COMS30040

Types & λ-calculus

Lecture Notes for 2022/23

Steven Ramsay
Department of Computer Science
University of Bristol

Contents

1 Proof: Propositional Reasoning 1

2 Proof: Quantifiers, Equality, Theories 9

3 Syntax: Terms and Binding 19

4 Syntax: Redexes, One-Step Reduction and Substitution 27

5 Semantics: One-Step Reduction via Contexts, and Simple Programming 31

6 Semantics: Reduction Sequences and Confluence 35

7 Semantics: Conversion and Recursion 41

8 Computability: Effectiveness and Turing-Completeness 47

9 Computability: Decidability and Undecidability 55

10 Types: Judgements and the Type System 61

11 Types: Derivations and Typability 67

A Proof Rules 73

1. Proof: Propositional Reasoning

In this first lecture, I want to introduce you to a way of thinking about mathematical proof as a kind of programming. The aim is to introduce you to proof (almost) from scratch, give you the rules you need to play the game and then look at examples of the rules in action.

1.1 Proof as a form of programming

Proofs are semi-formal arguments that make a case for the truth of some precise statement. The key word as far as proofs are concerned is *convincing*: a proof is a way of communicating your belief that something is true in a way that will convince another party.

Some proofs are more convincing than others. In mathematics, the statements that we want to give proofs for only have certain shapes — they are built from a few, standard logical operators. Over the years, mathematicians have developed equally standard approaches to dealing with the logical operators in proofs. By following these standard approaches, the benefit is that you will end up with a proof that will be convincing. Proofs of this kind are typically blocks of English text with a scattering of mathematical formulas.

It can be very helpful to think of the individual steps of a proof as being operations that manipulate the *proof state*. The proof state is a pair consisting of a set of *assumptions* and the current *goal*. The goal is the formula you are currently trying to prove and the assumptions are the resources that you have available in order to try to prove it. Each step of the proof (i.e. each application of one of the rules of natural deduction, signalled using an appropriate form of words) changes the proof state by either adding to the set of assumptions or changing the goal. For example, if we have assumptions $A_1, \ldots, A_k$ and we are aiming to prove $A \Rightarrow B$, then the magic words "assume A", which is a form of words used to signal a use of implication introduction, lead us to a proof state in which the assumptions are $A_1, \ldots, A_k, A$ and the goal is B. When you are reading a proof, it is very helpful to picture the proof state in your mind after each step of the proof.

The following example is a proof of a very simple fact from number theory: if two natural numbers n and m add up to zero, then m is not of the form $k + 1$ for any possible k.

Lemma 1.1. *For all $n, m, k \in \mathbb{N}$: if $n + m = 0$ then $m \neq k + 1$.*

Proof. Let n, m and k be natural numbers. Assume $n + m = 0$. To see that $m \neq k + 1$, we assume $m = k + 1$ and try to obtain a contradiction. Combining these assumptions we get $n + (k + 1) = 0$. By the associativity of addition this is the same as $(n + k) + 1 = 0$. Since $n + k$ is a natural number, we have $(n + k) + 1 \neq 0$. We have reached our contradiction. □

Proof step	Assumptions	Goal
	A, B, C	$\forall nmk \in \mathbb{N}.\ n + m = 0 \Rightarrow m \neq k + 1$
Let n, m and k be natural numbers.	A, B, C, $n, m, k \in \mathbb{N}$	$n + m = 0 \Rightarrow m \neq k + 1$
Assume $n + m = 0$.	A, B, C, $n, m, k \in \mathbb{N}$, $n + m = 0$	$m \neq k + 1$
To see that $m \neq k + 1$, we assume $m = k + 1$ and try to obtain a contradition.	A, B, C, $n, m, k \in \mathbb{N}$, $n + m = 0$, $m = k + 1$	$\bot$
Combining these we get $n + (k + 1) = 0$	A, B, C, $n, m, k \in \mathbb{N}$, $n + m = 0$, $m = k + 1$, $n + (k + 1) = 0$	$\bot$
By the associativity of addition, this is the same as $(n + k) + 1 = 0$,	A, B, C, $n, m, k \in \mathbb{N}$, $n + m = 0$, $m = k + 1$, $n + (k + 1) = 0$, $(n + k) + 1 = 0$	$\bot$
Since $n + k$ is a natural number, we have $(n + k) + 1 \neq 0$.	A, B, C, $n, m, k \in \mathbb{N}$, $n + m = 0$, $m = k + 1$, $n + (k + 1) = 0$, $(n + k) + 1 = 0$, $(n + k) + 1 \neq 0$	$\bot$
but this is a contradiction.	A, B, C, $n, m, k \in \mathbb{N}$, $n + m = 0$, $m = k + 1$, $n + (k + 1) = 0$, $(n + k) + 1 = 0$, $(n + k) + 1 \neq 0$, $\bot$	$\bot$

Table 1.1: Trace of the proof of $\forall nmk \in \mathbb{N}.\ n + m = 0 \Rightarrow m \neq k + 1$

Suppose someone gives you a piece of straightline C code (no loops) for computing some complicated function $f(x, y)$ of a pair of integers variables x and y, and asks you to check that it works. To understand what the code is doing, most likely you will start at the beginning of the code, read each statement in turn and keep track of the values of x and y in your head[1] as they are manipulated by the code. By the end, you should be able to tell whether or not the final value returned is $f(x, y)$ (for the original values of x and y) or not. Of course, to do this, you need to understand the "rules" of the C language, i.e. how the syntax fits together and how each statement affects the current state, i.e. the current values of x and y.

The same process is required to understand a proof. You will go through, statement by statement, keeping track of the proof state until, by the end, it should be clear whether or not the proof works, i.e. in the final proof state, the goal is something that is already known. Of course, to do this you will need to know the "rules" of mathematical proof. Giving you those rules is the purpose of the following section, but I think it is useful to go through this example first anyway so you can see what I mean.

One thing that we need to make clear before that, however, is what the starting point for the proof is.

In "real life" mathematicians are trying to prove statements that nobody has ever proven before (and may turn out to be false). To give themselves the best chance to prove their statement, they will employ all the mathematical knowledge that they have at their disposal: countless theorems already proven by the mathematicians that came before them, or that they proved during their career. In other words, they start the proof with a set of assumptions that is enormous (but they know each is justified by a proof, somewhere).

When learning mathematics, however, we are usually proving quite simple statements that someone else has already proven a long time ago. Hence, it can be a bit confusing to the student to know which already proven statements they are allowed to assume and which they are required to prove first. This will not be a problem during *Types and Lambda Calculus* because (I guess) you don't know any basic theorems of this theory, so we will just agree that you can assume only those statements that we have proven so far during the course of the unit (and anything you know about sets and strings).

The example above, however, is in number theory, and you probably already have a good idea of what simple statements are true about natural numbers and which are not. So, for this example, I will be very clear about what we are allowing ourselves to assume at the start of the proof:

(A) for all $p \in \mathbb{N}$: $p + 1 \neq 0$

(B) for all p, q and r in $\mathbb{N}$: $p + (q + r) = (p + q) + r$

(C) for all $p, q \in \mathbb{N}$: $p + q \in \mathbb{N}$

Statement B is what is referred to in the proof as "associativity of addition". Statements A and C are used in the last step, but are not referred to by any name. This probably means the author of the proof believes them to be so simple that they are not worth mentioning. Now let's analyse the proof step by step starting in the proof state containing only these assumptions A, B and C. A trace of the

[1] If you are not so experienced with C, or the code is very complicated, then it may be useful to write down the evolving values of x and y on a scrap of paper.

proof is given in Table 1.1, whose final two columns shows the proof state after executing each of the statements.

A few things about the general shape of this proof are worth observing.

- First, the proof appears to prove the statement, because in the final proof state the goal is already contained in the assumptions. However, we can't know that it is really a valid proof unless we check that each of the proof steps really transforms the proof state before-it into the proof state after-it, and this requires a knowledge of the rules.

- The set of assumptions grows larger over the course of the proof.

- The goal starts off quite large and complicated, but is progressively simplified over the first few proof steps and, once it is as simple as possible, it doesn't change from that point on.

Backward Reasoning Steps 1–3 in which the goal gets simpler are called *backwards reasoning*. It is quite typical that a proof starts by performing backward reasoning. Backwards reasoning steps are of the form "To prove a goal of shape A it is enough to prove the simpler goal(s) $B_1, \ldots, B_n$". In the case of the example, the particular backwards steps that are employed only generate a single simple goal. For example, step 2 uses the backwards reasoning rule for implication, which says "to prove A implies B, it is enough to assume A is true and then try to prove B". Backwards reasoning is very easy because the backwards reasoning steps you are allowed to make mostly just depend upon the syntactic shape of the goal: is it a conjunction, an implication, a for all, etc. Often, but not always, backwards reasoning not only simplifies the goal, but also adds new assumptions (which is helpful for fowards reasoning).

Forwards Reasoning Steps 4–7 are *forwards reasoning* steps. Forwards reasoning consists of those proof steps that do not change the goal, but simply deduce more and more assumptions (facts) by combining existing ones. The majority of a non-trivial proof is spent on forwards reasoning. Forwards reasoning is a bit more difficult only because you can build up a large set of assumptions and spotting which ones can be fruitfully combined to deduce new ones *that are actually relevant to proving your goal* requires some insight.

1.2 Rules of the game

In Appendix A are listed all the rules that suffice for any mathematical proof in this unit, organised by logical connective. For each logical connective there is a list of the forwards and backwards rules for that connective: the backwards rules tell you how you can simplify a goal which is built from that connective at the outer level, the forwards rules tell you how you can deduce new assumptions from an existing one built from that connective at the outer level.

Each rule is given in form of a table entry of shape:

Assms	Goal

"blah blah"

Assms	Goal

This tells you the syntax of the proof rule, e.g. the words "blah blah", and the semantics, i.e. how it changes the proof state. For a proof to be valid, the commands must hang together as a correct program, so you must ensure that you only use a rule when the proof state before has the correct shape. For example, the backwards reasoning rule for implication requires that the goal is literally an implication, i.e. has the shape $A \Rightarrow B$. It does not apply when the goal has shape, e.g. $\forall x \in X.\, A \Rightarrow B$, which is a for all (whose body is an implication), nor when the goal has hape $(A \Rightarrow B) \wedge C$, which is a conjunction (whose right conjunct is an implication). The exact form of words is not important, as long as it is clear from the context (i.e. the proof state) which rule are you trying to invoke.

Let's take a look at some of the more interesting ones.

Implication

$A \Rightarrow B$ A implies B If A then B. Suppose A, then B.

Backward Rule

To prove $A \Rightarrow B$ starting from some assumptions, it suffices to instead prove B with A added to the assumptions.

Assms	Goal
...	$A \Rightarrow B$

"Assume A. We prove B as follows..."

"To show $A \Rightarrow B$, we assume A. ..."

"Assume A. ..."

"Suppose A is true. ..."

Assms	Goal
$\ldots, A$	B

Forwards Rule

If you already know $A \Rightarrow B$ and you already know A, then you know B.

Assms	Goal
$\ldots, A \Rightarrow B, A$	C

"From $A \Rightarrow B$ and A we conclude B."
"Applying $A \Rightarrow B$ to A we obtain B."
"We have B by *modus ponens*."

Assms	Goal
$\ldots, A \Rightarrow B, A, B$	C

Negation

$\neg A$ not A A is false A is absurd

Backwards Rule

To prove $\neg A$ starting from some assumptions, it suffices to derive a contradiction (give a proof of false) starting from the same assumptions with A added.

Assms	Goal
$\ldots$	$\neg A$

"We assume A and try to obtain a contradiction."
"To show $\neg A$, we assume A..."
"Assume A."

Assms	Goal
$\ldots, A$	$\bot$

Forwards Rule

If you know $\neg A$ and you also know A, then you know absurdity.

Assms	Goal
$\ldots, \neg A, A$	C

"From A and $\neg A$ we obtain a contradiction."

Assms	Goal
$\ldots, \neg A, A, \bot$	C

1.3 Examples

The following is an example of pure logic. Let's suppose we do not know any facts about logic already (i.e. our starting assumptions are empty).

Lemma 1.2. *A implies $\neg\neg A$*

Proof. Assume A. We wish to show $\neg\neg A$, so we assume $\neg A$ and try to obtain a contradiction. From A and $\neg A$ we obtain the desired contradiction. □

Make sure you can follow which rule is being used in each step and how the proof state evolves. The first step in becoming proficient at proving is to at least be able to check that someone else's proof is valid. The following table gives a listing of the proof state at the end of each step in the proof.

Proof step	Assumptions	Goal
		$A \Rightarrow \neg\neg A$
"Assume A"	A	$\neg\neg A$
"... so we assume $\neg A$ and ... contradiction."	$A, \neg A$	$\bot$
"From A and $\neg A$ we obtain..."	$A, \neg A, \bot$	$\bot$

In the following example, we collect quite a few assumptions from our initial backward reasoning. In such cases it can sometimes be helpful to label them as we go for when we want to combine them in new ways during forward reasoning.

Lemma 1.3. *if A implies B then $\neg B$ implies $\neg A$*

Proof. Assume $A \Rightarrow B$ (A1) and assume $\neg B$ (A2). We aim to prove $\neg A$, so we will assume A (A3) and try to obtain a contradiction. From (A1) and (A3), we obtain B (A4). From (A4) and (A2) we obtain the desired contradiction. □

This is what is going on in my head when I read this proof back to myself. Let's again suppose that my starting knowledge is zero. Make sure that you can identify which backwards or forwards rule was used in each case.

Proof step	Assumptions	Goal
		$(A \Rightarrow B) \Rightarrow \neg B \Rightarrow A$
"Assume $A \Rightarrow B$"	A1:$A \Rightarrow B$	$\neg B \Rightarrow \neg A$
"assume $\neg B$"	A1:$A \Rightarrow B$, A2:$\neg B$	$\neg A$
"assume A ... try ... contradiction."	A1:$A \Rightarrow B$, A2:$\neg B$, A3:A	$\bot$
"From (A1) and (A3), we obtain B."	A1:$A \Rightarrow B$, A2:$\neg B$, A3:A, A4:B	$\bot$
"From ... obtain ... contradiction."	A1:$A \Rightarrow B$, A2:$\neg B$, A3:A, A4:B, $\bot$	$\bot$

One important reason that it is helpful to have the proof state in your head is that, because the proof steps are just certain phrases in English, sometimes the proof state is needed for disambiguation. For example, it may only be possible to understand whether the word "assume" heralds a use of implication introduction or a use of negation introduction, based on goal at the point at which the word was used.

The next example also uses conjunction.

Conjunction

$A \wedge B$ A and B

Backwards Rule

To prove $A \wedge B$ from some assumptions, it suffices to give two separate proofs: one proof of A from the assumptions and one proof of B from the assumptions.

Assms	Goal
...	$A \wedge B$

"We prove A and B separately. To prove A..."
"For A, we argue as follows ... For B, ..."

Assms	Goal
...	A

Assms	Goal
...	B

The proof splits into two at this point, so you have to keep in mind two states, although you can just work with one at a time.

Forwards Rule

If you know $A \wedge B$, then you can know A and you know B.

Assms	Goal
$\ldots, A \wedge B$	C

"" (too trivial to mention)

Assms	Goal
$\ldots, A \wedge B, A, B$	C

Lemma 1.4. *$A \Rightarrow (B \Rightarrow C)$ implies $A \wedge B \Rightarrow C$*

Proof. Assume $A \Rightarrow (B \Rightarrow C)$ (*). Assume A and B. From (*) and $A, B \Rightarrow C$ follows. From this and B, C follows. □

Proof step	Assumptions	Goal
		$(A \Rightarrow (B \Rightarrow C)) \Rightarrow (A \wedge B) \Rightarrow C$
"Assume $A \Rightarrow (B \Rightarrow C)$ (*)"	*:$A \Rightarrow (B \Rightarrow C)$	$A \wedge B \Rightarrow C$
"Assume A and B"	*:$A \Rightarrow (B \Rightarrow C), A, B$	C
"From (*) and $A, B \Rightarrow C$ follows."	*:$A \Rightarrow (B \Rightarrow C), A, B, B \Rightarrow C$	C
"From this and B, C follows."	*:$A \Rightarrow (B \Rightarrow C), A, B, B \Rightarrow C, C$	C

In this example I omitted the step between assuming $A \wedge B$ and deducing A and B separately because there is never any reason not to immediately deduce the two conjuncts separately.

2. Proof: Quantifiers, Equality, Theories

In this second part of our crash course on proof, we introduce first-order reasoning with the rules for the quantifiers, equality and how to work with theories.

2.1 Quantifier rules

Universal quantification

$\forall x \in X. A$	$\forall x : X. A$	for all x in X, A	A holds of all x in X

Backwards Rule

To give a proof of $\forall x \in X. A$ from some assumptions, it suffices to give a proof of A from the same assumptions with $x \in X$ added, as long as you have not made any prior assumptions about the name x (x doesn't appear in any of your existing assumptions).

Assms	Goal
...	$\forall x \in X. A$

with x not occurring free in the assms

"Let $x \in X$. We show A."
"Let x be an arbitrary member of X. We show A."
"Suppose x is in X. ... therefore A."

Assms	Goal
$\ldots, x \in X$	A

Forwards Rule

If you have know $\forall x \in X.\, A$ and you know $t \in X$, then you know A with all occurrences of x replaced by t.

Assms	Goal
$\ldots \forall x \in X.A\, t \in X$	C

"It follows that A holds of t"

Assms	Goal
$\ldots, \forall x \in X.A,\ t \in X, A[t/x]$	C

Existential Quantification

$\exists x \in X.\, A$	$\exists x : X.\, A$	there exists x in X such that A	A holds of some x in X

Backwards Rule

To prove $\exists x : X.\, A$ from some assumptions, it suffices to find a witness t and give proofs that $t \in X$ and of A with every occurence of x replaced by t, starting from the same assumptions.

Assms	Goal
$\ldots$	$\exists x \in X.\, A$

"We show that A holds of t."
"We take t as witness. To see $A[t/x]$..."
"We show that t is such an x."

Assms	Goal
$\ldots$	$t \in X$

Assms	Goal
$\ldots$	$A[t/x]$

Here, $A[t/x]$ means A with all free occurrences of x replaced by t.

Forwards Rule

Assms	Goal
$\ldots \exists x \in X . A$	C

with x not occurring free in the assms

"Let x be the witness..."

Assms	Goal
$\ldots \exists x \in X . A x \in X A$	C

2.2 Playing the game

There are generally many ways that one can complete a proof because, in any given proof state, several forwards and backwards rules may be applicable. In many cases, it requires trial and error until you find the right strategy for a proof of a given statement. However:

An excellent rule of thumb is to:

1. Apply backwards reasoning until the goal is as simple as possible (doesn't involve any more logical connectives) and all the delicious extra assumptions have been extracted.
2. Rewrite the goal using any non-inductive/recursive definitions that you have assumed (see Theories and Equality in Section 2.3). This may give you even more opportunities to apply backwards reasoning steps.
3. Step back and consider what you now know to be true (your assumptions). Then apply forwards reasoning to deduce more and more of them until you have finally deduced the goal you are looking to prove.

This strategy will not work in all cases, but I would always use it as my first attempt at a proof. In this unit, it will serve you very well for almost every proof.

Here's a slightly fiddly example that demonstrates some of the quantifier rules. In it we use the notation $A[y/x]$ to mean the formula A but with any (free) occurrence of x replaced by y. For example, if A is $x = 0 \vee x = 1$, then $A[y/x]$ is $y = 0 \vee y = 1$. In the statement that follows, the point is that $\forall x \in X . A$ is really the same as $\forall y \in X . A[y/x]$, we have just renamed a bound variable to give it a more convenient name, y.

Lemma 2.1. *Suppose x does not occur free in B. Then $(\exists x \in X . A) \Rightarrow B$ iff $\forall x \in X . A \Rightarrow B$.*

Proof. Assume x does not occur free in B. We proceed to prove both directions separately:

- In the forward direction, suppose $(\exists x \in X.\, A) \Rightarrow B$ (*). Our goal is to show $\forall x \in X.\, A \Rightarrow B$, but let us rename the bound variable x to avoid confusion with the x in our assumptions. So we prove $\forall y \in X.\, A[y/x] \Rightarrow B[y/x]$. Then let $y \in X$ and assume $A[y/x]$. I claim that $\exists x \in X.A$. To see why, take the witness to be y. We already know that $y \in X$ and $A[y/x]$ so that proves the claim. It follows from this and (*) that, therefore, B. Our goal is to show $B[y/x]$, but we assumed that B does not contain any occurrence of x, so replacing all occurrences of x in B by y is just B, i.e. $B = B[y/x]$. Hence, we have proven the goal.

- In the backward direction, suppose $\forall x \in X.\, A \Rightarrow B$ (*). Then suppose $\exists x \in X.\, A$. Let the witness be some $y \in X$ (so we know $A[y/x]$). Then it follows from (*) that $A[y/x] \Rightarrow B[y/x]$. Since we have the former already, we obtain the latter $B[y/x]$. But, we started by assuming that x does not occur in B, so $B[y/x]$ is exactly B.

□

Here is the trace of the backward direction part of the proof, the initial goal is a bit long so let us shorten it to $G \coloneqq (\forall x \in X.\, A \Rightarrow B) \Rightarrow ((\exists x \in X.\, A) \Rightarrow B)$.

Proof step	Assumptions	Goal
	$x \notin \mathsf{FV}(B)$	G
"suppose ... (*)"	$x \notin \mathsf{FV}(B)$, $\forall x \in X.\, A \Rightarrow B$	$(\exists x \in X.\, A) \Rightarrow B$
"Then suppose ..."	$x \notin \mathsf{FV}(B)$, $\forall x \in X.\, A \Rightarrow B$, $\exists x \in X.\, A$	B
"Let the witness be..."	$x \notin \mathsf{FV}(B)$, $\forall x \in X.\, A \Rightarrow B$, $\exists x \in X.\, A$, $y \in X, A[y/x]$	B
"The it follows from (*)"	$x \notin \mathsf{FV}(B)$, $\ldots, A[y/x], A[y/x] \Rightarrow B[y/x]$	B
"Since we have the former..."	$x \notin \mathsf{FV}(B)$, $\ldots, A[y/x], A[y/x] \Rightarrow B[y/x], B[y/x]$	B

Bending the rules

There are many, many ways that people take short-cuts when writing proofs. Sometimes it is just laziness, but other times it makes the difference between a proof that is a reasonable size and a proof that is large beyond comprehension. Generally, it's ok to leave steps implicit as long as you believe that your reader can fill in the gaps: it should be possible, in principal, to derive all the explicit steps that you missed out without having to do too much search.

Sometimes backwards reasoning steps are left implicit. For example, when proving a goal of the form $\forall x \in X.\, A \Rightarrow B$, often the proof will launch straight into proving B and, as part of that proof, will use A and $x \in X$ as if they are already assumed.

Conversely, when forwards reasoning from an assumptions of shape $\forall x \in X.\, A \Rightarrow B$ and $A[t/x]$ and $t \in X$, often the proof will immediately deduce $B[t/x]$, performing the forwards rule for forall and the forwards rule for implication in one combined step.

I will try not to leave backwards steps implicit, because they are crucial to understanding where the proof is headed, which is important if you are just starting out. I will, however, often combine forwards steps together if I think it remains clear what is happening.

2.3 Equality

We have talked about the generic, purely logical aspects of proof but, in reality, most proofs argue something in a specific domain (or, in the logical jargon, a specific theory). When you are working with a particular theory, e.g. the theory of arithmetic over the natural numbers, the theory gives you more to work with: for example, the language of formulas is enlarged to include new constants, function symbols and relations like 0, + and < respectively. Along with those symbols come axioms or definitions, and you can make use of these as extra assumptions in your proofs.

Most theories make use of some notion of equality over the inhabitants of the domain. The rules we can use for reasoning about equality are very straightforward:

At any time, you can assume $s = s$, for any term s, which is obviously true.

Assms	Goal
...	C

"" (too trivial to mention)

Assms	Goal
$\ldots, s = s$	C

If you know an equation, you can use it to rewrite the goal.

Assms	Goal
$\ldots, s = t$	$C[s/x]$

"$C[s/x]$ is just $C[t/x]$"
"$C[s/x]$ is therefore the same as $C[t/x]$"

Assms	Goal
$\ldots, s = t$	$C[t/x]$

The use of $C[s/x]$ here is just a way of saying that you have some proposition which contains an occurrence of s and the corresponding $C[t/x]$ indicates replacing that occurrence of s by t.

If you know an equation, you can use it to rewrite an assumption.

Assms	Goal
$\ldots, s = t, A[s/x]$	C

"$C[s/x]$ is just $C[t/x]$"
"$C[s/x]$ is therefore the same as $C[t/x]$"

Assms	Goal
$\ldots, s = t, A[s/x], A[t/x]$	C

2.4 Theories

A theory is just a collection of true sentences, typically organised around some theme, like a particular datatype and its operations "the theory of arithmetic" or a class of datatypes "the theory of groups". Theories bring with them definitions and existing theorems, which both contribute to your starting assumptions when you set out to prove something.

The following example uses the following recursive definition of mutliplication of natural numbers:

$$\begin{aligned} 0 * p &= 0 \\ (q+1) * p &= q * p + p \end{aligned}$$

If an equational definition of some function f is valid in a given theory, then we can just take the equations as extra assumptions when reasoning about f.

As usual, when we see a statement, such as the equations above, that contains free variables (here p and q), then the convention is to regard them as universally quantified, i.e:

$$\forall p \in \mathbb{N}.\ 0 * p = 0 \qquad \text{and} \qquad \forall pq \in \mathbb{N}.\ (q+1) * p = q * p + p$$

For this reason, it is extremely important to know your definitions well, because you will always be using them when you come to writing proofs. For the purpose of this example, we also allow ourselves to assume the commutativity of multiplication:

$$\forall pq \in \mathbb{N}.\ p * q = q * p$$

Lemma 2.2. *for all $n, m \in \mathbb{N}$: if $n = 0$ or $m = 0$ it follows that $n * m = 0$*

Proof. Let $n, m \in \mathbb{N}$ and assume $n = 0 \vee m = 0$. We show that $n * m = 0$ by case analysis.

- When $n = 0$ then the goal $n * m = 0$ is $0 * m = 0$ which is true by the definition of multiplication.
- When $m = 0$ then the goal $n * m = 0$ is $n * 0 = 0$ which, by the commutativity of mutliplication, is $0 * n = 0$ and this also true by definition.

□

In this proof we also used a rule to do with disjuncton – we worked forwards from an assumption that $n = 0 \vee m = 0$. The formulation of the rule is interesting:

Forwards Rule

If you have already know $A \vee B$ and, additionally, you can give the following two proofs:

- a proof of C starting from your assumptions and A
- a proof of C starting from your assumptions and B

Then you will also know C.

Assms	Goal
$\ldots, A \vee B$	C

"We proceed by cases on $A \vee B$.
– Assume A ... Hence C.
– Assume B ... Hence C."
"We analyse the two cases."
"We proceed by case analysis on $A \vee B$."

Assms	Goal
$\ldots, A \vee B, A$	C

Assms	Goal
$\ldots, A \vee B, B$	C

This forces the proof to split into two, and you're not done until you have completed both of them.

So, this is how the proof state evolves, with G abbreviating the original statement to be proven and the three assumptions that we know about from the theory abbreviated by their numbers:

(i) $\forall p.\ 0 * p = 0$

(ii) $\forall pq.\ (q+1) * p = q * p + p$

(iii) $\forall pq.\ p * q = q * p$

Proof step	Assumptions	Goal
	(i), (ii), (iii)	G
"Let $n, m \in \mathbb{N}$..."	(i), (ii), (iii), $n, m \in \mathbb{N}$	if $n = 0$ or $m = 0$, then $n * m = 0$
"and assume $n = 0 \vee m = 0$"	(i), (ii), (iii), $n, m \in \mathbb{N}$, $n = 0 \vee m = 0$	$n * m = 0$

"We show ... by case analysis"

"When $n = 0$"	(i), (ii), (iii), $n, m \in \mathbb{N}$, $n = 0 \vee m = 0$, $n = 0$	$n * m = 0$
"$n * m = 0$ is $0 * m = 0$ ... true by defn"	(i), (ii), (iii), $n, m \in \mathbb{N}$, $n = 0 \vee m = 0$, $n = 0$	$0 * m = 0$

"When $m = 0$"	(i), (ii), (iii), $n, m \in \mathbb{N}$, $n = 0 \vee m = 0$, $m = 0$	$n * m = 0$
"$n * m = 0$ is $n * 0 = 0$	(i), (ii), (iii), $n, m \in \mathbb{N}$, $n = 0 \vee m = 0$, $m = 0$	$n * 0 = 0$
"by commutativity is $0 * n = 0$, true by defn"	(i), (ii), (iii), $n, m \in \mathbb{N}$, $n = 0 \vee m = 0$, $m = 0$	$0 * n = 0$

Induction

Some theories even come with their own proof rules for reasoning about the datatypes involved. Induction over the natural numbers is an excellent example:

To prove $\forall n \in \mathbb{N}.\, A$ it suffices to prove $A[0/n]$ and, assuming $A[k/n]$, to prove $A[k+1/n]$ for all k.

Assms	Goal
...	$\forall n \in \mathbb{N}.\, A$

"We prove the result by induction on $x \in \mathbb{N}$.
– When $x = 0$ ... hence A is true of 0.
– When $x = k + 1$, assume A holds of k
... A is true of $k + 1$."

"By induction on $n \in \mathbb{N}$. We analyse the cases..."

Assms	Goal
...	$A[0/n]$

Assms	Goal
$\ldots, A[k/n]$	$A[k+1/n]$

Recall that $A[0/n]$ means the formula A but where n is replaced by 0. So, when we invoke induction on the natural numbers in order to prove a formula of the form $\forall n \in \mathbb{N}.\, A$, we are required to prove two new goals: A but with n being specifically 0, and A with n being $k + 1$ but under the assumption, usually given a special name for some reason: *the induction hypothesis*, of A with k replacing n. So, in the sense above, natural number induction is an alternative introduction rule (backward reasoning) for those formulas of shape $\forall x : X.\, A$ in the specific case that X is $\mathbb{N}$.

We get an induction rule every time we introduce a datatype defined by a context free grammar. For example, recall the grammar for the set of regular expressions, REGEXP, over {1}:

$$e ::= \epsilon \mid \emptyset \mid 1 \mid (e_1 + e_2) \mid (e_1 e_2) \mid e^*$$

From the structure of the grammar (think of the shape of the parse trees), we get a proof rule that can be used to prove formulas of the form "for all strings e in the language defined by the grammar, $P(e)$". A complete use of the rule consists of proving a number of subgoals, one for each of the different productions/alternatives in the grammar. In this case we have 6 alternatives and so there are 6 goals to prove. Each goal consists of proving the body of the original goal, the $P(e)$, for each

of the 6 different shapes of e. Where a production in the grammar is recursive, i.e. that alternative contains occurrences of the metavariable e again (or variations thereon, e.g. e_1, e_2 etc), then we can assume that P holds of those "smaller" occurrences when we proving the goal corresponding to that case.

Assms	Goal
...	$\forall e \in \mathsf{REGEXP}.\ A$

"We prove the result by induction on $e \in \mathsf{REGEXP}$.
– When $e = \epsilon$ ... hence A is true of ϵ.
– When $e = \emptyset$ ... hence A is true of $\emptyset$.
– When $e = 1$ ... hence A is true of 1.
– When e is of shape $e_1 + e_2$, assume A holds of e_1 and e_2 separately ... A is true of $(e_1 + e_2)$.
– When e is of shape $(e_1 e_2)$, assume A holds of e_1 and e_2 separately ... A is true of $(e_1 + e_2)$.
– When e is of shape e_1^*, assume A holds of e_1 ... A is true of e_1^*."

Assms	Goal
...	$A[\epsilon/e]$

Assms	Goal
...	$A[\emptyset/e]$

Assms	Goal
...	$A[1/e]$

Assms	Goal
$\ldots, A[e_1/e], A[e_2/e]$	$A[(e_1 + e_2)/e]$

Assms	Goal
$\ldots, A[e_1/e], A[e_2/e]$	$A[(e_1 e_2)/e]$

Assms	Goal
$\ldots, A[e_1/e],$	$A[e_1^*/e]$

In the interests of proving something interesting, let us define the language $L(e)$ denoted by a regular expression e, recursively, as follows:

$$L(\epsilon) = \{\epsilon\}$$
$$L(\emptyset) = \emptyset$$
$$L(1) = \{1\}$$

$$L(e_1 + e_2) = L(e_1) \cup L(e_2)$$
$$L(e_1 e_2) = \{uv \mid u \in L(e_1) \wedge v \in L(e_2)\}$$
$$L(e^*) = \bigcup_{n \in \mathbb{N}} e^n$$

Then we can prove the following fact about star-free languages:

Lemma 2.3. *For each regular expression e, if e does not contain *, then $L(E)$ is finite.*

Proof. We proceed by induction on e.

- When $e = \epsilon$, (suppose e does not contain *) then, by definition, $L(e) = \{\epsilon\}$, which is finite.

- When $e = \emptyset$, (suppose e does not contain *) then, by definition, $L(e) = \emptyset$, which is finite.
- When $e = 1$, (suppose e does not contain *) then, by definition, $L(e) = \{1\}$ which is finite.
- When e is of shape $(e_1 + e_2)$, proceed as follows. Assume the induction hypotheses:

  (i) If e_1 does not contain *, then $L(e_1)$ is finite.

  (ii) If e_2 does not contain *, then $L(e_2)$ is finite.

  Now, suppose $(e_1 + e_2)$ does not contain *. It follows that neither e_1 nor e_2 can contain *. Hence, from induction hypothesis (i), we obtain that $L(e_1)$ is finite and, from induction hypothesis (ii), we obtain that $L(e_2)$ is finite. By definition, $L(e_1 + e_2)$ is the union of $L(e_1)$ and $L(e_2)$, and the union of two finite sets is finite.
- When e is of shape $(e_1 e_2)$, proceed as follows. Assume the induction hypotheses:

  (i) If e_1 does not contain *, then $L(e_1)$ is finite.

  (ii) If e_2 does not contain *, then $L(e_2)$ is finite.

  Now, suppose $(e_1 e_2)$ does not contain *. It follows that neither e_1 nor e_2 can contain *. Hence, from induction hypothesis (i), we obtain that $L(e_1)$ is finite and, from induction hypothesis (ii), we obtain that $L(e_2)$ is finite. By definition, $L(e_1 e_2)$ is the set of all concatenated pairs of words uv with u from $L(e_1)$ and $v \in L(e_2)$. Since we now know $L(e_1)$ and $L(e_2)$ are finite, it follows that there are only finitely many such pairings.
- When e is of shape e_1^*, proceed as follows. Assume the induction hypothesis:

  (i) If e_1 does not contain * then $L(e_1)$ is finite.

  Suppose e_1^* does not contain *. Clearly (i.e. by the basic properties of strings) e_1^* contains *. Hence, we obtained a contradiction and the subgoal follows trivially.

□

3. Syntax: Terms and Binding

In this unit we are going to study a kind of prototypical functional programming language, called PCF, or more fully *Programming Computable Functions*. It has a bit of a strange heritage, because it was first introduced by Gordon Plotkin in 1977 as a programming language alternative to Dana Scott's *Logic of Computable Functions*, but it has become one of the standard programming languages that theorists use to investigate the central questions of programming languages and type systems.

In this lecture, we are going to get some intuitions about PCF, then we will see the formal definition of the syntax and finally we will introduce "capture-avoiding" substitution, which is the natural way to combine syntax with binders.

3.1 Intuitions

When we talk about programming languages, it usually suffices to talk about them abstractly as a group of features – the kinds of expression that they support. In these terms we would characterise PCF as a programming language characterised by supporting: *arithmetic*, *conditionals*, *higher-order functions*, and *general recursion*. Now let's explore the language informally by taking a look at each of these features.

Arithmetic

Arithmetic is provided by built-in natural numbers. The numbers are given in unary notation: there is a constant for zero, Z, and a constant for the successor of any given natural number, S. These behave like constructors in Haskell, so we could imagine that they have been introduced via a datatype declaration like:

```
data Nat = Z | S Nat
```

But our language is so simple that we will not have the ability to program new datatypes like this, S and Z are baked-in. With them we can construct the natural number k by applying the successor to the zero k-times, e.g the first three naturals are:

$$\mathsf{Z}, \qquad \mathsf{S}\ \mathsf{Z}, \qquad \mathsf{S}\ (\mathsf{S}\ \mathsf{Z}), \qquad \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))$$

There is also a built-in predecessor function pred which performs subtraction-by-1. When it's argument is zero it is defined so as to just return zero again:

$$\begin{aligned} \mathsf{pred}\ \mathsf{Z} &\approx \mathsf{Z} \\ \mathsf{pred}\ (\mathsf{S}\ N) &\approx N \end{aligned}$$

Here you can see that applying a function f to an input x is written using juxtaposition $f\ x$, as in typical functional programming languages. You can also see the $\approx$ notation we will use for equality of program terms. We are going to be careful about when two programs behave the same, as here, and when they are literally the same, i.e. the exact same source code, for which we will use the usual notation $=$ for identity.

Addition, subtraction, multiplication and so on are not built-in to the language, but we can define addition by iterating the successor, subtraction by iterating the predecessor, multiplication by iterating addition and so on.

Conditionals

To keep our language as simple as possible, there are no other base types other than natural numbers. In particular, there are no built-in Booleans. Hence, the built-in conditional, ifz, is perhaps not what you expect:

$$\mathsf{ifz}\ \mathsf{Z}\ N\ P\ \approx\ N$$
$$\mathsf{ifz}\ (\mathsf{S}\ M)\ N\ P\ \approx\ P$$

It takes three arguments as input, and returns the second N or third P depending on whether the first was zero or not. So we can think of the arguments as the guard, the then-branch and the else-branch respectively. When the guard is Z, it returns the then-branch N and when the guard is the successor of something, it returns the else-branch P.

Higher-order functions

Whilst most general purpose programming languages we meet will have the ability to perform arithmetic and do conditional branching, it is the support of higher-order functions that marks PCFas a *functional programming language*. Higher-order functions are written explicitly using λ*-abstraction*. A function that takes x as input and returns expression M (which will typically contain x) is written:

$$\lambda x.M$$

As remarked above, when we want to supply such a function with an input, say N, we write the two juxtaposed and say that the former is *applied* to the latter, and the result is M but with x everywhere replaced by N, which we write as $M[N/x]$.

$$(\lambda x.M)\ N\ \approx\ M[N/x]$$

In fact, as we shall see when we define the language precisely, abstraction and application are the *only* mechanism for building programs! This is what makes our language a λ*-calculus*.

For example, the function that takes a number x as input and returns $x+2$ is written: $\lambda x.\mathsf{S}\ (\mathsf{S}\ x)$. When we supply it with one as an argument, i.e. $\mathsf{S}\ \mathsf{Z}$, then the result is three, i.e:

$$(\lambda x.\mathsf{S}\ (\mathsf{S}\ x))\ (\mathsf{S}\ \mathsf{Z})\ \approx\ \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))$$

where we have taken the expression that is returned $\mathsf{S}\ (\mathsf{S}\ x)$ and replaced every occurrence of x by the argument, which was $\mathsf{S}\ \mathsf{Z}$.

The function that takes a number x and always returns 0 is written: $\lambda x.\,\mathsf{Z}$. When we apply it to the input 2, written $\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})$, then the output will be Z, because we compute by replacing all occurrences of the formal parameter x in the return expression Z by the actual parameter 2:

$$(\lambda x.\,\mathsf{Z})\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))\ \approx\ \mathsf{Z}$$

In Haskell, λ is written in the best ascii approximation, which was deemed to be the backslash \ and an arrow $\rightarrow$ is used to separate the input from the output instead of the full-stop, so the above application would be written:

```
(\x → S (S x)) (S Z)
```

We use iterated λ-abstractions to define functions of more than one argument. For example the function that takes a function f and a number x as input and returns the result of applying f to x twice is written:

$$\lambda f.\,\lambda x.\,f\ (f\ x)$$

We sometimes call such functions *curried*. Strictly speaking, this is the function that takes a function f as input and returns a function $\lambda x.\,f\ (f\ x)$, which itself takes a number x and returns $f\ (f\ x)$, but in practice we tend to think of it as a single function of two arguments f and x. When we apply this function to two arguments, say pred and the numeral for 3, $\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))$, then we can compute as follows:

$$\begin{aligned}
&(\lambda f.\,\lambda x.\,f\ (f\ x))\ \mathsf{pred}\ (\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})))\\
&\quad\approx\ (\lambda x.\,\mathsf{pred}\ (\mathsf{pred}\ x))\ (\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})))\\
&\quad\approx\ \mathsf{pred}\ (\mathsf{pred}\ (\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))))\\
&\quad\approx\ \mathsf{S}\ \mathsf{Z}
\end{aligned}$$

General recursion

Recursion (i.e. the ability to compute something by potentially unbounded repetition) is given by a built-in *fixed-point combinator* called fix. It's behaviour on any input M is as follows:

$$\mathsf{fix}\ M\ \approx\ M\ (\mathsf{fix}\ M)$$

It seems a bit mysterious and, indeed, we will need to devote a lecture to a proper discussion of fix, but for now let us just remark that we can use it to define e.g. addition:

$$\underline{\mathsf{add}} = \mathsf{fix}\ (\lambda f\,x\,y.\,\mathsf{ifz}\ x\ y\ (f\ (\mathsf{pred}\ x)\ (\mathsf{S}\ y)))$$

Let make a few points about this notation.

First, notice we have abbreviated the iterated abstraction $\lambda f.\,\lambda x.\,\lambda y.\,\ldots$ as $\lambda f\,x\,y.\,\ldots$, this is something we will commonly do. Second, notice that we have introduced a new name $\underline{\mathsf{add}}$ and stated that it is identical to $\mathsf{fix}\ (\lambda f\,x\,y.\,\mathsf{ifz}\ x\ y\ (f\ (\mathsf{pred}\ x)\ (\mathsf{S}\ y)))$.

PCF doesn't have a built-in mechanism for making definitions, for defining new functions. However, although this would make writing programs extremely painful in practice, it is not really a hindrance to doing theory because we can just agree among ourselves that $\underline{\mathsf{add}}$ is our abbreviation

for the term on the right-hand side. This means that whenever we use add, e.g. when we state an equivalence like:

$$\underline{\mathsf{add}}\ (\mathsf{S}\ \mathsf{Z})\ \mathsf{Z} \approx \mathsf{S}\ \mathsf{Z}$$

it is just an abbreviation for the expanded statement:

$$(\mathsf{fix}\ (\lambda f\, x\, y.\, \mathsf{ifz}\ x\ y\ (f\ (\mathsf{pred}\ x)\ (\mathsf{S}\ y))))\ (\mathsf{S}\ \mathsf{Z})\ \mathsf{Z} \approx \mathsf{S}\ \mathsf{Z}$$

This actually gives us the best of all worlds, because we can define abbreviations and build compact programs out of them, but when we analyse an arbitrary program we only have to consider a handful of primitive notions. Although our language is so small and simple, we will be adding everything the modern programmer could possibly need as abbreviations.

3.2 Syntax

We define the possible expressions of the programming language precisely, using a grammar. First, let us assume we have a countably infinite supply of program variable names. It actually won't matter what the variable names concretely look like, you could think of them as Haskell identifiers but we will never actually need to write them down. What we will do is use the letters towards the end of the alphabet (and variations): x, y, y', x_1, x_2 and so on to stand for arbitrary ones. Then program expressions, which we call *terms*, are built from variables and constants (the built-ins of PCF) using abstraction and application.

Definition 3.1. *The **terms** of PCF are those strings that can be generated by the following grammar:*

$$\begin{array}{rrcl} \textit{(Terms)} & M, N & ::= & x \mid c \mid (\lambda x.M) \mid (MN) \\ \textit{(Constants)} & c & ::= & \mathsf{fix} \mid \mathsf{Z} \mid \mathsf{S} \mid \mathsf{pred} \mid \mathsf{ifz} \mid \mathsf{wrong} \end{array}$$

*We use M, N, P, Q and other capital letters from the end of the alphabet to stand for arbitrary terms. In this grammar, x is any **variable**, c is any **constant**, the form $\lambda x.M$ is called a **λ-abstraction** and the form $M\ N$ is called **application**. We write Λ for the set of all terms of PCF.*

Except for wrong, you have seen examples of all the built-in constants already. We will use wrong to signal that something has gone wrong, in particular that some kind of type-related error has occurred. Some examples of terms are:

$$x \qquad \mathsf{ifz} \qquad \mathsf{Z} \qquad (\lambda x.(\mathsf{S}\ (\mathsf{S}\ x))) \qquad (\mathsf{pred}\ (\mathsf{S}\ \mathsf{wrong})) \qquad (\lambda f.(\lambda x.(f\ (f\ x))))$$

However, the following are not a terms: λ (only allowed as part of a λ-abstraction), $(\lambda(\mathsf{pred}\ x).(\mathsf{S}\ x))$ (only a variable is allowed between the λ and the . in an abstraction).

Definition 3.2. *A **subterm** of a term M is any substring of M that it itself a term, except for the variable that occurs in an abstraction between the λ and the dot if it does not occur outside of this position. This always includes M itself.*

For example, $(\lambda x.(x\,y))$ has x as a subterm, but $(\lambda x.\,y)$ does not have x as a subterm. The subterms of $(\lambda x.(((\mathsf{ifz}\ x)\ \mathsf{Z})(\mathsf{S}\ \mathsf{Z})))$ are:

$$(\lambda x.(((\mathsf{ifz}\ x)\ \mathsf{Z})(\mathsf{S}\ \mathsf{Z}))),\quad (((\mathsf{ifz}\ x)\ \mathsf{Z})(\mathsf{S}\ \mathsf{Z})),\quad ((\mathsf{ifz}\ x)\ \mathsf{Z}),\quad (\mathsf{ifz}\ x),\quad \mathsf{ifz},\quad x,\quad \mathsf{Z},\quad (\mathsf{S}\ \mathsf{Z}),\quad \mathsf{S}$$

You may have noticed that we are using rather more parentheses than we did in the informal introduction. According to the grammar we need parentheses around every function application $(M\ N)$, this means that, for example, we need to write the term $\mathsf{ifz}\ x\ y\ (f\ (\mathsf{pred}\ x)\ (\mathsf{S}\ y))$ which you saw earlier, more officially as $(((((\mathsf{ifz}\ x)\ y)\ (f\ (\mathsf{pred}\ x))\ (\mathsf{S}\ y))))$ – ouch!

Syntactic Sugar

We do need all these parentheses in order to specify a simple but unambiguous grammar. For example, without parentheses we wouldn't know whether a triple application $M\ N\ P$ should be parsed as $(M\ N)\ P$ – i.e. supplying P as input to the function returned by $M\ N$ – or as $M\ (N\ P)$ – i.e. supplying $N\ P$ as argument to the function M. However, it is rather inconvenient, so what we shall do is to agree some conventions between us for how to understand expressions when we don't put in quite so many parentheses.

Definition 3.3. *We adopt the following conventions regarding the omission of parentheses and the handling of λ:*

- We will omit any outermost parentheses. *For example, whenever we write $M\,N$ to mean a term, the term that we mean is $(M\,N)$.*
- In any term, we will assume that application associates to the left. *For example, whenever we write $M\,N\,P$, the term that we mean is $((M\,N)\,P)$.*
- In any term, we will assume that the body of an abstraction extends as far to the right as possible. *For example, when we write $\lambda x.M\,N$, the term that we mean is $(\lambda x.(M\,N))$.*
- In any term, iterated abstractions can be grouped. *For example, when we write $\lambda x\,y.M$, the term that we mean is $(\lambda x.(\lambda y.M))$*

It is still perfectly ok to write a term with all the parentheses, e.g. $((x\ y)\ z)$, but generally we will prefer to use as few parentheses as possible because it tends to be more readable. One exception to this is that we will always wrap a λ-abstraction in parentheses when it is an argument to some other function, for example, according to the agreed conventions we *can* write $(\lambda x.x)\ \lambda y.\mathsf{Z}$ to mean $((\lambda x.x)\ (\lambda y.\mathsf{Z}))$, but in practice I think everyone finds $(\lambda x.x)\ (\lambda y.\mathsf{Z})$ to be more readable.

While we are making life a bit more bearable, let us also agree some notation for numbers.

Definition 3.4. *For each natural number n, the PCF* numeral $\underline{n}$ *is the term defined recursively as follows:*

$$\underline{0} = \mathsf{Z}$$
$$\underline{n+1} = \mathsf{S}\,\underline{n}$$

For example, the numeral for three can be computed as follows:

$$\underline{3} = \underline{0+1+1+1} = \mathsf{S}\,\underline{0+1+1} = \mathsf{S}\,(\mathsf{S}\,\underline{0+1}) = \mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,\underline{0})) = \mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,\mathsf{Z}))$$

Now we can write some quite readable programs and know how to parse them:

What we write	What we mean
$x\ y\ (\mathsf{pred}\ y)$	$((x\ y)\ (\mathsf{pred}\ y))$
$\mathsf{ifz}\ x\ \underline{2}\ \underline{3}$	$(((\mathsf{ifz}\ x)\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})))\ (\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))))$
$\lambda f\, g x.f\ (g\ x)$	$(\lambda f.(\lambda g.(\lambda x.(f\ (g\ x)))))$
$\lambda x.x\ (\lambda y.y)$	$(\lambda x.(x\ (\lambda y.y)))$

3.3 Name Binding

Our intuition is that an abstraction $\lambda x.M$ is a function with formal parameter x that returns M. Intuitively, we think of the scope of x as being limited to M, so that if we see this term within something larger:

$$(\lambda x.M)\ \mathsf{Z}\ (\mathsf{pred}\ x)$$

Then we expect the x occurring on the right to be totally unconnected to any x occurring in M. In particular, when we compute with this term, we are supplying Z as input to $\lambda x.M$, so every x in M will be replaced by Z, but not the x on the right:

$$(M[\mathsf{Z}/x])\ (\mathsf{pred}\ x)$$

Recall that our notation $M[N/x]$, which is only informal for now, is meant to mean "the term M but with every occurrence of x replaced by N".

In fact, in this larger term, the x does not appear to be connected to anything – there is no outer function that it is the parameter of. We are going to say that any x in M is *bound* by the λx and, conversely, that the x on the right, which is not bound by any lambda, is *free*.

Definition 3.5. *The set of **free variables** of a term M is defined by recursion on the structure of M:*

$$\begin{aligned} \mathsf{FV}(x) &= \{x\} \\ \mathsf{FV}(c) &= \emptyset \\ \mathsf{FV}(PQ) &= \mathsf{FV}(P) \cup \mathsf{FV}(Q) \\ \mathsf{FV}(\lambda x.N) &= \mathsf{FV}(N) \setminus \{x\} \end{aligned}$$

*A variable that occurs in a term and which is not free is said to be **bound**. A term M without free variables is said to be **closed** or a **combinator**. The set of all closed terms is written Λ^0.*

By contrast, a term M for which $\mathsf{FV}(M) \neq \emptyset$ is said to be *open*. The term $(\lambda x.(\lambda y.(yx))$ is closed, because:

$$\begin{aligned} \mathsf{FV}(\lambda x.(\lambda y.(yx))) &= \mathsf{FV}(\lambda y.(yx)) \setminus \{x\} \\ &= \mathsf{FV}(yx) \setminus \{x, y\} \\ &= (\mathsf{FV}(x) \cup \mathsf{FV}(y)) \setminus \{x, y\} \\ &= (\{x\} \cup \{y\}) \setminus \{x, y\} \\ &= \emptyset \end{aligned}$$

We can think of an open term as being incomplete. Contrast x with $(\lambda x.x)$. We think of $(\lambda x.x)$ as being a function waiting for an input which, when supplied, will return it. Without more information we have no idea what the term x is supposed to do.

Now, since the scope of x in $\lambda x.M$ is completely delimited (i.e. we know that all the x bound by λx occur inside M), it doesn't really matter how we choose to name this formal parameter. For example, we could use y instead: $\lambda y.M[y/x]$ (as long as we rename x to y consistently everywhere inside M); $\lambda x.x$ is not different from $\lambda y.y$ in any essential way. This act of renaming is called a change of bound variable names, and it can be defined more precisely as follows.

Definition 3.6. *Suppose $\lambda x.P$ is a term and $y \notin \mathsf{FV}(P)$. Then the act of replacing $\lambda x.P$ by $\lambda y.P[y/x]$ is called a **change of bound variable name**. If two terms M and N can be made identical just by changing bound variable names to fresh names (not already used inside them), we say they are **α-equivalent**.*

Intuitively, the terms $\lambda x.xy$ and $\lambda z.zy$ are α-equivalent because, if I pick a fresh variable y' to use instead of x in the body of the first term and use the same y' instead of z in the second term, the terms become equal: $\lambda y'.y'y$.

Most of the time it will be obvious when two terms are α-equivalent, such as the case with $\lambda x.x$ and $\lambda y.y$, or $\lambda xy.x$ and $\lambda yx.y$. However, it is absolutely crucial that we only allow renaming of *bound* variables in the definition. The term $\lambda x.xy$ is *not* α-equivalent with $\lambda x.xz$ (assuming $y \neq z$).

You should think of free variables as being part of the explicit interface to the term. For example, consider the closed term $\lambda yz.(\lambda x.xy)(zy)$. The body of this abstraction consists of two subterms $\lambda x.xy$ and zy, with the former applied to the latter. We can think of these two subterms interacting via the variable y, which is free in both of them (but bound in the term of which they are both a part). On the other hand, a variable that is bound in one of these subterms, like x in the first, cannot possibly interact with the other because its whole lifetime exists within the abstraction, which is completely enclosed by that subterm.

Now, it is often the case that we would really like to do a quick change of bound variable names in order to make life easier for ourselves. This happens when computing with terms and when doing hypothetical reasoning about them. In fact, because it happens so frequently, and because α-equivalent terms are essentially the same anyway, we will just build α-equivalence into our definition of identity for terms:

> *From now on we will consider α-equivalent terms to be identical.*

The force of this statement is that, in the rest of this unit, we will not distinguish between e.g. $\lambda x.x$ and $\lambda y.y$, as far as we are concerned *they are the same string*. The situation is analogous to what you are used to with fractions and ratios. There are many different ways of writing down the same ratio $\frac{1}{2}$, $\frac{2}{4}$, $\frac{3}{6}$ and so on, but we consider them to be identical. And, as with fractions, we will feel free to choose the most convenient representative at any given moment – for fractions this is usually the fraction that is in reduced form, where the numerator and denominator have no common factor – for terms this is usually the term in which all bound variables have different names to all the free variables.

Moreover, in proofs, if we have to reason about an arbitrary term M, we can assume that it is in this most convenient form (just as when we reason about an arbitrary fraction $\frac{m}{n}$ we can assume that m and n have no common factors). This is so helpful it gets a name:

The Variable Convention. If $M_1, \ldots, M_k$ occur within the same scope, then in these terms you may assume that all bound variables are chosen to be different from any others.

You will appreciate this convention more fully once we have done some proofs where it is useful.

4. Syntax: Redexes, One-Step Reduction and Substitution

In this lecture we want to make precise the semantics of PCF, that is, how to compute with terms. Since PCF is a pure functional language, the most natural way to specify how it computes is by rewriting, that is a term M makes a computation step by replacing some subterm by a "rewritten" one to give N, and we will write this step as $M \rhd N$.

4.1 One-Step Reduction

Now we say what each basic kind of computational step looks like. To describe the behaviour of the built-in functions, we need to distinguish based on the shape of their input. For example, the predecessor function behaves differently whether it is given Z or a term starting with the successor $\mathsf{S}\ N$ or if it is given something like $(\lambda x.\, x)$ that does not look like a numeral at all. To this end, let us first describe the important shapes that data can take:

Definition 4.1. *The* values *are those terms V belonging to any of the following sets:*

- $\mathsf{Val}_{\mathsf{nat}} = \{\mathsf{Z}\} \cup \{\mathsf{S}\ M \mid M \in \Lambda\}$
- $\mathsf{Val}_{\mathsf{fun}} = \{\lambda x.\, M \mid M \in \Lambda\} \cup \{\mathsf{pred}, \mathsf{S}, \mathsf{fix}, \mathsf{ifz}\} \cup \{\mathsf{ifz}\ M \mid M \in \Lambda\} \cup \{\mathsf{ifz}\ M\ N \mid M, N \in \Lambda\}$
- $\mathsf{Val}_{\mathsf{wrong}} = \{\mathsf{wrong}\}$

Intuitively, the values are those terms where the "kind" of term they are, numerals, functions or wrong, is clear from their shape. Then the built-in functions can be described completely in terms of their behaviour on these values. We do this by listing all the possible patterns of function applications that can possibly make a step, the *redexes*, and next to each we list its *contraction*, which is result of making the step.

Definition 4.2. *We define the following redex/contraction pairs $M \ / \ N$. Any instance of the term on the left of the / is called a* ***redex*** *and the term on the right its* ***contraction****.*

$$\mathsf{pred}\ \mathsf{Z} \ /\ \mathsf{Z} \tag{4.1}$$
$$\mathsf{pred}\ (\mathsf{S}\ N) \ /\ N \tag{4.2}$$
$$\mathsf{ifz}\ \mathsf{Z}\ N\ P \ /\ N \tag{4.3}$$
$$\mathsf{ifz}\ (\mathsf{S}\ M)\ N\ P \ /\ P \tag{4.4}$$
$$(\lambda x.M)\ N \ /\ M[N/x] \tag{4.5}$$
$$\mathsf{fix}\ M \ /\ M\ (\mathsf{fix}\ M) \tag{4.6}$$
$$\mathsf{S}\ V \ /\ \mathsf{wrong} \quad \textit{if } V \notin \mathsf{Val}_{\mathsf{nat}} \tag{4.7}$$
$$\mathsf{pred}\ V \ /\ \mathsf{wrong} \quad \textit{if } V \notin \mathsf{Val}_{\mathsf{nat}} \tag{4.8}$$
$$V\ N \ /\ \mathsf{wrong} \quad \textit{if } V \notin \mathsf{Val}_{\mathsf{fun}} \tag{4.9}$$
$$\mathsf{ifz}\ V\ N\ P \ /\ \mathsf{wrong} \quad \textit{if } V \notin \mathsf{Val}_{\mathsf{nat}} \tag{4.10}$$

The terminology *redex* is a portmanteau of "reducible expression" and this kind of computation is sometimes called "reduction" because, in some sense, the expression (term) somehow gets simple – it reduces in complexity – as a result of performing a step. However, one has to take this with a pinch of salt because there are lots of occasions in which performing a step of reduction actually creates a term which seems much *more* complex that the original!

The above listing of redexes and their contractions tells us what is involved in computing with terms that are themselves redexes, but, more generally, we want to compute with terms that merely contain redexes, e.g.

$$\mathsf{ifz}\ (\mathsf{pred}\ \underline{2})\ M\ N$$

This term is not itself a redex – it cannot make a step at the top-level – but it contains a redex, namely $\mathsf{pred}\ \underline{2}$. If we were to contract this redex to give $\underline{1}$, we would then be in a position to compute the result of the conditional, because $\mathsf{ifz}\ \underline{1}\ M\ N$ is a redex (it's contraction is N).

The way we allow or disallow computational steps depending on where redexes occur in a given term gives rise to different kinds of reduction – *call-by-name*, *call-by-value*, *weak-head reduction* and so on. In this unit we take what is probably the most straightforward-to-use of these choices, the notion that is sometimes called *full β-reduction*. In full β-reduction, a term M makes a computational step by *contracting exactly one redex anywhere inside M*.

Definition 4.3 (Preliminary)**.** *If N is obtained from M by contracting exactly one redex anywhere inside M, then we write $M \rhd N$ and say that M has made **one reduction step** to N.*

We have marked this a preliminary definition because I am, at the moment, just going to rely on your intuitive understanding of strings to make sense of the phrase "anywhere inside". This will be perfectly fine for using this definition to actual perform computations, but we will want to be a bit more concrete about what forms "exactly one redex anywhere inside" can take when we want to reason about arbitrary computation steps later on.

For example, $\mathsf{ifz}\ (\mathsf{pred}\ \underline{2})\ M\ N \rhd \mathsf{ifz}\ \underline{1}\ M\ N$ by contracting the redex $\mathsf{pred}\ \underline{2}$. Some more examples:

- $(\lambda x.x)\ \underline{2} \rhd \underline{2}$
- $\mathsf{ifz}\ ((\lambda x.x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd \mathsf{ifz}\ \underline{2}\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y)$
- $\mathsf{ifz}\ ((\lambda x.x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd \mathsf{ifz}\ ((\lambda x.x)\ \underline{2})\ \underline{1}\ (\mathsf{S}\ y)$
- $(\lambda xy.x)\ (\lambda z.z)\ \underline{1} \rhd (\lambda y.(\lambda z.z))\ \underline{1}$

Note that there can be several possible reduction steps that we can perform, depending on which exactly one redex we choose (as in the second and third bullets). We do *not* have any of the following:

- $(\lambda x.\, x)\ \underline{2} \rhd (\lambda x.\, x)\ \underline{2}$ — requires contracting zero redexes.
- $\mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \not\rhd \mathsf{ifz}\ \underline{2}\ (\underline{1})\ (\mathsf{S}\ y)$ — requires contracting two redexes at once.
- $\mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd \mathsf{pred}\ \underline{2}$ — requires contracting two redexes in sequence.
- $(\lambda x y.\, x)\ (\lambda z.\, z)\ \underline{1} \rhd (\lambda x y.\, x)\underline{1}$ — the subterm $((\lambda z.\, z)\ \underline{1}$ is a redex, but it does *not* occur inside this term! With the proper parens this term is $((\lambda x y.\, x)\ (\lambda z.\, z))\ \underline{1}$, *not* $(\lambda x y.\, x)\ ((\lambda z.\, z)\ \underline{1})$.

4.2 Substitution

We have seen how to make a step, but so far we rely on a simple understanding of $M[N/x]$ which is that this is the term M with all occurrences of x replaced by N. This simple understanding is perfectly fine for computing $M[N/x]$ (actually performing the substitution) most of the time, but there are a couple of cases where this substitution interacts with binding that require a more sophisticated understanding. *However, these cases can always be avoided by renaming bound variables to ensure that no bound variable name clashes with any other.*

To motivate them, take a look at two computation steps in which, on the right-hand side we have obtained something unexpected by naïevely replacing every x by the actual parameter.

(1)

$$(\lambda x.\, (\lambda x.\, x)\ \underline{1})\ \underline{0} \rhd (\lambda x.\, \underline{0})\ \underline{1} \rhd \underline{0}$$

(2)

$$\begin{aligned}(\lambda x.\, (\lambda y x.\, y\ (\mathsf{S}\ x)\ (\mathsf{pred}\ x))\ (\mathsf{ifz}\ x))\ \underline{0} &\rhd (\lambda x.\, (\lambda x.\, (\mathsf{ifz}\ x)\ (\mathsf{S}\ x)\ (\mathsf{pred}\ x)))\ \underline{0} \\ &\rhd \lambda x.\, \mathsf{ifz}\ x\ (\mathsf{S}\ x)\ (\mathsf{pred}\ x)\end{aligned}$$

In (1), $(\lambda x.\, (\lambda x.\, x)\ \underline{1})\ \underline{0}$ should evaluate to $\underline{1}$. You can see this quite clearly if we rename the inner bound variable x, which we can do at any time, to obtain $(\lambda x.\, (\lambda y.\, y)\ \underline{1})\ \underline{0}$. The problem here is that we tried to replace a *bound* variable. In our definition below, we will only allow for replacing free variables. Note that when we make a step $(\lambda x.\, M)\ N \rhd M[N/x]$, any x that is bound by the outer λx in $\lambda x.\, M$ will be free if we just consider M on its own, as we do when we perform the substitution $M[N/x]$ on the right-hand-side. In (2), $(\lambda x.\, (\lambda y x.\, y\ (\mathsf{S}\ x)(\mathsf{pred}\ x))\ (\mathsf{ifz}\ x))\ \underline{0}$ should return the function $\lambda x.\, \mathsf{S}\ x$ that just adds one to its argument. Again, this becomes quite clear if we rename the two bound occurrences of x apart, e.g. $(\lambda z.\, (\lambda y x.\, y\ (\mathsf{S}\ x)(\mathsf{pred}\ x))\ (\mathsf{ifz}\ z))\ \underline{0}$. The problem here is that our actual parameter contained a free variable, x, that *became bound* as a result of being substituted into the body of the function; we say that x was *captured*. In our definition below we will not allow a substitution to occur if it would involve capture (you must perform a renaming first).

The version of substitution that we will define is called "capture-avoiding substitution" because we will outlaw such possibilities by leaving substitution undefined if a case like this would arise.

Definition 4.4. *We define* ***capture-avoiding substitution*** *of term N for variable x in term M, written $M[N/x]$, recursively on the structure of M:*

$$\begin{array}{rcll} c[N/x] & = & c & \\ y[N/x] & = & y & \text{if } x \neq y \\ y[N/x] & = & N & \text{if } x = y \\ (PQ)[N/x] & = & P[N/x]Q[N/x] & \\ (\lambda y.P)[N/x] & = & \lambda y.P & \text{if } y = x \\ (\lambda y.P)[N/x] & = & \lambda y.P[N/x] & \text{if } y \neq x \text{ and } y \notin \mathsf{FV}(N) \end{array}$$

Notice the last two cases. The first says that we only substitute for free occurrences of x: if you attempt to substitute inside a term where x is bound, such as $\lambda x.P$, then there will be no free occurrences of x that can be replaced and so the result of the substitution $(\lambda x.P)[N/x]$ is just $\lambda x.P$. The second condition avoids the possibility of capture, you may only replace a free occurrence of x by N in $\lambda y.P$ if you know that y does not itself occur freely in N. However, note that both of these conditions concern some bound variable y and so one can always avoid any problems by first performing a renaming of bound variables to change the bound variable y in $\lambda y.P$ for some other variable z that we haven't already used to obtain $\lambda z.P[z/y]$.

There is a potential for confusion due to the interaction between the notation for substitution $M[N/x]$ and our conventions for omitting parentheses when writing terms. So let's avoid it by agreeing that $MN[P/x]$ means M applied to $N[P/x]$ and $\lambda y.M[N/x]$ means an abstraction of y with body $M[N/x]$. Here are some examples of substitutions, make sure you follow the notation!

$$\begin{array}{rcl} (\lambda x.yx)[(\lambda z.z)/y] & = & \lambda x.(yx)[\lambda z.z/y] \\ & = & \lambda x.y[\lambda z.z/y]x[\lambda z.z/y] \\ & = & \lambda x.(\lambda z.z)x \\ & & \\ ((\lambda x.yx)x)[y/x] & = & (\lambda x.yx)[y/x]x[y/x] \\ & = & (\lambda x.yx)y \\ & & \\ (x(\lambda z.z)x)[\lambda z.z/x] & = & x[\lambda z.z/x](\lambda z.z)[\lambda z.z/x]x[\lambda z.z/x] \\ & = & (\lambda z.z)(\lambda z.z[\lambda z.z/x])(\lambda z.z) \\ & = & (\lambda z.z)(\lambda z.z)(\lambda z.z) \\ & & \\ (xx)[\underline{2}/y] & = & x[\underline{2}/y]x[\underline{2}/y] \\ & = & xx \end{array}$$

On the other hand $(\lambda x.yx)[\lambda z.xz/y]$ and $(y(\lambda x.x))[zx/y]$ are both undefined (even though, intuitively, the latter is not problematic).

5. Semantics: One-Step Reduction via Contexts, and Simple Programming

5.1 Reduction Step

In the previous lecture we introduced the *one-step reduction*, which is the central notion of computational step in PCF, but we fell short of giving a fully formal definition – we left "anywhere inside M" to your intuitions. However, as we start to look at reduction more closely, it will be helpful to make that a bit more concrete. For that we will use the notion of *one-hole context*.

Definition 5.1. *The **one-hole contexts** are those strings that can be generated by the following grammar:*

$$C[\,] ::= [\,] \mid M\ C[\,] \mid C[\,]\ N \mid \lambda x.\, C[\,]$$

Given a context $C[\,]$, we write $C[M]$ for the term obtained by replacing the hole $[\,]$ by M. We can also compose contexts by filling the hole in one context with another one: we write $C_1[C_2[\,]]$ for the context that arises by replacing the hole in $C_1[\,]$ with the context $C_2[\,]$.

Intuitively, the one-hole contexts are just terms that are missing exactly one subterm somewhere (which shows up as a single occurrence of the hole $[\,]$), so to make a one-hole context, you just need to think of a term and then replace some subterm by the hole. Examples of one-hole contexts are as follows:

$$[\,], \quad \lambda x y z.\, x\ [\,]\ (y\ z), \quad [\,]\ \underline{2}, \quad \mathsf{ifz}\ [\,]\ (\mathsf{S}\ x)\ (\mathsf{pred}\ y), \quad \mathsf{fix}\ (\lambda f\, x.\, \mathsf{S}\ (f\ (\mathsf{pred}\ [\,])))$$

The key operation associated with one-hole contexts is filling the hole. We can fill the hole in some context $C[\,]$ with a *term* N to obtain a *term* $C[N]$. For example, if we fill the hole of $\lambda x y z.\, x\ [\,]\ (y\ z)$ with z we obtain $\lambda x y z.\, x\ z\ (y\ z)$. Note, this operation does not respect binding, so we are happy to let variable capture happen (as in this case). Or, we could fill the same hole with another context, e.g. with $[\,]\ \underline{2}$, to obtain $\lambda x y z.\, x\ ([\,]\ \underline{2})\ (y\ z)$.

We can use one-hole contexts to say "anywhere inside M" as follows.

Definition 5.2. *Then **one-step reduction relation**, written infix by $\rhd$, is defined as $M \rhd N$ just if there is a context $C[\,]$ and a redex/contraction pair P / Q such that $M = C[P]$ and $N = C[Q]$.*

Let's revisit our one-step reductions from earlier and justify them according to this definition:

- $(\lambda x.\, x)\ \underline{2} \rhd \underline{2}$ – here $C[\,] = [\,]$.
- $\mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd \mathsf{ifz}\ \underline{2}\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y)$ – here $C[\,] = \mathsf{ifz}\ [\,]\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y)$.
- $\mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd \mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ \underline{1}\ (\mathsf{S}\ y)$ – here $C[\,] = \mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ [\,]\ (\mathsf{S}\ y)$.
- $(\lambda xy.\, x)\ (\lambda z.\, z)\ \underline{1} \rhd (\lambda y.\, (\lambda z.\, z))\ \underline{1}$ – here $C[\,] = [\,]\ \underline{1}$.

5.2 Simple Programming

Now that we have a clearer idea of how terms compute, we can do some simple programming. I want to start with some very simple programs that will, nevertheless, come in handy to illustrate important aspects of this language later.

These first programs are extremely simple, but quite well known in the theory of λ-calculus, where they are just known simply as combinators. On the left are the combinators and how we shall abbreviate them. On the right an illustrative example of how they behave.

$$\begin{array}{rcll} \underline{\mathsf{id}} & = & \lambda x.\, x & \underline{\mathsf{id}}\ M \rhd M \\ \underline{\mathsf{const}} & = & \lambda xy.\, x & \underline{\mathsf{const}}\ M\ N \rhd (\lambda y.\, M)\ N \rhd M \\ \underline{\mathsf{sub}} & = & \lambda xyz.\, xz(yz) & \underline{\mathsf{sub}}\ M\ N\ P \rhd \cdots \rhd M\ P\ (N\ P) \\ \underline{\mathsf{div}} & = & \mathsf{fix}\ \underline{\mathsf{id}} & \underline{\mathsf{div}} \rhd \underline{\mathsf{id}}\ \underline{\mathsf{div}} \rhd \underline{\mathsf{div}} \end{array}$$

The term $\underline{\mathsf{id}}$ is sometimes called the *identity* combinator, for obvious reasons. The term $\underline{\mathsf{const}}$ is sometimes called the *constant* combinator because, after accepting its first argument, say M, it then behaves like the "constantly M function" $\lambda y.\, M$ (i.e. that returns M no matter what its argument). The term $\underline{\mathsf{sub}}$ is sometimes called the *substitution* combinator because, historically, it has a role in simulating a kind of substitution if you don't have λ-abstraction. For us, it will not be so useful except when we need a handy example of a term that duplicates one of its arguments. Finally, the term $\underline{\mathsf{div}}$ is known as *divergence* because it implements a diverging computation – one that unproductively runs forever.

> Let me point out here that $\underline{\mathsf{id}}$, $\underline{\mathsf{const}}$ etc are *merely abbreviations* (you can think of them as macros) that allow us humans to avoid having to write out enormous terms. That is, we are not adding anything new to the programming language, terms are still only built from either variables, abstractions, applications or the six constants. At any point we can eliminate all such abbreviations by simply replacing them by their definition.
>
> Note: it *does not make any sense* to "define" a circular abbreviation like:
>
> $$\underline{\mathsf{fact}} = \lambda x.\, \mathsf{ifz}\ x\ \underline{1}\ (\underline{\mathsf{mult}}\ x\ (\underline{\mathsf{fact}}\ (\mathsf{pred}\ x))) \qquad \text{Nonsense!}$$
>
> (even assuming that we had defined $\underline{\mathsf{mult}}$). If we allowed something like this, then how could we make sense of a "term" like $\underline{\mathsf{fact}}\ y$? If we replace the "abbreviation" $\underline{\mathsf{fact}}$ by its definition, then we just end up with another "term", $(\lambda x.\, \mathsf{ifz}\ x\ \underline{1}\ (\underline{\mathsf{mult}}\ x\ (\underline{\mathsf{fact}}\ (\mathsf{pred}\ x))))\ y$, involving $\underline{\mathsf{fact}}$. We

could replace that occurrence by the definition of $\underline{\mathsf{fact}}$, but all this would do is create another one and there will never be an end to it. We will see in two lectures time that we can, in fact, define the factorial function and we will discuss recursive definition in general, and it will become clear then exactly what is going on.

Our version of PCF doesn't have tuple datatypes, but we can define them as a kind of abbreviation. We are going to use functions to represent tuples, which seems a bit strange at first but it works out quite nicely. We will use a technique called "Church Encoding", named after the inventor of λ-calculus, Alonzo Church.

Let's start by discussing pairs. Recall that a pair (M, N) packages up two data items, which for us will be represented by terms, in this case M and N. Programming languages with pairs have a way to construct the pair (M, N), given the items M and N, and they have a way to deconstruct the pair, e.g. projections fst and snd that behave like:

$$\mathsf{fst}\,(M, N) \rhd \cdots \rhd M \qquad \mathsf{snd}\,(M, N) \rhd \cdots \rhd N$$

In other words, there is a series of reduction steps (computation steps) that take us from $\mathsf{fst}\,(M\,N)$ to arrive at M, and similarly for the second projection.

In the Church encoding, a pair (M, N) will be represented by a function $\lambda p.\, p\, M\, N$. If you think about it, then, assuming M and N are in normal form (which we expect if they are meant to be data items), then $\lambda p.\, p\ M\ N$ is the simplest way to construct a closed term in normal form that involves M and N. The idea is that, if we supply this function $\lambda p.\, p\ M\ N$ with $\lambda xy.\, x$ as input then the result of that computation will be to extract M and, if we supply this function with $\lambda xy.\, y$ as input, then the result will be to project out N:

$$(\lambda p.\, p\ M\ N)\,(\lambda xy.\, x) \rhd (\lambda xy.\, x)\ M\ N \rhd (\lambda y.\, M)\ N \rhd M$$
$$(\lambda p.\, p\ M\ N)\,(\lambda xy.\, y) \rhd (\lambda xy.\, y)\ M\ N \rhd (\lambda y.\, y)\ N \rhd N$$

So all we need to do is define the projections fst and snd so that they take a pair (encoded as a function) as input and pass to it $\lambda xy.\, x$ or $\lambda xy.\, y$ respectively:

$$\underline{\mathsf{fst}} = \lambda p.\, p\ (\lambda xy.\, x) \qquad \underline{\mathsf{snd}} = \lambda p.\, p\ (\lambda xy.\, y)$$

To actually make a pair (M, N), we just need a function that takes M and N and returns the encoding $\lambda p.\, p\ M\ N$:

$$\underline{\mathsf{pair}} = \lambda mnp.\, p\ m\ n$$

Then we can check they everything works correctly by computing:

$$\underline{\mathsf{fst}}\ (\underline{\mathsf{pair}}\ M\ N) \rhd \cdots \rhd \underline{\mathsf{fst}}\ (\lambda p.\, p\ M\ N) \rhd \cdots \rhd M$$
$$\underline{\mathsf{snd}}\ (\underline{\mathsf{pair}}\ M\ N) \rhd \cdots \rhd \underline{\mathsf{snd}}\ (\lambda p.\, p\ M\ N) \rhd \cdots \rhd N$$

So, indeed our encoding satisfies our specification of pairing in programming languages.

Let's extend this idea to tuples of every arity. We'll call the projection that extracts the i^{th} component of a k-ary tuple more generically $\underline{\mathsf{proj}}^k_i$ and we'll call the function that constructs a k-ary tuple $\underline{\mathsf{tuple}}^k$. So, for example, our $\underline{\mathsf{fst}}$ arises as $\underline{\mathsf{proj}}^2_1$, $\underline{\mathsf{snd}}$ as $\underline{\mathsf{proj}}^2_2$ and $\underline{\mathsf{pair}}$ as $\underline{\mathsf{tuple}}^2$.

Definition 5.3. *For each natural number k and i, define k-ary pairing and projection functions as follows:*

$$\underline{\mathsf{proj}}_i^k = \lambda p.\, p\ (\lambda x_1 x_2 \ldots x_k.\, x_i)$$
$$\underline{\mathsf{tuple}}^k = \lambda x_1 x_2 \ldots x_k p.\, p\ x_1\ x_2 \cdots x_k$$

For ease of reading, lets agree to write each term of the form $\lambda p.\, p\ M_1 \cdots M_k$ rather as $(M_1, \ldots M_k)$.

6. Semantics: Reduction Sequences and Confluence

In this lecture we are going to look in more detail at reduction. In particular, we will see that reduction is *confluent*, which we can think of as a kind of eventual consistency. We will use these new tools in order to prove that certain programs are impossible to write.

6.1 Reduction

In the previous lecture we gave examples of reduction sequences, which were chains of steps one after another:

$$M_1 \rhd M_2 \rhd \cdots \rhd M_{n-1} \rhd M_n$$

Now we will formalise the idea that there exists such a reduction sequence leading from one term to another. In fact, a smooth way to do this to introduce, for each natural number n, a relation $M \rhd^k N$ to capture the idea that there exists a reduction sequence leading from M to N that comprises exactly k steps.

Definition 6.1. *Define a family of* n***-step reduction relations****, indexed by a natural number* n *and written* $P \rhd^n Q$*, recursively as follows:*

- $P \rhd^0 Q$ *just if* $P = Q$*.*
- $P \rhd^{k+1} Q$ *just if there is some* U *such that* $P \rhd^k U$ *and* $U \rhd Q$*.*

The ***reduces-to*** *relation, written infix* $\rhd^*$*, is defined by* $M \rhd^* N$ *just if there is some* n *such that* $M \rhd^n N$*. We can also write* $M \rhd^+ N$ *to signify that* $M \rhd^n N$ *with* $n > 0$*.*

This definition captures our aim for the notation $M \rhd^k N$ because if you unfold it for any specific k you will find it requires a k-step reduction sequence. For example, with $k = 3$:

$$\begin{aligned}
& M \rhd^3 N \\
\text{just if there is some } U_2 \text{ such that } \quad & M \rhd^2 U_2 \rhd N \\
\text{just if there is some } U_2, U_1 \text{ such that } \quad & M \rhd^1 U_1 \rhd U_2 \rhd N \\
\text{just if there is some } U_2, U_1, U_0 \text{ such that } \quad & M \rhd^0 U_0 \rhd U_1 \rhd U_2 \rhd N \\
\text{just if there is some } U_2, U_1 \text{ such that } \quad & M \rhd U_1 \rhd U_2 \rhd N
\end{aligned}$$

So, for most purposes, one can think of the notation $M \rhd^* N$ just to mean "there is a reduction sequence that leads from M to N". Note, this does not imply anything about N being terminal in any way – there may still be reduction steps possible from N. The first few non-examples of one-step reductions from the last lecture are all examples of many-step reduction:

- $\underline{\mathsf{id}}\ \underline{2} \rhd^* \underline{\mathsf{id}}\ \underline{2}$
  – because there is a 0-step sequence: $\underline{\mathsf{id}}\ \underline{2} \rhd^0 \underline{\mathsf{id}}\ \underline{2}$.
- $\mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd^* \mathsf{ifz}\ \underline{2}\ \underline{1}\ (\mathsf{S}\ y)$
  – because there is e.g. a 2-step sequence: $\mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd^2 \mathsf{ifz}\ \underline{2}\ \underline{1}\ (\mathsf{S}\ y)$.
- $\mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd^* \mathsf{pred}\ \underline{2}$
  – because there is e.g. a 2-step sequence: $\mathsf{ifz}\ ((\lambda x.\, x)\ \underline{2})\ (\mathsf{pred}\ \underline{2})\ (\mathsf{S}\ y) \rhd^2 \mathsf{pred}\ \underline{2}$.

The following terminology is not specific to PCF, but you will find it associated with any kind of computation by rewriting/reduction. It is very helpful for giving us a means to discuss reduction concisely.

Definition 6.2. *Some terminology:*

- *If $M \rhd^* N$ then the term N is said to be a **reduct** of M.*
- *If $M \rhd^+ N$ then the term N is said to be a **proper reduct** of M.*
- *A term M without proper reduct, i.e. for which there is no N such that $M \rhd N$, is a **normal form**.*
- *A term M that can reduce to normal form, i.e. for which there is a normal form N such that $M \rhd^* N$ is said to **have a normal form** or be **normalisable**.*
- *A term M that has no infinite reduction sequences is said to be **strongly normalisable**.*

Note: every term in normal form has a normal form and is strongly normalising. Note also that it is possible to have a normal form and to have infinite reduction sequences at the same time. Some examples to illustrate:

- The reducts of $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$ are:
  (i) $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$
  (ii) $(\lambda x.xy)(\lambda x.xy)$
  (iii) $(\lambda zx.xy)yy$
  (iv) $(\lambda x.xy)y$
- Except (i), all those reducts are proper.
- Every numeral, e.g. $\underline{2}$, is in normal form and, therefore, also has a normal form and is strongly normalising.
- The term $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$ has a normal form, namely yy. Since there are no infinite reduction sequences starting from this term, it is strongly normalising.

- The term $\underline{\mathsf{const}}\ \underline{1}\underline{\mathsf{div}}$ has a normal form, namely $\underline{1}$, which can be obtained by, e.g.:

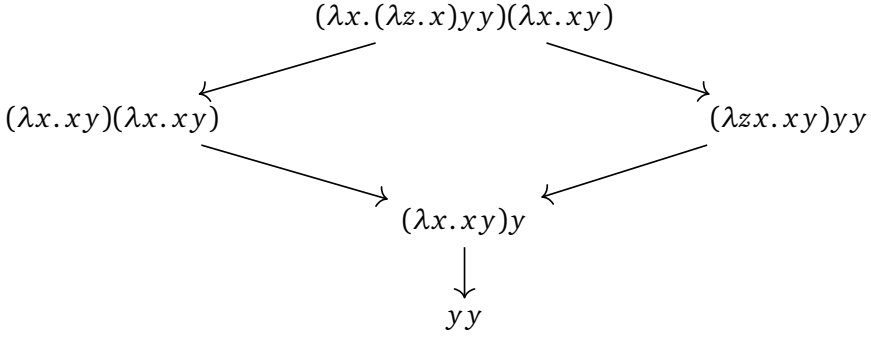
$$\underline{\mathsf{const}}\ \underline{1}\ \underline{\mathsf{div}} \rhd (\lambda y.\underline{1})\ \underline{\mathsf{div}} \rhd \underline{1}$$

  However, it also has an infinite reduction sequence, e.g.:

$$\underline{\mathsf{const}}\ \underline{1}\ \underline{\mathsf{div}} \rhd \underline{\mathsf{const}}\ \underline{1}\ (\underline{\mathsf{id}}\ \underline{\mathsf{div}}) \rhd \underline{\mathsf{const}}\ \underline{1}\ (\underline{\mathsf{id}}\ (\underline{\mathsf{id}}\ \underline{\mathsf{div}})) \rhd \underline{\mathsf{const}}\ \underline{1}\ (\underline{\mathsf{id}}\ (\underline{\mathsf{id}}\ (\underline{\mathsf{id}}\ \underline{\mathsf{div}}))) \rhd \cdots$$

6.2 Confluence

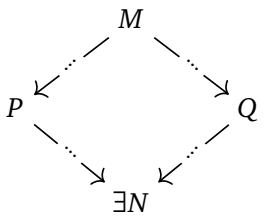
There is a sense in which one-step reduction is *non*deterministic: as we saw in the earlier examples, for certain terms there is more than one possible reduct. This can be pictured very nicely if we draw the reduction graph of a term. The reduction graph of a term P is just a rooted directed graph. The vertices of the graph are P and all of the terms that can be reached by making one-step reductions from P, one-step reductions from those terms and so on. There is an edge between two of these vertices M and N just if $M \rhd N$. For example, the reduction graph of $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$:



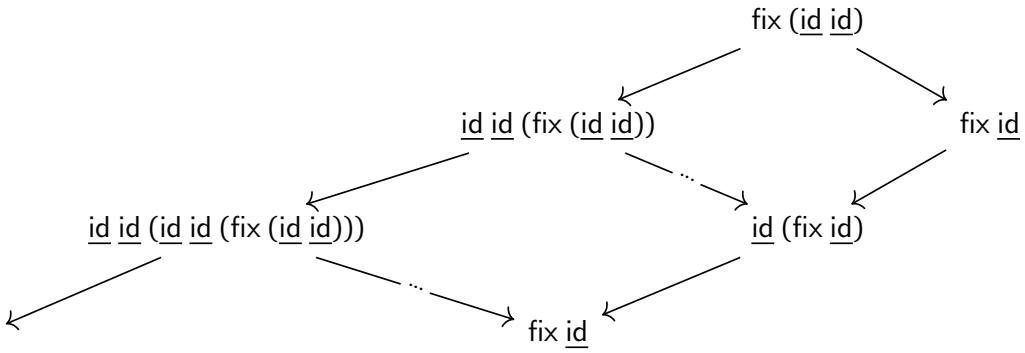
However, in this example, we can see that, although there are several possible ways to reduce $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$, they all eventually lead to yy. This is actually an important property of reduction, called *confluence*. We will state the theorem now and you will be expected to make use it from now on (i.e. you may assume it), but we will defer the proof until later.

Theorem 6.1 (Confluence)**.** *If $M \rhd^* P$ and $M \rhd^* Q$ then there exists a term N (not necessarily a normal form) such that $P \rhd^* N$ and $Q \rhd^* N$.*

This can be nicely illustrated by the following diagram, in which we use dots to indicate that more than one step may be involved:



It is a bit more nuanced than simply "if there are two different ways to reduce a term then they will always reach the same normal form" because it applies even when the term in question fails to have a normal form. Consider e.g. fix (id id). An illustration:



Use two fingers to trace out any two paths on this reduction graph for a while. Wherever each of your fingers got to, there is guaranteed to be a way to get to a common reduct.

There are many helpful consequences of confluence, but one particularly important one is that normal forms are unique when they exist.

Theorem 6.2 (Unique Normal Forms)**.** *If $M \rhd^* N$ and $M \rhd^* P$ with N and P normal forms, then $N = P$.*

Proof. Suppose $M \rhd^* N$ and $M \rhd^* P$. Then, by confluence, there is a common reduct Q, i.e. $N \rhd^* Q$ and $P \rhd^* Q$. However, N is a normal form, so it must be that $N \rhd^0 Q$, and P is a normal form, so $P \rhd^0 Q$. Hence, by definition, $N = Q = P$. □

Using the knowledge that terms have unique normal forms will actually allow us to exclude a variety of functions.

Theorem 6.3. *There is no PCF term* Halts *such that, for all M:*

$$\mathsf{Halts}\ M \rhd^* \begin{cases} \underline{1} & \text{if } M \text{ has a normal form} \\ \underline{0} & \text{otherwise} \end{cases}$$

Proof. We are trying to prove a statement of the form "not (there exists a term...)" so we assume there exists a term Halts that satisfies the given specification, i.e. we assume:

(i) for all M: $\mathsf{Halts}\ M \rhd^* \underline{1}$ if M has a normal form

(ii) for all M: $\mathsf{Halts}\ M \rhd^* \underline{0}$ if M does not have a normal form

Then we aim to find a contradiction. To that end, let us consider the behaviour of the program $(\lambda x.\, \mathsf{Halts}\ x)\ \underline{\mathsf{div}}$. Since x is in normal form, it follows from (i) that $\mathsf{Halts}\ x \rhd^* \underline{1}$, and thus:

$$(\lambda x.\, \mathsf{Halts}\ x)\ \underline{\mathsf{div}} \rhd^* (\lambda x.\, \underline{1})\ \underline{\mathsf{div}} \rhd \underline{1}$$

However, since $\underline{\mathsf{div}}$ does not have a normal form, it follows from (ii) that $\mathsf{Halts}\ \underline{\mathsf{div}} \rhd^* \underline{0}$, and thus:

$$(\lambda x.\, \mathsf{Halts}\ x)\ \underline{\mathsf{div}} \rhd \mathsf{Halts}\ \underline{\mathsf{div}} \rhd^* \underline{0}$$

By the unique normal forms theorem, we obtain that $\underline{0} = \underline{1}$ which is clearly absurd (they are not identical strings). □

7. Semantics: Conversion and Recursion

We usually use the term recursive function for those whose definition is given in terms of itself. The classic example is the factorial function on natural numbers:

$$\begin{array}{rcl} \mathsf{fact} & : & \mathbb{N} \to \mathbb{N} \\ \mathsf{fact}(n) & = & \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot \mathsf{fact}(n-1) \end{array}$$

The presence of the function's name, fact, on both sides of the $=$ is the tell-tale sign that this is a recursive function. So, it seems that names are important for giving recursive function definitions. In λ-calculus there is no formal support for attaching names to functions — no syntax for naming that is built into the notion of reduction or conversion. So a question arises: can we even define recursive functions in the λ-calculus and, if not, how can we perform repetitive computations?

7.1 Recursive functions

To answer this question we have to understand more fully what we mean when we say that the above equation is a definition the factorial function. As computer scientists, and especially as functional programmers, we are a bit spoilt by the very liberal notion of function definition given in languages like Haskell. In Haskell every equation of the form $f\,x = e$ (for some Haskell expression e), assuming it type-checks, is a valid function definition. Consequently, we may be led to believe that, in general, if we write a name f followed by a formal parameter x and then $=$ followed by some expression e, we have defined a function.

However, this is not the case. Consider the following "definition" of a function yuck over the integers:

$$\begin{array}{rcl} \mathsf{yuck} & : & \mathbb{Z} \to \mathbb{Z} \\ \mathsf{yuck}(n) & = & \mathsf{yuck}(n) + 1 \end{array}$$

Is this a valid function definition? The answer is no: we can prove it. Assume, for the purposes of obtaining a contradiction, that there is a function yuck on integers with the property that the following integer equation is true: $\mathsf{yuck}(n) = \mathsf{yuck}(n) + 1$. Then we can subtract $\mathsf{yuck}(n)$ from both sides, to obtain the true equation $0 = 1$, which is absurd.

To understand this, we have to be clear on what we mean by the word function. Intuitively, we think of a function f as some black box which associates to each input x an output $f(x)$. This intuition is made precise through the *definition* of function in Zermelo Fraenkel (ZF) set theory, and this is usually the meaning of the word function in mathematics more generally. In ZF, a function f from a set A to a set B is a subset of $A \times B$ in which each element of A is associated with exactly one element of B.

According to this definition, a function is just a tabulation of its inputs and outputs, one can say that the notion of function and the notion of *graph of a function* are identified. So, strictly speaking, the factorial function is a certain set of pairs, starting off like:

$$\begin{array}{rcl} 0 & \mapsto & 1 \\ 1 & \mapsto & 1 \\ 2 & \mapsto & 2 \\ 3 & \mapsto & 6 \\ & \vdots & \end{array}$$

Since there is exactly one output for each input, the sentence "the output of fact corresponding to input 3" unambiguously refers to exactly one natural number, 6. However, it is a bit long-winded to say, so in 1734 Leonard Euler invented a more concise notation: fact(3).

Unfortunately, the department's budget does not stretch far enough to allow me to purchase enough paper to give a complete listing of the factorial function. So we have to look for some finite means by which we can *specify* this infinite set, unambiguously. This is purpose of the equation. The equation constitutes a *definition* of factorial in the following sense.

Definition 7.1. *We define* fact *as the unique function on natural numbers (i.e. subset of* $\mathbb{N} \times \mathbb{N}$ *with the property that exactly one output number is associated with each input number) that satisfies the equation (in parameter* F*):*

$$F(n) = \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot F(n-1)$$

So, in a sense, Definition 7.1 is the "official" definition of the factorial function, but, in practice we omit the parameter F and instead write the equation directly using fact, as we did at the start of this chapter. It should remind you of the way that we make inductive definitions of sets.

Of course, if we want to make such a definition then there is some obligation on us to be sure that the equation has a unique solution. However, we know a few syntactic tricks that make it often straightforward to tell. For example, if we do write the equation in the form $f(x) = e$ *and* there is a well-founded order on the domain *and* we only make recursive calls in e on strictly smaller values, then we can be sure that this equation specifies a genuine function.

We can now see what is the key mistake in our "definition" of yuck: there are *no* functions over integers that satisfy the equation (in parameter F):

$$F(n) = F(n) + 1$$

No wonder that by assuming that one exists, named yuck, we are led inevitably to a contradiction!

Before we leave this excursion into the world of recursive definitions outside of the λ-calculus, let me return briefly to Haskell. The equation:

$$\begin{array}{rcl} \mathsf{yuck} & :: & \mathsf{Int} \to \mathsf{Int} \\ \mathsf{yuck}\ n & = & \mathsf{yuck}\ n + 1 \end{array}$$

has a solution in the space of Haskell programs. There is a Haskell program that satisfies these equations – the program that never terminates on any input!

7.2 Conversion

Now, let's turn to PCF. To understand when a PCF term satisfies an equational specification, we should first say what we mean by an equation between PCF programs.

When you learned about functional programming, you probably did some short proofs using equational reasoning, like:

$$\begin{aligned}\mathsf{map}\ (+1)\ [3,8,6] &= 4 : \mathsf{map}\ (+1)\ [8,6] \\ &= 4 : 9 : \mathsf{map}\ (+1)\ [6] \\ &= 4 : 9 : 7 : \mathsf{map}\ (+1)\ [\,] \\ &= 4 : 9 : 7 : [\,] \\ &= [4,9,7]\end{aligned}$$

What do we really mean by $\mathsf{map}\ (+1)\ [3,8,6] = 4 : \mathsf{map}\ (+1)\ [8,6]$? When I am doing arithmetic, I know what I mean by $3+1=4$, I mean that the left and right sides of the $=$ are literally the same natural number, they are identical: $3+1$ is a number and it is the very same number has 4. However, that's not what I mean in $\mathsf{map}\ (+1)\ [3,8,6] = 4 : \mathsf{map}\ (+1)\ [8,6]$: it's plain to see that the left and right sides of this equation, which are Haskell expressions (i.e. strings), are *not* identical: the string on the left starts with 'm' whereas the one on the right starts with '4'.

You might complain at that and say that we are not talking about Haskell expressions as strings, but rather Haskell expressions as (compiled) programs, and that the left and right hand sides are identical programs. However, that is clearly untrue also, since the program on the left has more steps to evaluate than the one on the right before reaching a normal form.

What we need is some kind of "equality modulo computation".

We could perhaps make sense of these equations by justifying them by saying $\mathsf{map}\ (+1)\ [3,8,6] = 4 : \mathsf{map}\ (+1)\ [8,6]$ because the left-hand side *evaluates to* the right-hand side. However, this too seems like a dishonest way to interpret an equation, because if

$$\mathsf{map}\ (+1)\ [3,8,6] = 4 : \mathsf{map}\ (+1)\ [8,6]$$

then it should also be that

$$4 : \mathsf{map}\ (+1)\ [8,6] = \mathsf{map}\ (+1)\ [3,8,6]$$

but surely it is not the case that $4 : \mathsf{map}\ (+1)\ [8,6]$ evaluates to $\mathsf{map}\ (+1)\ [3,8,6]$.

A more satisfying approach is to say that two expressions are equal modulo computation when they *each evaluate to the same thing*. We say that they are *convertible*.

Definition 7.2. *We write $M \approx N$ just if there is a term P such that $M \rhd^* P$ and $N \rhd^* P$. In this case we say that they are **convertible**.*

For short, we will sometimes rephrase this as "convertible terms have a common reduct". The following are examples of convertible pairs of terms:

- $\mathsf{ifz}\ \underline{0}\ (\mathsf{pred}\ \underline{3})\ \underline{1} \approx \underline{2}$ – the common reduct is $\underline{2}$
- $\mathsf{ifz}\ x\ (\mathsf{pred}\ \underline{3})\ \underline{1} \approx \mathsf{ifz}\ x\ \underline{2}\ (\mathsf{pred}\ (\mathsf{pred}\ \underline{3}))$ – a common reduct is $\mathsf{ifz}\ x\ \underline{2}\ \underline{2}$.

- $\underline{1}\ \underline{2} \approx \mathsf{pred}\ \underline{\mathsf{id}}$ – the common reduct is wrong.

In general, conversion is weaker than reduction, in the sense that $M \rhd^* N$ implies $M \approx N$, but not necessarily vice-versa.

What is perhaps not so clear from this definition is that conversion behaves anything like an equality, that it is an equivalence. However, using confluence, it is quite straightforward to show. The proof is the subject of this week's exercises.

Lemma 7.1. *Convertibility is a congruence, that is an equivalence relation that is compatible with contexts, i.e. all of the following hold:*

Reflexivity *For all M: $M \approx M$.*

Symmetry *For all M, N: $M \approx N$ implies $N \approx M$.*

Transitivity *For all M, N and P: $M \approx P$ and $P \approx N$ implies $M \approx N$.*

Compatibility *For all M, N and $C[]$: if $M \approx N$ then $C[M] \approx C[N]$.*

7.3 Recursive functions in PCF

Conversion allows us to specify recursive equations in PCF. It makes sense, for example, to ask: *is there a PCF-term M satisfying:*

$$M\,x \approx x(Mx)$$

I hope you agree that, if there is such a term M then, we can think of it as a recursive function. In fact, there is a term satisfying this equation, fix, because $\mathsf{fix}\ x \rhd^* x(\mathsf{fix}\ x)$.

Not all recursive equations have solutions (terms that satisfy them) in PCF. For example, the following equation has no solution for M:

$$(\lambda x.\lambda y.y)M \approx (\lambda x.\lambda yz.y)M$$

How can you argue it? To show that it has *no* solution, we assume it has one, so suppose M is a solution. Then we have $(\lambda x.\lambda y.y)M \approx (\lambda x.\lambda yz.y)M$ is true of this M. By equational reasoning (which makes sense since conversion is an equivalence):

$$\lambda y.y \approx (\lambda x.\lambda y.y)M \approx (\lambda x.\lambda yz.y)M \approx \lambda yz.y$$

Since the term on the left and the term on the right are convertible, by definition there must be a common a reduct Q. But these two terms are not identical, so Q must be a proper reduct of at least one of them. However, they are both in normal form: a contradiction.

Anyway, it is still a meaningful question to ask whether this recursive equation has a solution, but I agree that it is not a natural way to ask about recursive function definitions.

It might be more reasonable to restrict our attention to equations of the form:

$$M\,x_1, \ldots, x_n \approx N$$

for some fixed term N (which may, of course, contain occurrences of M). The equation that we started with, $Mx \approx x(Mx)$ has this shape, with N being $x(Mx)$. We will see that, in PCF, all such recursive equations have a solution for M.

Fixpoints

The fact that we can always find solutions to such equations follows from one important property enjoyed by PCF terms. Although very simple, some people would say that it is *the* fundamental property of untyped programs.

We start by transferring the concept of fixed point (or fixpoint) of a function to λ-terms. On mathematical (set-theoretic) functions, the concept is defined as follows. Let A be a set and f be a function from A to A. Then $a \in A$ is said to be a *fixed point* of f just if $f(a) = a$. For example, 0 is a fixed point of the "doubling" function $z \mapsto 2 \cdot z$ on natural numbers, because $2 \cdot 0 = 0$. Some functions on naturals do not posess any fixed points, for example, there is no natural z that is equal to its own successor $z + 1$, so the successor function has no fixed points. On the other hand, every natural is a fixed point of the identity function on naturals.

Let's transfer this idea to PCF.

Definition 7.3. *A term N is said to be a **fixed point** of another term M just if $MN \approx N$.*

According to this definition there are analogues to the examples that we discussed above, but it's not a perfect correspondence. For example, every natural number is a fixed point of the identity:

$$\underline{\mathsf{id}}\,\underline{n} \approx \underline{n}$$

It's also the case that no *numeral* is the fixed point of the successor function.

$$\mathsf{S}\,\underline{k} \not\approx \underline{k}$$

However, this implementation of the successor function *does* posess a fixed point! I can tell you that without even looking at its definition, because of the following theorem.

Theorem 7.1 (First Recursion Theorem)**.** *Every term has a fixed point.*

Proof. We claim that $\mathsf{fix}\,M$ is a fixed point of M. We calculate:

$$\mathsf{fix}\,M \rhd M\,(\mathsf{fix}\,M)$$

Hence, $M\,(\mathsf{fix}\,M) \approx \mathsf{fix}\,M$. □

The proof of the First Recursion Theorem tells us more, not only does every term possess a fixed point, but the fixed point of M can be computed by a PCF program, $\mathsf{fix}\,M$. The primitive fix is an example of a *fixed point combinator*, so called because it computes the fixed point of another term.

Fixpoints and recursive functions

We can use fixed point combinators to solve certain kinds of recursive equations, including those of shape:

$$Mx_1 \cdots x_n \approx N$$

where, of course, N may contain some occurrences of M and $x_1, \ldots, x_n$. There is a certain recipe that we can follow in order to find M.

First, we can use the compatibility property of conversion to reduce the problem to finding an M that satisfies the following equation instead:

$$M \approx \lambda x_1, \ldots, x_n.N$$

This is because we can go from the second equation to the first one: we just apply $x_1, \ldots, x_n$ in order to both sides – this amounts to using compatibility with $C[] = [] \, x_1 \cdots x_n$.

Next, we can "abstract out M", we just look at N and replace all the occurrences of M in there by the same fresh variable. Let's say that variable f does not occur anywhere else, and N' is the result of replacing all the occurrences of M in N by f. Then it is sufficient to find an M that satisfies:

$$M \approx (\lambda f. \lambda x_1, \ldots, x_n.N')M$$

We know about this shape. The First Recursion Theorem guarantees that there is a solution, and fix allow us to describe it directly. For example, fix $(\lambda f. \lambda x_1, \ldots, x_n.N')$ is an M that satisfies this equation and hence also satisfies the very first equation.

8. Computability: Effectiveness and Turing-Completeness

In this lecture we will show that in PCF one can program every function that is "intuitively" or "effectively" computable.

8.1 Computability

It is worth pausing to remind ourselves exactly what is meant by this, and to do that I want to go back to the origins of computability.

In the late 1920s and early 1930s a great unsolved problem preoccupied mathematical logicians. This problem is called the *entscheidungsproblem*, which is German for "decision problem". Nowadays we are very familiar with decision problems, but back then the phrase referred to one in particular, which was the problem of determining the validity of a statement of first-order logic. The fact that there might be a step-by-step method, requiring no insight on the part of the practicioner, and by which one could determine the truth of any mathematical conjecture was an *entirely plausible* to the mathematicians at the turn of the 20th century.

The entscheidungsproblem was one pillar of the famous programme of David Hilbert. Hilbert really put meta-mathematics on the map. In the half-century before, mathematicians had become interested in putting more rigour into mathematics. On the one hand they were concerned by certain infinite constructions, like the limits of infinite series that arise in analysis, on the other there had been many important developments in mathematical logic which put a greater emphasis on axiomatisation. Hilbert was interested in analysing the axiomatisations themselves to understand when axioms could be said to be independent, when systems of axioms could be said to be consistent or complete and whether there was a procedure that could be followed in order to calculate the consequences of a set of axioms — hence the entsheidungsproblem.

If you believed that the decision problem has a positive solution, you need only propose an algorithm that will solve any instance. Even though the digital computer would not be invented for many years to come, logicians of the 1930s were quite able to recognise an algorithm when they saw one — (roughly speaking) a procedure of discrete steps, requiring no insight and which, if followed faithfully, would yield the desired result. However, if you believed that the decision problem has a negative solution things are a little more difficult. To show that there can be *no* algorithm capable of solving the problem requires that you can make precise the class of all algorithms. What else can you do? You would need to give a mathematical definition of the intuitive notion of, as it was sometimes

called at the time, *effective calculability*, which was thought of as meaning the quality of being solvable by an algorithm.

One of the key difficulties was to propose a precise definition of the set of computable functions that the mathematical community would agree captures the fuzzy idea of being "intuitively" or "effectively computable". Whatever definition is proposed, someone may challenge and say "I think there are functions that can be computed by an algorithm that lie outside the set you defined... we just haven't thought a specific example yet...'. The trouble is that it is not possible to *prove* that a definition of the set of computable functions completely captures the intuitive idea of computablility, because only the former lives in the world of mathematics – the latter is more of a gut a feeling.

Anyway, the first serious proposal for a definition of effective calculability was given by Alonzo Church, using his recently invented pure, untyped λ-calculus. That is, untyped PCF, but *without any constants* including fix! We now agree that the pure, untyped λ-calculus *does* capture all the intuitively computable functions, but at the time Church proposed it, it was not generally accepted. However Soon after, Alan Turing proposed his own notion of (what is now know as) Turing machine. Turing's great achievement was to give an informal, but very convincing argument as to why his theoretical machines – a kind of souped up finite automata – could do any computation that can ever be done. He also proved that his machines and the λ-calculus expressed exactly the same class of functions, so if you believe his argument then you must also believe that the pure, untyped λ-calculus captures computability as well.

The definition of computable functions that you saw last year was a bit more modern, going via *While*-programs, so let me first remind you of what a While program looks like:

$$\begin{array}{rrcl} \text{(Expressions)} & e & ::= & n \mid x \mid e + e \mid e - e \mid e * e \\ \text{(Booleans)} & b & ::= & e = e \mid e \leq e \mid \mathsf{true} \mid \mathsf{false} \mid \,!\, b \mid b \wedge b \\ \text{(Statements)} & s & ::= & \mathsf{skip} \mid x := e \mid \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \mid \mathsf{while}\ b\ \mathsf{do}\ s \mid s_1\ ;\ s_2 \end{array}$$

In the while language you studied last year, n could range over any integer, but for simplicity we will only consider natural numbers $n \in \mathbb{N}$. Consequently, we will need to say what it means to subtract a larger number from a smaller one (which would normally go negative) and we will opt to truncate at 0, so that e.g. $4 - 6 = 0$.

Recall that the *state* of a While program is a map σ from program variables to their values. However, we are going to simplify this a bit in our presentation by assuming that all the variables in a given While program are numbered: i.e. they are called x_1, x_2, x_3 and so on for some initial segment of the natural numbers. Then we can represent a state as a tuple of values $(n_1, \ldots, n_k)$ rather than a map $\sigma(x_i) = n_i$. For example:

$$\text{We will use:} \quad (1, 6, 23, 4, 19) \quad \text{rather than} \quad \begin{bmatrix} x_1 & \mapsto & 1 \\ x_2 & \mapsto & 6 \\ x_3 & \mapsto & 23 \\ x_4 & \mapsto & 4 \\ x_5 & \mapsto & 19 \end{bmatrix}$$

I won't go through the whole definition of the small-step semantics for While programs, because we are not actually going to give a formal proof in the end, I just want you to get the idea. However, I should remind you of a couple of important points. First, recall that, although Expressions e and

Booleans b cannot change the state of the program (they cannot embed an assignment or some such, as is possible in e.g. C), to understand their value ($[\![e]\!]$ and $[\![b]\!]$ respectively), we do need to supply the current state of the program, because they may contain variables:

$$[\![3 * 4]\!](1,6,23,4,19) = 12 \qquad [\![\underline{\mathsf{true}} \wedge \,!\,\underline{\mathsf{false}}]\!](1,6,23,4,19) = \underline{\mathsf{true}}$$
$$[\![3 + (x_2 * x_4)]\!](1,6,23,4,19) = 27 \qquad [\![(x_3 = 23) \wedge (x_4 \leq x_2)]\!](1,6,23,4,19) = \underline{\mathsf{true}}$$
$$[\![x_5 + (x_2 - x_4)]\!](1,6,23,4,19) = 19 \qquad [\![!\,(x_1 + 3 \leq x_4)]\!](1,6,23,4,19) = \underline{\mathsf{false}}$$

Then the small-step semantics is a binary relation $\Rightarrow$ on *configurations*, that are pairs $\langle s, \sigma \rangle$ of a statement and a state. For example, the factorial of $x_2 = 3$ can be computed as follows:

	Statement	State (before)
	$x_1 := 1$; while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(0,3)$
$\Rightarrow$	while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(1,3)$
$\Rightarrow$	$x_1 := x_1 * x_2$; $x_2 := x_2 - 1$; while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(1,3)$
$\Rightarrow$	$x_2 := x_2 - 1$; while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(3,3)$
$\Rightarrow$	while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(3,2)$
$\Rightarrow$	$x_1 := x_1 * x_2$; $x_2 := x_2 - 1$; while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(3,2)$
$\Rightarrow$	$x_2 := x_2 - 1$; while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(6,2)$
$\Rightarrow$	while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(6,1)$
$\Rightarrow$	$x_1 := x_1 * x_2$; $x_2 := x_2 - 1$; while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(6,1)$
$\Rightarrow$	$x_2 := x_2 - 1$; while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(6,1)$
$\Rightarrow$	while $1 \leq x_2$ do $(x_1 := x_1 * x_2 \,;\, x_2 := x_2 - 1)$	$(6,0)$
$\Rightarrow$	skip	$(6,0)$

We recall the notation $[x \mapsto n]$ for the state in which x is set to n and every other variable is set to 0 and we extend it to $[x_i \mapsto n_i \mid 1 \leq i \leq p]$, to stand for the state in which each variable x_i is set to n_i, for i between 1 and p, and all other variables are set to 0. We write $\mathbb{N}^p$ for the set of all p-tuples $(n_1, \ldots, n_p)$ of natural numbers.

Definition 8.1. *A While program S **computes** a partial function $f : \mathbb{N}^p \to \mathbb{N}$ on p-tuples of natural numbers (with respect to variables $x_1, \ldots, x_p$) just if:*

$$f(n_1, \ldots, n_p) = m \qquad \textit{iff} \qquad \langle s, [x_i \mapsto n_i \mid 1 \leq i \leq p] \rangle \Rightarrow^* \langle \mathsf{skip}, [x_1 \mapsto f(n_1, \ldots, n_p)] \rangle$$

*We say that a partial function f on p-tuples of natural numbers is **While-computable** just if there is a While program s which computes it with respect to variables $x_1, \ldots, x_p$.*

Thesis 8.1 (Church-Turing)**.** *The (intuitively) computable partial functions on natural numbers are exactly the While-computable functions (equivalently, the Turing-computable functions, the Haskell-definable functions etc).*

8.2 Simulating While Programs in PCF

Now our aim is to show that every While program can be simulated faithfully by a term of PCF. Given a While program s over k variables, we will construct a PCF term $(\!|s|\!)$ such that, for all possible

states $\sigma = (n_1, \ldots, n_k)$, $\langle s, \sigma\rangle \Rightarrow \langle s', \sigma'\rangle$ just if $(\!|s|\!)\ \sigma \approx (\!|s'|\!)\,\sigma'$. Note: here we are using σ as a While program state on the left of the "just if" and as a tuple of numerals (i.e. a PCF term) on the right. Note that, by the definition of conversion, if $(\!|s'|\!)\,\sigma'$ is a normal form then we will have $(\!|s|\!)\ \sigma \rhd^* (\!|s'|\!)\,\sigma'$.

Following the definition of the While syntax, we break our translation from While into PCF down into separate translations for Expressions, Booleans and Statements. Just as each of these syntactic classes requires a state in order to be evaluated correctly, the translation of an expression e, a Boolean b or a statement s will be a function $(\!|e|\!)$, $(\!|b|\!)$ and $(\!|s|\!)$ respectively, that takes a state as input and delivers a state as output. In each case, the only difficulty is to thread the state of the computation through so that all the subterms have access to the values of the variables. Throughout we will assume we are dealing with a program of k variables.

Numeric Expressions

The translation of expressions is given recursively on the syntax:

$$\begin{aligned}
(\!|n|\!) &= \underline{\mathsf{const}}\ \underline{n}\\
(\!|x_i|\!) &= \underline{\mathsf{proj}}_i^k\\
(\!|e_1 + e_2|\!) &= \lambda z.\, \underline{\mathsf{add}}\ ((\!|e_1|\!)\ z)\ ((\!|e_2|\!)\ z)\\
(\!|e_1 - e_2|\!) &= \lambda z.\, \underline{\mathsf{sub}}\ ((\!|e_1|\!)\ z)\ ((\!|e_2|\!)\ z)\\
(\!|e_1 * e_2|\!) &= \lambda z.\, \underline{\mathsf{mult}}\ ((\!|e_1|\!)\ z)\ ((\!|e_2|\!)\ z)
\end{aligned}$$

Here the z parameter that shows up in the last three abstractions is meant to represent the state. This ensures that $[\![e]\!](\sigma) = n$ iff $(\!|e|\!)\ \sigma \rhd^* \underline{n}$, in other words: a While expression e evaluates to n in a given state iff applying the PCF term $(\!|e|\!)$ to the same tuple reduces to the numeral for n. For example we have:

$$(\!|3 + (x_2 * x_4)|\!) = \lambda z.\, \underline{\mathsf{add}}\ (\underline{\mathsf{const}}\ \underline{3}\ z)((\lambda z.\, \underline{\mathsf{mult}}\ (\underline{\mathsf{proj}}_2^5\ z)(\underline{\mathsf{proj}}_4^5\ z))\ z)$$

Then recall that $[\![3 + (x_2 * x_4)]\!](1, 6, 23, 4, 19) = 27$ and observe:

$$\begin{aligned}
&(\lambda z.\, \underline{\mathsf{add}}\ (\underline{\mathsf{const}}\ \underline{3}\ z)((\lambda z.\, \underline{\mathsf{mult}}\ (\underline{\mathsf{proj}}_2^5\ z)(\underline{\mathsf{proj}}_4^5\ z))\ z))\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19})\\
&\rhd^* \underline{\mathsf{add}}\ (\underline{\mathsf{const}}\ \underline{3}\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}))((\lambda z.\, \underline{\mathsf{mult}}\ (\underline{\mathsf{proj}}_2^5\ z)(\underline{\mathsf{proj}}_4^5\ z))\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}))\\
&\rhd^* \underline{\mathsf{add}}\ \underline{3}\ ((\lambda z.\, \underline{\mathsf{mult}}\ (\underline{\mathsf{proj}}_2^5\ z)(\underline{\mathsf{proj}}_4^5\ z))\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}))\\
&\rhd^* \underline{\mathsf{add}}\ \underline{3}\ (\underline{\mathsf{mult}}\ (\underline{\mathsf{proj}}_2^5\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}))(\underline{\mathsf{proj}}_4^5\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19})))\\
&\rhd^* \underline{\mathsf{add}}\ \underline{3}\ (\underline{\mathsf{mult}}\ \underline{6}\ \underline{4})\\
&\rhd^* \underline{\mathsf{add}}\ \underline{3}\ (\underline{24})\\
&\rhd^* \underline{27}
\end{aligned}$$

Boolean Expressions

The translation of Booleans is given recursively on the syntax:

$$\begin{aligned}
(\!| e_1 = e_2 |\!) &= \lambda z.\, \underline{\mathsf{eq}}\ ((\!| e_1 |\!)\ z)\ ((\!| e_2 |\!)\ z) \\
(\!| e_1 \leq e_2 |\!) &= \lambda z.\, \underline{\mathsf{leq}}\ ((\!| e_1 |\!)\ z)\ ((\!| e_2 |\!)\ z) \\
(\!| \mathsf{true} |\!) &= \underline{\mathsf{const}}\ \underline{\mathsf{true}} \\
(\!| \mathsf{false} |\!) &= \underline{\mathsf{const}}\ \underline{\mathsf{false}} \\
(\!| !\, b |\!) &= \lambda z.\, \underline{\mathsf{not}}\ ((\!| b |\!)\ s) \\
(\!| b_1 \wedge b_2 |\!) &= \lambda z.\, \underline{\mathsf{and}}\ ((\!| b_1 |\!)\ z)\ ((\!| b_2 |\!)\ z)
\end{aligned}$$

This ensures that, for all Boolean expressions b and all states σ: $[\![b]\!](\sigma) = \mathsf{true}$ *iff* $(\!| b |\!)\ \sigma \rhd^* \underline{\mathsf{true}}$, and $[\![b]\!](\sigma) = \mathsf{false}$ *iff* $(\!| b |\!)\ \sigma \rhd^* \underline{\mathsf{false}}$. For example, we have:

$$(\!| !\,(x_1 + 3 \leq x_4) |\!) = \lambda z.\, \underline{\mathsf{not}}\ ((\lambda z.\, \underline{\mathsf{leq}}\ ((\lambda z.\, \underline{\mathsf{add}}\ (\underline{\mathsf{proj}}_1^5\ z)\ (\underline{\mathsf{const}}\ \underline{3}\ z))\ z)\ (\underline{\mathsf{proj}}_4^5\ z))\ z)$$

Then recall that $[\![!\,(x_1 + 3 \leq x_4)]\!](1, 6, 23, 4, 19) = \mathsf{false}$, and observe:

$$\begin{aligned}
&(\lambda z.\, \underline{\mathsf{not}}\ ((\lambda z.\, \underline{\mathsf{leq}}\ ((\lambda z.\, \underline{\mathsf{add}}\ (\underline{\mathsf{proj}}_1^5\ z)\ (\underline{\mathsf{const}}\ \underline{3}\ z))\ z)\ (\underline{\mathsf{proj}}_4^5\ z))\ z))\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}) \\
&\rhd^*\ \underline{\mathsf{not}}\ ((\lambda z.\, \underline{\mathsf{leq}}\ ((\lambda z.\, \underline{\mathsf{add}}\ (\underline{\mathsf{proj}}_1^5\ z)\ (\underline{\mathsf{const}}\ \underline{3}\ z))\ z)\ (\underline{\mathsf{proj}}_4^5\ z))\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19})) \\
&\rhd^*\ \underline{\mathsf{not}}\ (\underline{\mathsf{leq}}\ ((\lambda z.\, \underline{\mathsf{add}}\ (\underline{\mathsf{proj}}_1^5\ z)\ (\underline{\mathsf{const}}\ \underline{3}\ z))\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}))\ (\underline{\mathsf{proj}}_4^5\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}))) \\
&\rhd^*\ \underline{\mathsf{not}}\ (\underline{\mathsf{leq}}\ ((\lambda z.\, \underline{\mathsf{add}}\ (\underline{\mathsf{proj}}_1^5\ z)\ (\underline{\mathsf{const}}\ \underline{3}\ z))\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}))\ \underline{4}) \\
&\rhd^*\ \underline{\mathsf{not}}\ (\underline{\mathsf{leq}}\ (\underline{\mathsf{add}}\ (\underline{\mathsf{proj}}_1^5\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19}))\ (\underline{\mathsf{const}}\ \underline{3}\ (\underline{1}, \underline{6}, \underline{23}, \underline{4}, \underline{19})))\ \underline{4}) \\
&\rhd^*\ \underline{\mathsf{not}}\ (\underline{\mathsf{leq}}\ (\underline{\mathsf{add}}\ \underline{1}\ \underline{3})\ \underline{4}) \\
&\rhd^*\ \underline{\mathsf{not}}\ (\underline{\mathsf{leq}}\ \underline{4}\ \underline{4}) \\
&\rhd^*\ \underline{\mathsf{not}}\ \underline{\mathsf{true}} \\
&\rhd^*\ \underline{\mathsf{false}}
\end{aligned}$$

Statements

The translation of statements is given recursively on the syntax:

$$\begin{aligned}
(\!| \mathsf{skip} |\!) &= \underline{\mathsf{id}} \\
(\!| x_i := e |\!) &= \lambda z.\, \underline{\mathsf{id}}\ (\underline{\mathsf{proj}}_1^k\ z, \ldots, \underline{\mathsf{proj}}_{i-1}^k\ z, (\!| e |\!)\ z, \underline{\mathsf{proj}}_{i+1}^k\ z, \ldots, \underline{\mathsf{proj}}_k^k\ z) \\
(\!| \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 |\!) &= \lambda z.\, \underline{\mathsf{if}}\ ((\!| b |\!)\ z)\ ((\!| s_1 |\!)\ z)\ ((\!| s_2 |\!)\ z) \\
(\!| \mathsf{while}\ b\ \mathsf{do}\ s |\!) &= \mathsf{fix}\ (\lambda f z.\, \underline{\mathsf{if}}\ ((\!| b |\!)\ z)\ (f\ \ ((\!| s |\!)\ z))\ z) \\
(\!| s_1 ; s_2 |\!) &= \lambda z.\, (\!| s_2 |\!)\ ((\!| s_1 |\!)\ z)
\end{aligned}$$

8.3 PCF is Turing-Complete

Now we are in a position to show that every While program can be simulated by a PCF program.

Lemma 8.1 (Simulation Lemma)**.** *If* $\langle s, \sigma\rangle \Rightarrow \langle s', \sigma'\rangle$ *then* $(\!|s|\!)\ \sigma \approx (\!|s'|\!)\ \sigma'$

Proof. By case analysis on the structure of while statement s:

- When s is skip, there is no possible transition, so the result holds vacuously.
- When s is of shape $x_i := e$, assume $\langle x_i := e, \sigma\rangle \Rightarrow \langle s', \sigma'\rangle$ – it must be that $s' = \mathsf{skip}$ and σ' is σ but with the i^{th} component of the tuple replaced by $[\![e]\!](\sigma)$. By definition:

  $$(\!| x_i := e |\!) = \lambda z.\, \underline{\mathsf{id}}\ (\underline{\mathsf{proj}^k_1}\ z, \ldots, \underline{\mathsf{proj}^k_{i-1}}\ z, (\!| e |\!)\ z, \underline{\mathsf{proj}^k_{i+1}}\ z, \ldots, \underline{\mathsf{proj}^k_k}\ z)$$

  Now $(\!|e|\!)\ \sigma \rhd^* \underline{[\![e]\!](\sigma)}$, so $(\underline{\mathsf{proj}^k_1}\ \sigma, \ldots, \underline{\mathsf{proj}^k_{i-1}}\ \sigma, (\!|e|\!)\ \sigma, \underline{\mathsf{proj}^k_{i+1}}\ \sigma, \ldots, \underline{\mathsf{proj}^k_k}\ \sigma) \rhd^* \sigma'$ and hence:

  $$(\!| x_i := e |\!)\ \sigma \rhd^* \underline{\mathsf{id}}\ \sigma' = (\!|\mathsf{skip}|\!)\ \sigma'$$

- When s is of shape $\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$, assume $\langle \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \sigma\rangle \Rightarrow \langle s', \sigma'\rangle$. Necessarily $\sigma = \sigma'$, but we need to analyse cases on $[\![b]\!](\sigma)$ to know what s' is:
  - If $[\![b]\!](\sigma) = \mathsf{true}$, then $s' = s_1$. By definition, $(\!|\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2|\!)\ \sigma \rhd^* \underline{\mathsf{if}}\ ((\!|b|\!)\ \sigma)\ ((\!|s_1|\!)\ \sigma)\ ((\!|s_2|\!)\ \sigma)$. Now, in this case, we have that $(\!|b|\!)\ \sigma \rhd^* \underline{\mathsf{true}}$ and so also:

    $$(\!|\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2|\!)\ \sigma \rhd^* \underline{\mathsf{if}}\ ((\!|b|\!)\ \sigma)\ ((\!|s_1|\!)\ \sigma)\ ((\!|s_2|\!)\ \sigma) \rhd^* (\!|s_1|\!)\ \sigma$$

    which is exactly as required.
  - If $[\![b]\!](\sigma) = \mathsf{false}$, then $s' = s_2$. By definition, $(\!|\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2|\!)\ \sigma \rhd^* \underline{\mathsf{if}}\ ((\!|b|\!)\ \sigma)\ ((\!|s_1|\!)\ \sigma)\ ((\!|s_2|\!)\ \sigma)$. Now, in this case, we have that $(\!|b|\!)\ \sigma \rhd^* \underline{\mathsf{false}}$ and so also:

    $$(\!|\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2|\!)\ \sigma \rhd^* \underline{\mathsf{if}}\ ((\!|b|\!)\ \sigma)\ ((\!|s_1|\!)\ \sigma)\ ((\!|s_2|\!)\ \sigma) \rhd^* (\!|s_2|\!)\ \sigma$$

    which is exactly as required.
- When s is of shape $\mathsf{while}\ b\ \mathsf{do}\ s$, assume $\langle \mathsf{while}\ b\ \mathsf{do}\ s, \sigma\rangle \Rightarrow \langle s', \sigma'\rangle$. Note that:

  $$\begin{aligned}(\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ \sigma &\rhd^* \underline{\mathsf{if}}\ ((\!|b|\!)\ \sigma)\ ((\lambda z.\, (\mathsf{fix}\ (\lambda f z.\, \underline{\mathsf{if}}\ ((\!|b|\!)\ z)\ (f\ ((\!|s|\!)\ z))\ (\underline{\mathsf{id}}\ z)))((\!|s|\!)\ z))\ \sigma)\ (\underline{\mathsf{id}}\ \sigma)\\ &= \underline{\mathsf{if}}\ ((\!|b|\!)\ \sigma)\ ((\lambda z.\, (\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ ((\!|s|\!)\ z))\ \sigma)\ (\underline{\mathsf{id}}\ \sigma)\end{aligned}$$

  In this case, necessarily, $\sigma = \sigma'$, but we need to analyse cases on $[\![b]\!](\sigma)$ to know what s' is:
  - If $[\![b]\!](\sigma) = \mathsf{true}$ then $s' = s\ ;\ \mathsf{while}\ b\ \mathsf{do}\ s$. Now, we have that $(\!|b|\!)\ \sigma \rhd^* \underline{\mathsf{true}}$ and so we can continue the sequence started above:

    $$(\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ \sigma \rhd^* \underline{\mathsf{if}}\ ((\!|b|\!)\ \sigma)\ ((\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ ((\!|s|\!)\ \sigma))\ \sigma \rhd^* (\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ ((\!|s|\!)\ \sigma)$$

    Finally, observe that, by definition $(\!|s\ ;\ \mathsf{while}\ b\ \mathsf{do}\ s|\!) = \lambda z.\, (\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ ((\!|s|\!)\ z)$, so $(\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ ((\!|s|\!)\ \sigma) \approx (\!|s\ ;\ \mathsf{while}\ b\ \mathsf{do}\ s|\!)\ \sigma$.

– If $[\![b]\!](\sigma) = \mathsf{false}$ then $s' = \mathsf{skip}$. Now, we have that $(\!| b |\!)\ \sigma \rhd^* \underline{\mathsf{false}}$ and so we can continue the sequence started above:

$$(\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ \sigma \rhd^* \underline{\mathsf{if}}\ ((\!| b |\!)\ \sigma)\ ((\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)\ ((\!| s |\!)\ \sigma))\ \sigma \rhd^* \sigma$$

Since $(\!|\mathsf{skip}|\!) = \underline{\mathsf{id}}$, we have $\sigma \approx (\!|\mathsf{skip}|\!)\ \sigma$.

- Finally, when s is of shape $s_1 ; s_2$, assume $\langle s_1 ; s_2, \sigma\rangle \Rightarrow \langle s', \sigma'\rangle$. By definition, $(\!| s_1 ; s_2 |\!) = \lambda z.\, (\!| s_2 |\!)\ ((\!| s_1 |\!)\ z)$. According to the semantics, there are three possibilities:
  - If $s_1 = \mathsf{skip}$ then $s' = s_2$ and $\sigma' = \sigma$. In this case $(\!| s_1 ; s_2 |\!)\ \sigma = (\lambda z.\, (\!| s_2 |\!)\ (\underline{\mathsf{id}}\, z))\ \sigma \rhd (\!| s_2 |\!)\ (\underline{\mathsf{id}}\, z) \rhd (\!| s_2 |\!)\ \sigma$, as required.
  - If $s_1 \neq \mathsf{skip}$ but $\langle s_1, \sigma\rangle \Rightarrow \langle \mathsf{skip}, \sigma_1\rangle$ then it follows from the induction hypothesis that $(\!| s_1 |\!)\ \sigma \approx (\!|\mathsf{skip}|\!)\ \sigma_1 = \underline{\mathsf{id}}\ \sigma_1$ and hence $(\!| s_1 |\!)\ \sigma \rhd^* \sigma_1$ (since σ_1 is a normal form). In this case, $s' = s_2$ and $\sigma' = \sigma_1$. To see the result, we can compute:

$$(\!| s_1 ; s_2 |\!)\ \sigma \rhd^* (\!| s_2 |\!)\ ((\!| s_1 |\!)\ \sigma) \rhd^* (\!| s_2 |\!)\ \sigma'$$

  - If $s_1 \neq \mathsf{skip}$ but $\langle s_1, \sigma\rangle \Rightarrow \langle s_1', \sigma_1\rangle$ with $s_1' \neq \mathsf{skip}$, then it follows from the induction hypothesis that $(\!| s_1 |\!)\ \sigma \approx (\!| s_1' |\!)\ \sigma_1$. In this case, by definition, $s' = s_1' ; s_2$ and $\sigma' = \sigma_1$. To see the result, we compute:

$$(\!| s_1 ; s_2 |\!)\ \sigma \approx (\!| s_2 |\!)\ ((\!| s_1 |\!)\ \sigma) \approx (\!| s_2 |\!)\ ((\!| s_1' |\!)\ \sigma') \approx (\!| s_1' ; s_2 |\!)\ \sigma'$$

□

It is easy to make precise what it means for a PCF program to compute a function on natural numbers:

Definition 8.2. *We say that a partial function $f : \mathbb{N}^p \to \mathbb{N}$ is **PCF-definable** just if there is a PCF term M such that, for all $n_1, \ldots, n_p \in \mathbb{N}$:*

- *If f is defined at $(n_1, \ldots, n_p)$, then $M\ \underline{n_1}\ \cdots\ \underline{n_p} \approx \underline{f(n_1, \ldots, n_p)}$.*
- *Otherwise $M\ \underline{n_1}\ \cdots\ \underline{n_p}$ does not have a normal form.*

Now, assuming the Church-Turing thesis, we can show that PCF can define all "intuitively computable" functions.

Theorem 8.1 (PCF is Turing-Complete)**.** *Every computable function on natural numbers is PCF-definable.*

Proof. For lack of time, and because it is all we need in the following, we will only prove the case for total functions. Suppose $f : \mathbb{N}^p \to \mathbb{N}$ is a computable *total* function on p-tuples of natural numbers. Then, by the Church-Turing thesis, there is a While program s that computes f with respect to variables $x_1, \ldots, x_p$. Let us suppose that this program contains $k \geq p$ variables. To show that f is also PCF-definable, we construct the following term:

$$\lambda x_1 \ldots x_p.\, \underline{\mathsf{proj}}_1^k\ ((\!| s |\!)\ (x_1, \ldots, x_p, \underline{0}, \ldots, \underline{0}))$$

(where $(x_1, \ldots, x_p, 0, \ldots, 0)$ denotes the tuple with x_i in i^{th} position, from $i = 1$ to p and zeroes thereafter up to position k). We claim that this term PCF-defines f, so let $(n_1, \ldots, n_p) \in \mathbb{N}^p$. Since f is total, f is defined at $(n_1, \ldots, n_p)$ and so, by definition, $\langle s, [x_i \mapsto n_i \mid 1 \leq i \leq p]\rangle \Rightarrow^* \langle \mathsf{skip}, [x_1 \mapsto f(n_1, \ldots, n_p)]\rangle$. It follows from the Simulation Lemma that, therefore $(\!|s|\!)\ [x_i \mapsto n_i \mid 1 \leq i \leq p] \approx (\!|\mathsf{skip}|\!)\ [x_1 \mapsto f(n_1, \ldots, n_p)]$. Hence, we can compute:

$$
\begin{aligned}
&(\lambda x_1 \ldots x_p.\, \underline{\mathsf{proj}}_1^k\ ((\!|s|\!)\ (x_1, \ldots, x_p, \underline{0}, \ldots, \underline{0})))\ \underline{n_1}\ \cdots\ \underline{n_p} \\
&\quad \approx\ \underline{\mathsf{proj}}_1^k\ ((\!|s|\!)\ (\underline{n_1}, \ldots, \underline{n_p}, \underline{0}, \ldots, \underline{0})) \\
&\quad \approx\ \underline{\mathsf{proj}}_1^k\ ((\!|\mathsf{skip}|\!)\ (\underline{f(n_1, \ldots, n_p)}, \underline{0}, \ldots, \underline{0})) \\
&\quad \approx\ \underline{\mathsf{proj}}_1^k\ (\underline{f(n_1, \ldots, n_p)}, \underline{0}, \ldots, \underline{0}) \\
&\quad \approx\ \underline{f(n_1, \ldots, n_p)}
\end{aligned}
$$

□

9. Computability: Decidability and Undecidability

This material was not covered in lectures and **will not be examined**. However, it is interesting!

9.1 Decidable Properties of Terms

First, I want to talk a little bit about the fact that, throughout what we have done so far, we have assumed that the functions that we are interested in computing are functions on natural numbers. This is completely standard, and it makes sense if you think about it. Your experience of digital computers should suggest to you that all kinds of data — strings, arrays, trees, graphs etc — can all ultimately be represented using sequences of bits. Since we can think of a sequence of bits as a binary numeral, we can thus encode any kind of data as some natural number. Operations on data can then be implemented as operations on natural numbers, and to all intents and purposes basic arithmetic is all that is needed.

Consequently, formal systems (comprising syntax and rules) are especially powerful if they allow for the representation of numbers and the implementation of arithmetic. One way to see this precisely is to look at the consequences of encoding parts of the system within itself. This can typically done because the a formal system is just a collection of strings and strings can be encoded as numbers.

This was the approach of Gödel for obtaining his famous incompleteness results about the theory of arithmetic in the early 1930s. In the theory of arithmetic, the central concept is provability. By encoding the formulas and proof system of arithmetic within the theory of arithmetic he was able to derive the conclusion that there are some formulas which, although true, are not provable. In the PCF the concept we are interested in is computability (stated precisely as PCF-definability), and the same technique will allow us to conclude that there are a large number of functions that are not computable.

The technique of encoding syntax by numbers takes Gödel's name. Here we are interested in encoding PCF-terms.

Definition 9.1. *A **Gödel numbering** is a pair of computable functions:*

- $\# : \Lambda \to \mathbb{N}$
- $\#^{-1} : \mathbb{N} \to \mathsf{Maybe}\ \Lambda$

with the property that:

$$\#^{-1}\ (\#\ M) = \mathsf{Just}\ M$$

Sometimes it is required that a Gödel encoding be a bijection, but we won't need that. There are actually lots of left-invertible injections # that we could define in order to satisfy the required properties and, for the rest of our development, it doesn't matter which one we choose. However, it is worth seeing one of them to make sure we all believe that it is possible. Let's consider an adaptation of Gödel's original.

We start by making an arbitrary assignment code of natural numbers to the letters that make up the strings in the language Λ. For example maybe we decide to assign:

$$\begin{aligned} \mathsf{code}(\text{`}\lambda\text{'}) &= 1 & \mathsf{code}(\text{`S'}) &= 5 \\ \mathsf{code}(\text{`(`}) &= 2 & \mathsf{code}(\text{`pred'}) &= 6 \\ \mathsf{code}(\text{`)'}) &= 3 & \mathsf{code}(\text{`fix'}) &= 7 \\ \mathsf{code}(\text{`.'}) &= 4 & \mathsf{code}(\text{`ifz'}) &= 8 \\ & & \mathsf{code}(\text{`wrong'}) &= 9 \end{aligned}$$

and then enumerate the countably many variables as $\mathsf{code}(\text{`}x_0\text{'}) = 10$, $\mathsf{code}(\text{`}x_1\text{'}) = 11$, $\mathsf{code}(\text{`}x_2\text{'}) = 12$ and so on. The key is then to map a whole string to a single number in such a way that we ensure that distinct strings get mapped to distinct numbers. The way Gödel did this was to use products of primes, with each prime raised to the power corresponding to the code of the letter:

$$\#(\text{``}(\mathsf{fix}\ (\lambda x_1.\, x_1))\text{''}) = 2^2 \cdot 3^7 \cdot 5^2 \cdot 7^1 \cdot 11^{10} \cdot 13^4 \cdot 17^{10} \cdot 19^3 \cdot 23^3$$

It then follows from the fundamental theorem of arithmetic that different strings will give rise to different Gödel numbers.

Since we can represent numbers in PCF by our PCF numerals, let us investigate what we can say about PCF-terms computing with codings of PCF-terms. Our main aim is to be able to deduce some undecidability results about problems to do with PCF. For example, we will show that the problem of determining whether a PCF-term M terminates is undecidable.

To that end, we first need to make precise what we mean by property of terms and what it means for such a property to be decidable. In mathematics, a property of a class of objects is often just represented as a the set of those objects which have the property. For example, the property of being even is just the set of even numbers. Then we can say that an object satisfies a property just if it is a member of the set. We will say that a property is decidable just when membership in the set is decidable.

Definition 9.2 (Decidability)**.** *Let $\Phi \subseteq \Lambda$ be a property of PCF-terms. We say that Φ is **decidable** or **recursive** just if the characteristic function of Φ is computable, i.e. there is an (intuitively) total computable function $\chi : \Lambda \to \mathbb{N}$ satisfying, for all terms $M \in \Lambda$:*

$$\chi(M) = \begin{cases} 1 & \text{if } M \in \Phi \\ 0 & \text{if } M \notin \Phi \end{cases}$$

To deduce that a particular problem is decidable, all we need is to convince ourselves that the characteristic function of the corresponding set is computable. For example:

Lemma 9.1. *The problem* $\{M \mid M \text{ is an abstraction}\}$ *is decidable.*

Proof. Given M, we can compute whether or not M is an abstraction by parsing M. □

In practice, a short sketch of an algorithm such as the one above always suffices as a proof.

9.2 Undecidability and the Second Recursion Theorem

To show that a problem is undecidable is, conceptually, a lot more difficult. However, there is a very powerful theorem, independently discovered by Scott and Curry for the pure untyped λ-calculus, that allows us to conclude undecidability relatively straightforwardly, in many cases. We start by considering some computable functions on the codes of terms.

Lemma 9.2. *There exist terms* $\underline{\mathsf{app}}$ *and* $\underline{\mathsf{gnum}}$ *such that:*

$$\underline{\mathsf{app}}\ \underline{\#M}\ \underline{\#N} \approx \underline{\#(MN)} \qquad \textit{and} \qquad \underline{\mathsf{gnum}}\ \underline{n} \approx \underline{\#\underline{n}}$$

Proof. Given that $\#$ and $\#^{-1}$ are intuitively computable:

- The function $\mathsf{app} : \mathbb{N}^2 \to \mathbb{N}$ satisfying $\mathsf{app}(\#M, \#N) = \#(MN)$ is, intuitively, a computable function (I could come up with an algorithm to implement it).
- Likewise, the function $\mathsf{gnum} : \mathbb{N} \to \mathbb{N}$ satisfying $\mathsf{gnum}(n) = \#(\underline{n})$ is, intuitively, a computable function.

Hence, it follows from the Church-Turing thesis that there are (probably quite complicated) terms $\underline{\mathsf{app}}$ and $\underline{\mathsf{gnum}}$ to define them in PCF. □

It is worth considering the two expressions M and $\underline{\#M}$. *Both of these expressions are terms*. The best way to think about the situation is that M is a program, $\#M$ is its source code and $\underline{\#M}$ is it's source code represented as a datatype that we can compute with in PCF (i.e. a PCF numeral). Therefore, we can understand the combinator $\underline{\mathsf{app}}$ as being the program that takes the source code of M and the source code of N and returns the source code for MN. If we specialise the equation for gnum so that n is, in fact, $\#M$, the code for some program, then we can understand gnum as being the program that takes the source code for some program M (represented as a PCF numeral) and returns the source code for that numeral.

The following important result is due to Kleene (for pure λ-calculus).

Theorem 9.1 (Second Recursion Theorem)**.** *For all terms* F*, there exists a term* M *such that:*

$$F\ \underline{\#M} \approx M$$

Proof. Let F be an arbitrary term. Then define N as $\lambda x.F\,(\underline{\mathsf{app}}\ x\ (\underline{\mathsf{gnum}}\ x))$. Finally, take $N\ \underline{\#N}$ to be the witness M: it is easily seen that this term satisfies $F\ \underline{\#M} \approx M$. □

First, pay special attention to the quantification here: it says that *for all* terms *there exists* an input which behaves in a certain way. A common mistake is to look at the defining equation and assume that the theorem is stating the existence of an self-interpreter F for terms. There is such a theorem, but this is not it. It's a bit more difficult to attach an intuitive meaning to the theorem, except that it is stating the existence of fixed points *modulo coding*.

9.3 The Scott-Curry Theorem

The theorem is valuable because it says that, if you construct a program that is supposed to compute with the codes of λ-terms, then no matter how you implement it (for all F), there will always be some input $\underline{\#M}$ about which you have no control over the corresponding output — it is always equal to M, whether that is what you want or not!

If you want to show that some set Φ is decidable then you must exhibit an F computing the characteristic function of Φ, that is it must be equal to $\underline{0}$ or $\underline{1}$ depending on whether the input is the code of a term M in Φ or not. What is the chance that you can construct an F with this property given that there will always be some input $\underline{\#M}$ over which you no control about the corresponding output? I'll tell you, it's not good.

In fact, for any property worth its salt, we can construct a rather tricksy term that precisely demonstrates the folly. When we say "any property worth its salt" what we mean is that the property should be a *non-trivial* and *behavioural*. Trivial properties are properties which are satisfied by no terms or by all terms. These kinds of properties are too easy to decide, because the characteristic function is constantly 0 or constantly 1 and these are obviously computable. Non-behavioural properties concern only the syntax of terms and not *what they compute*. For example, the property $\{M \in \Lambda \mid M \text{ is an abstraction }\}$ is not behavioural because there are terms like $\underline{\mathsf{idconst}}$ which do not satisfy the property and yet are convertible with a term $\underline{\mathsf{const}}$ that does. Non-behavioural properties are easy to compute because we just need to interrogate the syntax. If a property is both non-trivial and behavioural then it is undecidable.

Theorem 9.2 (Scott-Curry)**.** *Let $\Phi \subseteq \Lambda$ satisfy the following properties:*

- *Φ is **non-trivial**: $\emptyset \neq \Phi \neq \Lambda$*
- *Φ is **behavioural**: if $M \in \Phi$ and $M \approx N$, then $N \in \Phi$*

Then it follows that Φ is undecidable.

Proof. Since Φ is non-trivial, there is some $P \in \Phi$ and some $Q \notin \Phi$. Suppose, for the purposes of obtaining a contradiction, that Φ is decidable. Then it follows that there is some term F that computes its characteristic function, i.e. satisfying:

$$F\ \underline{\#M} \approx \begin{cases} \underline{1} & \text{if } M \in \Phi \\ \underline{0} & \text{if } M \notin \Phi \end{cases}$$

Define $G \coloneqq \lambda x.\,\mathsf{ifz}\ (F\ x)\ P\ Q$. It follows from the Second Recursion Theorem that there is some term N such that $G\ \underline{\#N} = N$. One of $N \in \Phi$ or $N \notin \Phi$ is certainly true, so let us examine the two cases:

- If $N \in \Phi$, then by the properties of N, F and ifz:

$$N \;\approx\; G\;\underline{\#N} \;\approx\; \mathsf{ifz}\;(F\;\underline{\#N})\;P\;Q \;\approx\; Q$$

  but we assumed that $N \in \Phi$ and Φ is closed under $\approx$, so we are forced to conclude that $Q \in \Phi$ which is a contradiction.

- If $N \notin \Phi$, then by the properties of N, F and ifz:

$$N \;\approx\; G\;\underline{\#N} \;\approx\; \mathsf{ifz}\;(F\;\underline{\#N})\;P\;Q \;\approx\; P$$

  but we assumed that $N \notin \Phi$ and Φ is closed under $\approx$, so we are forced to conclude that $P \notin \Phi$ which is a contradiction.

□

The theorem is a direct analogue of Rice's Theorem for properties of Turing machines (you may have seen a version for While programs). Whether or not you have come across it, I recommend you look it up and compare its statement to the Scott-Curry Theorem.

So which reasonable properties of λ-terms are decidable? Essentially none of them.

Corollary 9.1. *The problem* $\Phi_{norm} := \{M \in \Lambda \mid M \text{ is normalising}\}$ *is undecidable.*

Proof. The set Φ_{norm} is non-trivial because $\underline{\mathsf{id}}$ is a member but $\underline{\mathsf{div}}$ is not. Moreover, it is closed under $\approx$. To see this let M be normalising and $M \approx N$. Since M is normalising, there is a normal form P with $M \approx P$. Therefore, also $N \approx P$ and it follows that, therefore $N \rhd {}^{*}P$. Consequently, N is normalising. Hence, undecidability of Φ_{norm} follows from the Scott-Curry theorem. □

10. Types: Judgements and the Type System

So far we have been considering the untyped syntax of PCF, in which it is perfectly possible to create strange looking terms like $\underline{1}\ \underline{\mathsf{id}}$ and compute with them, although the computation does not end well! In practice, we would like that such combinations of terms simply cannot arise, that they are forbidden, and this is where the type system comes in.

10.1 Intuitions

Typically, we are only interested in applications of a function to data that is within its domain and which yield data from its codomain — applications that are *well typed*. *Type theory* concerns the formalisation of this idea of type and rules for deducing well-typedness. In the type theory that we study in this second half of the unit, this is made precise by the *judgement*:

$$\Gamma \vdash M : A$$

which we can read as "Under assumptions Γ, M has type A". There are two particularly fruitful views of type theory, which help to explain the meaning of the words "has type" intuitively.

Types describe abstract properties

In this view, "type" and "property" are taken to be synonymous. Type theories are a syntax for reasoning about the properties of terms. For example, to say $M : A \to B$ is to assert that M has a certain property, namely: given any input satisfying the property A, its output will satisfy the property B. An example is the following:

$$\lambda x.\, x + 1 : \{n \in \mathbb{Z} \mid n \text{ is even}\} \to \{n \in \mathbb{Z} \mid n \text{ is odd}\}$$

The successor function has the property that, given an even integer, it will always return an odd integer. In this view it is typical that a term may enjoy the association of many different types. For example, the same term also satisfies the property $\{n \in \mathbb{Z} \mid n \geq 0\} \to \{n \in \mathbb{Z} \mid n \neq 0\}$.

This view finds important applications in the *analysis of programs*. The second property is exactly what you need to know about $\lambda x.\, x + 1$ in order to deduce that the following program does not raise an exception:

$$100 \div (\mathsf{if}\ n \geq 0\ \mathsf{then}\ (\lambda x.\, x + 1)\ n\ \mathsf{else}\ n)$$

Types prescribe abstract classifications

In this view, "type" and "classification" are taken as synonymous. Type theories are a syntax for enforcing the classification of terms. To say $M : A \to B$ is to assert that M belongs to a certain class, namely: the class of functions with domain A and codomain B. An example is the following:

$$\lambda x.\, x +\!\!+ [1] : [\mathsf{int}] \to [\mathsf{int}]$$

The append-one function is a function from list of integers to lists of integers. In this view it is typical that we think of the type as being an intrinsic part of the description of the function. For example, we consider $\lambda x.\, x +\!\!+ [1] : [\mathsf{float}] \to [\mathsf{float}]$ to be a *different function*, it's just a coincidence that you happen to follow the same rule in order to compute its output.

Of course, these two views are far from mutually exclusive! On the one hand, it may only be sensible to infer properties of terms that are already classified, for example, the properties above only make sense for functions of type $\mathbb{Z} \to \mathbb{Z}$. On the other, a classification is a kind of very coarse property, for if M is classified as a function of type $[\mathsf{int}] \to [\mathsf{int}]$ then it certainly has the property that it maps inputs with the property of being a list of integers to outputs with the same property.

Just as there are an amazing variety of programming languages out there, there is an even greater variety of type systems. We will study the "simple type system" that is usually associated with PCF (here: *simple* has a technical connotation). Simple types go back all the way to the logicians of the early 20th Century, but they form the core part of any of the modern type systems that you will find attached to programming languages today.

10.2 Types and Environments

We start by saying exactly what the types of this system actually look like. There are two kinds of types in this system, the *types* and the *type schemes*.

Definition 10.1. *We assume a countable set of* ***type variables*** $\mathbb{A}$*, ranged over by* a*,* b*,* c *and other lowercase letters from the start of the alphabet. The* ***types****, written* $\mathbb{T}$ *are a set of strings* A, B *etc defined by the following grammar:*

$$\textit{(Types)} \quad A, B \quad ::= \mathsf{Nat} \mid a \mid (A \to B)$$

We immediately adopt the following conventions to make our life less tedious:

- *We omit the outermost parenthesis when writing types.*
- *We assume that the arrow associates to the right: i.e. when we write* $A_1 \to A_2 \to A_3$*, the type that we mean is* $(A_1 \to (A_2 \to A_3))$*.*

Here are some examples of types and type schemes written with and without all their parentheses.

$$\begin{array}{rl} \mathsf{Nat} \to \mathsf{Nat} & (\mathsf{Nat} \to \mathsf{Nat}) \\ a \to b \to b & (a \to (b \to b)) \\ (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} & ((\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}) \end{array}$$

We will consistently use A, B, C and other upper case letters at the start of the alphabet to refer to types generically.

Types that are of the form $A_1 \to A_2$ are variously called *arrow types* or *function types* and the idea is that if M has type $A_1 \to A_2$ then it represents a function that maps terms of type A_1 to terms of type A_2. You can think of type variables as placeholders for any type. So, for example, from the fact that we will be able to assign the type $a \to a$ to the identity function $\underline{\mathsf{id}} : a \to a$ we can infer that we could assign *any* type of shape $A \to A$ to this function. However, this is in itself not very useful because our system cannot exploit this fact, it is not *polymorphic*. The real value of type variables will come later when we discuss inference.

The whole point of types is to classify terms, so we next need to say how that comes about. Let's first agree on how to write it.

Definition 10.2. *A **type assignment** is a pair of a term M and a type $\forall \overline{a}.A$, written $M : \forall \overline{a}.A$. The term part of a type assignment is called the **subject** and the type part the **predicate**.*

Clearly, not all type assignments $M : A$ should be allowed. For example, we would not expect $\underline{1}$ to have type $\mathsf{Nat} \to \mathsf{Nat}$. On the other hand, we would expect that $\lambda x.\mathsf{S}\ x : \mathsf{Nat} \to \mathsf{Nat}$ or even $\lambda x.x : a \to a$.

The purpose of a *type system* is to provide a collection of rules in order to determine when $M : A$ is valid. In general M may contain free variables and then the type of M will usually depend on the type of the free variables. For example, term $\lambda x.y$ should be allowed to have a type of the form $A \to B$ where B is the type of y. So, this begs the question, what should be the type of a free variable?

We allow for *hypothetical* deductions. That is, we define when $M : A$ is valid only *with respect to* some given *assumptions* (also known as *hypotheses*) concerning the types of the free variables. These hypotheses are just type assignments of the form $x : B$, meaning that free variable x is assumed to have type B, which are collected together in a *type environment*, usually written Γ. We require that deductions can be made only under consistent assumptions.

Definition 10.3. *A **type environment**, generically written Γ, is a finite set of type assignments of the form $x : A$ which is, moreover, **consistent**, in the sense that if $x : A \in \Gamma$ and $x : B \in \Gamma$, then $A = B$.*

*The **subjects** of Γ is the set, written $\mathsf{dom}\ \Gamma$, of those term variables x for which there is some A such that $x : A \in \Gamma$.*

A concrete example of a type environment is $\{x : a \to b, y : \mathsf{Nat} \to \mathsf{Nat}, z : \mathsf{Nat}\}$ (whenever we write a type environment explicitly like this, we will assume that the *subjects* x, y and z are distinct variables).

10.3 Type System

The notion that a term may be assigned a type under some assumptions is made precise through the notion of *typing judgement*. A *type system* gives rules by which typing judgements can be deduced. We start by saying what should be the types of the built-in constants of the language.

Definition 10.4. *Let $\mathbb{C}$ be the following collection of type assignments:*

$$\begin{aligned}&\{\mathsf{Z} : \mathsf{Nat}\}\\ &\cup \{\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}\}\\ &\cup \{\mathsf{pred} : \mathsf{Nat} \to \mathsf{Nat}\}\\ &\cup \{\mathsf{ifz} : \mathsf{Nat} \to A \to A \to A \mid A \in \mathbb{T}\}\\ &\cup \{\mathsf{fix} : (A \to A) \to A \mid A \in \mathbb{T}\}\end{aligned}$$

You can see from this definition that we expect terms to be assigned several types, in general. For example, the ifz constant is assigned every type of the form $\mathsf{Nat} \to A \to A \to A$ for some type A. This should ring true with your intuitions, because we expect the ifz constant to take three arguments: an number, the then branch and the else branch and it should return one of those branches. The branches can be numbers of type Nat: ifz x $\underline{1}$ $\underline{2}$; or they could be functions of type $\mathsf{Nat} \to \mathsf{Nat}$: ifz x $\underline{\mathsf{id}}$ S; or of any other type A, as long as the type of the then branch and the else branch match. Recall that the idea of the fixed point combinator fix is that it takes a function as input and returns a fixed point of that function. This only makes sense for functions that have the same domain as their codomain because, if N is a fixed point of M then $M\ N \approx N$ – it returns something of the same type as that taken as input. Hence, fix should take a function of any type $A \to A$ (i.e. where the domain and codomain match) and return a fixed point, which is of course something of this type A.

Definition 10.5 (Type System)**.** *A **type judgement** is a triple consisting of a type environment Γ, a term M and a type A, which we write:*

$$\Gamma \vdash M : A$$

*Then the **type system** is the following collection of rules by which one can justify such type judgements using proof trees:*

$$x{:}A \in \Gamma \ \frac{}{\Gamma \vdash x : A}\ \textit{(TVar)} \qquad c{:}A \in \mathbb{C}\ \frac{}{\Gamma \vdash c : A}\ \textit{(TCst)}$$

$$\frac{\Gamma \vdash M : B \to A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : A}\ \textit{(TApp)} \qquad x \notin \mathsf{dom}\ \Gamma\ \frac{\Gamma \cup \{x : B\} \vdash M : A}{\Gamma \vdash \lambda x.\, M : B \to A}\ \textit{(TAbs)}$$

*In type theory, a proof tree justifying a type judgement is usually called a **type derivation**.*

Since we are never in any danger of confusion, whenever we write out a type environment explicitly as part of a judgement, we will omit the braces part of the set notation. So, for example, we will write $x{:}\mathsf{Nat} \to \mathsf{Nat},\ y : \mathsf{Nat} \vdash x\,y : \mathsf{Nat}$ instead of the more cumbersome $\{x{:}\mathsf{Nat} \to \mathsf{Nat},\ y{:}\mathsf{Nat}\} \vdash x\,y : \mathsf{Nat}$. For the same reason, if we make no assumptions then we will simply write $\vdash M : A$ rather than $\emptyset \vdash M : A$.

The way to read a typing judgement $\Gamma \vdash M : A$ is "under assumptions Γ, M has type A" and these rules allow us to conclude sound judgements – i.e. where that statement is true. Each rule should be read as:

> "If any statement(s) above the line are true and any condition to the left is satisfied, then the statement below the line is also true"

The rule (TVar) allows us to make use of our assumptions, it says that, at any time (i.e. without premise), if we assumed $x{:}A$ then we can conclude $x : A$. The rule (TCst) plays a similar role for constants. It says that if $c{:}A$ is a primitive typing that we assigned to the constants in $\mathbb{C}$, then we can conclude exactly $c : A$. A difference between the assumptions on variables and the built-in assignment of types to the constants is that the former will typically change during a derivation, whereas the latter never changes.

The rule (TApp) allows us to make use of function types. It says that if we know M has a function type $B \to A$ and N has a matching type B then it follows that the application MN can be assigned the type according to what M is known to return, which is A.

The rule (TAbs) says that, in order to conclude that $\lambda x.\, M$ has type $B \to A$ from some assumptions Γ, I just have to show that I can conclude that M has type A (recall that M is what the function returns, and A is the return type of $B \to A$) under the additional assumption that x has type B.

The following is a proof tree showing that the function that adds two to its argument is of type $\mathsf{Nat} \to \mathsf{Nat}$. That is, that the following judgement is true: $\vdash \lambda x.\, \mathsf{S}\ (\mathsf{S}\ x) : \mathsf{Nat} \to \mathsf{Nat}$. We will take a look at proof trees in more detail in the next lecture.

$$\dfrac{\dfrac{\dfrac{}{x{:}\mathsf{Nat} \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}}\ (\mathsf{TCst}) \quad \dfrac{\dfrac{}{x{:}\mathsf{Nat} \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}}\ (\mathsf{TCst}) \quad \dfrac{}{x{:}\mathsf{Nat} \vdash x : \mathsf{Nat}}\ (\mathsf{TVar})}{x{:}\mathsf{Nat} \vdash \mathsf{S}\ x : \mathsf{Nat}}\ (\mathsf{TApp})}{x{:}\mathsf{Nat} \vdash \mathsf{S}\ (\mathsf{S}\ x) : \mathsf{Nat}}\ (\mathsf{TApp})}{\vdash \lambda x.\, \mathsf{S}\ (\mathsf{S}\ x) : \mathsf{Nat} \to \mathsf{Nat}}\ (\mathsf{TAbs})$$

11. Types: Derivations and Typability

In this lecture we look at more detail at type derivations/proof trees, some nuances of the system and some of the key problems associated with typing.

11.1 Proof Trees

Let me remind you of the rules of the type system:

$$x{:}A \in \Gamma \;\; \frac{}{\Gamma \vdash x : A}\;(\mathsf{TVar}) \qquad c{:}A \in \mathbb{C} \;\; \frac{}{\Gamma \vdash c : A}\;(\mathsf{TCst})$$

$$\frac{\Gamma \vdash M : B \to A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : A}\;(\mathsf{TApp}) \qquad x \notin \mathsf{dom}\;\Gamma \;\; \frac{\Gamma \cup \{x : B\} \vdash M : A}{\Gamma \vdash \lambda x.M : B \to A}\;(\mathsf{TAbs})$$

We can use the rules in order to construct *proof trees* (also called type derivations in this setting). A *proof tree* for this system is tree whose nodes are labelled by judgements, in such a way that, if a node is labelled $\Gamma \vdash M : A$ and its children are labelled $\Gamma_1 \vdash M_1 : A_1 \;\ldots\; \Gamma_k \vdash M_k : A_k$, then it must be that:

$$\frac{\Gamma_1 \vdash M_1 : A_1 \qquad \ldots \qquad \Gamma_k \vdash M_k : A_k}{\Gamma \vdash M : A}$$

is an instance of one of the given rules: (TVar), (TCst), (TAbs) or (TApp). An *instance* of a rule is just a copy of the rule but with the generic M, N, x, Γ and so on replaced *consistently* by particular objects like $(\lambda z.z)$, $\{x{:}\mathsf{Nat} \to \mathsf{Nat}\}$ or y.

For example:

$$\frac{x{:}\mathsf{Nat} \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat} \quad x{:}\mathsf{Nat} \vdash x : \mathsf{Nat}}{x{:}\mathsf{Nat} \vdash \mathsf{S}\;x : \mathsf{Nat}}\;(\mathsf{TApp})$$

is an instance of the (TApp) rule, with $M := \mathsf{S}$, $N := x$, $\Gamma := \{x{:}\mathsf{Nat}\}$, $A := \mathsf{Nat}$, $B := \mathsf{Nat}$. Similarly,

$$\frac{}{x{:}\mathsf{Nat} \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}}\;(\mathsf{TCst})$$

is an instance of the (TCst) rule, with $\Gamma := \{x{:}\mathsf{Nat}\}$, $c = \mathsf{S}$, and $A := \mathsf{Nat} \to \mathsf{Nat}$.

When we put rule instances together, in such a way that every branch of the tree is closed off by one of the axioms (those rules without premises) (TVar) or (TCst), then we obtain a proof tree.

The judgement at the root of the tree is then shown to be true. For example, the example from the previous lecture:

$$\dfrac{\dfrac{\dfrac{}{x{:}\mathsf{Nat} \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}}\,(\mathsf{TCst}) \quad \dfrac{}{x{:}\mathsf{Nat} \vdash x : \mathsf{Nat}}\,(\mathsf{TVar})}{x{:}\mathsf{Nat} \vdash \mathsf{S}\ x : \mathsf{Nat}}\,(\mathsf{TApp})}{x{:}\mathsf{Nat} \vdash \mathsf{S}\ (\mathsf{S}\ x) : \mathsf{Nat}}\,(\mathsf{TApp})}{\vdash \lambda x.\mathsf{S}\ (\mathsf{S}\ x) : \mathsf{Nat} \to \mathsf{Nat}}\,(\mathsf{TAbs})$$

When you want to construct a type derivation for a judgement, it is usually best to start from the bottom and build upwards. This type system has a particularly nice property that makes this approach work very well. In this system, there is exactly one rule for each possible shape of term. So given $\Gamma \vdash M : A$ there is only one rule that can be used to obtain this judgement in the conclusion, and it is completely determined by the shape of M. Type systems with this property are called *syntax-directed* because the syntax of the term directs you in building a derivation.

However, that is not to say that every type derivation is completely mechanical. The rule (TApp) requires some creativity. Imagine building a derivation that has $\Gamma \vdash MN : A$ in the conclusion. The shape MN determines that we must use the (TApp) rule, but to use this rule we have to somehow invent the type B. The type B is not necessarily mentioned in the conclusion (below the line) so it requires a little bit of thought (but usually not too much) to come up with a correct instantiation. Consider building the following derivation from the ground up, with $\Gamma = \{x : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}\}$ (I have omitted the rule names so that it fits on the page, make sure you can reconstruct them):

$$\dfrac{\dfrac{\dfrac{}{\Gamma, y{:}\mathsf{Nat} \to \mathsf{Nat} \vdash x : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}} \quad \dfrac{}{\Gamma, y{:}\mathsf{Nat} \to \mathsf{Nat} \vdash y : \mathsf{Nat} \to \mathsf{Nat}}}{\Gamma, y{:}\mathsf{Nat} \to \mathsf{Nat} \vdash xy : \mathsf{Nat}}}{\Gamma \vdash \lambda y.xy : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}} \quad \dfrac{\dfrac{}{\Gamma, z{:}\mathsf{Nat} \vdash z : \mathsf{Nat}}}{\Gamma \vdash \lambda z.z : \mathsf{Nat} \to \mathsf{Nat}}}{\Gamma \vdash (\lambda y.xy)(\lambda z.z) : \mathsf{Nat}}$$

The tree is concluded using an instance of the rule (TApp) (this is the rule used at the root), with $B := \mathsf{Nat} \to \mathsf{Nat}$, but there is not much of a clue to why you should pick $B := \mathsf{Nat} \to \mathsf{Nat}$ if you only have the root judgement to go off.

11.2 Types involving type variables

So far we have been deriving judgements in which the types involved were all combinations of the base type Nat and the arrow (or function type). However, we can also prove judgements involving types with type variables. For example, let me abbreviate the environment $\{x{:}a \to b \to c,\ y{:}a \to b, z{:}a\}$ as Γ and omit the rule names in the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : a \to b \to c} \quad \dfrac{}{\Gamma \vdash z : a}}{\Gamma \vdash xz : b \to c} \quad \dfrac{\dfrac{}{\Gamma \vdash y : a \to b} \quad \dfrac{}{\Gamma \vdash z : a}}{\Gamma \vdash yz : b}}{\Gamma \vdash xz(yz) : c}}{x : a \to b \to c,\ y : a \to b \vdash \lambda z.xz(yz) : (a \to b) \to a \to c}}{x : a \to b \to c \vdash \lambda yz.xz(yz) : (a \to b) \to a \to c}}{\vdash \lambda xyz.xz(yz) : (a \to b \to c) \to (a \to b) \to a \to c}$$

What does it mean for a term like $\lambda xyz.\, x\ z\ (y\ z)$ to have a type of shape $(a \to b \to c) \to (a \to b) \to a \to c$? You can think of type variables as placeholders: it is possible to show that, if a provable judgement $\Gamma \vdash M : A$ involves type variables, then any consistent replacement of the type variables will also yield a provable judgement. As a simpler example, consider the proof:

$$\dfrac{\dfrac{\dfrac{}{x{:}a,\, y{:}b \vdash x : a}\,(\text{TVar})}{x{:}a \vdash \lambda y.\, x : b \to a}\,(\text{TAbs})}{\vdash \lambda xy.\, x : a \to b \to a}\,(\text{TAbs})$$

Any replacement of a and b by any other types will also yield a valid proof tree. E.g. $a := (\mathsf{Nat} \to \mathsf{Nat})$, $b := \mathsf{Nat}$:

$$\dfrac{\dfrac{\dfrac{}{x{:}\mathsf{Nat} \to \mathsf{Nat},\, y{:}\mathsf{Nat} \vdash x : \mathsf{Nat} \to \mathsf{Nat}}\,(\text{TVar})}{x{:}\mathsf{Nat} \to \mathsf{Nat} \vdash \lambda y.\, x : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}}\,(\text{TAbs})}{\vdash \lambda xy.\, x : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}}\,(\text{TAbs})$$

In fact, there is a sense in which the type $a \to b \to a$ tells you everything about the possible typings for the term $\lambda xy.\, x$: we shall see when we look at inference that they all arise as instances of $a \to b \to a$.

No Polymorphism

However, this is an external property. The fact that I can assign a type like $a \to b \to a$ to $\lambda xy.\, x$ tells me, the human, that I will be able to construct a derivation of $\vdash \lambda xy.\, x : A \to B \to A$ for any particular types A and B. However, this is not a property that I can exploit within the system. This is because, unlike Haskell's type system, the simple type system we look at in this unit does not have *polymorphism*: that is, the ability to *use* a type involving type variables at different instances *within* a derivation. For example, consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\vdots\quad \dfrac{x{:}a \to a \not\vdash x : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat} \qquad \dfrac{}{x{:}a \to a \vdash \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}}}{x{:}a \to a \vdash x\ \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}}}{x{:}a \to a \vdash \mathsf{ifz}\ (x\ \underline{0})\ (x\ \mathsf{S}) : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}} \qquad \dfrac{}{x{:}a \to a \vdash \mathsf{pred} : \mathsf{Nat} \to \mathsf{Nat}}}{x{:}a \to a \vdash \mathsf{ifz}\ (x\ \underline{0})\ (x\ \mathsf{S})\ \mathsf{pred} : \mathsf{Nat} \to \mathsf{Nat}}$$

If you consider the invalid step $x{:}a \to a \not\vdash x : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$, we know that x has type $a \to a$, so, in principle, whatever term x represents, we could type it by the replacement $a := \mathsf{Nat} \to \mathsf{Nat}$. However, there is no rule within the system that allows us to do this in the middle of the type derivation. When we have $x{:}a \to a$ as an assumption, the *only* thing we can do with it is to conclude $x : a \to a$.

This is in contrast to Haskell, which has a type system with a limited form of polymorphism. In Haskell, for example, supposing id is the identity function, which has Haskell type $a \to a$, and succ is the successor function and pred the predecessor with types Int → Int, the expression:

if (**id** 0) **then** (**id succ**) **else pred**

is, in fact typable, because the Haskell polymorphic type system allows $\mathsf{id} : a \to a$ to be used at any instance (here it is used at Int → Int on the left and (Int → Int) → Int → Int on the right).

11.3 Typability and other Problems

When we introduced the system, we said that not all type assignments $M : A$ should be allowed (to be derived).

Definition 11.1. *We say that a closed term M is **typable** just if some type A such that $\vdash M : A$ is derivable in the type system.*

It's possible to talk about the typability of open terms too, by extending this definition, but a little less neat, so we will stick to closed terms. To show that a closed term M is typable requires us to exhibit a type and to show that the corresponding judgement is derivable.

Not all closed terms are typable. For example, the term $\lambda x.\, xx$ is not typable. To prove it, we will rely on analysing the possible shape of *any* typing derivation that has $\lambda x.\, xx$ in the conclusion.

Lemma 11.1. *The term $\lambda x.\, xx$ is untypable.*

Proof. We suppose $\lambda x.\, xx$ is typable and try to obtain a contradiction. Since $\lambda x.\, xx$ is typable, there is a type A such that $\vdash \lambda x.\, xx : A$ is derivable. Since this judgement is derivable, and its subject is an abstraction, the derivation must conclude using (TAbs) at the root:

$$\dfrac{\begin{array}{c}\vdots\\ x : B \vdash xx : C\end{array}}{\vdash \lambda x.\, xx : A}\ (\mathsf{TAbs})$$

but, for this to be a correct derivation, it must be that A has shape $B \to C$ for some monotypes B and C. Now consider the next part of this derivation, rooted at $x : B \vdash xx : C$. This can only be justified using the (TApp) rule:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ x : B \vdash x : D \to C\end{array} \quad \begin{array}{c}\vdots\\ x : B \vdash x : D\end{array}}{x : B \vdash xx : C}\ (\mathsf{TApp})}{\vdash \lambda x.\, xx : B \to C}\ (\mathsf{TAbs})$$

which means that there must be some type D such that the operator (on the left of the application) has type $D \to C$ and the operand (on the right of the application) has type D. In this case, both operator and operand are x. Both of the remaining judgements have x as the subject, and so must be justified using instances of (TVar):

$$\dfrac{\dfrac{\dfrac{}{x : B \vdash x : D \to C}\ (\mathsf{TVar}) \quad \dfrac{}{x : B \vdash x : D}\ (\mathsf{TVar})}{x : B \vdash xx : C}\ (\mathsf{TApp})}{\vdash \lambda x.\, xx : B \to C}\ (\mathsf{TAbs})$$

Although we don't know what B is exactly, we know it is a *monotype*, so there are no quantified type variables to be instantiated in B. Hence, for the left-hand instance of (TVar) to be a correct, it can only be that B and $D \to C$ are the same type. Similarly, from the right-hand instance of (TVar), it must be that B and D are the same type. Therefore, if this was a valid derivation, we would have that type

$D = B = D \to C$. This is impossible! A type is just a string, and there is no finite string D that is equal to a string $D \to C$ that properly contains it. □

These insights into what shape the various types have based on the form of a given judgement, which we used throughout the foregoing proof, are neatly summarised in the following *Inversion* theorem.

Theorem 11.1 (Inversion)**.** *Suppose $\Gamma \vdash M : A$ (is derivable), then:*

- *If M is a variable x, then there is a type scheme $\forall \overline{a}.B$ (with $\overline{a}$ possibly empty) and $A = B[\overline{C}/\overline{a}]$ for some monotypes $\overline{C}$.*
- *If M is an application PQ, then there is a type B such that $\Gamma \vdash P : B \to A$ and $\Gamma \vdash Q : B$.*
- *If M is an abstraction $\lambda x.P$, then there are types B and C such that $A = B \to C$, and $\Gamma, x : B \vdash P : C$.*

Proof. By inspection of the rules. □

I want to conclude this section by describing three of the main computational problems that are considered in connection to type systems. For simplicity, we give their definitions for the case of closed terms. The problems all extend to open terms, but the definitions are slightly messier.

- **Typability.** Given a closed term M, is M typable?
- **Type checking.** Given a closed term M and a type A, is $\vdash M : A$ derivable?
- **Type inference.** Given a closed term M, compute all types A such that $\vdash M : A$.

The three problems are all related, and it is likely that if one is decidable/computable for a given type system, then all of them are. For the simple type system that we study, all three are decidable/-computable.

A. Proof Rules

Implication

$A \Rightarrow B$	A implies B	If A then B.	Suppose A, then B.

Backward Rule

To prove $A \Rightarrow B$ starting from some assumptions, it suffices to instead prove B with A added to the assumptions.

Assms	Goal
...	$A \Rightarrow B$

"Assume A. We prove B as follows..."

"To show $A \Rightarrow B$, we assume A. ..."

"Assume A. ..."

"Suppose A is true. ..."

Assms	Goal
$\ldots, A$	B

Forwards Rule

If you already know $A \Rightarrow B$ and you already know A, then you know B.

Assms	Goal
$\ldots, A \Rightarrow B, A$	C

"From $A \Rightarrow B$ and A we conclude B."
"Applying $A \Rightarrow B$ to A we obtain B."
"We have B by *modus ponens*."

Assms	Goal
$\ldots, A \Rightarrow B, A, B$	C

Conjunction

$A \wedge B$ A and B

Backwards Rule

To prove $A \wedge B$ from some assumptions, it suffices to give two separate proofs: one proof of A from the assumptions and one proof of B from the assumptions.

Assms	Goal
$\ldots$	$A \wedge B$

"We prove A and B separately. To prove A..."
"For A, we argue as follows ... For B, ..."

Assms	Goal
$\ldots$	A

Assms	Goal
$\ldots$	B

The proof splits into two at this point, so you have to keep in mind two states, although you can just work with one at a time.

Forwards Rule

If you know $A \wedge B$, then you can know A and you know B.

Assms	Goal
$\ldots, A \wedge B$	C

"" (too trivial to mention)

Assms	Goal
$\ldots, A \wedge B, A, B$	C

Disjunction

$A \vee B$ A or B

Backwards Rules

To prove $A \vee B$ from some assumptions, it suffices to give a proof of A from those assumptions.

Assms	Goal
$\ldots$	$A \vee B$

"To see $A \vee B$, observe that A is true since..."
"We prove A."
"It suffices to prove A."

Assms	Goal
$\ldots$	A

To prove $A \vee B$ from some assumptions, it suffices to give a proof of B from those assumptions.

Assms	Goal
$\ldots$	$A \vee B$

"To see $A \vee B$, observe that B is true since..."
"We prove B."
"It suffices to prove B."

Assms	Goal
$\ldots$	B

Forwards Rule

If you have already know $A \vee B$ and, additionally, you can give the following two proofs:

- a proof of C starting from your assumptions and A
- a proof of C starting from your assumptions and B

Then you will also know C.

Assms	Goal
$\ldots, A \vee B$	C

"We proceed by cases on $A \vee B$.
Assume A ... Hence C.
Assume B ... Hence C."
"We analyse the two cases."
"We proceed by case analysis on $A \vee B$."

Assms	Goal
$\ldots, A \vee B, A$	C

Assms	Goal
$\ldots, A \vee B, B$	C

This forces the proof to split into two, and you're not done until you have completed both of them.

Negation

$\neg A$ not A A is false A is absurd

Backwards Rule

To prove $\neg A$ starting from some assumptions, it suffices to derive a contradiction (give a proof of false) starting from the same assumptions with A added.

Assms	Goal
$\ldots$	$\neg A$

"We assume A and try to obtain a contradiction."
"To show $\neg A$, we assume A..."
"Assume A."

Assms	Goal
$\ldots, A$	$\bot$

Forwards Rule

If you know $\neg A$ and you also know A, then you know absurdity.

Assms	Goal
$\ldots, \neg A, A$	C

"From A and $\neg A$ we obtain a contradiction."

Assms	Goal
$\ldots, \neg A, A, \bot$	C

If and only if

$\Leftrightarrow \equiv$ iff

Backwards Rule

An iff can be thought of as two implications, one in each direction. So to prove an iff it suffices to prove each separately.

Assms	Goal
$\ldots$	$A \Leftrightarrow B$

"In the forwards direction...
... in the backwards direction..."
"We prove each direction separately..."

Assms	Goal
$\ldots$	$A \Rightarrow B$

Assms	Goal
$\ldots$	$B \Rightarrow A$

Forwards Rule

Conversely, to use one, first just extract the two directions.

Assms	Goal
$\ldots, A \Leftrightarrow B$	C

"" (too trivial to mention)

Assms	Goal
$\ldots, A \Leftrightarrow B, A \Rightarrow B, B \Rightarrow A$	C

False

$\bot$ false absurdity

Forwards Rule

If you know absurdity, then you know everything.

Assms	Goal
$\ldots, \bot$	C

"Hence we obtain the desired result."
"*Ex falso quodlibet*"
"A follows trivially"

Assms	Goal
$\ldots, \bot, A$	C

Note, this A can be anything you like, including C.

Universal quantification

$\forall x \in X.\, A$	$\forall x : X.\, A$	for all x in X, A	A holds of all x in X

Backwards Rule

To give a proof of $\forall x \in X.\, A$ from some assumptions, it suffices to give a proof of A from the same assumptions with $x \in X$ added, as long as you have not made any prior assumptions about the name x (x doesn't appear in any of your existing assumptions).

Assms	Goal
...	$\forall x \in X.A$

with x not occurring free in the assms

"Let $x \in X$. We show A."
"Let x be an arbitrary member of X. We show A."
"Suppose x is in X. ... therefore A."

Assms	Goal
$\ldots, x \in X$	A

Forwards Rule

If you have know $\forall x \in X.\, A$ and you know $t \in X$, then you know A with all occurrences of x replaced by t.

Assms	Goal
$\ldots \forall x \in X.A\, t \in X$	C

"It follows that A holds of t"

Assms	Goal
$\ldots, \forall x \in X.A,\ t \in X, A[t/x]$	C

Existential Quantification

$\exists x \in X.\, A$	$\exists x : X.\, A$	there exists x in X such that A	A holds of some x in X

Backwards Rule

To prove $\exists x : X.\, A$ from some assumptions, it suffices to find a witness t and give proofs that $t \in X$ and of A with every occurence of x replaced by t, starting from the same assumptions.

Assms	Goal
...	$\exists x \in X.\, A$

"We show that A holds of t."
"We take t as witness. To see $A[t/x]$..."
"We show that t is such an x."

Assms	Goal
...	$t \in X$

Assms	Goal
...	$A[t/x]$

Here, $A[t/x]$ means A with all free occurrences of x replaced by t.

Forwards Rule

Assms	Goal
... $\exists x \in X.A$	C

with x not occurring free in the assms

"Let x be the witness..."

Assms	Goal
... $\exists x \in X.A\, x \in X\, A$	C

Equality

At any time, you can assume $s = s$, for any term s, which is obviously true.

Assms	Goal
...	C

"" (too trivial to mention)

Assms	Goal
$\ldots, s = s$	C

If you know an equation, you can use it to rewrite the goal.

Assms	Goal
$\ldots, s = t$	$C[s/x]$

"$C[s/x]$ is just $C[t/x]$"
"$C[s/x]$ is therefore the same as $C[t/x]$"

Assms	Goal
$\ldots, s = t$	$C[t/x]$

The use of $C[s/x]$ here is just a way of saying that you have some proposition which contains an occurrence of s and the corresponding $C[t/x]$ indicates replacing that occurrence of s by t.

If you know an equation, you can use it to rewrite an assumption.

Assms	Goal
$\ldots, s = t, A[s/x]$	C

"$C[s/x]$ is just $C[t/x]$"
"$C[s/x]$ is therefore the same as $C[t/x]$"

Assms	Goal
$\ldots, s = t, A[s/x], A[t/x]$	C