Elements of mathematics and logic for computer program analysis

Frédéric Blanqui

Elements of mathematics and logic for computer program analysis

Frédéric Blanqui (INRIA)

12 March 2013

These notes have been written for a 7-days school organized at the Institute of Applied Mechanics and Informatics (IAMA) of the Vietnamese Academy of Sciences and Technology (VAST) at Ho Chi Minh City, Vietnam, from Tuesday 12 March 2013 to Tuesday 19 March 2013. The mornings were dedicated to theoretical lectures introducing basic notions in mathematics and logic for the analysis of computer programs (these notes). The afternoons were practical sessions introducing the OCaml programming language and the Coq proof assistant (see the companion notes [11] and [10]).

Contents

1 Introduction

In order to be able to rigorously prove the correctness of a program, one must have a formal definition of:

- what is a program, syntactically;
- how it is evaluated, that is, what is its *semantics*;
- how to formulate the properties we are interested in;
- and how to prove them.

All this requires to understand some basic mathematical notions like induction, terms, formulas, deduction, etc. These notes are intended to give an introduction to some of these notions.

2 Induction and sequences

2.1 Induction on natural numbers

Induction is a very important tool in mathematics and computer science. We are going to see various induction principles in the course of these notes.

Let $\mathbb{N}$ be the set of natural numbers $0, 1, 2, \ldots$ and $P(x)$ be a property on natural numbers ($x \in \mathbb{N}$) (we will see a more formal definition of what P is later).

The induction principle on natural numbers is a way to prove that $P(x)$ holds for *all* natural number x, *i.e.* $(\forall x)P(x)$, by showing:

1. $P(x)$ holds for $x = 0$: $P(0)$;

2. $P(x)$ is preserved by taking the successor, *i.e.* if x is any natural number for which $P(x)$ holds (induction hypothesis), then $P(x+1)$ holds too: $(\forall x)P(x) \Rightarrow P(x+1)$ (*).

Indeed, since that $P(0)$ holds then, by (*), $P(1)$ holds and, by (*) again, $P(2)$ holds, etc.

Example 1 Let $\Sigma_{i=0}^{n} i$ be the sum of all integers i from 0 to $n \geq 0$, and assume that we want to prove that $\Sigma_{i=0}^{n} i = n(n+1)/2$. To do so, we can prove by induction on n the property $P(n) = ((\Sigma_{i=0}^{n} i) = n(n+1)/2)$.

1. We first check $P(0)$ holds: $\Sigma_{i=0}^{0} i = 0$ and $0(0+1)/2 = 0$.

2. Now, let k be any natural number such that $P(k)$ holds, *i.e.* $\Sigma_{i=0}^{k} = k(k+1)/2$ (induction hypothesis), and let's try to prove that $P(k+1)$ holds, *i.e.* $\Sigma_{i=0}^{k+1} i = (k+1)(k+2)/2$. We have $\Sigma_{i=0}^{k+1} i = \Sigma_{i=0}^{k} i + (k+1) = k(k+1)/2 + (k+1)$ by induction hypothesis, and $k(k+1)/2 + (k+1) = (k(k+1) + 2(k+1))/2 = (k+1)(k+2)/2$.

2.2 Words and sequences

Programs are texts or sequences of characters from some alphabet X (a set whose elements are called letters). How to define sequences of letters (words) formally?

We can see a word w of length $|w| = n \geq 0$ as a finite map $w : n \to X$, identifying 0 with the empty set $\emptyset$ and $n \geq 1$ with the set $\{0, 1, \ldots, n-1\}$. Then, $w(i)$ or w_i is the $(i+1)$-th character of w if $i < |w|$. We often write ε for the empty word.

Another definition is as follows. Assume that there is an (infinite) set $\mathcal{U}$ closed by pairing, *i.e.* if x and y are elements of $\mathcal{U}$, then the pair (x, y) is also an element of $\mathcal{U}$. Assume also that $\mathcal{U}$ contains all the letters ($X \subseteq \mathcal{U}$). Then, we can define the set of words on X, written X^*, as the *smallest* subset S of $\mathcal{U}$ that satisfies the properties (1) and (2) below, *i.e.* the intersection of the all the subsets $S \subseteq \mathcal{U}$ that satisfy the properties (1) and (2) below:

$$X^* = \bigcap \{S \subseteq \mathcal{U} \mid S \text{ satisfies (1) and (2)}\}.$$

where the properties (1) and (2) are:

1. the empty word ε belongs to S ($\varepsilon \in S$);

2. if x is a letter ($x \in X$) and w is a word ($w \in S$), then (x, w) is a word ($(x, w) \in S$).

Is X^* well defined? To this end, we need to check that there is at least one set $S \subseteq \mathcal{U}$ satisfying the properties (1) and (2). If we assume that $\varepsilon \in \mathcal{U}$ ($\mathcal{U}$ is not empty), then $\mathcal{U}$ itself satisfies (1) and (2) since it is closed by pairing.

Words are similar to chained lists in programming. In the following, we write xw instead of (x, w). For instance, if $X = \{a, b, c\}$, then $X^* = \{\varepsilon, a, b, c, aa, ab, abab c, \ldots\}$.

Why do we need to take the *smallest* subset? There are many sets satisfying (1) and (2), including $\mathcal{U}$ itself! But we want a word to be only empty or of the form (x, w) with x a letter and w another word, and $\mathcal{U}$ contains (infinitely) many elements that are not words (*e.g.* $(\varepsilon, \varepsilon)$).

2.3 A digression on set theory

Why do we need to consider a set $\mathcal{U}$? To avoid logical consistencies, you cannot form any set. Consider for instance the set x of all the sets y that do not belong to themselves ($y \notin y$): $x = \{y \mid y \notin y\}$. Do we have $x \in x$ or $x \notin x$? If $x \in x$ then, by definition of x, $x \notin x$, which is a contradiction. Otherwise, $x \notin x$. But, then, by definition of x, $x \in x$! This problem is called Russel's paradox [43] (but it has been found independently by Zermelo). To avoid it, the formation of sets is restricted:

- You have the empty set $\emptyset$.
- Given two sets x and y, you can form the set $\{x, y\}$ whose elements are x and y. If $x = y$, then $\{x, y\}$ is the singleton set $\{x\}$.
- More generally, given a set x, you can take the union of its elements: $\bigcup x = \bigcup_{y \in x} y = \{z \mid (\exists y) z \in y \wedge y \in x\}$. A particular case is the binary union: $x \cup y = \bigcup\{x, y\}$, which can be used to define the (ordered) (Kuratowski) pair $(x, y) = \{\{x\}, \{x, y\}\}$ (you can check that $(x, y) \neq (y, x)$ if $x \neq y$).
- Given a set x, you can also form the set of all its subsets $\mathcal{P}(x) = \{z \mid z \subseteq x\} = \{z \mid (\forall y) y \in z \Rightarrow y \in x\}$.
- But, given a property P on sets, you can form the set $\{z \mid P\}$ only if P is of the form $z \in x \wedge Q$, and we generally write $\{z \in x \mid Q\}$.

Is that all? Well, not exactly, but this is another story... However, it is enough to define many other standard constructions:

- The binary intersection $x \cap y = \{z \in x \mid z \in y\}$.
- The generalized intersection of a *non-empty* family $x \neq \emptyset$ of subsets of some set A: $\bigcap x = \bigcap_{y \in x} y = \{z \in A \mid (\forall y) y \in x \Rightarrow z \in y\}$.
- The (cartesian) binary product $x \times y = \{z \in \mathcal{P}(\mathcal{P}(x \cup y)) \mid (\exists a)(\exists b) z = (a, b) \wedge a \in x \wedge b \in y\}$ that is the set of pairs (a, b) made of an element $a \in x$ and an element $b \in y$.

How to get a set $\mathcal{U}$ closed by pairing? Starting from any set X, you can define $S^0(X) = X$ and $S^{n+1}(X) = S^n(X) \cup \mathcal{P}(S^n(X))$. Then, let $\mathcal{U} = S(X) = \bigcup_{i \in \mathbb{N}} S^i(X)$.

2.4 Induction on words

Now, how can we reason about words? Let for instance $P(w)$ be a property on words. How to prove that $P(w)$ holds for all words $((\forall w)P(w))$? We could use the induction principle on natural numbers by considering the length of the word, that is, we could prove by induction on $n \in \mathbb{N}$ the property:

$$Q(n) = (\forall w \in X^*)|w| = n \Rightarrow P(w).$$

We can also remark that, $P(w)$ holds for all words iff (if and only if):

$$\{w \in X^* \mid P(w)\} = X^*,$$

that is, iff P satisfies the following properties (**Exercise 1**):

(P1) $P(\varepsilon)$ holds

(P2) if x is a letter and w is a word such $P(w)$ holds (induction hypothesis), then $P((x, w))$ holds.

This is the induction principle on words. This easily extends to any set defined as the *smallest* set satisfying some properties.

Note that this gives a justification to the induction principle on natural numbers by seeing $\mathbb{N}$ as the smallest set $S \subseteq \mathcal{U}$ such that:

1. $0 \in S$,
2. if $x \in S$, then $x + 1 \in S$.

Now, how to define sets, relations or functions on words? For instance, the concatenation $u \circ v$ of two words u and v? In this case, we would like to proceed as follows:

- If $u = \varepsilon$, then $u \circ v = v$.
- Otherwise, there are x and u' such that $u = (x, u')$. Then, we would like to say that $u \circ v = (x, u' \circ v)$.

But why is it valid? A function is nothing but a relation, which is nothing but a set of pairs. And a relation R is a function if it satisfies the following property: if x is in relation with both y and y' (xRy and xRy'), then $y = y'$ (*), that is, an element x in the domain of R is mapped to a unique element that we then usually write $R(x)$. Now, we could define $\circ$ as the smallest set S such that:

- $((\varepsilon, v), v) \in S$,
- if $((u', v), w) \in S$ then, for all $x \in X$, $((xu', v), xw) \in S$.

We remark that the two cases are disjoint ($u = \varepsilon$ and $u = (x, u')$), cover all possible cases and that $(x, u') \circ v$ is defined from $u' \circ v$, like in the induction principle $P((x, u'))$ is proved from $P(u')$. In the following, we will directly admit such inductive definitions.

Before moving to another subject, note that $(X^*, \circ)$ is a semi-group (**Exercise 2**), that is:

- it admits ε as neutral element: $\varepsilon \circ v = v \circ \varepsilon = v$,
- and it is associative: $(u \circ v) \circ w = u \circ (v \circ w)$.

Hence, we do not need to write parentheses. So, in the following, we will simply write concatenation by juxtaposition, that is, uv instead of $u \circ v$.

2.5 Grammar rules and string rewriting

Now, programs are not arbitrary character sequences. There are grammatical rules to respect for a program to be accepted by a computer. For instance, arithmetical expressions are for instance defined (not counting spaces and comments) as a number or two arithmetical expressions around an infix operator symbol. This can be described by giving generating or production rules:

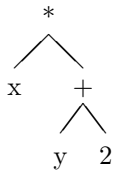
$$\begin{array}{lcl} E & \to & N \\ E & \to & E\ O\ E \\ E & \to & (\ E\ ) \\ O & \to & + \\ O & \to & * \\ N & \to & D\ N \\ D & \to & 0 \\ D & \to & 1 \end{array}$$
$$\ldots$$

All syntactically correct programs can then be generated by following these rules. For instance, $E \to EOE \to NOE \to NON \to 1ON \to 1 + N \to 1 + 1$. This can be defined more formally by considering the notion of string rewrite system. A string rewrite rule is a pair of words (l, r), written $l \to r$. Given a set $\mathcal{R}$ of string rewrite rules, we say that a word w rewrites to another word w' if there is a rule $l \to r \in \mathcal{R}$ and two words w_1 and w_2 (the rule context) such that $w = w_1 l w_2$ and $w' = w_1 r w_2$. In other words, if a word contains the left-hand side of a rule $l \to r \in \mathcal{R}$, then it can be rewritten by replacing l by r. Note that, *a priori*, a word can be rewritten at different positions and with different rules.

3 Terms

3.1 Definition of terms

We have seen how to formalize the notion of text, possibly following some grammar. But it is almost impossible to define the semantics (meaning) of a program if we consider a program as merely a sequence of characters. In general, space, new lines, comments, etc. are not meaningful. It is therefore necessary to abstract these syntactic details away. This is the purpose of the notion of *abstract syntax tree* or *term* to make the grammatical structure apparent. For instance, the text "x*(y+2)", which is the sequence of characters 'x', '*', '(', 'y', '+', 2' and ')', corresponds to the term:



A term is a tree (connected graph with no cycle) whose nodes are labeled. Again, a term can be defined as a map from the set of nodes to the set of labels. But it is more convenient to define the set of terms inductively as follows. Given a set $\mathcal{F}$ of symbols, let the set of terms over $\mathcal{F}$ be the *smallest* set (included in $\mathcal{U}$[1]) such that:

- if $\mathsf{f} \in \mathcal{F}$ is a symbol and $t_1, \ldots, t_n$ is a sequence of terms, then $(\mathsf{f}, t_1, \ldots, t_n)$ is a term that we usually write $\mathsf{f}(t_1, \ldots, t_n)$ and simply f if $n = 0$.

What is the induction principle on terms? Proceeding as before, we get that a property P holds for all term t if the following property is satisfied:

- if $\mathsf{f} \in \mathcal{F}$ is a symbol and $t_1, \ldots, t_n$ is a sequence of terms such that $P(t_1)$, ..., $P(t_n)$ hold, then $P(\mathsf{f}(t_1, \ldots, t_n))$ holds.

Note that there is (apparently) no base case!... In fact, there is one hidden: when $n = 0$. In this case, we have: if $\mathsf{f} \in \mathcal{F}$ is a symbol, then $P(\mathsf{f})$ holds.

The task of converting a sequence of characters into a tree is called *parsing*. There are tools (*e.g.* `yacc` [12]) that generate a parser from a grammar. For more details about that and compilation, see for instance [1, 2].

3.2 Knaster-Tarski's fixpoint theorem

We are going to see other ways to understand/define the set of terms.

Exercise 3 Consider the function $f : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ such that $f(X) = X \cup \{\mathsf{f}(t_1, \ldots, t_n) \mid \mathsf{f} \in \mathcal{F}, t_1, \ldots, t_n \in X\}$. Prove that the set $\mathcal{T}$ of terms over

[1] We will not mention it anymore.

$\mathcal{F}$ is a fixpoint of f, *i.e.* $f(\mathcal{T}) = \mathcal{T}$, and prove also that f is *monotone*, *i.e.* $f(X) \subseteq f(Y)$ if $X \subseteq Y$.

Exercise 4 Prove that, for any set $\mathcal{U}$, the set $\mathcal{P}(\mathcal{U})$ of all subsets of $\mathcal{U}$, is a *complete lattice for set inclusion*, that is, any subset $\mathcal{E}$ of $\mathcal{P}(\mathcal{U})$ has a *least upper bound* $\text{lub}(\mathcal{E}) \in \mathcal{P}(\mathcal{U})$ and a *greatest lower bound* $\text{glb}(\mathcal{E}) \in \mathcal{P}(\mathcal{U})$ both wrt to set inclusion, where $\text{lub}(\mathcal{E})$ is a least upper bound of $\mathcal{E}$ if:

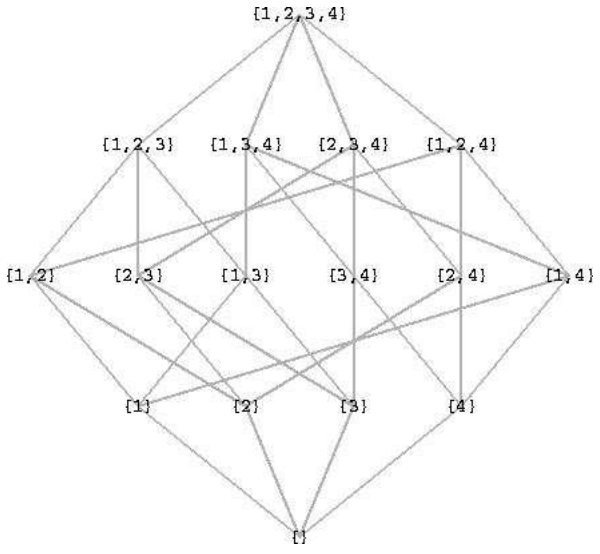
- $\text{lub}(\mathcal{E})$ is an upper bound of $\mathcal{E}$: $\text{lub}(\mathcal{E})$ is bigger wrt set inclusion than every element of $\mathcal{E}$, *i.e.* every element of $\mathcal{E}$ is included in $\text{lub}(\mathcal{E})$;
- $\text{lub}(\mathcal{E})$ is smaller wrt set inclusion than any upper bound of $\mathcal{E}$: if U is an upper bound of $\mathcal{E}$, *i.e.* if every element of $\mathcal{E}$ is included in U, then $\text{lub}(\mathcal{E})$ itself is included in U;

and, dually (change $\subseteq$ to $\supseteq$), $\text{glb}(\mathcal{E})$ is a greatest lower bound of $\mathcal{E}$ if:

- $\text{glb}(\mathcal{E})$ is a lower bound of $\mathcal{E}$: $\text{glb}(\mathcal{E})$ is smaller wrt set inclusion than every element of $\mathcal{E}$, *i.e.* $\text{glb}(\mathcal{E})$ is included in every element of $\mathcal{E}$;
- $\text{glb}(\mathcal{E})$ is bigger wrt set inclusion than any lower bound of $\mathcal{E}$: if L is a lower bound of $\mathcal{E}$, *i.e.* L is included in every element of $\mathcal{E}$, then L is included in $\text{glb}(\mathcal{E})$ itself.

Finally, prove that $\text{lub}(\mathcal{P}(\mathcal{U})) = \text{glb}(\emptyset) = \mathcal{U}$ is the greatest/biggest element of $\mathcal{P}(\mathcal{U})$ and that $\text{glb}(\emptyset) = \text{lub}(\mathcal{P}(\mathcal{U})) = \emptyset$ is the lowest/smallest elements of $\mathcal{P}(\mathcal{U})$.

Example: the Hasse diagram of the lattice $\mathcal{P}(\{1,2,3,4\})$[2]:



[2]Picture taken from `http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/hogg96a-html/node4.html`, HTML version of *Quantum Computing and Phase Transitions in Combinatorial Search*, Tad Hogg (Xerox Palo Alto Research Center), in *Journal of Artificial Intelligence Research* 4:91-128, 1996.

Knaster and Tarski proved in 1927 that, for any set $\mathcal{U}$, *any* monotone function $f : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ admits a fixpoint, *i.e.* there is $X \subseteq \mathcal{U}$ such that $f(X) = X$ [36]. Later, Tarski extended this to any complete lattice, *i.e.* any set $\mathcal{U}$ equipped with an ordering relation $\leq$ and having arbitrary lub's and glb's [45].

3.3 Kleene's fixpoint theorem

Exercise 5 Let f^0 be the identity on $\mathcal{P}(\mathcal{U})$ and $f^{n+1} = f \circ f^n$. Prove that $\mathcal{T} = \bigcup_{n\in\mathbb{N}} f^n(\emptyset)$ and that f is *ω-continuous*, that is, for all $\mathbb{N}$-indexed family $(S_i)_{i\in\mathbb{N}}$ of elements of $\mathcal{P}(\mathcal{U})$, $f(\bigcup_{i\in\mathbb{N}} S_i) = \bigcup_{i\in\mathbb{N}} f(S_i)$.

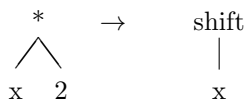
Lattice theory and Knaster, Tarski and Kleene fixpoint theorems [34] are the basis of *abstract interpretation*, a powerful technique for detecting common bugs in programs invented by Patrick and Radhia Cousot in 1977 [17]. See for instance the web page of the Astrée software: "In Nov. 2003, Astrée was able to prove completely automatically the absence of any run-time error (RTE) in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C analyzed in 1h20 on a 2.8 GHz 32-bit PC using 300 Mb of memory."

More can be found on lattice theory for instance in [9]. For abstract interpretation, see for instance [18] and the web page of Patrick Cousot.

3.4 Pattern matching and term rewriting

Now that we have a formal representation of a program as a term, how can we formalize program transformations? For instance, before compiling a program into binary code, it may be interesting to transform it in order to do some optimizations. Assume for instance that a program contains an arithmetic expression *of the form* $t \times 2$. Then, we can simply replace it by $\mathsf{shift}(t)$, which is more efficient.

How to formalize such term transformations? Forms of terms or *patterns* can be defined as terms with holes that can be replaced by any term and, to refer to these terms, we can use *term variables*:



For representing patterns, we therefore extend the algebra of terms with some set $\mathcal{X}$ of (term) *variables*. A term or pattern is now either a variable $x \in \mathcal{X}$ or a symbol $\mathsf{f} \in \mathcal{F}$ applied to some sequence of terms $t_1, \ldots, t_n$. We say that a term is *closed* or *ground* if it contains no variable. A map from variables to terms is called a *substitution*. Applying a substitution σ to a term t consists in replacing in t every variable x by $\sigma(x)$. This gives the map $\widehat{\sigma}$ from terms to terms such that $\widehat{\sigma}(x) = \sigma(x)$ and $\widehat{\sigma}(\mathsf{f}(t_1, \ldots, t_n)) = \mathsf{f}(\widehat{\sigma}(t_1), \ldots, \widehat{\sigma}(t_n))$. But, for

the sake of simplicity, we often write σ instead of $\widehat{\sigma}$. Hence, a (closed) term t *matches* a pattern p if there is a substitution σ such that $t = \sigma(p)$.

Many term transformation can then be represented by extending string rewriting to terms [37]. Let a term rewrite rule be a pair of terms (l, r) written $l \to r$. Given a set $\mathcal{R}$ of rewrite rules, a term t rewrites to a term t', written $t \to_{\mathcal{R}} t'$, if there are a term C (the context) with a unique occurrence of some variable x, a rule $l \to r \in \mathcal{R}$ and a substitution σ such that $t = C\{x \mapsto \sigma(l)\}$ and $t' = C\{x \mapsto \sigma(r)\}$.

Exercise 6 Define a function match which, given a pattern p and a term t, returns $\perp$ (fails) if t does not match p, or some substitution σ such that $t = \sigma(p)$ otherwise. Prove its correctness. What is its complexity wrt (with respect to) the size of its inputs?

Exercise 7 Prove that variable names are not significant in rules. Let $l \to r \in \mathcal{R}$ be a rule, π be a permutation on variables, and let $\mathcal{S} = \mathcal{R} \cup \{\pi(l) \to \pi(r)\}$. Then, the two relations $\to_{\mathcal{R}}$ and $\to_{\mathcal{S}}$ are equivalent.

More on term rewriting can be found for instance in [6, 46].

3.5 Models of a term algebra

In general, a term has no meaning in itself although, in practice, we often give informative names to symbols and variables. We can give some meaning or *semantics* to a term by interpreting its variables in some particular set and its function symbols by operations in this set. For the sake of simplicity, assume that a function symbol f is always applied to the same *fixed* number $\alpha_{\mathsf{f}} \geq 0$ of arguments called its *arity*. A model of the term algebra over $\mathcal{F}$ is given by a set A and, for each function symbol f of arity n, a function $\mathsf{f}_A : A^n \to A$. The interpretation of a term t in A wrt a *valuation* map $\mu : \mathcal{X} \to A$, written $[\![t]\!]_\mu$, can then be defined inductively as follows: $[\![x]\!]_\mu = \mu(x)$ and $[\![\mathsf{f}(t_1, \ldots, t_n)]\!]_\mu = \mathsf{f}_A([\![t_1]\!]_\mu, \ldots, [\![t_n]\!]_\mu)$. Remark how term substitution is a particular case of term interpretation (**Exercise 8**).

In the following, we will only consider function symbols of fixed arity. More general notions of terms and patterns can otherwise be considered. For instance, in XML, function symbols are called of flexible arity since they can be applied to any number of arguments, and the order of arguments is not significant. In Xpath, a notion of pattern and pattern-matching is defined that is more complicated than the one for terms.

Exercise 9 Prove that, for all term t, substitution σ and valuation μ, we have $[\![\sigma(t)]\!]_\mu = [\![t]\!]_{\mu \circ \sigma}$.

4 Lambda-calculus

4.1 Definition of λ-calculus

We are going to see a notation for representing arbitrary "computable" functions introduced by Alonzo Church at the end of 1920's [13, 16]. The idea is simple. It consists in considering the term algebra over the set of symbols made of the symbol app of arity 2 for function application and, for each variable x, of the symbol abs_x of arity 1 for abstraction (function formation). A term $\mathsf{app}(t, u)$ is intended to represent the application of the function t to an argument u and is usually simply written by juxtaposition tu. A term $\mathsf{abs}_x t$ is intended to represent the function mapping x to t and is usually written λxt. Hence, λ-terms are usually defined as follows:

$$t = x \mid \lambda xt \mid tt$$

Then, the only computation rule, called β-reduction, consists in, when applying an abstraction λxt to a term u, replacing by u every occurrence of x in t:

$$(\lambda xt)u \to_\beta t\{x \mapsto u\}$$

However, substitution has to be defined differently than previously. Indeed, in some cases, we need to rename variables bound by a λ to avoid that a variable that was not bound becomes bound (variable capture). Consider for instance the case where $t = \lambda yx$ and $u = y$. The term λxt is intended to represent the function that maps x to the function that maps y to x which, in some sense, is equivalent to the function that maps (x, y) to x. And we apply it to an argument called y. Informally, we know that the function that maps (x, y) to x is the same as the one that maps (x', y') to x': variable names are not significant. So, applied to $x' = y$, we should get the function that maps y' to y, *i.e.* the constant function equal to y. But, if we do not do this renaming, we get the function that maps y to y, *i.e.* the identity function.

We therefore need to define substitution carefully and consider that two terms equal after renaming of their bound variables are equivalent (a relation called α-equivalence, α-conversion or α-congruence). Then, whenever it is necessary, the bound variables of a λ-term can be assumed disjoint from any finite set (assuming that the set $\mathcal{X}$ of variables is infinite) and that functions and properties used on this λ-term are indeed invariant (stable) by α-equivalence.

Here is a possible definition due to Curry and Feys [19]. They first define what are the *free* (*i.e.* unbound) variables in a term t:

- $\mathrm{FV}(x) = \{x\}$
- $\mathrm{FV}(tu) = \mathrm{FV}(t) \cup \mathrm{FV}(u)$
- $\mathrm{FV}(\lambda xt) = \mathrm{FV}(t) - \{x\}$

Then, assuming an enumeration of variables, *i.e.* a bijection $x : \mathbb{N} \to \mathcal{X}$, they define the substitution of x by v in a term t, written $t\{x \mapsto v\}$:

- $x\{x \mapsto v\} = v$
- $y\{x \mapsto v\} = y$ if $x \neq y$
- $(tu)\{x \mapsto v\} = (t\{x \mapsto v\})(u\{x \mapsto v\})$
- $(\lambda xt)\{x \mapsto v\} = \lambda xt$
- $(\lambda yt)\{x \mapsto v\} = \lambda z((t\{y \mapsto z\})\{x \mapsto v\})$ if $x \neq y$, where:
  - $z = y$ if $x \notin \mathrm{FV}(\lambda yt)$ and $y \notin \mathrm{FV}(v)$
  - $z = x_i$ where i is the smallest integer such that $x_i \notin \mathrm{FV}(\lambda yt) \cup \mathrm{FV}(v)$.

One can extend this definition to the simultaneous substitution σ of a finite number of variables $x_1, \ldots, x_n$ by the corresponding terms $v_1, \ldots, v_n$. In this case, let $\mathrm{FV}(\sigma) = \mathrm{FV}(v_1) \cup \ldots \cup \mathrm{FV}(v_n)$.

Finally, they define α-equivalence as the smallest relation $=_\alpha$ on terms such that:

- $\lambda xt =_\alpha \lambda y(t\{x \mapsto y\})$ if $y \notin \mathrm{FV}(t)$
- $t =_\alpha t'$ and $u =_\alpha u'$ implies $tu =_\alpha t'u'$
- $t =_\alpha t'$ implies $\lambda xt =_\alpha \lambda xt'$
- $t =_\alpha t$
- $t =_\alpha u$ implies $u =_\alpha t$
- $t =_\alpha u$ and $u =_\alpha v$ implies $t =_\alpha v$

Exercise 10 Prove that FV is invariant by α-equivalence.

Exercise 11 We define the set of *immediate subterms* of t as follows: $\mathrm{Sub}(x) = \emptyset$, $\mathrm{Sub}(\lambda xt) = \{t\}$ and $\mathrm{Sub}(tu) = \{t, u\}$. Prove that Sub is not stable by α-equivalence.

4.2 Church-computable functions

Surprisingly, this language is powerful enough to express arbitrary computable functions. This is due to the fact that there is no restriction on abstraction and application. For instance, a variable can be applied to itself! So, the following λ-terms are valid: xx, $(\lambda xx)(\lambda xx)$, λxxx, $(\lambda xxx)(\lambda xxx)$, $(\lambda xy(xx))(\lambda xy(xx))$.

Exercise 12 What are the β-reducts of these terms, *i.e.* the terms that we can get by β-rewriting them as long as possible?

But what does "computable" mean after all? Difficult question... Well, historically, λ-calculus was the first attempt to give a mathematical definition of the intuitive notion of "computability". This has been done by Kleene, a student of Church, in his PhD in 1934 [33]. Soon later, other definitions have been proposed and, in particular, Turing's definitions in 1936 [47], but they all have been proved equivalent to Church λ-calculus. Kleene himself provided an alternative definition called the "μ-recursive functions". That's why most people think that we found the right mathematical definition of the intuitive notion of "computability" (Church-Turing's thesis). To know more on the history of the notion of computability, see for instance [35].

But how to represent numbers in λ-calculus? The idea is to use function iteration: we represent $n \in \mathbb{N}$ by the λ-term $\widehat{n} = \lambda f \lambda x f(f(\ldots(fx)))$ (Church's numeral) where f is applied n times to x. So, $\widehat{0} = \lambda f \lambda x x$, $\widehat{1} = \lambda f \lambda x f x$, $\widehat{2} = \lambda f \lambda x f(fx)$, etc. Then, for instance, we can define the addition on Church's numerals: $\widehat{+} = \lambda p \lambda q \lambda f \lambda x p f(qfx)$.

Exercise 13 Check that $\widehat{+}\widehat{2}\widehat{2}$ rewrites (or reduces) in zero or more β-steps into $\widehat{4}$, written $\widehat{+}\widehat{2}\widehat{2} \to_\beta^* \widehat{4}$.

Exercise 14 Define multiplication on Church's numerals, *i.e.* define a λ-term $\widehat{\times}$ such that, for all $p, q \in \mathbb{N}$, $\widehat{\times}\widehat{p}\widehat{q} \to_\beta^* \widehat{p \times q}$.

Exercise 15 Define a λ-term ifzero such that $\mathsf{ifzero}\, tuv \to_\beta^* u$ if $t \to_\beta^* \widehat{0}$, and $\mathsf{ifzero}\, tuv \to_\beta^* v$ otherwise.

Exercise 16 How to represent pairs? Define λ-terms pair, π_1 and π_2 such that, for all λ-terms u and v, $\pi_1(\mathsf{pair}\, uv) \to_\beta^* u$ and $\pi_2(\mathsf{pair}\, uv) \to_\beta^* v$.

A total function $f : \mathbb{N}^n \to \mathbb{N}$ ($n \geq 0$), *i.e.* a function that is defined on all elements of $\mathbb{N}^n$, is Church-computable (or λ-definable) if there is a λ-term $\widehat{f}$ such that, for all $k_1, \ldots, k_n \in \mathbb{N}$, $\widehat{f}\widehat{k_1}\ldots\widehat{k_n} \to_\beta^* \widehat{f(k_1, \ldots, k_n)}$. More generally, a partial function $f : \mathbb{N}^n \to \mathbb{N}$ is Church-computable (or λ-definable) if there is a λ-term $\widehat{f}$ such that, for all $(k_1, \ldots, k_n)$ in the domain of f, $\widehat{f}\widehat{k_1}\ldots\widehat{k_n} \to_\beta^* \widehat{f(k_1, \ldots, k_n)}$ and, for all $(k_1, \ldots, k_n)$ not in the domain of f, $\widehat{f}\widehat{k_1}\ldots\widehat{k_n}$ does not terminate, *i.e.* can be β-reduced for ever.

But what about functions on other data structures (lists, trees, graphs, etc.)? Well, once we can encode pairs, we can encode any finite data structure. The empty list can be encoded by 0 and a non empty list $(x_1, \ldots, x_n)$ can be encoded by the pair $(x_1, (x_2, \ldots (x_n, 0)\ldots))$. Etc.

More on pure λ-calculus can be found for instance in [7, 38]. Examples of programming languages based on pure λ-calculus: Lisp, Scheme.

4.3 Kleene-computable functions

Kleene defines the set of (partial) *recursive* or computable functions as the smallest set of functions from $\mathbb{N}^n$ to $\mathbb{N}$ (for any $n \geq 0$) containing:

- the projection functions: $(x_1, \ldots, x_n) \mapsto x_i$ for all $i \in [1, n]$;
- the constant functions equal to 0: $(x_1, \ldots, x_n) \mapsto 0$;
- the successor function: $x \mapsto x + 1$;

and closed by:

- composition: if $h : \mathbb{N}^m \to \mathbb{N}$ is computable and $g_1 : \mathbb{N}^n \to \mathbb{N}$, ..., $g_m : \mathbb{N}^n \to \mathbb{N}$ are computable, then $(x_1, \ldots, x_n) \mapsto h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ is computable;
- recursion: if $g : \mathbb{N}^n \to \mathbb{N}$ is computable and $h : \mathbb{N}^{n+2} \to \mathbb{N}$ is computable, then the function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ such that:
  - $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$,
  - $f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$,

  is computable;
- minimization: if $g : \mathbb{N}^{n+1} \to \mathbb{N}$ is computable, then the function $f : \mathbb{N}^n \to \mathbb{N}$ such that $f(x_1, \ldots, x_n)$ is the least integer $y \in \mathbb{N}$ such that $g(x_1, \ldots, x_n, y) = 0$ (this is a partial function if there is no such y), is computable.

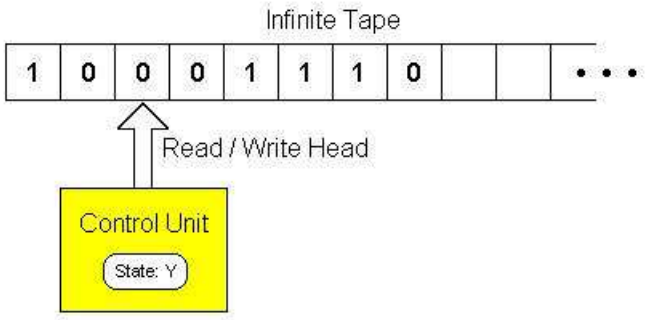
Exercise 17 Given three functions $f, g, h : \mathbb{N}^n \to \mathbb{N}$, let $\mathsf{ifzero}^n_{f,g,h} : \mathbb{N}^n \to \mathbb{N}$ be the function such that $\mathsf{ifzero}^n_{f,g,h}(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$ if $f(x_1, \ldots, x_n) = 0$, and $\mathsf{ifzero}^n_{f,g,h}(x_1, \ldots, x_n) = h(x_1, \ldots, x_n)$ otherwise. Show that addition, multiplication and ifzero^n are Kleene-computable.

But what about functions on other data structures (lists, trees, graphs, etc.)? Well, once we can encode pairs of natural numbers, we can encode any finite data structure. So, how to encode pairs? It suffices to find a computable bijection from $\mathbb{N}^2$ to $\mathbb{N} - \{0\}$, whose inverses π_1 and π_2 are computable too. Take for instance the function

$$p; q = (p + q)(p + q + 1)/2 + p + 1$$

Using this function, any pair of numbers (p, q) can be seen as a number $p; q$, and any number $l \neq 0$ can be seen as pair of numbers (p, q). And once we can encode pairs, we can encode lists, trees, etc.

4.4 Turing-computable functions



Turing's definition looks more like an automaton, a discrete dynamical system, or a computer. A (multi-tape) Turing machine[3] is defined by:

- a finite set Σ of symbols (alphabet) containing at least the following three distinct symbols: $\square$ (start), 0 (blank) and 1;
- a number $k \geq 1$ of (memory) *tapes*;
- a finite set Q of (control) *states* containing at least two different states q_i (initial) and q_f (final);
- a finite (partial) transition function (program) $\delta : \Sigma^k \times (Q - \{q_f\}) \to \Sigma^k \times Q \times \{-1, 0, 1\}$ describing what the machine should do at each step.

At each moment, the configuration of a Turing-machine is then given by:

- the symbols written on the tapes, *i.e.* for each tape i, a function $m_i : \mathbb{N} \to \Sigma$;
- the position $h \in \mathbb{N}$ of the head;
- its state $q \in Q$.

We now describe how a Turing machine $M = (\Sigma, k, Q, q_i, q_f, \delta)$ evolves. Assume that at time $t \in \mathbb{N}$, it is in the configuration $c_t = (m, h, q)$ and that $\delta((m_1(h), \ldots, m_k(h)), q) = ((b_1, \ldots, b_k), q', r)$. Then, at time $t + 1$, it moves to the configuration $c_{t+1} = (m', h + r, q')$ (we consider discrete times in $\mathbb{N}$) with $m'_i(h) = b_i$ and $m'_i(l) = m_i(l)$ if $l \neq h$, that is, $m_i(h)$ is replaced by b_i, q is replaced by q', and the head is moved to the left if $r = -1$ and $h > 0$, to the right if $r = 1$, and stay at the same position otherwise. Note that, if $q' = q_f$ then the machine cannot evolve anymore: it stops.

[3]Picture taken from `http://science.slc.edu/~jmarshall/courses/2002/fall/cs30/Lectures/week08/Computation.html`, a web page of Jim Marshall (Sarah Lawrence College, Bronxville, NY, USA).

Then a function $f : \mathbb{N}^n \to \mathbb{N}$ is Turing-computable if there is a Turing-machine $M = (\Sigma, k, Q, q_i, q_f, \delta)$ such that $k \geq n+1$ and, for all tuple $(k_1, \ldots, k_n) \in \mathbb{N}^n$, starting from the initial configuration $c_0 = (m, 0, q_i)$ where $m_i(0) = \Box$ if $i \in [1, n+1]$, $m_i(l) = 1$ if $i \in [1, n]$ and $l \in [1, k_i]$, and $m_i(l) = 0$ otherwise:

- if $f(k_1, \ldots, k_n)$ is defined then the machine stops in configuration $(m', 0, q_f)$ where $m'_{n+1}(0) = \Box$, $m'_{n+1}(l) = 1$ if $l \in [1, f(k_1, \ldots, k_n)]$, and $m'_{n+1}(l) = 0$ otherwise;
- if $f(k_1, \ldots, k_n)$ is undefined then the machine never reaches state q_f.

Exercise 18 Prove that Kleene-computability implies Turing-computability.

Exercise 19 Prove that Turing-computability implies Kleene-computability.

An important question about computability is the following: is there a total function (*i.e.* defined on all inputs) that is not computable? The answer is yes. In particular, Turing proved that the total boolean function $\mathsf{halt} : \mathbb{N}^2 \to \mathbb{N}$ saying if a Turing machine halts on some input (where $\mathsf{false} = 0$ and $\mathsf{true} = 1$) is not computable. Indeed, a Turing machine M can be encoded by a natural number $\widehat{M}$ so that any natural number n corresponds to a Turing machine. Then, $\mathsf{halt}(x, y)$ returns true if the Turing machine corresponding to x halts on input y, and returns false otherwise. We now prove that halt is not computable by showing that it is different from any possible computable function $f : \mathbb{N}^2 \to \mathbb{N}$. Given any computable function $f : \mathbb{N}^2 \to \mathbb{N}$, let $G_f : \mathbb{N} \to \mathbb{N}$ be the function such that $G_f(x) = \mathsf{false}$ if $f(x, x) = \mathsf{false}$, and $G_f(x)$ is undefined otherwise. Since G_f is computable, there is a Turing machine M_{G_f} that computes it. Let then $k = \widehat{M_{G_f}}$ be the natural number representing it. We now prove that $f(k, k) \neq \mathsf{halt}(k, k)$. There are two cases:

- $f(k, k) = \mathsf{false}$. Then, $G_f(k) = \mathsf{false}$ and, by definition, $\mathsf{halt}(k, k) = \mathsf{true}$.
- $f(k, k)$ is undefined or $f(k, k) \neq \mathsf{false}$. Then, $G_f(k)$ is undefined and, by definition, $\mathsf{halt}(k, k) = \mathsf{false}$.

Turing machines provide also a way to measure and compare the time and memory needed to compute a function. You can find more information about this subject in textbooks on computational complexity like [39, 44, 5].

5 Simply-typed lambda-calculus

5.1 Curry-style simply-typed λ-calculus

In this section, we are going to see some restriction of λ-calculus based on the use of *types*. Types like `int`, `string`, `float`, etc. are common in programming. This is an important tool to avoid some programming errors. But they have been first introduced by logicians in order to avoid inconsistencies. Indeed, since

there is no restriction on abstraction and application, we can encode Russell's paradox in λ-calculus: if you represent set formation by λ-abstraction (λxt is the set of x's satisfying t) and membership by application (tu means that $u \in t$), and take $\delta = \lambda x \neg(xx)$, then we have $\delta\delta =_\beta \neg(\delta\delta)$.

In the *simple type* discipline, we assume given a non-empty set $\mathcal{B}$ of type constants (*e.g.* `int`, ...). Then, a type is either a type constant $\mathsf{B} \in \mathcal{B}$ or an arrow or function type $U \to V$ where U and V are types, representing the type of functions from U to V. Hence, a type is nothing but a *term* on the set of symbols $\mathcal{F} = \mathcal{B} \cup \{\to\}$ with type constants of arity 0, $\to$ of arity 2 and $\to (T, U)$ written $T \to U$. For instance, $\mathsf{B} \to \mathsf{B}$ is the type of functions that take as argument a value of type B and return a value of type B; $\mathsf{B} \to (\mathsf{B} \to \mathsf{B})$, also written $\mathsf{B} \to \mathsf{B} \to \mathsf{B}$ (assuming that $\to$ "associates" to the right) is the type of functions that take two arguments of type B and return a value of type B; $(\mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ if the type of functions that take a function of type $\mathsf{B} \to \mathsf{B}$ as argument and return a value of type B.

Then, a λ-term of type $U \to V$ can only be applied to an argument of type U. For instance, a function of type `int` $\to$ `int` cannot be applied to a value of type `float`. Formally, we represent this by inductively defining a subset $\rhd$ of the set $\mathcal{E} \times \mathcal{L} \times \mathcal{S}$ where $\mathcal{L}$ is the set of λ-terms, $\mathcal{S}$ is the set of simple types, and $\mathcal{E}$ is the set of finite maps from $\mathcal{X}$ to $\mathcal{S}$, declaring what are the types of variables, and we write $E \rhd t : T$ if $(E, t, T) \in \rhd$, and say that t has type T in typing environment E:

$$\text{(var)}\ \frac{(x, T) \in E}{E \rhd x : T}$$

$$\text{(abs)}\ \frac{E \cup \{(x, U)\} \rhd v : V \quad x \notin \mathrm{dom}(E)}{E \rhd \lambda xv : U \to V}$$

$$\text{(app)}\ \frac{E \rhd t : U \to V \quad E \rhd u : U}{E \rhd tu : V}$$

This means that $\rhd$ is the smallest set of tuples (E, t, T) such that:

(var) A variable x has type T in typing environment E if x is mapped to T in E, *i.e.* is declared to have type T in E.

(abs) An abstraction λxv has type $U \to V$ in typing environment E if, assuming that x has no declared type in E (which can always be done by α-conversion), the body v has type V in typing environment E extended by declaring x of type U.

(app) An application tu has type V in typing environment E if there is a type U such that t is of type $U \to V$ in E, and u is of type U in E.

Exercise 20 What are the types of λxx and λxxx?

Exercise 21 Given two environments E and F, a substitution σ is of type E in F, written $F \rhd \sigma : E$ if, for all $(x, T) \in E$, $F \rhd \sigma(x) : T$. Prove that, if $F \rhd \sigma : E$ and $E \rhd t : T$, then $F \rhd \sigma(t) : T$.

Exercise 22 Prove that $\rhd$ is invariant by α-equivalence.

Is typing decidable? That is, given E, t and T, is it decidable to know whether $E \rhd t : T$ or not? Said again otherwise, is there a computable boolean function that, given E, t and T, returns true if $E \rhd t : T$, and false otherwise? Another similar problem is type *inference* (as opposed to type *checking*): given t, can we find E and T such that $E \rhd t : T$? It is not easy to answer these problems because, in the case of an abstraction λxv, we have to find some type U for x. What types are possible for a bound variable x depends on the terms to which λxv is applied and of which λxv is the argument. Moreover, we have seen that a term may have infinitely many types in some fixed environment. However, it seems that all these types follow the same pattern. But is it always the case? If this is the case, then does there exist a *most general* pattern from which all other types can be deduced? To solve these problems, we will introduce the notion of *unification* [30, 42].

5.2 Unification

Let a unification problem be a finite set of equations between terms, an equation being a pair of terms (t, u) written $t =^? u$. A solution to a unification problem $\{t_1 =^? u_1, \ldots, t_n =^? u_n\}$ is a substitution σ such that, for all i, $\sigma(t_i) = \sigma(u_i)$.

Exercise 23 Prove that this problem is decidable, that is, there is a function which, given a unification problem P, returns $\bot$ (fails) if P has no solution, and some solution σ otherwise. Prove its correctness. What is its complexity?

Given a set X of variables, we say that a substitution σ is *more general* than another substitution θ on X, written $\sigma \leq_X \theta$, if there is a substitution ρ such that $\theta =_X \widehat{\rho} \circ \sigma$, *i.e.* $\theta(x) = \rho(\sigma(x))$ for all $x \in X$. We say that two substitutions σ and θ are *equivalent* on X, written $\sigma \simeq_X \theta$, if $\sigma \leq_X \theta$ and $\theta \leq_X \sigma$.

Exercise 24 Prove that $\leq_X$ is a quasi-ordering, that is, a reflexive and transitive relation. Prove that $\simeq_X$ is an equivalence relation, that is, a reflexive, symmetric and transitive relation.

Exercise 25 Prove that $\sigma \simeq_X \theta$ iff there is a permutation $\pi : \{\mathrm{FV}(\theta(x)) \mid x \in X\} \to \{\mathrm{FV}(\sigma(x)) \mid x \in X\}$ such that $\sigma =_X \widehat{\pi} \circ \theta$.

Exercise 26 Prove that a solvable unification problem has a most general solution σ, that is, σ is more general than any other solution.

5.3 Type inference

We now have all the necessary tools to prove that type inference is decidable and that any typable term has a most general type and, finally, that type checking

also is decidable.

Exercise 27 Prove that type inference is decidable, and define a function computing the most general type of a term.

Exercise 28 Is typing decidable? That is, is there a computable boolean function that, given E, t and T, returns true if $E \rhd t : T$, and false otherwise?

5.4 Church-style simply-typed λ-calculus

To make type checking and type inference simpler (as in programming languages like C, Java, etc), and decidable in more complex type systems (*e.g.* Coq), it is necessary to annotate bound variables with their types. In this case, λ-terms are defined as follows: $t = x \mid \lambda x^T t \mid tt$, and the typing rule for abstraction has to be replaced by:

$$\text{(abs)}\ \frac{E \cup \{(x, U)\} \rhd v : V \quad x \notin \mathrm{dom}(E)}{E \rhd \lambda x^U v : U \to V}$$

Exercise 29 Prove that with type annotations, a λ-term can have at most one type in a given typing environment.

More on typed λ-calculus can be found for instance in [8].

6 First-order logic

6.1 Formulas and truth

We now have all the tools to define logical formulas. Given a family $(\mathcal{F}_n)_{n\in\mathbb{N}}$ of function symbols of fixed arity, a family $(\mathcal{P}_n)_{n\in\mathbb{N}}$ of predicate symbols of fixed arity, and an infinite set $\mathcal{X}$ of variables, let $\mathcal{T}$ be the set of (object) terms on $(\mathcal{F}_n)_{n\in\mathbb{N}}$, and let the set $\mathcal{L}$ of (logical) formulas be the smallest set such that:

- if P is a predicate symbol of arity n, *i.e.* $\mathsf{P} \in \mathcal{P}_n$, and $t_1, \ldots, t_n$ are (object) terms, then $\mathsf{P}(t_1, \ldots, t_n)$ is a formula;
- $\bot$ and $\top$ are formulas;
- if ϕ is a formula, then $\neg\phi$ is a formula;
- if ϕ and ψ are formulas, then $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \Rightarrow \psi$ and $\phi \Leftrightarrow \psi$ are formulas;
- if x is a variable and ϕ is a formula, then $\forall x\phi$ and $\exists x\phi$ are formulas.

So, $\mathcal{L}$ can be seen as the set of terms on the set of symbols $\mathcal{P} \cup \{\bot, \top, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall_x, \exists_x\}$ with the elements of $\mathcal{P} \cup \{\bot, \top\}$ of arity 0, $\neg, \forall_x, \exists_x$ of arity 1, and $\wedge, \vee, \Rightarrow, \Leftrightarrow$ of arity 2. Moreover, as in λ-calculus, formulas are identified modulo renaming of their bound variables: $\forall x \mathsf{P}(x) =_\alpha \forall y \mathsf{P}(y)$.

Now, how to define the truth (true or false) of a formula *in a model* $(A, (\mathsf{f}_A)_{\mathsf{f}\in\mathcal{F}}$ of the object terms? We need an interpretation $\mathsf{P}_A : A^n \to \mathsf{Bool}$ where $\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$ (or, equivalently, a subset $\mathsf{P}_A \subseteq A^n$) for each predicate symbol P of arity n. Then, given a finite valuation $\mu : \mathcal{X} \to A$, the interpretation of a formula ϕ, written $[\![\phi]\!]_\mu$, can be defined as follows:

- $[\![\mathsf{P}(t_1, \ldots, t_n)]\!]_\mu = \mathsf{P}_A([\![t_1]\!]_\mu, \ldots, [\![t_n]\!]_\mu)$
- $[\![\bot]\!]_\mu = \mathsf{false}$
- $[\![\top]\!]_\mu = \mathsf{true}$
- $[\![\neg\phi]\!]_\mu = \mathsf{not}([\![\phi]\!]_\mu)$
- $[\![\phi \vee \psi]\!]_\mu = \mathsf{or}([\![\phi]\!]_\mu, [\![\psi]\!]_\mu)$
- $[\![\phi \wedge \psi]\!]_\mu = \mathsf{and}([\![\phi]\!]_\mu, [\![\psi]\!]_\mu)$
- $[\![\phi \Rightarrow \psi]\!]_\mu = \mathsf{impl}([\![\phi]\!]_\mu, [\![\psi]\!]_\mu)$
- $[\![\phi \Leftrightarrow \psi]\!]_\mu = \mathsf{equiv}([\![\phi]\!]_\mu, [\![\psi]\!]_\mu)$
- $[\![\forall x\phi]\!]_\mu = \mathsf{forall}(\{[\![\phi]\!]_{\mu\cup\{x,a\}} \mid a \in A\})$ if $x \notin \mathrm{dom}(\mu)$
- $[\![\exists x\phi]\!]_\mu = \mathsf{exists}(\{[\![\phi]\!]_{\mu\cup\{x,a\}} \mid a \in A\})$ if $x \notin \mathrm{dom}(\mu)$

where the boolean functions not, or, ... are defined as usual, and $\mathsf{forall}, \mathsf{exists} : \mathcal{P}(\mathsf{Bool}) \to \mathsf{Bool}$ are defined as follows:

- $\mathsf{forall}(S) = \mathsf{true}$ iff $\mathsf{false} \notin S$
- $\mathsf{exists}(S) = \mathsf{true}$ iff $\mathsf{true} \in S$

Given a formula ϕ with free variables $x_1, \ldots, x_n$, its universal closure $\overline{\forall}\phi = \forall x_1 \ldots \forall x_n \phi$ and its existential closure $\overline{\exists}\phi = \exists x_1 \ldots \forall x_n \phi$.

Then, we say that a formula ϕ is *satisfiable* if there is a model $\mathcal{A} = (A, (\mathsf{f}_A)_{\mathsf{f}\in\mathcal{F}}, (\mathsf{P}_A)_{\mathsf{P}\in\mathcal{P}})$ in which $[\![\overline{\exists}\phi]\!] = \mathsf{true}$, and that ϕ is *valid* if, in every model $\mathcal{A}$, $[\![\overline{\forall}\phi]\!] = \mathsf{true}$.

A formula ψ is a *semantical consequence* of another formula ϕ, written $\phi \models \psi$, if $\phi \Rightarrow \psi$ is valid. Two formulas ϕ and ψ are *semantically equivalent*, written $\phi \equiv \psi$, if $\phi \models \psi$ and $\psi \models \phi$, *i.e.* if $\phi \Leftrightarrow \psi$ is valid.

Exercise 30 Prove that $\phi \Rightarrow \phi$ is valid, $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$ and $\neg(\forall x\phi) \equiv \exists x \neg\phi$.

In fact, we could simplify the definition of the logic because many connectors are definable from others.

6.2 Provability and deduction systems

How to know that a formula is valid or not? For a formula of the form $\forall x\phi$, we cannot check that ϕ is valid in *all* models. Instead, we generally use the rules of *logic* to *prove* that a formula is true or not. For instance, to prove that some property P implies another property Q, we may try to prove Q assuming that P holds. And if we prove later that P holds, then we can deduce that Q holds too. Etc. There are different ways to formalize these practices. One is Gentzen's *natural deduction* system invented in 1935 [23, 41], where each connector has one or two (for $\vee$) introduction rules (except $\bot$ of course), explaining how to prove a formula headed by this connector, and one or two (for $\wedge$) elimination rules explaining how to use a formula headed by this connector.

Gentzen's provability relation $\vdash$ on pairs (Γ, ϕ) where Γ is a finite set of formulas (the assumptions) and ϕ a formula, is the smallest relation $\vdash$ satisfying the following properties, where $(\Gamma, \phi) \in \vdash$ is written $\Gamma \vdash \phi$:

$$\text{(axiom)}\ \frac{\phi \in \Gamma}{\Gamma \vdash \phi}$$

$$(\bot\text{-elim})\ \frac{\Gamma \vdash \bot}{\Gamma \vdash \phi}$$

$$(\Rightarrow\text{-intro})\ \frac{\Gamma \cup \{\phi\} \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$$

$$(\Rightarrow\text{-elim})\ \frac{\Gamma \vdash \phi \Rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

$$(\wedge\text{-intro})\ \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$$

$$(\wedge\text{-elim-left})\ \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}$$

$$(\wedge\text{-elim-right})\ \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$$

$$(\vee\text{-intro-left})\ \frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi}$$

$$(\vee\text{-intro-right})\ \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}$$

$$(\vee\text{-elim})\ \frac{\Gamma \vdash \phi \vee \psi \quad \Gamma \cup \{\phi\} \vdash \chi \quad \Gamma \cup \{\psi\} \vdash \chi}{\Gamma \vdash \chi}$$

$$(\forall\text{-intro})\ \frac{\Gamma \vdash \phi \quad x \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash \forall x\phi}$$

$$\text{(}\forall\text{-elim)}\quad \frac{\Gamma \vdash \forall x \phi}{\Gamma \vdash \phi\{x \mapsto t\}}$$

$$\text{(}\exists\text{-intro)}\quad \frac{\Gamma \vdash \phi\{x \mapsto t\}}{\Gamma \vdash \exists x \phi}$$

$$\text{(}\exists\text{-elim)}\quad \frac{\Gamma \vdash \exists x \phi \quad \Gamma \cup \{\phi\} \vdash \chi \quad x \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(\chi)}{\Gamma \vdash \chi}$$

$$\text{(EM)}\quad \Gamma \vdash \phi \vee \neg\phi$$

We say that ϕ is *provable* under the assumptions Γ if $\Gamma \vdash \phi$ holds. Note that, since $\vdash$ is defined inductively, if $\Gamma \vdash \phi$ holds, then there must be some witness/proof of this fact that can be represented by a deduction tree.

Exercise 31 Prove that this deduction system is correct, *i.e.* $\vdash \subseteq \models$, by interpreting a finite set of formulas $\Gamma = \{\phi_1, \ldots, \phi_n\}$ by $\phi_1 \wedge \ldots \wedge \phi_n$. That is, if one can deduce ϕ from Γ, then ϕ is a semantical consequence of Γ. In particular, if one can deduce ϕ using no assumption, then ϕ is valid.

And does the converse hold? That is, if ϕ is valid, then ϕ is provable. In 1929, Gödel proved that this is indeed the case when no axiom is assumed (empty theory) [26, 27]. But he also proved in 1931 that this does not hold when some axioms powerful enough to do arithmetic are assumed [28]. In particular, the consistency of a theory powerful enough to do arithmetic cannot be deduced using arithmetic only.

We have introduced a notion of provability but is it decidable, that is, is there a computer program that can tell us in finite time whether *any* mathematical theorem is provable or not? Said otherwise, is there a computable boolean function that, for *all* Γ and ϕ, returns true if $\Gamma \vdash \phi$, and false otherwise. The answer is again no, if the language is rich enough to do arithmetic, because then we can encode the halting problem as a formula [15, 47].

However, it is decidable to check that a candidate deduction tree is correct or not, by checking that each rule is correctly applied and every side condition is fulfilled. This is what proof assistants like Coq do: they provide tools to build correct deduction trees.

6.3 Proof terms and Curry-Howard correspondence

But there is another way to represent deduction proofs. Indeed, it has been progressively discovered that formulas can be seen as types and proofs as λ-terms, so that, verifying the correctness of a deduction tree reduces to type-checking that a λ-term has a given type, a relation called the Curry-Howard correspondence [32].

For instance, the simply-typed λ-calculus corresponds to the logic with connectors restricted to $\Rightarrow$. In this correspondence:

- type constants correspond to atomic propositions, *i.e.* formulas of the form $\mathsf{P}(t_1, \ldots, t_n)$;
- the type $T \to U$ corresponds to the formula $T \Rightarrow U$;
- a typing environment $E = \{(x_1, T_1), \ldots, (x_n, T_n)\}$ corresponds to the set of assumptions $\{T_1, \ldots, T_n\}$ where every assumption T_i is given a name x_i;
- the application tu corresponds to the application of the rule ($\Rightarrow$-elim);
- the abstraction λxt corresponds to the application of the rule (A-intro).

Hence, for instance, λxx can be seen as a proof of $\phi \Rightarrow \phi$. So, λ-calculus can not only be used to define functions: it can also be used to represent proofs.

The Coq proof assistant is based on this correspondence. Its language is in fact some extension of the λ-calculus with a rich type system. When trying to prove some formula ϕ, the user in fact builds a λ-term that, in the end, is type-checked against ϕ to verify its correctness.

More on typed λ-calculus and logic can be found for instance in [25, 38].

7 To go further

To go further, I recommend to read the following books or articles.

- On the foundations of programs and proofs: [20].
- On computational complexity: [39, 44, 5].
- On the operational semantics of programs: [40].
- On the denotational semantics of programs: [50, 29].
- On Hoare's logic and the Why3 software: [31, 21, 22].
- On abstract interpretation and the Astrée software: [18].
- On program compilation: [1, 2].
- On rewriting theory: [6, 46].
- On typed λ-calculus: [25, 38, 8].
- On pure λ-calculus: [7].

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[2] A. W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998. See also [3] and [4].

[3] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[4] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[5] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. Draft and additional material available on `http://www.cs.princeton.edu/theory/complexity/`.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd edition, 1984.

[8] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 2. Oxford University Press, 1992.

[9] G. Birkhoff. *Lattice theory*. Number 25 in Colloquium Publications. American Mathematical Society, 3rd edition, 1967.

[10] F. Blanqui. Introduction to the Coq proof assistant. Lecture notes available on `https://who.rocq.inria.fr/Frederic.Blanqui/`, March 2013.

[11] F. Blanqui. Introduction to the OCaml programming language. Lecture notes available on `https://who.rocq.inria.fr/Frederic.Blanqui/`, March 2013.

[12] D. Brown, J. Levine, and T. Mason. *lex & yacc*. O'Reilly Media, 2nd edition, 1992.

[13] A. Church. A set of postulates for the foundations of logic. *Annals of Mathematics*, 33(2):346–366, 1932. Corrections in [14].

[14] A. Church. A set of postulates for the foundations of logic (second paper). *Annals of Mathematics*, 34(4):839–864, 1933.

[15] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[16] A. Church. *The calculi of lambda conversion*. Princeton University Press, 1941.

[17] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[18] P. Cousot and R. Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In J. Esparza, O. Grumberg, and M. Broy, editors, *Logics and Languages for Reliability and Security*, NATO Science Series III: Computer and Systems Sciences, pages 1–29. IOS Press, 2010.

[19] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[20] G. Dowek. *Proofs and Algorithms. An introduction to Logic and Computability*. Undergraduate Topics in Computer Science. Springer, 2011.

[21] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of the 19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 4590, 2007.

[22] J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *Proceedings of the 22th European Symposium on Programming*, Lecture Notes in Computer Science 7792, 2013.

[23] G. Gentzen. Untersuchungen über das logische schließsen I. *Mathematische Zeitschrift*, 39:176–210, 1935. English translation in [24].

[24] G. Gentzen. Investigations into logical deduction I. *American Philosophical Quarterly*, 1(4):288–306, 1964. English translation by M. E. Szabo.

[25] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1988.

[26] K. Gödel. *Über die Vollständigkeit des Logikkalküls*. PhD thesis, University of Vienna, Austria, 1929.

[27] K. Gödel. Die vollständigkeit der axiome des logischen funktionenkalküls. *Monatshefte für Mathematik un Physik*, 37(1):349–360, 1930.

[28] K. Gödel. ber formal unentscheidbare sätze der principia mathematica und verwandter systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English translation in [49].

[29] C. A. Gunter. *Semantics of Programming Languages. Structures and Techniques*. MIT Press, 1992.

[30] J. Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Faculté des sciences de Paris, France, 1930.

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, 1969.

[32] W. A. Howard. The formulae-as-types notion of construction (1969). In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[33] S. C. Kleene. *A Theory of Positive Integers in Formal Logic*. PhD thesis, Princeton, USA, 1934.

[34] S. C. Kleene. *Introduction to metamathematics*. North-Holland, 1952.

[35] S. C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, 1981.

[36] B. Knaster and A. Tarski. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématiques*, 6:133–134, 1928.

[37] D. Knuth and P. Bendix. Simple word problems in universal algebra. In *Computational problems in abstract algebra, Proceedings of a Conference held at Oxford in 1967*, pages 263–297. Pergamon Press, 1970.

[38] J.-L. Krivine. *Lambda-calculus, types and models*. Series in computers and their applications Computers and their applications. Ellis Horwood, 1993. `http://cel.archives-ouvertes.fr/cel-00574575/`.

[39] C. Papadimitriou. *Computational complexity*. Addison Wesley, 1993.

[40] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[41] D. Prawitz. *Natural deduction: a proof-theoretical study*. Almqvist and Wiksell, Stockholm, 1965. `http://su.diva-portal.org/smash/get/diva2:563092/FULLTEXT01.pdf`.

[42] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[43] B. Russel. *The principle of mathematics*. Cambridge University Press, 1903. `http://quod.lib.umich.edu/cgi/t/text/text-idx?c=umhistmath;idno=AAT1273`.

[44] M. Sipser. *Introduction to the Theory of Computation*. South-Western College Publishing, 2012. Third revised edition.

[45] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[46] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[47] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937. Corrections in [48].

[48] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, 43:544–546, 1938.

[49] J. v. Heijenoort, editor. *From Frege to Gödel, a source book in mathematical logic, 1879-1931*. Harvard University Press, 1977.

[50] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

8 Solutions to exercises

8.1 Section 2: Induction and sequences

Solution of Exercise 1

Let $S = \{w \in X^* \mid P(w)\}$. By definition, we have $S \subseteq X^*$. Now, $X^* \subseteq S$ if S satisfies the properties (1) and (2):

1. $\varepsilon \in S$, *i.e.* $P(\varepsilon)$. That is (P1).
2. If x is a letter and $w \in S$, *i.e.* $P(w)$, then $xw \in S$, *i.e.* $P(xw)$. That is (P2).

Solution of Exercise 2

We have $\varepsilon \circ v = v$ by definition. We can prove that $v \circ \varepsilon = v$ by induction on v. If $v = \varepsilon$, then $v \circ \varepsilon = \varepsilon = v$. Assume now that $v \circ \varepsilon = v$ and let x be a letter. Then, $(xv) \circ \varepsilon = x(v \circ \varepsilon) = xv$.

For the associativity, we can proceed by induction on u. If $u = \varepsilon$, then $(u \circ v) \circ w = v \circ w = u \circ (v \circ w)$. Assume now that $(u \circ v) \circ w = u \circ (v \circ w)$ and let x be a letter. Then, $((xu) \circ v) \circ w = (x(u \circ v)) \circ w = x((u \circ v) \circ w) = x(u \circ (v \circ w)) = (xu) \circ (v \circ w)$.

8.2 Section 3: Terms

Solution of Exercise 3

By definition $\mathcal{T}$ is the smallest subset S of $\mathcal{U}$ satisfying the property $P(S)$: for all $\mathsf{f} \in \mathcal{F}$ and $t_1, \ldots, t_n \in S$, $\mathsf{f}(t_1, \ldots, t_n) \in S$, *i.e.* $\mathcal{T} = \bigcap\{S \subseteq \mathcal{U} \mid P(S)\}$. By definition of f, $f(\mathcal{T}) = \mathcal{T} \cup \{\mathsf{f}(t_1, \ldots, t_n) \mid \mathsf{f} \in \mathcal{F}, t_1, \ldots, t_n \in \mathcal{T}\}$. Thus, $\mathcal{T} \subseteq f(\mathcal{T})$ and we are left to prove that $f(\mathcal{T}) \subseteq \mathcal{T}$. Since $\mathcal{T}$ is the smallest subset of $\mathcal{U}$ satisfying P, we have $f(\mathcal{T}) \subseteq \mathcal{T}$ if, for all $S \subseteq \mathcal{U}$ satisfying $P(S)$, $f(\mathcal{T}) \subseteq S$. So, let $t \in f(\mathcal{T})$. By definition of f, there are two cases:

- $t \in \mathcal{T}$. Then, $t \in S$ by definition of $\mathcal{T}$.
- $t = \mathsf{f}(t_1, \ldots, t_n)$ with $\mathsf{f} \in \mathcal{F}$ and $t_1, \ldots, t_n \in \mathcal{T}$. Since $\mathcal{T} \subseteq S$, we have $t_1, \ldots, t_n \in S$. Since S satisfies P, we have $t \in S$.

Note that this proof is in fact a proof by induction on $t \in f(\mathcal{T})$ that $t \in S$. We now check that f is monotone. Assume that $X \subseteq Y$ and let $t \in f(X)$. By definition of f, there are two cases:

- $t \in X$. Since $X \subseteq Y$ and $Y \subseteq f(Y)$, we have $t \in f(Y)$.
- $t = \mathsf{f}(t_1, \ldots, t_n)$ with $\mathsf{f} \in \mathcal{F}$ and $t_1, \ldots, t_n \in X$. Since $X \subseteq Y$, we have $t_1, \ldots, t_n \in Y$. Thus, by definition of f, $t \in f(Y)$.

Solution of Exercise 4

Take $\mathrm{lub}(\mathcal{E}) = \bigcup \mathcal{E}$, $\mathrm{glb}(\emptyset) = \mathcal{U}$ and $\mathrm{glb}(\mathcal{E}) = \bigcap \mathcal{E}$ if $\mathcal{E} \neq \emptyset$. Why the greatest lower bound of $\emptyset$ is $\mathcal{U}$? $\mathcal{U}$ is a lower bound of $\emptyset$ since it is smaller than any

element of $\emptyset$ for there is no element in $\emptyset$! In fact, *any* subset of $\mathcal{U}$ *is* a lower bound of $\emptyset$. And if L is a lower bound of $\emptyset$, *i.e.* if L is any subset of $\mathcal{U}$, then L is included in $\mathcal{U}$.

Solution of Exercise 5

Let $A = \bigcup_{n\in\mathbb{N}} f^n(\emptyset) \subseteq \mathcal{T}$. We first prove that $f^n(\emptyset) \subseteq \mathcal{T}$ by induction on $n \in \mathbb{N}$. For $n = 0$, this is trivial. Assume that $f^n(\emptyset) \subseteq \mathcal{T}$ and let $t \in f^{n+1}(\emptyset)$. By definition of f, there are two cases:

- $t \in f^n(\emptyset)$. By induction hypothesis, $f^n(\emptyset) \subseteq \mathcal{T}$. So, $t \in \mathcal{T}$.
- $t = \mathsf{f}(t_1, \ldots, t_n)$ with $\mathsf{f} \in \mathcal{F}$ and $t_1, \ldots, t_n \in f^n(\emptyset)$. By induction hypothesis, $f^n(\emptyset) \subseteq \mathcal{T}$. So, $t_1, \ldots, t_n \in \mathcal{T}$ and, since $f(\mathcal{T}) \subseteq \mathcal{T}$, we have $t \in \mathcal{T}$.

Therefore, $A \subseteq \mathcal{T}$. We now prove that $\mathcal{T} \subseteq A$. Since $\mathcal{T}$ is the smallest set S satisfying $P(S)$: for all $\mathsf{f} \in \mathcal{F}$ and $t_1, \ldots, t_n \in S$, $\mathsf{f}(t_1, \ldots, t_n) \in S$, *i.e.* $\mathcal{T} = \bigcap\{S \subseteq \mathcal{U} \mid P(S)\}$, it suffices to prove that A satisfies P. Let $\mathsf{f} \in \mathcal{F}$ and $t_1, \ldots, t_k \in A$. By definition of A, for each i, there is n_i such that $t_i \in f^{n_i}(\emptyset)$. Now, remark that $(f^n(\emptyset))_{n\in\mathbb{N}}$ is an increasing sequence of sets, *i.e.* $f^n(\emptyset) \subseteq f^{n+1}(\emptyset)$ (easy proof). Therefore, for each i, $t_i \in f^n(\emptyset)$ where n is the maximum of $\{n_1, \ldots, n_k, 0\}$ (we add 0 to make the set non empty). Hence, $t \in f^{n+1}(\emptyset) \subseteq A$.

Solution of Exercise 6

Let a pattern-matching problem P be a set of pairs (p, t) made of a pattern p and a term t. A substitution σ is a solution of $P = \{(p_1, t_1), \ldots, (p_n, t_n)\}$ if $\sigma(p_1) = t_1, \ldots, \sigma(p_n) = t_n$.

Let now match be the function defined as follows:

- $\mathsf{match}(\{(\mathsf{f}(t_1, \ldots, t_m), \mathsf{g}(u_1, \ldots, u_n))\} \cup P) = \mathsf{match}(\{(t_1, u_1), \ldots, (t_m, u_m)\} \cup P)$ if $\mathsf{f} = \mathsf{g}$ and $m = n$,
- $\mathsf{match}(\{(\mathsf{f}(t_1, \ldots, t_m), \mathsf{g}(u_1, \ldots, u_n))\} \cup P) = \bot$ if $\mathsf{f} \neq \mathsf{g}$ or $m \neq n$,
- $\mathsf{match}(\{(x, t), (x, u)\} \cup P) = \bot$ if $t \neq u$.

If $\mathsf{match}(P) = \bot$, then P has no solution. Otherwise, $\mathsf{match}(P) = \{(x_1, t_1), \ldots, (x_n, u_n)\}$ with $t_i = t_j$ if $x_i = x_j$, that is, $\mathsf{match}(P)$ is the unique solution of P.

Solution of Exercise 7

We have $\rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{S}}$. Assume now that $t \rightarrow_{\mathcal{S}} t'$. Then, there are C, $l \rightarrow r \in \mathcal{S}$ and σ, such that $t = C\{x \mapsto \sigma(l)\}$ and $t' = C\{x \mapsto \sigma(r)\}$. If $l \rightarrow r \in \mathcal{R}$, then we are done. Otherwise, there is a rule $g \rightarrow d \in \mathcal{R}$ such that $l \rightarrow r = \pi(g) \rightarrow \pi(d)$. Hence, $t \rightarrow_{\mathcal{R}} t'$ since $t = C\{x \mapsto (\sigma \circ \pi^{-1})(g)\}$ and $t' = C\{x \mapsto (\sigma \circ \pi^{-1})(d)\}$.

Solution of Exercise 8

The model is the set A of terms itself with the interpretation function $\mathsf{f}_A(x_1, \ldots, x_n) = \mathsf{f}(x_1, \ldots, x_n)$.

Solution of Exercise 9

By induction on t. If $t = x$, then $[\![\sigma(t)]\!]_\mu = [\![\sigma(x)]\!]_\mu = (\mu \circ \sigma)(x) = [\![x]\!]_{\mu\circ\sigma}$. Otherwise, $t = \mathsf{f}(t_1, \ldots, t_n)$, $[\![\sigma(t)]\!]_\mu = [\![\mathsf{f}(\sigma(t_1), \ldots, \sigma(t_n))]\!]_\mu = \mathsf{f}_A([\![\sigma(t_1)]\!]_\mu, \ldots, [\![\sigma(t_n)]\!]_\mu)$. By induction hypothesis, $[\![\sigma(t_i)]\!]_\mu = [\![t_i]\!]_{\mu\circ\sigma}$. Therefore, $[\![\sigma(t)]\!]_\mu = [\![t]\!]_{\mu\circ\sigma}$.

8.3 Section 4: Lambda-calculus

Solution of Exercise 10

We prove by induction on t that, for all t' such that $t =_\alpha t'$, we have $\mathrm{FV}(t) = \mathrm{FV}(t')$.

- Case $t = x$. Then, $t' = x$ and $\mathrm{FV}(t) = \mathrm{FV}(t')$.
- Case $t = uv$. Then, $t' = u'v'$ with $u =_\alpha u'$ and $v =_\alpha v'$. By induction hypothesis, $\mathrm{FV}(u) = \mathrm{FV}(u')$ and $\mathrm{FV}(v) = \mathrm{FV}(v')$. Therefore, $\mathrm{FV}(t) = \mathrm{FV}(u) \cup \mathrm{FV}(v) = \mathrm{FV}(u') \cup \mathrm{FV}(v') = \mathrm{FV}(t')$.
- Case $t = \lambda xu$. Then, $t' = \lambda yv$ with $y \notin \mathrm{FV}(\lambda xu)$ and $u =_\alpha v\{y \mapsto x\}$. We have $\mathrm{FV}(t) = \mathrm{FV}(u) - \{x\}$ and $\mathrm{FV}(t') = \mathrm{FV}(v) - \{y\}$. By induction hypothesis, $\mathrm{FV}(u) = \mathrm{FV}(v\{y \mapsto x\})$. If $y = x$, then $\mathrm{FV}(u) = \mathrm{FV}(v)$ and we are done. Otherwise, $\mathrm{FV}(u) = (\mathrm{FV}(v) - \{y\}) \cup \{x\}$. Therefore, $\mathrm{FV}(t) = \mathrm{FV}(t')$.

Solution of Exercise 11

We have $\lambda xx =_\alpha \lambda yy$ but $\mathrm{Sub}(\lambda xx) = \{x\}$ and $\mathrm{Sub}(\lambda yy) = \{y\}$.

Solution of Exercise 12

- xx, λxxx have no β-reduct: they are in normal form.
- $(\lambda xx)(\lambda xx) \to_\beta \lambda xx$.
- $(\lambda xxx)(\lambda xxx)$ β-rewrites to itself: $(\lambda xxx)(\lambda xxx) \to_\beta (\lambda xxx)(\lambda xxx)$.
- $(\lambda xy(xx))(\lambda xy(xx))$ is a fixpoint of y: $(\lambda xy(xx))(\lambda xy(xx))$ $\to_\beta y((\lambda y(xx))(\lambda y(xx))) \to_\beta y(y((\lambda y(xx))(\lambda y(xx)))) \to_\beta \ldots$

Solution of Exercise 13

$\widehat{+}\widehat{2}\widehat{2} \to^*_\beta \lambda f\lambda x\widehat{2}f(\widehat{2}fx) \to^*_\beta \lambda f\lambda x\widehat{2}f(f(fx)) \to^*_\beta \lambda f\lambda xf(f(f(fx))) = \widehat{4}$.

Solution of Exercise 14

Note that $\widehat{p}fx \to^*_\beta f(f(\ldots(fx)))$ (f applied p times to x) and that $p \times q = q + q + \ldots + q$ (p times). So, we can take $\widehat{\times} = \lambda p\lambda q\lambda f\lambda xp(\widehat{+}q)\widehat{0}fx$.

Solution of Exercise 15

We can take $\mathsf{ifzero} = \lambda t\lambda u\lambda vtu(\lambda zv)$ where z is any variable not occurring in v (λzv is then the constant function equal to v). The term $\mathsf{ifzero}\,\widehat{n}uv$ iterates n times λzv on u. If $n = 0$, then we get u (λzv is never applied). Otherwise, we get v (λzv is applied at least once).

Solution of Exercise 16

We can take $\mathsf{pair} = \lambda u \lambda v \lambda z \mathsf{ifzero} z u v$, $\pi_1 = \lambda p p \widehat{0}$ and $\pi_2 = \lambda p p \widehat{1}$.

Solution of Exercise 17

To make things easier, we can give names to the functionals used to define K-computable functions. Let π_i^n be the function mapping $(x_1, \ldots, x_n)$ to x_i, 0^n be the function mapping $(x_1, \ldots, x_n)$ to 0, s be the function mapping x to $x+1$, $\mathsf{comp}_m^n(h, g_1, \ldots, g_m)$ be the function mapping $(x_1, \ldots, x_n)$ to $h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$, $\mathsf{rec}^n(g, h)$ the function f such that $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$ and $f(x_1, \ldots, x_n, y{+}1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, *y))$, and $\mathsf{min}^n(g)$ be the function f such that $f(x_1, \ldots, x_n)$ is the least integer y such that $g(x_1, \ldots, x_n, y) = 0$.

Addition can be defined by recursion as follows: $x + 0 = x$ and $x + (y+1) = (x+y)+1$. Hence, the addition is $\mathsf{add} = \mathsf{rec}^1(g, h)$ where $g(x) = x$, *i.e.* $g = \pi_1^1$, and $h(x, y, z) = z + 1$, *i.e.* $h = \mathsf{comp}_1^3(\mathsf{s}, \pi_3^3)$. Therefore, the addition is K-computable.

Multiplication can be defined by recursion too: $x \times 0 = 0$ and $x \times (y+1) = x + (x \times y)$. Hence, the multiplication is $\mathsf{mul} = \mathsf{rec}^1(g, h)$ where $g = 0^1$ and $h(x, y, z) = x + z$, *i.e.* $h = \mathsf{comp}_2^3(\mathsf{add}, \pi_1^3, \pi_2^3)$.

The function $\mathsf{ifzero}_{f,g,h}^n$ can be defined by doing a recursion on $f(x_1, \ldots, x_n)$, *i.e.* $\mathsf{ifzero}_{f,g,h}^n(x_1, \ldots, x_n) = k(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$, *i.e.* $\mathsf{ifzero}_{f,g,h}^n = \mathsf{comp}_{n+1}^n (k, \pi_1^n, \ldots, \pi_n^n, f)$, where k is defined by recursion as follows: $k(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$ and $k(x_1, \ldots, x_n, y+1) = h(x_1, \ldots, x_n)$, ie $k = \mathsf{rec}^n(g, h')$ where $h'(x_1, \ldots, x_n, y, z) = h(x_1, \ldots, x_n)$, *i.e.* $h' = \mathsf{comp}_n^{n+2}(h, \pi_1^{n+2}, \ldots, \pi_n^{n+2})$.

Solution of Exercise 18

We proceed by induction on the way K-computable functions are defined.

Turing machine for π_i^n. The machine has to copy the 1's that are on the i-th tape to the $(n+1)$-th tape and go back to position 0 before stoping. To this end, we will use two new states q_c (copy) and q_b (back):

- $\delta((x_1, \ldots, x_{n+1}), q_i) = ((x_1, \ldots, x_{n+1}), q_c, 1)$ (we start by moving the head to the right and change to state q_c);
- if $x_i = 1$, then $\delta((x_1, \ldots, x_n, x_{n+1}), q_c) = ((x_1, \ldots, x_n, 1), q_c, 1)$ (to copy the 1's of tape i to tape $(n+1)$);
- if $x_i = 0$, then $\delta((x_1, \ldots, x_{n+1}), q_c) = ((x_1, \ldots, x_{n+1}), q_b, -1)$ (to start moving back to position 0 when we read a 0);
- if $x_i = 1$, then $\delta((x_1, \ldots, x_{n+1}), q_b) = ((x_1, \ldots, x_{n+1}), q_b, -1)$ (to move back);
- if $x_i = \square$, then $\delta((x_1, \ldots, x_{n+1}), q_b) = ((x_1, \ldots, x_{n+1}), q_f, 0)$ (to stop once we have reached position 0).

Turing machine for 0^n. Just take $\delta((x_1, \ldots, x_n), q_i) = ((x_1, \ldots, x_n), q_f, 0)$ (stop immediately).

Turing machine for s. It suffices to copy the 1's of tape 1 to tape 2 (state q_c), add one more 1 on tape 2, and come back to position 0 (state q_b):

- $\delta((x_1, x_2), q_i) = ((x_1, x_2), q_c, 1)$ (we start by moving the head to the right and change to state q_c);
- $\delta((1, x_2), q_c) = ((1, 1), q_c, 1)$ (to copy the 1's of tape 1 to tape 2);
- $\delta((0, x_2), q_c) = ((0, 1), q_b, -1)$ (to add one 1 on tape 2 and start moving back to position 0);
- $\delta((1, x_2), q_b) = ((1, x_2), q_b, -1)$ (to move back);
- $\delta((\square, x_2), q_b) = ((\square, x_2), q_f, 0)$ (to stop once we have reached position 0).

Turing machine for $\mathsf{comp}^n_m(h, g_1, \ldots, g_m)$. By induction hypothesis, g_1 is computed by $T_1 = (\Sigma^1, k_1, Q^1, q_i^1, q_f^1, \delta^1)$, $\ldots$, g_m by $T_m = (\Sigma^m, k_n, Q^m, q_i^m, q_f^m, \delta^m)$, and h by $T_{m+1} = (\Sigma^{m+1}, k_{m+1}, Q^{m+1}, q_i^{m+1}, q_f^{m+1}, \delta^{m+1})$. Let $l_i = \Sigma_{j=1}^{i} k_i$. Any machine $T = (\Sigma, k, Q, q_i, q_f, \delta)$ computing a function $f : \mathbb{N}^n \to \mathbb{N}$ can be easily transformed into a machine $T' = (\Sigma, k, Q, q_i, q_f, \delta')$ where tapes are permuted so that it reads its arguments on the tapes $j_1, \ldots, j_k$ and outputs its result on tape j_{k+1}, where $j_1, \ldots, j_{k+1}$ are pairwise distinct integers of $[1, k]$ (the order of tapes does not matter). Any machine can also be easily transformed into a machine with additional unused tapes. Let $l_j = n + 1 + \Sigma_{i=1}^{j-1} k_j$ and $k = l_{m+2}$. For all $j \in [1, m]$, we transform T_j into a machine $U_j = (\Sigma^j, k, Q^j, q_i^j, q_f^j, \varepsilon^j)$ that reads its arguments on tapes $l_j + 1, \ldots, l_j + n$ and outputs its result on tape $l_j + n + 1$ (the tapes used by T_j are translated by l_j). And we transform T_{m+1} into a machine $U_{m+1} = (\Sigma^{m+1}, k, Q^{m+1}, q_i^{m+1}, q_f^{m+1}, \varepsilon^{m+1})$ that reads its arguments on tapes $l_1 + n + 1, \ldots, l_m + n + 1$ and outputs its result on tape $n + 1$. Without loss of generality, we can also assume that the sets Σ^i, Q^i and $Q' = \{q_i, q_c, q_b\}$ are pairwise disjoint. Then, $\mathsf{comp}^n_m(h, g_1, \ldots, g_m)$ can be computed by taking $T = (\Sigma, k, Q \cup Q', q_i, q_f^{m+1}, \varepsilon \cup \tau)$ where $\Sigma = \bigcup_{i=1}^{m+1} \Sigma^i$, $Q = \bigcup_{i=1}^{m+1} Q^i$, $\varepsilon = \bigcup_{i=1}^{m+1} \varepsilon^i$, and τ is defined as follows:

- $\tau((x_1, \ldots, x_k), q_i) = ((x_1, \ldots, x_k), q_c, 0)$ (to start copying the arguments $(x_1, \ldots, x_n)$ to compute $g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)$);
- if there is $i \in [1, n]$ such that $x_i = 1$, then $\tau((x_1, \ldots, x_k), q_c) = ((y_1, \ldots, y_k), q_c, 1)$ where $y_j = x_j$ except, for all $j \in [1, m]$ and $q \in [1, n]$, $y_{l_j+q} = x_j$;
- if there is no $i \in [1, n]$ such that $x_i = 1$, then $\tau((x_1, \ldots, x_k), q_c) = ((x_1, \ldots, x_k), q_b, -1)$ (to start to move the head back);
- if $x_1 \neq \square$, then $\tau((x_1, \ldots, x_k), q_b) = ((x_1, \ldots, x_k), q_b, -1)$ (to move back);
- $\tau((\square, \ldots, \square), q_b) = ((\square, \ldots, \square), q_i^1, 0)$ (to compute $g_1(x_1, \ldots, x_n)$);
- $\tau((x_1, \ldots, x_k), q_f^1) = ((x_1, \ldots, x_k), q_i^2, 0)$ (to compute $g_2(x_1, \ldots, x_n)$);
- $\ldots$
- $\tau((x_1, \ldots, x_k), q_f^m) = ((x_1, \ldots, x_k), q_i^{m+1}, 0)$ (to compute $g_2(x_1, \ldots, x_n)$).

Turing machine for recursion and minimization. ...

Solution of Exercise 19

Encode the configuration of a Turing machine into some tuple of natural numbers and show that the function $c_t \mapsto c_{t+1}$ is Kleene-computable.

...

8.4 Section 5: Simply-typed lambda-calculus

Solution of Exercise 20

Any type for λxx must be of the form $T \to T$, and every type of the form $T \to T$ is a type of λxx. On the other hand, there is no type T such that $\rhd \lambda xxx : T$. Indeed, assume that $\rhd \lambda xxx : T$, then T must be of the form $U \to V$ with $\{(x, U)\} \rhd xx : V$. But, for this to hold, we must have $U = U \to V$ which is not possible (the size of $U \to V$ is strictly bigger than the size of U).

Solution of Exercise 21

By induction on the definition of $\rhd$.

- $E \rhd x : T$ since $(x, T) \in E$. We have $F \rhd \sigma(x) : T$ since $F \rhd \sigma : E$.
- $E \rhd \lambda xv : U \to V$ since $E \cup \{(x, U)\} \rhd v : V$ and $x \notin \mathrm{dom}(E)$. By renaming, we can assume that $x \notin \mathrm{dom}(F) \cup \mathrm{FV}(\sigma)$. Therefore, $\sigma(\lambda xv) = \lambda x \sigma(v)$. By induction hypothesis, $F \cup \{(x, U)\} \rhd \sigma(v) : V$ since $F \cup \{(x, U)\} \rhd \sigma : E \cup \{(x, U)\}$. Therefore, $F \rhd \sigma(\lambda xv) : U \to V$.
- $E \rhd tu : V$ since $E \rhd t : U \to V$ and $E \rhd u : U$. We have $\sigma(tu) = \sigma(t)\sigma(u)$. By induction hypothesis, $E \rhd \sigma(t) : U \to V$ and $E \rhd \sigma(u) : U$. Therefore, $E \rhd \sigma(tu) : V$.

Solution of Exercise 22

We prove by induction on the definition of $\rhd$ that, if $E \rhd t : T$ and $t =_\alpha t'$, then $E \vdash t' : T$.

- Case $t = x \in \mathcal{X}$ and $(x, T) \in E$. Then, $t' = x$ and we are done.
- Case $t = \lambda xv$, $T = U \to V$, $x \notin \mathrm{dom}(E)$ and $E \cup \{(x, U)\} \rhd v : V$. Then, $t' = \lambda yw$, $v =_\alpha w\{y \mapsto x\}$ and $y \notin \mathrm{FV}(\lambda xv)$.

Solution of Exercise 23

Consider the following rewrite rules on unification problems:

- $P \cup \{t =^? t\} \to P$
- $P \cup \{\mathsf{f}(t_1, \ldots, t_n) =^? \mathsf{g}(u_1, \ldots, u_p)\} \to \bot$ if $\mathsf{f} \neq \mathsf{g}$
- $P \cup \{\mathsf{f}(t_1, \ldots, t_n) =^? \mathsf{f}(u_1, \ldots, u_n)\} \to P \cup \{t_1 =^? u_1, \ldots, t_n =^? u_n\}$
- $P \cup \{x =^? t\} \to P\{x \mapsto t\} \cup \{x =^? t\}$ if $x \notin \mathrm{FV}(t)$ and $x \in \mathrm{FV}(P)\}$

- $P \cup \{t =^? x\} \to P\{x \mapsto t\} \cup \{x =^? t\}$ if $x \notin \mathrm{FV}(t)$ and $x \in \mathrm{FV}(P)\}$
- $P \cup \{x =^? t\} \to \bot$ if $x \in \mathrm{FV}(t)$ and $t \neq x$
- $P \cup \{t =^? x\} \to \bot$ if $x \in \mathrm{FV}(t)$ and $t \neq x$

Let $S(P)$ be the set of substitutions satisfying P. The rules are correct: if $P \to Q$, then every solution of Q is a solution of P ($S(Q) \subseteq S(P)$):

- A solution of P is a solution of $P \cup \{t =^? t\}$ since $\sigma(t) = \sigma(t)$.
- There is no solution of $\bot$.
- Let σ be a solution of $P \cup \{t_1 =^? u_1, \ldots, t_n =^? u_n\}$. Then, σ is a solution of $P \cup \{\mathsf{f}(t_1, \ldots, t_n) =^? \mathsf{f}(u_1, \ldots, u_n)\}$ since $\sigma(\mathsf{f}(t_1, \ldots, t_n)) = \mathsf{f}(\sigma(t_1), \ldots, \sigma(t_n))$.
- Let $\rho = \{x \mapsto t\}$ and σ be a solution of $Q = \rho(P) \cup \{x =^? t\}$. This means that $\sigma(x) = \sigma(t)$ and, for all equation $a =^? b \in P$, $\sigma(\rho(a)) = \sigma(\rho(b))$. Hence, $\sigma \circ \rho$ is a solution of P. Since $x \notin \mathrm{FV}(t)$, $\rho(t) = t$ and $\sigma \circ \rho$ is also a solution of $x =^? t$.
- There is no solution of $\bot$.

The rules are also complete: if $P \to Q$, then every solution of P is a solution of Q ($S(P) \subseteq S(Q)$).

- A solution of $P \cup \{t =^? t\}$ is a solution of P since P is included in $P \cup \{t =^? t\}$.
- There is no solution for $P \cup \{\mathsf{f}(t_1, \ldots, t_n) =^? \mathsf{g}(u_1, \ldots, u_p)\}$ if $\mathsf{f} \neq \mathsf{g}$.
- A solution of $P \cup \{\mathsf{f}(t_1, \ldots, t_n) =^? \mathsf{f}(u_1, \ldots, u_n)\}$ is clearly is solution for $P \cup \{t_1 =^? u_1, \ldots, t_n =^? u_n\}$ too.
- Let σ be a solution of $P \cup \{x =^? t\}$. This means that $\sigma(x) = \sigma(t)$ and, for all equation $a =^? b \in P$, $\sigma(a) = \sigma(b)$. Let θ be the substitution such that $\theta(x) = x$ and, for all $y \neq x$, $\theta(y) = \sigma(y)$. We have $\sigma = \theta \cup \rho$ where $\rho = \{x \mapsto t\}$. We also have $\sigma = \theta \circ \rho$. Indeed, $\theta(\rho(x)) = \theta(t) = \sigma(t)$ since $x \notin \mathrm{FV}(t)$ and, for all $y \neq x$, $\theta(\rho(y)) = \theta(y) = \sigma(y)$. Hence, σ is a solution of $\rho(P) \cup \{x =^? t\}$.
- There is no solution of $x =^? t$ if $x \in \mathrm{FV}(t)$ and $t \neq x$ because the size of $\sigma(t)$ must be strictly bigger than the size of $\sigma(x)$.

What is the shape of a problem that cannot be rewritten (we say that it is in normal form)? It is either $\bot$ or a set P of equations e of the form $x =^? t$ or $t =^? x$ with $x \notin \mathrm{FV}(t)$ and $x \notin \mathrm{FV}(P - \{e\})$. If it is $\bot$, then there is no solution. Otherwise, P can be transformed into the solution σ such that $\sigma(x) = t$ if $x =^? t$ or $t =^? x$ is in P, and $\sigma(x) = s$ otherwise.

But can we always get to a normal form? To this end, we need to prove that the rules cannot be applied for ever, *i.e.* the rewrite system terminates. To this end, we can split a problem into two parts (P, Q) where Q, empty at the

beginning, gathers all the equations $x =^? t$ or $t =^? x$ obtained by the rules 4 and 5. Then, the rules are applied to the P part only. We remark then that the number of distinct variables is not increased by rules. And, in the case where the number of distinct variables do not strictly decrease, *i.e.* in the decomposition rule 3, the size of the problem strictly decreases.

One can easily see this algorithm is of complexity n^2 in the size n of the unification problem: the number of rewrite steps is of order n and each substitution propagation in rules 4 and 5 is of order n too.

Solution of Exercise 24

- $\leq_X$ is reflexive. For all substitution σ, we have $\sigma \leq_X \sigma$ since $\sigma =_X \widehat{\iota} \circ \sigma$ where ι is the identity substitution.
- $\leq_X$ is transitive. Assume that $\sigma_1 \leq_X \sigma_2$ and $\sigma_2 \leq_X \sigma_3$. Then, there are ρ_1 and ρ_2 such that $\sigma_2 =_X \widehat{\rho_1} \circ \sigma_1$ and $\sigma_3 =_X \widehat{\rho_2} \circ \sigma_2$. Therefore, $\sigma_1 \leq_X \sigma_3$ since $\sigma_3 =_X \widehat{\rho_2} \circ (\widehat{\rho_1} \circ \sigma_1) = (\widehat{\rho_2} \circ \widehat{\rho_1}) \circ \sigma_1$ (associativity of function composition).
- $\simeq_X$ is reflexive. Note that $\simeq_X = \leq_X \cap \geq_X$. Therefore, $\simeq_X$ is reflexive since $\leq_X$ so is.
- $\simeq_X$ is transitive. Since $\leq_X$ is transitive.
- $\simeq_X$ is symmetric. By definition.

Solution of Exercise 25

Assume that $\sigma \simeq_X \theta$. Then, there ρ and ω such that $\sigma =_X \widehat{\rho} \circ \theta$ and $\theta =_X \widehat{\omega} \circ \sigma$. Thus, $\sigma =_X \widehat{\rho} \circ \widehat{\omega} \circ \sigma$. It follows that, for every $y \in \{\mathrm{FV}(\sigma(x)) \mid x \in X\}$, $\omega(y)$ must be a variable $z \in \{\mathrm{FV}(\theta(x)) \mid x \in X\}$, and $\rho(z) = y$.

Solution of Exercise 26

When applying the previous algorithm, we obtain a problem in normal form that has exactly the same solutions as the initial problem. The substitution corresponding to the problem in normal form is the most general one mgu (up to renaming of variables).

Solution of Exercise 27

We extend the algebra $\mathcal{S}$ of types with an infinite set of type variables $X, Y, \ldots$ to represent type patterns. We then define a computable function $\mathsf{typconstr}$ which, given a typing environment E, a term t and a type T, returns the constraints that should be satisfied for having $E \rhd t : T$:

- $\mathsf{typconstr}(E, x, T) = \{T =^? U\}$ if $(x, U) \in E$,
- $\mathsf{typconstr}(E, \lambda x v, T) = \{T =^? X \to Y\} \cup \mathsf{typconstr}(E \cup \{(x, X)\}, v, Y)$ where X and Y are new variables not occurring in E or T,
- $\mathsf{typconstr}(E, tu, T) = \mathsf{typconstr}(E, t, X \to T) \cup \mathsf{typconstr}(E, u, X)$ where X is a new variable not occurring in E or T,

- $\mathsf{typconstr}(E, t, T) = \bot$ otherwise.

Then, we can define the computable function infer which, given a term t, returns a pair (E, T) such that $E \rhd t : T$, as follows. Assume that $\mathrm{FV}(t) = \{x_1, \ldots, x_n\}$. Let $X_1, \ldots, X_n, Y$ be new distinct variables. If $\mathsf{typconstr}(E, t, Y)$ returns some unification problem P admitting a most general solution σ, then let $\mathsf{infer}(t) = (\sigma(E), \sigma(Y))$. Otherwise, let $\mathsf{infer}(t) = \bot$.

Solution of Exercise 28

We define a computable function check which, given an environment E, a term t and a type T, returns true if $E \rhd t : T$, and false otherwise, as follows. If $\mathsf{typconstr}(E, t, T)$ returns some unification problem P admitting a most general solution σ such that $\sigma(E) = E$ and $\sigma(T) = T$, then let $\mathsf{check}(E, t, T) = \mathsf{true}$. Otherwise, let $\mathsf{check}(E, t, T) = \mathsf{false}$.

Solution of Exercise 29

We prove that, if $E \rhd t : T$ and $E \rhd t : T'$, then $T = T'$, by induction on $E \rhd t : T$.

- Case $t = x$. Then, $T = E(x) = T'$.
- Case $t = \lambda x^U v$. Then, $T = U \to V$ and $T' = U \to V'$. But, by induction hypothesis, $V = V'$.
- Case $t = ab$. Then, there are U and U' such that $E \rhd a : U \to T$ and $E \rhd a : U' \to T'$. But, by induction hypothesis, $U \to T = U' \to T'$.

8.5 Section 6: First-order logic

Solution of Exercise 30

We consider a model on A and a valuation μ.

- $[\![\phi \Rightarrow \phi]\!]_\mu = \mathsf{imply}([\![\phi]\!]_\mu, [\![\phi]\!]_\mu) = \mathsf{true}$ by definition of imply.
- $[\![\neg\phi \vee \psi]\!]_\mu = \mathsf{or}(\mathsf{not}([\![\phi]\!]_\mu), [\![\psi]\!]_\mu) = \mathsf{imply}([\![\phi]\!]_\mu, [\![\phi]\!]_\mu) = [\![\phi \Rightarrow \phi]\!]_\mu$.
- $[\![\neg(\forall x \phi)]\!]_\mu = \mathsf{not}(\mathsf{forall}(\{[\![\phi]\!]_{\mu \cup \{x,a\}} \mid a \in A\}))$
  $= \mathsf{exists}(\{\mathsf{not}([\![\phi]\!]_{\mu \cup \{x,a\}}) \mid a \in A\}) = [\![\exists x \neg \phi]\!]_\mu$.

Solution of Exercise 31

We proceed by induction on the definition of $\vdash$. Let $\Gamma = \{\phi_1, \ldots, \phi_n\}$. In the following, we identity Γ and the formula $\phi_1 \wedge \ldots \wedge \phi_n$. Let A be a model and μ a valuation such that $\mathrm{dom}(\mu) \subseteq \mathrm{FV}(\Gamma)$ and $[\![\Gamma]\!]_\mu = \mathsf{true}$.

($\wedge$-intro) We have $[\![\phi \wedge \psi]\!] = \mathsf{and}([\![\phi]\!]_\mu, [\![\psi]\!]_\mu)$. By induction hypothesis, $[\![\phi]\!]_\mu = \mathsf{true}$ and $[\![\psi]\!]_\mu = \mathsf{true}$. Therefore, $[\![\phi \wedge \psi]\!] = \mathsf{true}$.

($\wedge$-elim-left) By induction hypothesis, we have $[\![\phi \wedge \psi]\!]_\mu = \mathsf{and}([\![\phi]\!]_\mu, [\![\psi]\!]_\mu) = \mathsf{true}$. Therefore, $[\![\phi]\!]_\mu = \mathsf{true}$.

($\wedge$-elim-right) Similar.

($\forall$-intro) We have $[\![\forall x \phi]\!]_\mu = \mathsf{forall}(\{[\![\phi]\!]_{\mu \cup \{x,a\}} \mid a \in A\})$. But, for all $a \in A$, by induction hypothesis, we have $[\![\phi]\!]_{\mu \cup \{x,a\}} = \mathsf{true}$. Therefore, $[\![\forall x \phi]\!]_\mu = \mathsf{true}$.

($\forall$-elim) By induction hypothesis, we have $[\![\forall x \phi]\!] = \mathsf{forall}(\{[\![\phi]\!]_{\mu \cup \{x,a\}} \mid a \in A\}) = \mathsf{true}$. Therefore, by definition of forall, we have $[\![\phi\{x \mapsto t\}]\!]_\mu = \mathsf{true}$ since $[\![\phi\{x \mapsto t\}]\!]_\mu = [\![\phi]\!]_{\mu \cup \{x \mapsto a\}}$ where $a = [\![t]\!]_\mu$ (can be proved by induction on ϕ).

...