a -> 'a) =
    match b with
    | Zero -> z
    | Succ(i) -> s (encode i z s);;
```

This gives us the following encodings:

$$\begin{aligned} N &\implies \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha \\ z &\implies \Lambda\alpha.\lambda z : \alpha.\lambda s : \alpha \to \alpha.z \\ s(e) &\implies \Lambda\alpha.\lambda z : \alpha.\lambda s : \alpha \to \alpha.s(e\alpha zs) \\ \text{iter}(e, z \to e_z, s(x) \to e_s) : X &\implies eXe_z(\lambda x : X.e_s) \end{aligned}$$

And we can also do lists recursively too:

```
type list 'x = Nil \; | \; Cons of ('x * list);

let encode (l: list 'x) (n: 'a) (c: 'x -> 'a -> 'a) =
    match l with
    | Nil -> n
    | Cons(v, l) -> c v (encode l n c);;
```

Which gives the following:

$$\begin{aligned} listX &\implies \forall\alpha.\alpha \to (X \to \alpha \to \alpha) \to \alpha \\ [] &\implies \Lambda\alpha.\lambda n : \alpha.\lambda c : X \to \alpha \to \alpha.n \\ e :: e' &\implies \Lambda\alpha.\lambda n : \alpha.\lambda c : X \to \alpha \to \alpha.ce(e'\alpha nc) \end{aligned}$$

And we can define the three main list functions on this representation

$$\begin{aligned} fold(e, [] \to e_n, x :: r \to e_c) : Z = eZe_n(\lambda x : X.\lambda r : Z.e_c) \\ map(e, f) : listZ = fold(e, [] \to [], x :: r \to (fx) :: r) : listZ \\ filter(e, f) : listZ = fold(e, [] \to [], x :: r \to if(fx, x :: r, r)) : listZ \end{aligned}$$

Existential types

Within modern programming languages, we use polymorphism not only to allow methods to run on any number of types, as we saw in the list example above, but we also use polymorphism to define interfaces behind which we hide implementation. For example, in Java we might define an interface as follows:

```
interface Bool {
    public Bool getTrue();
    public Bool getFalse();
  public <T> T eval(T ifTrue, T ifFalse);
}
```

We can then require our argument to be any implementation of this interface, but we do not care about the actual way that the boolean is stored. We might have some obvious implementation backed by a boolean:

```
class BoolBool implements Bool {
    boolean value;

    private BoolBool(boolean value) {
        this.value = value;
    }

    public Bool getTrue() {
        return new BoolBool(true);
    }

    public Bool getFalse() {
        return new BoolBool(false);
    }

    public <T> T eval(T ifTrue, T ifFalse) {
        return value ? ifTrue : ifFalse;
    }
}
```

Or an integer:

```
class IntBool implements Bool {
    private int value;

    private IntBool(int value) {
        this.value = value;
    }

    public Bool getTrue() {
        return new IntBool(1);
    }

    public Bool getFalse() {
        return new IntBool(0);
    }

    public <T> T eval(T ifTrue, T ifFalse) {
        return value == 1 ? ifTrue : ifFalse;
    }
}
```

Or any number of other implementations. The only important addition to being able to query the value is to also be able to construct the value.

The way that we can capture this concept within our type system is existential types, written $\cdot;\cdot \vdash e : \exists\alpha.A$ to mean 'Expression e requires some implementation α of interface A'

We add the grammar for these implementations as follows:

$$\begin{aligned}
&\text{Types } A ::= \ldots \mid \exists\alpha.A \\
&\text{Terms } e ::= \ldots \mid \text{pack}_{\alpha.B}(A, e) \mid \text{let pack}(\alpha, x) = e \text{ in } e' \\
&\text{Values } v ::= \ldots \mid \text{pack}_{\alpha.B}(A, v)
\end{aligned}$$

$$\frac{\Theta, \alpha \vdash B \text{ type} \qquad \Theta \vdash A \text{ type} \qquad \Theta; \Gamma \vdash e : [A/\alpha]B}{\Theta; \Gamma \vdash \text{pack}_{\alpha.B}(A, e) : \exists\alpha.B}\ \exists I$$

$$\frac{\Theta; \Gamma \vdash e : \exists\alpha.A \qquad \Theta, \alpha; \Gamma, x : A \vdash e' : C \qquad \Theta \vdash C \text{ type}}{\Theta; \Gamma \vdash \text{let pack}(\alpha, x) = e \text{ in } e' : C}\ \exists E$$

The concept here is that:

- We can 'pack' both a type A and an expression e together to form an expression of type $\exists\alpha.B$, where α can appear within B, as long as e has type $[A/\alpha]B$
- We can use this packed value in an expression e' if it has both a free type α and a free variable x of type B (note that in the rules above we result in a pack expression with type $\exists\alpha.B$ but then use a pack expression of type $\exists\alpha.A$).

When we use a packed value in an expression, we want to substitute both the type and the value in at once. This gives us the operational semantics:

$$\frac{e \rightsquigarrow e'}{\text{pack}_{\alpha.B}(A, e) \rightsquigarrow \text{pack}_{\alpha.B}(A, e')}$$

$$\frac{e \rightsquigarrow e'}{\text{let pack}(\alpha, x) = e \text{ in } t \rightsquigarrow \text{let pack}(\alpha, x) = e' \text{ in } t}$$

$$\frac{}{\text{let pack}(\alpha, x) = \text{pack}_{\alpha.B}(A, v) \text{ in } e \rightsquigarrow [A/\alpha, v/x]e}$$

To see how this works in practice, consider our boolean example. A boolean interface needs to provide a true instance, a false instance, and an if statement.

If α encodes a boolean value, then an if statement would have the signature

$$\text{if}(a, b, c) : \alpha \to \delta \to \delta \to \delta$$

Therefore the boolean existential type might look something like:

$$\exists\alpha.(\alpha \times \alpha \times (\forall\delta.\alpha \to \delta \to \delta \to \delta))$$

Remember that we encode a tuple in System F as $\forall\beta.(X \to Y \to \beta) \to \beta$, so we can expand the tuple:

$$\exists\alpha.\forall\beta.(\alpha \to \alpha \to (\forall\delta.\alpha \to \delta \to \delta \to \delta) \to \beta) \to \beta$$

This gives us the type that we want for the boolean pack, which we can validate by working from the pack.

To build a specific pack implementation we need a type and a value. The type that we had for our booleans before was $\forall\alpha.\alpha \to \alpha \to \alpha$. We also have true as $\Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.x$ and false as $\Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.y$. This means that our implementation **value** would be:

$$\begin{gathered}(\Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.x,\\ \Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.y,\\ \Lambda A.\lambda b : \forall\alpha.\alpha \to \alpha \to \alpha.\lambda x : A.\lambda y : A.bxy)\end{gathered}$$

AKA a true value, a false value, and an if statement. Using the tuple encoding we get a System F value:

$$\begin{aligned}&\Lambda B.\lambda c : (\forall\alpha.\alpha \to \alpha \to \alpha) \to\\ &\quad(\forall\alpha.\alpha \to \alpha \to \alpha) \to\\ &\quad(\forall\delta.(\forall\alpha.\alpha \to \alpha \to \alpha) \to \delta \to \delta \to \delta)\\ &\quad\to B.\\ &\qquad c(\Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.x)\\ &\qquad(\Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.y)\\ &\qquad(\Lambda A.\lambda b : \forall\alpha.\alpha \to \alpha \to \alpha.\lambda x : A.\lambda y : A.bxy)\end{aligned}$$

Unpleasant but it's all there. If you want to try to read this and understand where it all comes in, imagine that c is a selector function that returns either its first argument (a value representing **True**), its second argument (a value representing **False**) or its third argument (a function representing **if**) If we type this value we get the following:

$$\begin{gathered}\forall\beta.((\forall\alpha.\alpha \to \alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha \to \alpha) \to\\ (\forall\delta.(\forall\alpha.\alpha \to \alpha \to \alpha) \to \delta \to \delta \to \delta) \to \beta) \to \beta\end{gathered}$$

Which has a lot of $\forall\alpha.\alpha \to \alpha \to \alpha$ in it. Luckily this is also the type of our boolean implementation, and the idea of the pack expression is that we hide our specific implementation type and value. So we observe that this type is the same as:

$$\begin{gathered}[\forall\alpha.\alpha \to \alpha \to \alpha/\gamma](\forall\beta.(\gamma \to \gamma \to\\ (\forall\delta.\gamma \to \delta \to \delta \to \delta) \to \beta) \to \beta)\end{gathered}$$

And voilà we have arrived at our boolean pack type again.

It is worth noting that we haven't actually extended the computational power of System F since we can encode existential types using type polymorphism:

$$\begin{aligned}\exists\alpha.B &\implies \forall\beta.(\forall\alpha.B \to \beta) \to \beta\\ \text{pack}_{\alpha.B}(A, e) &\implies \Lambda\beta.\lambda k : \forall\alpha.B \to \beta.kAe\\ \text{let pack}(\alpha, x) = e \text{ in } e' : C &\implies eC(\Lambda\alpha/\lambda X : B.e')\end{aligned}$$

System F Termination

We already know that structural and rule induction were non-starters when it came to proving termination within STLC, and we can assume that the same applies here. Instead, System F can be proven to terminate using similar methods to when we proved termination for STLC.

We define a *Semantic Type* to be a similar concept to the *Halt* sets when we were proving termination for STLC - A *Semantic Type X* is a set of terms such that:

- (**Halting**)

  If some expression is in the semantic type then it halts

  $e \in X \implies \exists v. e \rightsquigarrow^* v$

- (**Closure**)

  If some expression in the semantic type steps to or is stepped to by another expression, that other expression is also in the semantic type

  $e \rightsquigarrow e' \implies (e \in X \iff e' \in X)$

The reason that we can't use the same *Halt* sets as STLC is that not all types are valid, so we can't have all *Halt* sets existing. The solution is to include Θ and only make Semantic Types over well-formed types.

From the two conditions above, just like our definition of Halt, we can see that each Semantic Type X with at least one expression has an associated type T. This can be seen by the type safety of stepping within the second point - if an expression with type T is in the Semantic Type then the normal form of that expression must also be in the Semantic Type, which means that all expressions that reduce to that normal form must be in the semantic type. We can't use this property to assume that all expressions of a type are in a Semantic Type since we are currently proving that System F actually terminates, so we haven't yet proven that all expressions of a type have a normal form.

Recall that we use Θ to represent a type variable context. That is a set of type variables $\alpha, \beta, ...$

We can define θ to be a specific substitution of the generic types to specific semantic types, i.e.

$$\theta ::= \cdot \mid (\theta, X/\alpha)$$

Then we define a function $[\![-]\!]$ such that:

$$[\![-]\!] \in WellFormedType \rightarrow VarInterpretation \rightarrow SemanticType$$

Since θ is a mapping from well formed types to semantics types, we can define the following:

$$[\![\Theta \vdash \alpha \text{ Type}]\!] = \theta(\alpha)$$

We then know that all other valid types follow one of two other forms, either $A \rightarrow B$ or $\forall\alpha.B$

The $A \rightarrow B$ case is very similar to Halt; e is in $[\![\Theta \vdash A \rightarrow B \text{ Type}]\!]\theta$ if:

- e halts
- For all e' in $[\![\Theta \vdash A \text{ Type}]\!]\theta$ we have that (ee') is in $[\![\Theta \vdash B \text{ Type}]\!]\theta$

That is, e preserves halting of expressions when applied to an expression. This too is similar to the interpretation of *Halt*.

The new type form of System F is $\forall\alpha.B$; e is in $[\![\Theta \vdash \forall\alpha.B \text{ Type}]\!]\theta$ if:

- e halts
- For all Types A and Semantic Types X we have that (eA) is in $[\![\Theta, \alpha \vdash B \text{ Type}]\!](\theta, X/\alpha)$

This can be interpreted as e halting no matter what type it is instantiated with. Note that the type A and semantic type X don't have to be linked. The reason that we can do this is that technically type applications are only used for book keeping - they never modify the behaviour of a program. This is similar logic to type erasure in Java in that once we have proven that an expression is well-typed, the types can be stripped away and the program can be run without any of the type information. Therefore we can allow for any type A without checking that the semantic type X corresponds to the same type since the type A that we choose will not modify whether or not the program halts, only the semantic type X could possibly modify the halting behaviour of the program.

We have a few properties that we can prove on these Semantic Types:

- (**Closure**)
  If θ is an interpretation for Θ, then $[\![\Theta \vdash A \text{ type}]\!]\theta$ is a semantic type.
- (**Exchange**)
  $[\![\Theta, \alpha, \beta, \Theta' \vdash A \text{ type}]\!] = [\![\Theta, \beta, \alpha, \Theta' \vdash A \text{ type}]\!]$
- (**Weakening**)
  If $\Theta \vdash A$ type, then $[\![\Theta, \alpha \vdash A \text{ type}]\!](\theta, X/\alpha) = [\![\Theta \vdash A \text{ type}]\!]\theta$
- (**Substitution**)
  If $\Theta \vdash A$ type and $\Theta, \alpha \vdash B$ type then
  $[\![\Theta \vdash [A/\alpha]B \text{ type}]\!]\theta = [\![\Theta, \alpha \vdash B \text{ type}]\!](\theta, [\![\Theta \vdash A \text{ type}]\!]\theta)$

These properties can be proven by induction on the three structures of well formed types.

Which leads us to the fundamental lemma: If we have that

- $\Theta = \alpha_1, ..., \alpha_k$
- $\Gamma = x_1 : A_1, ..., x_n : A_n$
- $\Theta \vdash \Gamma$ ctx
- $\Theta; \Gamma \vdash e : B$
- θ interprets Θ
- For all $x_i : A_i$ in Γ we have some $e_i \in [\![\Theta \vdash A_i \text{ type }]\!]\theta$

Then we can substitute all free type variables with arbitrary types and all free variables with the expressions and gain the properties given by the semantic type set:

$$[C_1/\alpha_1, ..., C_k/\alpha_k][e_1/x_1, ..., e_n/x_n]e \in [\![\Theta \vdash B \text{ type}]\!]\theta$$

And so, since all values within System F halt by definition, and all expressions can be built up by substitution of values into values, all expressions must be in their respective Semantic Type. And since all expressions in a Semantic Type halt, all expressions in System F must halt.

Second Order Intuitionistic Propositional Logic

We have just seen that we can introduce both Universal and Existential quantifiers to our type system. This would imply that we can show that Second Order Intuitionistic propositions are true by finding instances of higher order types.

For example, imagine that we have propositions $P(x)$ and $Q(x)$ with corresponding types $\alpha \vdash T_P$ type and $\alpha \vdash T_Q$ type. Note the free type variables α modelling the parameter x. Then we may want to prove the following second order statement:

$$\exists x.P(x) \wedge \forall x.(P(x) \implies Q(x)) \implies \exists x.Q(x)$$

This has the following type:

$$\exists \alpha.T_P \to (\forall \alpha.T_P \to T_Q) \to \exists \alpha.T_Q$$

We can build the following term in System F with the above type:

$$\begin{array}{l}
\lambda a : \exists \alpha.T_P.(\\
\quad \lambda b.\forall \alpha.T_P \to T_Q.(\\
\quad\quad \text{let pack}(\alpha, x) = a \text{ in pack}_{\alpha.T_Q}(\alpha, b\alpha x) \\
\quad) \\
)
\end{array}$$

We will discuss what this means further when we introduce Classical Logic.

State and Stores

Sometimes it'd be useful to actually harvest output from our code. The most intuitive way to do this is with a store and state, as if modelling a file system or memory block.

Borrowing from IB Semantics we can imagine a language with references to state locations that we can update. We might want to create a fresh reference (akin to malloc) using an instruction like **ref e**, read a reference with **!e** and update a reference with **e := e'**.

Small Aside:

Something not mentioned on the slides is that we will also need a sequence operator, as we will want to run instructions in sequence while we modify the state. However our operational semantics state that function application only occurs once both the function and the argument are values, so we can encode sequence in the following way:

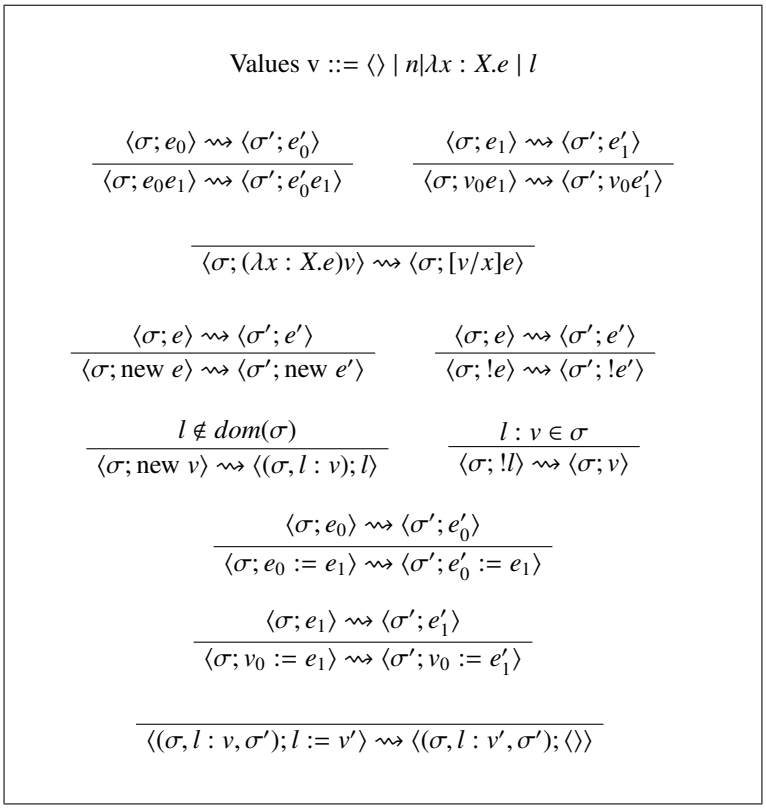
$$e_1; e_2 \implies (\lambda x.)e_2)e_1$$

Here e_1 must be fully evaluated before being substituted, and then e_2 will be fully evaluated and returned.

We first define our language grammar:

$$\begin{aligned}
\text{Types X} &::= 1 \mid N \mid X \to Y \mid \text{ref } X \\
\text{Terms e} &::= \langle\rangle \mid n \mid \lambda x : X.e \mid ee' \mid \text{new } e \mid !e \mid e := e' \mid l \\
\text{Stores } \sigma &::= \cdot \mid \sigma, l : v \\
\text{Contexts } \Gamma &::= \cdot \mid \Gamma, x : X
\end{aligned}$$

We then get the following operational semantics. Note that they are mostly just copied and pasted from STLC, but we keep a state σ along for the ride. The only exceptions are the instructions which modify the state:



When it comes to typing this language, we need to make sure that we can type the store. We keep this information in an extra variable Σ that we pass around our type derivations.

The first five typing rules are taken from STLC and Σ is just added to the things we keep track of. The last four deal with references, but only the last one actually uses the contents of Σ.

Store Typings $\Sigma ::= \cdot \mid \Sigma, l : X$

$$\frac{x : X \in \Gamma}{\Sigma; \Gamma \vdash x : X}\ \text{HYP} \qquad \frac{}{\Sigma; \Gamma \vdash \langle\rangle : 1}\ 1\text{I} \qquad \frac{}{\Sigma; \Gamma \vdash n : \mathbb{N}}\ \mathbb{N}\text{I}$$

$$\frac{\Sigma; \Gamma, x : X \vdash e : Y}{\Sigma; \Gamma \vdash \lambda x : X.e : X \to Y}\ \to\text{I}$$

$$\frac{\Sigma; \Gamma \vdash e : X \to Y \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash ee' : Y}\ \to\text{E}$$

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \text{new } e : \text{ref } X}\ \text{RefL} \qquad \frac{\Sigma; \Gamma \vdash e : \text{ref } X}{\Sigma; \Gamma \vdash !e : X}\ \text{RefGet}$$

$$\frac{\Sigma; \Gamma \vdash e : \text{ref } X \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e := e' : 1}\ \text{RefSet}$$

$$\frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \text{ref } X}\ \text{RefBar}$$

We now would want to talk about this language having type safety, but to do this we need to be able to describe that the store at any point of execution is well typed. We're not able to show that an expression is type safe if it is possible for the expression to load something that isn't a valid type from the store.

We define that a store is well typed recursively:

$$\frac{}{\Sigma \vdash \cdot : \cdot}\ \text{StoreNil} \qquad \frac{\Sigma \vdash \sigma' : \Sigma' \qquad \Sigma; \cdot \vdash v : X}{\Sigma \vdash (\sigma', l : v) : (\Sigma', l : X)}\ \text{StoreCons}$$

$$\frac{\Sigma \vdash \sigma : \Sigma \qquad \Sigma; \cdot \vdash e : X}{\langle \sigma; e \rangle : \langle \Sigma; X \rangle}\ \text{ConfigOK}$$

We can interpret this in the following way:

- (**Store Nil**)

  An empty store is well typed

- (**Store Cons**)

  A store can be grown by a single location l and variable v with type X if it can be shown that v has type X

- (**Config OK**) A store is valid for a store configuration if every location in the store matches with the store configuration

Unfortunately if we were to now try to prove type safety naively we'd find ourselves stuck when we try to do structural induction on *new*. This is because the store grows, and so the typing for the store is no longer valid. This leads us to the idea of **Store Monotonicity**, or that The Store Only Grows.

We define $\Sigma \leq \Sigma'$ to mean there is some other Σ'' such that $\Sigma' = \Sigma, \Sigma''$. Note that this Σ'' may just be $\cdot$, hence the less than *or equal to* relation.

We can show that if the store grows then the typing of a store and an expression are still valid. That is,

$$\Sigma; \Gamma \vdash e : X \implies \Sigma'; \Gamma \vdash e : X$$
$$\Sigma \vdash \sigma_0 : \Sigma_0 \implies \Sigma' \vdash \sigma_0 : \Sigma_0$$

With these rules we can define progress and preservation:

1. (**Progress**)

   Well-typed programs and stores are not stuck: they can always take a step of progress (or are done).

   $\langle \sigma; e \rangle : \langle \Sigma; X \rangle \implies (e \in v) \vee (\exists\, \sigma', e'. \langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle)$

2. (**Preservation**)

   If a well-typed program and store take a step, the program will stay well-typed and the store will only grow.

   $(\langle \sigma; e \rangle : \langle \Sigma; X \rangle) \wedge (\langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle) \implies \exists\, \Sigma' \geq \Sigma. \langle \sigma'; e' \rangle : \langle \Sigma'; X' \rangle$

Accidental Looping

Using our store, we can store references to functions. This is an issue, because functions can themselves read those references when they are run. This lets us create recursion by creating a function that:

1. Creates a function that loads a reference and calls the reference
2. Stores the function in a memory location
3. Runs the function with the memory location

This can be illustrated with Landin's Knot:

```
let knot : ((int -> int) -> int -> int) -> int -> int =
    fun f ->
        let r = ref (fun n -> 0) in
        let recur = fun n -> !r n in
        let () = r := fun n -> f recur n in recur
```

This means that we can build programs that loop forever and so our type system does not result in **Termination**. This is a shame and it would be nice if we could separate our effectful code from our pure code in a way that prevents infinite loops...

Monads

To stop infinite loops, we need to prevent the formation of structures like Landin's Knot. Landin's Knot was able to be formed because we could both dereference a value and apply that value as a function to an argument within the same program.

Monads introduce two separate types of instructions and two separate environments in which these instructions execute, called Pure and Impure.

Pure terms operate like traditional lambda calculus - we have values, functions and function compositions. The only thing that we add is that impure terms can be passed around like values. We can't interrogate or execute these impure terms within a pure environment, instead they are opaque and remain unevaluated.

Impure terms *have effects*. They may read from or write to a store, perform IO, read and write from a terminal. We may also execute Pure terms.

This means that Pure terms may not execute their containing Impure terms, but Impure terms may execute their Pure terms.

One interesting result of this is that a program that has a Pure term as its top level structure must not have any effects, since the Pure term cannot initiate execution of an Impure term.

An initial grammar for a monadic language might be as follows:

$$
\begin{aligned}
\text{Pure Terms } e &::= \langle\rangle \mid n \mid \lambda x : X.e \mid ee' \mid l \mid t \\
\text{Impure Terms } t &::= \text{new } e \mid !e \mid e := e' \mid \text{let } x = e; t \mid \text{return } e \\
\text{Impure Terms } e &::= \langle\rangle \mid n \mid \lambda x : X.e \mid l \mid t \\
\text{Values } v &::= \langle\rangle \mid n \mid \lambda x : X.e \mid l \mid \{t\} \\
\text{Stores } \sigma &::= \cdot \mid \sigma, l : v \\
\text{Contexts } \Gamma &::= \cdot \mid \Gamma, x : X \\
\text{Store Typings } \Sigma &::= \cdot \mid \Sigma, l : X
\end{aligned}
$$

The differences to notice are:

1. We have two new types, ref X and TX. The first describes the type of a store location, and the second describes the type as the result of an Impure expression, or Monad. For example, 3 has the type $\mathbb{N}$ while $\{return 3\}$ has the type $T\mathbb{N}$
2. We have two new Pure terms, l and $\{t\}$. The first represents a location which should be generated by new e under normal circumstances, but should be able to be typechecked so that we can show **Type Safety** later on. The second is how we embed Impure expressions within pure ones, by wrapping the impure statement

in curly braces. This links into one of our new value types, as Impure expressions wrapped in curly braces are values and so when encountered in a Pure context are not reduced at all.

3. We have a new category called Impure types which consist of everything that can manipulate the store, as well as the **let** and **return** expressions.

   We can see the let expression as temporarily exposing the content of a monad. That is, the let expression is a bit like a function of the type $T\alpha \to (\alpha \to T\beta) \to T\beta$, except its second argument isn't given as a function but instead an expression with a free variable.

   The **return** expression is the counterpart to let and allows us to create a new monad, that exposes whatever is returned to the let expression.

   One way of viewing these two commands (and monads in general) is as a many-layered object like a burrito or an onion. When an object is wrapped in curly brackets we wrap the monad with an extra layer, obscuring the mess inside. This wrapped object is itself inert and cannot be queried or evaluated. However a **let** expression allows us to peel back one layer of the monad and access the value inside, so long as the expression that uses the value wraps the result back up at the end.

Since it can be quite hard to read this grammar, here are some examples of programs:

$$(\lambda x : T\,1.x)\{\text{let } x = 1; x := 2\}$$

This first equation reduces to $\{\text{let } x = 1; x := 2\}$ and then halts. This is because an Impure term wrapped in curly braces is a value, said to be a **Suspended Computation**, and is not itself evaluated. This means that any expression for which the 'top level' component is Pure must have no side effects.

$$\text{let } y = \lambda x : 1.x; \text{let } z = \text{new } y$$

This second expression displays that we can still store functions in the store. We could even load this store and run the function. The difference is that this function itself cannot load from the store, since functions must be Pure. This is how we avoid the formation of Landin's Knot.

$$\text{let } x = (\text{if True then } \{\text{return } 1\} \text{ else } \{\text{return } 2\}); \mathit{return}!x$$

This program illustrates the nesting of Pure and Impure programs. The outer layer is Impure with a **let** statement, then within this is a Pure **lambda** expression, which contains two impure **return** statements.

More formally, here are the typing rules. Note the two different forms of typing derivations, using $\Sigma; \Gamma \vdash e : \tau$ for Pure expressions and $\Sigma; \Gamma \vdash e \div \tau$ for Impure expressions.

Also notice that HYP, 1I, $\mathbb{N}$I, $\to$I and $\to$E are exactly the same as STLC just with a Σ passed through.

$$\frac{x : X \in \Gamma}{\Sigma; \Gamma \vdash x : X}\ \text{HYP} \qquad \frac{}{\Sigma; \Gamma \vdash \langle\rangle : 1}\ 1\text{I} \qquad \frac{}{\Sigma; \Gamma \vdash n : \mathbb{N}}\ \mathbb{N}\text{I}$$

$$\frac{\Sigma; \Gamma, x : X \vdash e : Y}{\Sigma; \Gamma \vdash \lambda x : X.e : X \to Y}\ \to\text{I}$$

$$\frac{\Sigma; \Gamma \vdash e : X \to Y \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash ee' : Y}\ \to\text{E}$$

$$\frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \text{ref } X}\ \text{RefBar} \qquad \frac{\Sigma; \Gamma \vdash t \div X}{\Sigma; \Gamma \vdash \{t\} : TX}\ \text{TI}$$

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \text{new } e \div \text{ref } X}\ \text{RefL} \qquad \frac{\Sigma; \Gamma \vdash e : \text{ref } X}{\Sigma; \Gamma \vdash !e \div X}\ \text{RefGet}$$

$$\frac{\Sigma; \Gamma \vdash e : \text{ref } X \qquad \Sigma; \Gamma \vdash e' : X}{\Sigma; \Gamma \vdash e := e' \div 1}\ \text{RefSet}$$

$$\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \text{return } e \div X}\ \text{TRet}$$

$$\frac{\Sigma; \Gamma \vdash e : TX \qquad \Sigma; \Gamma, x : X \vdash t \div Z}{\Sigma; \Gamma \vdash \text{let } x = e; t \div Z}\ \text{TLet}$$

We then define our operational semantics on two levels. We have our very basic function application semantics for Pure expressions, and then our operational semantics for Impure expressions which caries with it a store. Note that we could extend our monad system however we wanted to include any other kinds of effects, and these would all be tracked inside this impure semantics.

$$\frac{e_0 \rightsquigarrow e_0'}{e_0 e_1 \rightsquigarrow e_0' e_1} \qquad \frac{e_1 \rightsquigarrow e_1'}{v_0 e_1 \rightsquigarrow v_0 e_1'}$$

$$\frac{}{(\lambda x : X.e)v \rightsquigarrow [v/x]e}$$

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \text{new } e \rangle \rightsquigarrow \langle \sigma; \text{new } e' \rangle} \qquad \frac{l \notin dom(\sigma)}{\langle \sigma; \text{new } e \rangle \rightsquigarrow \langle (\sigma, l : v); \text{return } l \rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \sigma; !e \rangle \rightsquigarrow \langle \sigma; !e' \rangle} \qquad \frac{l : v \in \sigma}{\langle \sigma; !l \rangle \rightsquigarrow \langle (\sigma, l : v); \text{return } v \rangle}$$

$$\frac{e_0 \rightsquigarrow e_0'}{\langle \sigma; e_0 := e_1 \rangle \rightsquigarrow \langle \sigma; e_0' := e_1 \rangle} \qquad \frac{e_1 \rightsquigarrow e_1'}{\langle \sigma; v_0 := e_1 \rangle \rightsquigarrow \langle \sigma; v_0 := e_1' \rangle}$$

$$\frac{}{\langle (\sigma, l : v, \sigma'); l := v' \rangle \rightsquigarrow \langle (\sigma, l : v', \sigma'); \text{return } \langle\rangle \rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \text{return } e \rangle \rightsquigarrow \langle \sigma; \text{return } e' \rangle}$$

$$\frac{e \rightsquigarrow e'}{\langle \sigma; \text{let } x = e; t \rangle \rightsquigarrow \langle \sigma; \text{let } x = e'; t \rangle}$$

$$\frac{}{\langle \sigma; \text{let } x = \{\text{return } v\}; t_1 \rangle \rightsquigarrow \langle \sigma; [v/x]t_1 \rangle}$$

$$\frac{\langle \sigma; t_0 \rangle \rightsquigarrow \langle \sigma'; t_0' \rangle}{\langle \sigma; \text{let } x = \{t_0\}; t_1 \rangle \rightsquigarrow \langle \sigma'; \text{let } x = \{t_0'\}; t_1 \rangle}$$

Things that are worth noting here:

- Since all effectful code is strictly linear, and the only way to deviate from the chosen path of execution is function application which is done on the Pure level, we have preserved our halting properties from before.

- Also, since all effectful code is strictly linear, the only rule that uses progress of Impure code is the final **let** rule. This rule propagates the execution step into the argument to be assigned to the variable until the argument is a value. Every other Impure rule either steps a containing Pure argument (and so doesn't change the store), or does some work on itself, possibly changing the store. This demonstrates the seperation of Pure and Impure code in that any containing Pure code is not handed the store and cannot change the store, so must be independent of the result of any Impure code.
- As seen in the lectures, we can see that the Operational Semantics for Pure statements are the first three rules and no more. If we add other effects such as **print** or **read** then we may need to change the store that is threaded around the Impure rules, but the Pure part of this system always stays the same.

We can prove **Weakening**, **Exchange** and **Substitution** for the set Γ for both Pure and Impure types. Since the proof of each property on Pure expressions would require the property to be true on Impure types and the other way around too, prove each property on both Pure and Impure at the same time (mutually inductively) so as to be able to use structural induction using every rule for every possible expression form. This way you can use the inductive hypothesis on both Pure and Impure expressions. **Progress** and **Preservation** can be proven mutually inductively in a similar way, giving us **Type Safety**.

And as we discussed earlier, we can see that Pure expressions cannot depend on the results of Impure expressions, so Pure expressions must still have the property of **Termination** that we showed for STLC.

Extra Information:

Often, this **let** expression as described above is instead called **bind**, and has exactly the signature that we described before:

$$bind : m\alpha \to (\alpha \to m\beta) \to m\beta$$

We also have the **return** function, which has the type

$$return : \alpha \to m\alpha$$

However there is a more simple function, called **join**, which simply takes to monadic 'black boxes' and combines them together:

$$join : m(m\alpha) \to m\alpha$$

We can represent this join function in our language using two let expressions:

$$\text{let } x = v; \text{let } y = x; \text{return } y$$

Classical Logic

We saw with the Curry Howard correspondence that Simply Typed lambda calculus can be used to create instances of types that prove corresponding First Order Intuitionistic Propositional statements. We then saw that we could further introduce Universal and Existential quantifiers through System F to gain more proof power, allowing us to create instances of types that prove corresponding *Second Order* Intuitionistic Propositional statements.

However we are still one Double Negation Elimination axiom away from being able to prove any true statement within Classical logic.

The issue is that within Intuitionistic Logic we have a slightly different notion of Falsehood. We write $\neg P$ to mean that $P \implies \bot$, or that $T_P \to \bot$. In other words, if we are able to give an instance of type T_P, then we are also able to give an instance of type $\bot$, however since we cannot ever have a type of $\bot$ we must not be able to give an instance of T_P.

This means that, within Intuitionistic Logic, we have ideas of *Proven* and *Unprovable*, rather than Classical Logic's *True* and *False*.

We could therefore begin to build up to Classical Logic by treating Refutations, or Proofs of Unprovability, as a first-class notion of Falsehood. This moves away from STLC where the only first-class object is a type and instances of a type represent proofs of a true proposition. By having two different first-class objects, *true* and *false*, we find that an instance of a type is now either a proof that a proposition is true or that it is false.

To expand on this, imagine first we want to prove that some proposition P holds. Within STLC, it is sufficient to show that some expression with type T_P exists. However within our new system, we will want to show that some expression with type T_P *true* exists. This allows us to prove that a proposition P is false by showing that some expression in our system has the type T_P $false$.

We now must establish what exactly these expressions with these types are. We know that our types now have an extra tag stating *true* or *false*, but we need to devise a type of expression which has these types.

The trick is in going backwards; we know that we must be able to use our expressions as proofs of the proposition that their type corresponds to, so we can let our expressions be the encoding of proofs of statements within Classical Logic.

That is, if we can establish firstly a system for proving all true or false statements in Classical Logic, and then secondly a way to encode these proofs as expressions, then each expression can have the type that corresponds to the proven true or false proposition.

We know from IB Logic and Proof that we can use sequents to prove propositions within Classical Logic:

$$\begin{aligned}
\text{Propositions } A &::= \top \mid A \wedge B \mid \bot \mid A \vee B \mid \neg A \\
\text{True contexts } \Gamma &::= \cdot \mid \Gamma, A \\
\text{False contexts } \Delta &::= \cdot \mid \Delta, A \\
\text{Typing Judgements} &::= \Gamma; \Delta \vdash A \text{ true} \mid \Gamma; \Delta \vdash A \text{ false} \mid \Gamma; \Delta \vdash \text{contr}
\end{aligned}$$

$$\frac{A \in \Gamma}{\Gamma; \Delta \vdash A \text{ true}} \text{ HypP} \qquad \frac{A \in \Delta}{\Gamma; \Delta \vdash A \text{ false}} \text{ HypR}$$

$$\frac{}{\Gamma; \Delta \vdash \top \text{ true}} \top\text{P} \qquad \frac{}{\Gamma; \Delta \vdash \bot \text{ false}} \bot\text{R}$$

$$\frac{\Gamma; \Delta \vdash A \text{ true} \qquad \Gamma; \Delta \vdash B \text{ true}}{\Gamma; \Delta \vdash A \wedge B \text{ true}} \wedge\text{P}$$

$$\frac{\Gamma; \Delta \vdash A \text{ false} \qquad \Gamma; \Delta \vdash B \text{ false}}{\Gamma; \Delta \vdash A \vee B \text{ false}} \vee\text{R}$$

$$\frac{\Gamma; \Delta \vdash A \text{ true}}{\Gamma; \Delta \vdash A \vee B \text{ true}} \vee\text{P}_1 \qquad \frac{\Gamma; \Delta \vdash B \text{ true}}{\Gamma; \Delta \vdash A \vee B \text{ true}} \vee\text{P}_2$$

$$\frac{\Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash A \wedge B \text{ false}} \wedge\text{R}_1 \qquad \frac{\Gamma; \Delta \vdash B \text{ false}}{\Gamma; \Delta \vdash A \wedge B \text{ false}} \wedge\text{R}_2$$

$$\frac{\Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash \neg A \text{ true}} \neg\text{P} \qquad \frac{\Gamma; \Delta \vdash A \text{ true}}{\Gamma; \Delta \vdash \neg A \text{ false}} \neg\text{R}$$

$$\frac{\Gamma; \Delta, A \vdash \text{contr}}{\Gamma; \Delta \vdash A \text{ true}} \qquad \frac{\Gamma, A; \Delta \vdash \text{contr}}{\Gamma; \Delta \vdash A \text{ false}}$$

$$\frac{\Gamma; \Delta \vdash A \text{ true} \qquad \Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash \text{contr}} \text{ Contr}$$

With these rules we've gained the ability to prove the Double Negation Elimination rule:

$$\dfrac{\dfrac{\dfrac{}{A;\cdot \vdash A \text{ true}}\text{HYP}}{A;\cdot \vdash \neg A \text{ false}}\neg\text{R}}{A;\cdot \vdash \neg\neg A \text{ true}}\neg\text{P} \qquad \dfrac{\dfrac{}{\neg\neg A; A \vdash \neg\neg A \text{ true}} \qquad \dfrac{\dfrac{}{\neg\neg A; A \vdash A \text{ false}}}{\neg\neg A; A \vdash \neg A \text{ true}}}{\neg\neg A; A \vdash \neg\neg A \text{ false}}}{\dfrac{\neg\neg A; A \vdash \text{contr}}{\neg\neg A;\cdot \vdash A \text{ true}}}$$

We can now devise a way to encode these rules as values within our new language. We have a parity between the rules, where every proposition form has exactly two sequents that it corresponds to; one for if the proposition is true and one for if it was false. We must therefore include in our encoding whether an expression is true or false, which we can do by introducing an alternative variation of each value, dubbed a Continuation.

This gives us the following grammar:

$$\begin{aligned}
\text{Values } e &::= \langle\rangle \mid \langle e, e' \rangle \mid Le \mid Re \mid \text{not}(k) \mid \mu u : A.c \\
\text{Continuations } k &::= [] \mid [k, k'] \mid \text{fst } k \mid \text{snd } k \mid \text{not}(e) \mid \mu x : A.c \\
\text{Contradictions } c &::= \langle e |_A k \rangle
\end{aligned}$$

Note the correspondence between Values and Continuations.

We can then type these expressions with the types that we established earlier:

$$\frac{x : A \in \Gamma}{\Gamma; \Delta \vdash x : A \text{ true}} \text{ HypP} \qquad \frac{x : A \in \Delta}{\Gamma; \Delta \vdash x : A \text{ false}} \text{ HypR}$$

$$\frac{}{\Gamma; \Delta \vdash \langle\rangle : \top \text{ true}} \top\text{P} \qquad \frac{}{\Gamma; \Delta \vdash [] : \bot \text{ false}} \bot\text{R}$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true} \qquad \Gamma; \Delta \vdash e' : B \text{ true}}{\Gamma; \Delta \vdash \langle e, e' \rangle A \wedge B \text{ true}} \wedge\text{P}$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false} \qquad \Gamma; \Delta \vdash k' : B \text{ false}}{\Gamma; \Delta \vdash [k, k'] : A \vee B \text{ false}} \vee\text{R}$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true}}{\Gamma; \Delta \vdash \text{L}e : A \vee B \text{ true}} \vee\text{P}_1 \qquad \frac{\Gamma; \Delta \vdash e : B \text{ true}}{\Gamma; \Delta \vdash \text{R}e : A \vee B \text{ true}} \vee\text{P}_2$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \text{fst } k : A \wedge B \text{ false}} \wedge\text{R}_1$$

$$\frac{\Gamma; \Delta \vdash k : B \text{ false}}{\Gamma; \Delta \vdash \text{snd } k : A \wedge B \text{ false}} \wedge\text{R}_2$$

$$\frac{\Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \text{not}(k) : \neg A \text{ true}} \neg\text{P} \qquad \frac{\Gamma; \Delta \vdash e : A \text{ true}}{\Gamma; \Delta \vdash \text{not}(e) : \neg A \text{ false}} \neg\text{R}$$

$$\frac{\Gamma; \Delta, u : A \vdash c \text{ contr}}{\Gamma; \Delta \vdash \mu u : A.c : A \text{ true}} \qquad \frac{\Gamma, x : A; \Delta \vdash c \text{ contr}}{\Gamma; \Delta \vdash \mu x : A.c : A \text{ false}}$$

$$\frac{\Gamma; \Delta \vdash e : A \text{ true} \qquad \Gamma; \Delta \vdash k : A \text{ false}}{\Gamma; \Delta \vdash \langle e |_A k \rangle \text{ contr}} \text{ Contr}$$

Note that the set of things that we have assumed to be true Γ and the set of things we have assumed to be false Δ now become our variable contexts. This gives added meaning to an expression having no free variables - it must correspond to a tautology.

As with the Curry Howard Correspondence, we can imagine that evaluation of an expression within this new language would be akin to normalising the proof for it. Since most proofs within this language will take the form of a contradiction, we can observe a few things about proving with contradictions:

- If a contradiction exists with the type $A \wedge B$, and the evidence that $A \wedge B$ is false is of the form fst k, then there must actually be a contradiction specifically with A.
- If a contradiction exists with the type $\neg A$ and both evidence are of the form not(e) and not(k), then a contradiction must exist with A with the evidence being e and k.
- If a contradiction exists with the type A, and the evidence $\mu u : A.c$ that A is true directly uses a contradiction, then we can simply use the evidence of the contradiction as the contradiction.

To see that last point more clearly, consider the following two proof trees:

$$\dfrac{\dfrac{\dfrac{\Gamma; \Delta, A \vdash B \text{ true} \qquad \Gamma; \Delta, A \vdash B \text{ false}}{\Gamma; \Delta, A \vdash \text{contr}}}{\Gamma; \Delta \vdash A \text{ true}} \qquad \Gamma; \Delta \vdash A \text{ false}}{\Gamma; \Delta \vdash \text{contr}}$$

$$\dfrac{\Gamma; \Delta \vdash B \text{ true} \qquad \Gamma; \Delta \vdash B \text{ false}}{\Gamma; \Delta \vdash \text{contr}}$$

Both show the same thing, which is that $\Gamma; \Delta$ leads to a contradiction. We can transform the first into the second since we can take the leftmost branch of the first tree and substitute our proof that A is false in the rightmost branch.

This gives us the following operational semantics:

$$\begin{aligned}
\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{fst } k \rangle &\rightsquigarrow \langle e_1 \mid_A k \rangle \\
\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{snd } k \rangle &\rightsquigarrow \langle e_2 \mid_B k \rangle \\
\langle Le \mid_{A \vee B} [k_1, k_2] \rangle &\rightsquigarrow \langle e \mid_A k_1 \rangle \\
\langle Re \mid_{A \vee B} [k_1, k_2] \rangle &\rightsquigarrow \langle e \mid_B k_2 \rangle \\
\langle \text{not}(k) \mid_{\neg A} \text{not}(e) \rangle &\rightsquigarrow \langle e \mid_A k \rangle \\
\langle \mu u : A.c \mid_A k \rangle &\rightsquigarrow [k/u]c \\
\langle e \mid_A \mu x : A.c \rangle &\rightsquigarrow [e/x]c
\end{aligned}$$

The Computational Ability of Classical Logic

When inventing our new Classical Logic computation system, we've removed the lambda expressions. Luckily, if you squint hard enough, our new contradiction terms look a bit like lambda expressions. In fact, we have the operational semantics $\langle \mu u : A.c \mid_A k \rangle \rightsquigarrow [k/u]c$, which looks to me like $(\lambda u : A.c)k \rightsquigarrow [k/u]c$.

Progress can be written that if $\cdot;\cdot \vdash c$ contr then $c \rightsquigarrow c'$ (or c final). Unfortunately, if Classical Logic is consistent (which it is), we know that there do not exist any such expressions that are both closed and contradictions. Therefore our language is functionally useless.

We can try to fix this if we want by introducing extra values, called *halt*, *ans* and *done* to be able to form contradictions where there are none and so allow for arbitrary computations. Unfortunately, by doing this we completely break all of the consistency of Classical Logic; if any contradiction can be formed at any time, we can prove that anything holds.

This appears to give us a choice: we can either use our Classical Logic in a consistent state, or in a computationally useful way. We will see that there is a different approach that we can take towards this whole thing that allows us to have both.

Classical Embedding

We introduce the concept of Embedding Classical logic with an observation:

$$\lambda k : ((X \to \bot) \to \bot) \to \bot . \lambda a : X . k(\lambda q : X \to \bot . qa))$$
$$\text{has the type}$$
$$((X \to \bot) \to \bot) \to \bot \to (X \to \bot)$$

If we use our previous observation that we can equate the proposition $\neg A$ to the type $A \to \bot$, we have proven that:

$$\neg\neg\neg X \implies \neg X$$

This is called triple negation, and holds within Intuitionistic logic.

Small Aside

Why is it then that Double Negation Elimination doesn't hold but Triple Negation Elimination does?
The trick is in the Intuitionistic interpretation of False as 'Not Able to be Proven'.
Imagine that you and a group of researchers find some ancient cave writings. These writings might be written by aliens, making the statements written True, or they might be scribbles written by children, making them Nonsensical and Unable to be proven True. The children's scribbles may be completely True statements about the meaning of Life, the Universe and Everything, however unfortunately the children's handwriting is not good enough for us to ever decode these statements.
Now imagine that I told you that the writing is *Not* the children's. This means that the statements contained must be True, and follows the classical idea of Double Negation Elimination.
Imagine instead that I told you that we are *Unable to Prove* that the writing is the Children's. Then we do not know anything more about the validity of the contained statements, and so Double Negation Elimination does not hold.
Now imagine that I told you that we would *Never be Able to Prove* whether or not we are *Unable to Prove* that the writing is the Children's. Then we will also *Never be Able to Prove* that we are ***Able** to Prove* that the writing is the Alien's, since showing that the writing is the Alien's proves that we are unable to show that the writing is the Children's. Therefore we will never know whether the contained statements are True or not, which means that we are functionally in the same place as if we knew that the writing was the Children's. This extra layer of Unprovability allows us to essentially use the Law of Excluded Middle, that $\neg\neg\neg A \vee \neg\neg A$ holds.

Note that we can define a far more relaxed version of negation where Triple Negation Elimination still holds, called **quasi-negation**.

We take any proposition p and define quasi-negation $\sim X$ as $X \to T_p$. This type equates to the proposition $X \implies P$ for any P. This logically makes sense since we know that $\bot \implies$ *anything* by the Principle of Explosion ("*ex falso quodlibet*").

Using this quasi-negation, we define a translation for Classical Statements into STLC as follows:

$$\begin{aligned}(\neg A)^\circ &= \sim A^\circ \\ \top^\circ &= 1 \\ (A \wedge B)^\circ &= A^\circ \times B^\circ \\ \bot^\circ &= p \\ (A \wedge B)^\circ &= \sim\sim(A^\circ + B^\circ)\end{aligned}$$

Note the extra double-negation on the union term. This is only one possible encoding of Classical Logic into a form for which Double Negation Elimination holds. For example, we could place a double negation in front of every term (known as the Kolmogorov Translation), or the de Morgan duality for disjunction $\sim(\sim A^\circ \times \sim B^\circ)$ to avoid unions all together.

We can prove that double negation elimination holds for all of these encoded terms within Intuitionistic logic by creating functions that have the following types:

$$\cdot \vdash \text{dne}_A : \sim\sim A^\circ \to A^\circ$$

$$\begin{aligned}\cdot \vdash \text{dne}_\top &: \sim\sim 1 \to 1 \\ \text{dne}_\top &= \lambda q.\langle\rangle\end{aligned}$$

$$\begin{aligned}\cdot \vdash \text{dne}_{A\wedge B} &: \sim\sim A^\circ \times B^\circ \to A^\circ \times B^\circ \\ \text{dne}_{A\wedge B} &= \lambda q.\left\langle \begin{matrix}\text{dne}_A(\lambda k.q(\lambda p.k(\text{fst } p))) \\ \text{dne}_B(\lambda k.q(\lambda p.k(\text{snd } p)))\end{matrix}\right\rangle\end{aligned}$$

$$\begin{aligned}\cdot \vdash \text{dne}_\bot &: \sim\sim p \to p \\ \text{dne}_\bot &= \lambda q.q(\lambda x.x)\end{aligned}$$

$$\begin{aligned}\cdot \vdash \text{dne}_{A\vee B} &: \sim\sim\sim\sim(A^\circ + B^\circ) \to \sim\sim(A^\circ + B^\circ) \\ \text{dne}_{A\vee B} &= \lambda q.(\lambda k.\lambda a.k(\lambda q.qa))q\end{aligned}$$

$$\begin{aligned}\cdot \vdash \text{dne}_{\neg A} &: \sim\sim(\sim A^\circ) \to \sim A^\circ \\ \text{dne}_{\neg A} &= \lambda q.(\lambda k.\lambda a.k(\lambda q.qa))q\end{aligned}$$

Note the similarity between the terms for $\text{dne}_{A\vee B}$ and $\text{dne}_\neg A$, and the Triple Negation Elimination term $\lambda k.\lambda a.k(\lambda q.qa)$. We call this term *tne* for short.

Now that we have established an embedded form of Intuituinistic logic for which Double Negation Elimination holds, we can begin mapping our encoding of Classical logic into this embedding.