Lecture Notes
Semantics Course WS 2013

Gert Smolka with Steven Schäfer
Saarland University

February 3, 2014

Preface

The course is about the theory of programming languages. We study in depth a number of standard models for programming languages. In these notes, the term programming language includes logical languages like the language of the proof assistant Coq. We use Coq throughout the course to formalize the models we study. Coq will provide us with an example of a powerful logical programming language.

We assume that you are familiar with Coq (first half of the ICL course). With Coq we mean several things: The type theory underlying Coq, the Coq programming language, and the Coq programming system. In this course you will learn a lot about the type theory underlying Coq. However, we assume that you are already familiar with the basic ideas of type theory and their realization in Coq.

Contents

1 Untyped Lambda Calculus, First Encounter

The basic model in the theory of programming languages is the untyped lambda calculus [Church 1932]. The untyped lambda calculus can be seen as a small functional programming language that can express all computable functions. Turing showed in 1937 that his machines and untyped lambda calculus have equivalent computational power. While Turing's machines compute with strings, the untyped lambda calculus computes with functions.

We start with a informal presentation of the untyped lambda calculus and discuss important ideas and results. Later in the course we will formalize the lambda calculus and prove some results.

From a programming point of view, every object of the untyped lambda calculus is a function. Thus other objects like numbers must be represented as functions.

Coq encompasses a typed version of the lambda calculus. Thus experience with Coq will be helpful in understanding the untyped lambda calculus.

The standard reference for untyped lambda calculus is Barendregt [1]. A more gentle textbook introduction is Hindley and Seldin [2].

1.1 Terms

The abstract syntax of the untyped lambda calculus consists of **terms**.

$$s, t ::= x \mid \lambda x.s \mid st \qquad (x \in \mathbb{N})$$

Terms of the form $\lambda x.s$ are called **abstractions** and introduce local variables. One often says that the symbol λ acts as a **variable binder**. A term is **closed** if no variable occurs free in it (i.e., every variable occurrence is local). Closed terms are also called **combinators**.

Terms that are equal up to consistent renaming of bound variables are identified. For instance, $(\lambda x.x) = (\lambda y.y)$. We call this important assumption the **α-law**. The α-law considerably complicates the formal definition of terms (to be studied later).

The closed terms of the untyped lambda calculus describe functions. Every function of the lambda calculus takes one argument. However, since functions return functions, we can apply a function to any number of arguments. This is reflected in a basic notation for terms:

$$stu \rightsquigarrow (st)u$$

Here is a list of **prominent combinators**:

$$\begin{aligned} I &:= \lambda x.x \\ K &:= \lambda xy.x \\ B &:= \lambda fgx.f(gx) \\ C &:= \lambda fxy.fyx \\ S &:= \lambda fgx.fx(gx) \\ \omega &:= \lambda x.xx \\ \Omega &:= \omega\omega \end{aligned}$$

The combinator B can be seen as a composition operator. Sometimes we will write $s \circ t$ for Bst.

1.2 Substitution and Beta Reduction

The idea that terms describe functions is captured by a computation rule known as β-reduction. For the definition of β-reduction we need the notion of substitution.

The notation s_t^x stands for the term that is obtained from the term s by replacing every free occurrence of the variable x with the term t. From the α-law it follows that we have

$$\begin{aligned} (\lambda x.fxy)_x^y &= \lambda z.fzx \\ (\lambda x.fxy)_x^y &\neq \lambda x.fxx \end{aligned}$$

if x, y, z, and f are distinct variables. One speaks of **capture-free substitution**.

A **β-redex** is a term of the form $(\lambda x.s)t$. A **β-reduction** replaces a β-redex $(\lambda x.s)t$ with the term s_t^x. We write $s \succ t$ if t can be obtained from s by reducing some β-redex in s. For instance, we have

$$(\lambda x.s)t \succ s_t^x$$

for a top level β-reduction. The formal definition of **β-reduction** is inductive.

$$\frac{}{(\lambda x.s)t \succ s^x_t} \qquad \frac{s \succ s'}{\lambda x.s \succ \lambda x.s'} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

We write $s \succ^n t$ if t can be obtained from s with n β-reductions, and $s \succ^* t$ if $s \succ^n t$ for some $n \geq 0$ (...reflexive transitive closure). Here are examples.

$$\begin{aligned} \omega\omega &\succ \omega\omega \\ SKK &\succ^2 \lambda x.Kx(Kx) \succ I \end{aligned}$$

The first example shows that β-reduction does not always terminate. This is to be expected for a Turing-complete system.

A term is **normal** if it cannot be β-reduced. Obviously, a term is normal if and only if it contains no β-redex. While the combinators I, K, S, and ω are normal, the combinator Ω is not.

Given two terms s and t, we say that **s evaluates to t** and write **$s \Downarrow t$** if $s \succ^* t$ and t is normal. If s evaluates to t, we say that t is a **normal form** of s. Note that the term Ω has no normal form. A main result about the lambda calculus says that every term has at most a normal form.

An **interpreter** for the lambda calculus is an algorithm that given a term computes a normal form of the term whenever there is one. From the term $KI\Omega$ we learn that the naive strategy that reduces some beta redex as long as there is one does not suffice for an interpreter (since $KI\Omega \succ KI\Omega$ but also $KI\Omega \succ I$). It is known that the strategy that always reduces the leftmost outermost β-redex finds a normal form whenever there is one.

In Coq we have β-redexes and β-reduction for typed terms. The typing ensures that β-reduction always terminates. Thus every term has a normal form in Coq.

We say that a term is **weakly normalizing** if it has a normal form. A term is **strongly normalizing** if there is no infinite β-reduction chain issuing from it. The term $KI\Omega$ is weakly normalizing but not strongly normalizing. If a term is strongly normalizing, we can compute a normal form of it by just reducing β-redexes as long as we can. We can define strong normalization inductively.

$$\frac{\forall t.\, s \succ t \to SN\ t}{SN\ s}$$

According to this definition, normal terms are strongly normalizing since they do not have successors and thus all their successors are strongly normalizing.

Exercise 1 (Substitution) Convince yourself that the following equations hold.

$$\begin{array}{rcll}
(st)^x_u & = & s^x_u\,t^x_u & \\
s^x_x & = & s & \\
s^x_t & = & s & \text{if } x \text{ not free in } s \\
(\lambda x.s)^y_t & = & \lambda x.s^y_t & \text{if } x \neq y \text{ and } x \text{ not free in } t \\
\lambda x.s & = & \lambda y.s^x_y & \text{if } y \text{ not free in } s
\end{array}$$

1.3 Beta Equivalence

The lambda calculus comes with a canonical equivalence relation on terms. Informally, two closed terms are equivalent if and only if they describe the same function. We leave the notion of described function informal and define equivalence of terms formally.

Beta equivalence is the equivalence closure β-reduction. The inductive definition of β-equivalence looks as follows.

$$\frac{s \succ t}{s \equiv t} \qquad \frac{}{s \equiv s} \qquad \frac{s \equiv t}{t \equiv s} \qquad \frac{s \equiv t \quad t \equiv u}{s \equiv u}$$

Beta equivalence satisfies

$$s \succ^* t \;\rightarrow\; s \equiv t$$

and the **congruence laws**

$$\begin{array}{l}
s \equiv s' \;\rightarrow\; \lambda x.s \equiv \lambda x.s' \\
s \equiv s' \;\rightarrow\; t \equiv t' \;\rightarrow\; st \equiv s't'
\end{array}$$

Another important property of β-equivalence is **substitutivity**.

$$s \equiv t \;\rightarrow\; s^x_u \equiv t^x_u$$

A nontrivial property of β-equivalence is the **Church-Rosser property**:

$$s \equiv t \;\rightarrow\; \exists u.\, s \succ^* u \wedge t \succ^* u$$

Thus two terms are β-equivalent if and only if there is a term to which they both reduce. Here are three important consequences of the Church-Rosser property (all of them have straightforward proofs).

$$\begin{array}{l}
s \equiv t \rightarrow s \Downarrow u \rightarrow t \Downarrow u \\
s \equiv t \rightarrow t \text{ normal} \rightarrow s \Downarrow t \\
s \equiv t \rightarrow s, t \text{ normal} \rightarrow s = t
\end{array}$$

These facts are of great importance for computational correctness proofs. If we want to argue that a term s evaluates to a term t or that two terms are equivalent as it comes to evaluation, we can use equational reasoning based on β-equivalence. This is much easier than arguing about β-reduction directly.

Beta equivalence is an example of what is called a convertibility relation. We will say that two terms are **convertible** if they are β-equivalent.

Beta equivalence does not satisfy the so-called **η-law**:

$$\lambda x.sx \equiv s \qquad \text{if } x \text{ is not free in } s$$

This is in contrast to the convertibility relation underlying Coq, which satisfies a typed version of the η-law. It is possible to consider untyped lambda calculus with both β- and η-reduction. For now, we will just consider β-reduction.

Exercise 2 Prove the following equivalences.

a) $SKK \equiv I$

b) $BCCfxy \equiv fxy$

1.4 Church Numerals

Church encoded a natural number n as a function iterating a given function n-times:

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

We call the term $\hat{n}$ the **Church numeral for n**. Note that the Church numerals are normal combinators. This ensures that numerals for different numbers are not β-equivalent (Church-Rosser property).

Sometimes it is helpful to think of a numeral $\hat{n}$ as an operator $\lambda f.f^n$ that applied to a function f yields the function f^n.

1.4.1 Successor Function

We express the successor function as the following normal combinator.

$$\mathit{succ} := \lambda nfx.f(nfx)$$

The proof of the correctness statement

$$succ\,\hat{n} \Downarrow \widehat{Sn}$$

is straightforward:

$$succ\,\hat{n} \succ \lambda fx.\ f(\hat{n}fx) \succ^2 \lambda fx.\ f(f^n x) = \widehat{Sn}$$

At this point an explicit definition of the notation $s^n t$ is helpful.

$$\begin{aligned} s^0 t &:= t \\ s^{Sn} t &:= s(s^n t) \end{aligned}$$

We have the following equivalence for the successor function (f and g are distinct variables).

$$succ\,f\,g \;\equiv\; g \circ fg$$

1.4.2 Addition, Multiplication, Exponentiation

The following equations fully characterize addition, multiplication, and exponentiation of natural numbers.

$$\begin{array}{ccc} 0 + n = n & 0 \cdot m = 0 & m^0 = 1 \\ Sm + n = S(m+n) & Sm \cdot n = n + m \cdot n & m^{Sn} = m \cdot m^n \end{array}$$

We refer to these equations as **Dedekind equations**. They provide the basis for our encoding of addition, multiplication, and exponentiation.

$$\begin{aligned} add &:= \lambda m.m\,succ \\ mul &:= \lambda mn.m\,(add\,n)\,\hat{0} \\ exp &:= \lambda mn.n\,(mul\,m)\,\hat{1} \end{aligned}$$

One can show that *add*, *mul*, and *exp* satisfy the Dedekind equations for Church numerals modulo β-equivalence (e.g., $add\ \hat{0}\ \hat{n} \equiv \hat{n}$). With natural induction we can then prove the following equivalences:

$$\begin{aligned} add\,\hat{m}\,\hat{n} &\equiv \widehat{m+n} \\ mul\,\hat{m}\,\hat{n} &\equiv \widehat{m \cdot n} \\ exp\,\hat{m}\,\hat{n} &\equiv \widehat{m^n} \end{aligned}$$

Since the right hand sides of the equivalences are normal, we obtain the following correctness properties with the Church-Rosser property.

$$\begin{aligned} add\,\hat{m}\,\hat{n} &\Downarrow \widehat{m+n} \\ mul\,\hat{m}\,\hat{n} &\Downarrow \widehat{m \cdot n} \\ exp\,\hat{m}\,\hat{n} &\Downarrow \widehat{m^n} \end{aligned}$$

1.4.3 More Equivalences for Church Numerals

One can show the following β-equivalences.

$$\begin{aligned}\widehat{m+n} &\equiv \lambda f.\ \widehat{m}f \circ \widehat{n}f\\ \widehat{m \cdot n} &\equiv \widehat{m} \circ \widehat{n}\end{aligned}$$

The equivalences can be used to encode addition and multiplication. In the lambda calculus with β- and η-reduction, we have the following equivalence for exponentiation.

$$\widehat{m^n} \equiv nm$$

This equivalence does not hold without η-reduction:

$$\widehat{m^0} = \hat{1} = \lambda fx.fx \not\equiv \lambda x.x \prec \hat{0}m$$

1.5 Pairs, Predecessor, and Primitive Recursion

At first, writing a predecessor function for Church numerals seems difficult. The trick is to iterate on pairs. We start from the pair $(0,0)$ and iterate n-times to obtain the pair $(n, n-1)$.

$$(0,0) \rightsquigarrow (1,0) \rightsquigarrow (2,1) \rightsquigarrow \ \cdots \ \rightsquigarrow (n, n-1)$$

1.5.1 Pairs

We encode pairs as follows.

$$\begin{aligned}\mathit{pair} &:= \lambda xyf.fxy\\ \mathit{fst} &:= \lambda p.p(\lambda xy.x)\\ \mathit{snd} &:= \lambda p.p(\lambda xy.y)\end{aligned}$$

The following equivalences are easy to prove:

$$\begin{aligned}\mathit{fst}\ (\mathit{pair}\, x\, y) &\equiv x\\ \mathit{snd}\ (\mathit{pair}\, x\, y) &\equiv y\end{aligned}$$

1.5.2 Primitive Recursion

Primitive recursion is a definition scheme for functions on the natural numbers introduced by Peano in 1889 as a compagnon to natural induction. We will define a primitive recursion combinator *prec* satisfying the equivalences

$$\begin{aligned} \mathit{prec}\,x\,f\,\hat{0} &\equiv x \\ \mathit{prec}\,x\,f\,\widehat{Sn} &\equiv f\,\hat{n}\,(\mathit{prec}\,x\,f\,\hat{n}) \end{aligned}$$

The trick is to iterate on pairs:

$$(\hat{0}, s) \rightsquigarrow (\hat{1}, t\,\hat{0}\,s) \rightsquigarrow (\hat{2}, t\,\hat{1}\,(t\,\hat{0}\,s)) \rightsquigarrow \cdots$$

This leads to the following definition.

$$\begin{aligned} a &:= \lambda x.\ \mathit{pair}\ \hat{0}\ x \\ \mathit{step} &:= \lambda f p.\ \mathit{pair}\ (\mathit{succ}\ (\mathit{fst}\ p))\ (f\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)) \\ \mathit{prec} &:= \lambda x f n.\ \mathit{snd}\ (n\ (\mathit{step}\ f)\ (a x)) \end{aligned}$$

Showing the first correctness equivalence for *prec* is easy. For the second correctness equivalence we need the following lemma.

$$\widehat{Sn}\ (\mathit{step}\ f)\ (ax) \equiv \mathit{pair}\ \widehat{Sn}\ (f\ \hat{n}\ (\mathit{snd}\ (\hat{n}\ (\mathit{step}\ f)\ (ax))))$$

The lemma follows by induction on n.

1.5.3 Predecessor

The predecessor operation can now be expressed with primitive recursion.

$$\mathit{pred} := \mathit{prec}\ \hat{0}\ (\lambda x y.x)$$

The correctness proof is straightforward using the correctness equivalences for *prec*.

$$\begin{aligned} \mathit{pred}\,\hat{0} &\equiv \hat{0} \\ \mathit{pred}\,\widehat{Sn} &\equiv \hat{n} \end{aligned}$$

1.6 Church Numerals in Coq

We will now represent Church numerals and their operations in Coq and prove the correctness of the operations. This will deepen our understanding of Church numerals and raise some interesting issues about Coq.

Since Coq is typed we must represent Church numerals as typed functions. We represent Church numerals as members of the type

```
Definition Nat : Prop := ∀ X : Prop, (X → X) → X → X.
```

It is crucial that the variable X ranges over propositions rather than general types. This will be explained later.

We define a function N that maps a number n to the numeral $\hat{n}$.

```
Definition zero : Nat := fun X f x ⇒ x.
Definition succ : Nat → Nat := fun n X f x ⇒ f (n X f x).
Definition N : nat → Nat := fun n ⇒ nat_iter n succ zero.
```

Use the command *Compute N 7* to see the Church numeral for 7. Following the Dedekind equations, we express addition, multiplication, and exponentiation as follows.

```
Definition add : Nat → Nat → Nat := fun m ⇒ m Nat succ.
Definition mul : Nat → Nat → Nat := fun m n ⇒ m Nat (add n) (N 0).
Definition exp : Nat → Nat → Nat := fun m n ⇒ n Nat (mul m) (N 1).
```

We can now prove the following correctness statements.

- succ (N n) = N (S n)
- add (N m) (N n) = N (m + n).
- mul (N m) (N n) = N (m * n).
- exp (N m) (N n) = N (pow m n).

All proofs are straightforward and are based on the characteristic equations for the operations, which hold by conversion. The proofs for addition and multiplication are by induction on m, and the proof for exponentiation is by induction on n, as one would expect from the definitions. The correctness proof for addition looks as follows.

```
Lemma add_correct m n :
  add (N m) (N n) = N (m + n).
Proof.
  induction m; simpl.
  - reflexivity.
  - change (add (N (S m)) (N n)) with (succ (add (N m) (N n))).
    now rewrite IHm.
Qed.
```

1.6.1 Predicativity of Coq's Universe *Type*

We now explain why the definition

```
Definition Nat : Type := ∀ X : Type, (X → X) → X → X.
```

does not work. The reason is that *Type* is a **predicative universe**. This means that function of a type $A := \forall X : Type.\ s$ can only be applied to types that are *smaller than* A. In particular, if f is a function of type A, then f cannot be applied to A. As a consequence, the definition

```
Definition add : Nat → Nat → Nat := fun m ⇒ m Nat succ.
```

will not type check since $m : Nat$ is applied to *Nat*.

In contrast, *Prop* is an **impredicative universe** where a size restriction on types does not exist. The universe *Type* can not be made impredicative since this would result in an inconsistent system where *False* is provable. This is a basic fact of logic that cannot be massaged away.

Since *Nat* is a proposition, it follows that we express numerals and operations on numerals as proofs. So our representation of Church numerals shows that Coq's proof language has considerable computational power. Since numerals are proofs, we cannot show in Coq that the embedding function $N : nat \to Nat$ is injective (because of the elim restriction). Nevertheless, we can observe from the outside that N yields different numerals for different numbers.

1.6.2 Church Exponentiation

Since Coq has η-conversion, we can prove the equation $\widehat{m^n} = \hat{m}\hat{n}$ and use it to obtain an exponentiation function. We speak of **Church exponentiation**. For the proof to go through, we need suitable equations for multiplication and addition. The following works.

```
Definition add : Nat → Nat → Nat := fun m n X f x ⇒ m X f (n X f x).
Definition mul : Nat → Nat → Nat := fun m n X f ⇒ m X (n X f).
Definition exp : Nat → Nat → Nat := fun m n X ⇒ n (X → X) (m X).
Lemma add_correct m n : N (m + n) = add (N m) (N n).
Lemma mul_correct m n : N (m * n) = mul (N m) (N n).
Lemma exp_correct m n : N (pow m n) = exp (N m) (N n).
```

Interestingly, the above encodings of addition, multiplication, and exponentiation will type check if *Nat* is defined with *Type* rather than *Prop* since they do not require an application of a numeral to the type *Nat*. However, once we encode the predecessor operation, the typing problem will reoccur and cannot be avoided.

1.7 Fixed Point Combinators and Recursive Functions

A **fixed point** of a function f is an argument x such that $fx = x$. A **fixed point combinator** is a combinator R such that

$$Rs \equiv s(Rs)$$

for every term s. We can see a fixed point combinator as a function that yields a fixed point for every function of the lambda calculus. Turing defined a fixed point combinator T as follows:[1]

$$\begin{aligned} A &:= \lambda xf.\ f(xxf) \\ \Theta &:= AA \end{aligned}$$

Since $\Theta f \succ^3 f(\Theta f)$, we have that Θ is a fixed point combinator. It follows that every function of the lambda calculus has a fixed point.

With a fixed point combinator we can construct recursive functions. To do so, we represent the desired recursive function as an ordinary function $F := \lambda fx.s$ where f serves as the name of the recursive function. We call F the **functional** for the recursive function. Given a fixed point combinator R, the term RF describes the desired recursive function:

$$RF \equiv F(RF) \equiv \lambda x.s^f_{RF}$$

This straightforward construction of recursive functions may be surprising at first. What is technically needed are functional arguments and self application of functions.

Exercise 3 Prove that the following term is a fixed point combinator.

$$Y := \lambda f.\ (\lambda x.f(xx))\,(\lambda x.f(xx))$$

Exercise 4 Prove that no normal fixed point combinator exists. The following facts are helpful (WN means weakly normalizing).

$$\begin{aligned} WN\ s &\rightarrow WN\,(sx) \\ s \Downarrow t &\rightarrow xs \Downarrow xu \end{aligned}$$

Exercise 5 Let s be a term and f and x be distinct variables. Find a term t such that $tx \equiv s^f_t$. Prove that your term t satisfies the equivalence. One says that t solves the equation $fx = s$ for the function variable f. Note that the existence of t justifies the usual equational form of recursive definitions.

[1] Barendregt [1] writes Θ for T.

1.8 Scott Numerals

Church numerals are not the simplest representation of numbers. Given that we have unrestricted functional recursion, it suffices that the functions representing the numbers provide for the case analysis coming with natural numbers (as match does in Coq). In fact, it suffices that the numerals satisfy the following equivalences ($\overline{n}$ is the numeral for n):

$$\begin{aligned} \overline{0}\,x\,f &\equiv x \\ \overline{Sn}\,x\,f &\equiv f\,\overline{n} \end{aligned}$$

Note that the Church numerals do not satisfy these equivalences. However, the definition of numerals satisfying the equivalences is straightforward:

$$\begin{aligned} \overline{0} &:= \lambda x f.x \\ \overline{Sn} &:= \lambda x f.\,f\,\overline{n} \end{aligned}$$

We call these numerals **Scott numerals** in honor of Dana Scott who identified the scheme for general constructor types (the trick is always to model the match coming with the constructor type). Note that Scott numerals are normal combinators, and that Scott numerals for different numbers are different normal terms. It is straightforward to define functions that yield successors and predecessors.

$$\begin{aligned} \mathit{succ} &:= \lambda n x f.\,f n \\ \mathit{pred} &:= \lambda n.\,n\,\overline{0}\,I \end{aligned}$$

The equivalence $\mathit{pred}\,(\mathit{succ}\ s) \equiv s$ follows for every term s by β-reduction:

$$\mathit{pred}\,(\mathit{succ}\ s) \succ \mathit{succ}\ s\,\overline{0}\,I \succ^3 Is \succ s$$

Following the Dedekind equations, we can now define addition, multiplication, and exponentiation as follows:

$$\begin{aligned} \mathit{add} &:= \Theta\,(\lambda f m n.\ m\,n\,(\lambda m'.\ \mathit{succ}\,(f m' n))) \\ \mathit{mul} &:= \Theta\,(\lambda f m n.\ m\,\hat{0}\,(\lambda m'.\ \mathit{add}\,n\,(f m' n))) \\ \mathit{exp} &:= \Theta\,(\lambda f m n.\ n\,\hat{1}\,(\lambda n'.\ \mathit{mul}\,m\,(f m n'))) \end{aligned}$$

Exercise 6 Verify that our representation of pairs is Scott style.

Exercise 7 Do the following for Scott numerals.

a) Verify the correctness of addition, multiplication, and exponentiation.

b) Define and verify a factorial function.

c) Define and verify a higher-order function for primitive recursion.

Exercise 8 Represent lists in Scott style.

a) Define *nil* and *cons*.

b) Define and verify *hd* and *tl*.

c) Define and verify length and concatenation.

1.9 SK-Terms and SK-Reduction

SK-terms are defined inductively:

1. Variables are SK-terms.
2. K and S are SK-terms.
3. st is an SK-term if s and t are SK-terms.

Theorem 1 Every term is equivalent to an SK-term.

The proof of the theorem is based on an **abstraction operator** that for a variable x and an SK-term s yields an SK-term ${}^x s$ such that ${}^x s \equiv \lambda x.s$. The abstraction operator is defined by structural recursion on SK-terms:

$$\begin{array}{rcll} {}^x x & := & SKK & \\ {}^x s & := & Ks & \text{if } x \text{ not free in } s \\ {}^x st & := & S\,{}^x s\,{}^x t & \text{otherwise} \end{array}$$

Exercise 9 Prove ${}^x s \equiv \lambda x.s$ for every variable x and every SK-term s.

We now define a **translation operator** $[s]$ translating every term into an equivalent SK-term by structural recursion on terms.

$$\begin{array}{rcl} [x] & := & x \\ [st] & := & [s][t] \\ [\lambda x.s] & := & {}^x[s] \end{array}$$

Exercise 10 Let s be a term. Prove that $[s]$ is an SK-term such that $[s] \equiv s$.

Weak reduction $s \succ_w t$ is a binary relation inductively defined on terms:

$$\frac{}{Kst \succ_w s} \qquad \frac{}{Sstu \succ_w su(tu)} \qquad \frac{s \succ_w s'}{st \succ_w s't} \qquad \frac{t \succ_w t'}{st \succ_w st'}$$

Here are facts about weak reduction.

- Weak reduction does not involve substitution.
- Weak reduction cannot happen below lambda (i.e., inside abstractions).
- Weak reduction applies only to β-redexes of the form Kst or $Sstu$.
- If $s \succ_w t$, then $s \succ^2 t$ or $s \succ^3 t$.
- If $s \succ_w t$ and s is an SK-term, then t is an SK-term. In other words, SK-terms are closed under weak reduction.

Exercise 11 Find two terms s and t such that $s \succ t$ but not $[s] \succ_w^* [t]$. Hint: Consider $\lambda x.Ix$.

1.10 Combinatory Logic

Suppose we just consider SK-terms and weak reduction. It turns out that this subsystem of the lambda calculus is still Turing-complete. Moreover, in this system there is no need for lambdas, all we need are two constants taking the role of the combinators S and K. The resulting system is known as **combinatory logic** (CL).[2]

The **terms** of the CL are defined as follows.

$$s, t \ ::= \ x \mid K \mid S \mid st \qquad (x \in \mathbb{N})$$

Note that S and K are now constants rather than λ-terms. The **reduction** relation for combinatory logic is defined like weak reduction.

$$\frac{}{Kst \succ s} \qquad \frac{}{Sstu \succ su(tu)} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

Equivalence of CL-terms is defined analogously to the lambda calculus.

$$\frac{s \succ t}{s \equiv t} \qquad \frac{}{s \equiv s} \qquad \frac{s \equiv t}{t \equiv s} \qquad \frac{s \equiv t \quad t \equiv u}{s \equiv u}$$

As in the lambda calculus, equivalence in CL satisfies the Church-Rosser property. Thus normal forms are unique and correctness proofs in CL have the structure familiar from lambda calculus. Moreover, leftmost outermost reduction yields a normal form whenever there exists one.

[2] CL originated in Haskell Curry's doctoral dissertation in 1930 (supervised by Hilbert in Göttingen). Some of the ideas including the combinators S and K already appeared in 1924 in a very readable paper by Moses Schönfinkel [3].

Since CL has no binders, the definition of **substitution** is straightforward.

$$\begin{aligned} x^x_u &:= u \\ y^x_u &:= y && \text{if } x \neq y \\ K^x_u &:= K \\ S^x_u &:= S \\ (st)^x_u &= s^x_u\, t^x_u \end{aligned}$$

Reduction and equivalence in CL are substitutive.

We define combinators describing identity and composition.

$$\begin{aligned} I &:= SKK \\ B &:= S(KS)K \end{aligned}$$

The reductions verifying the correctness of our definitions are easy to show.

$$\begin{aligned} Ix &\succ^* x \\ Bxyz &\succ^* x(yz) \end{aligned}$$

We represent the natural numbers in Scott-style.

$$\begin{aligned} \overline{0} &:= S(KK)I \\ \overline{Sn} &:= K(SI(K\overline{n})) \end{aligned}$$

The correctness of the representation is easy to verify.

$$\begin{aligned} \overline{0}xf &\succ^* x \\ \overline{Sn}xf &\succ^* f\overline{n} \end{aligned}$$

There is also a fixed point combinator.

$$\begin{aligned} \omega &:= SII \\ A &:= B(SI)\omega \\ \Theta &:= AA \end{aligned}$$

We have

$$\begin{aligned} \omega x &\succ^* xx \\ \Theta f &\succ^* f(\Theta f) \end{aligned}$$

1.11 Abstraction Operator for CL

We have already seen that S and K can simulate abstraction in the lambda calculus. Thus it is not surprising that we can define an abstraction operator for CL. We define an **abstraction operator** for CL following Hindley and Seldin [2].[3]

$$\begin{array}{rcll} {}^x x & := & I & \\ {}^x s & := & Ks & \text{if } x \text{ does not occur in } s \\ {}^x sx & := & s & \text{if } x \text{ does not occur in } s \\ {}^x st & := & S\,{}^x s\,{}^x t & \text{otherwise} \end{array}$$

Note that x does not occur in the term ${}^x s$. The defined abstraction in CL behaves much like the native lambda abstraction in lambda calculus. We have the following [2].

$$\begin{array}{rcll} ({}^x s)t & \succ^* & s^x_t & \\ ({}^{x_1 \dots x_n} s)t_1 \dots t_n & \succ^* & s^{x_1 \dots x_n}_{t_1 \dots t_n} & \text{if } x_1 \dots x_n \text{ are distinct variables not appearing in } t_1 \dots t_n \\ ({}^x s)^y_u & = & {}^x(s^y_u) & \text{if } x \neq y \text{ and } x \text{ does not occur in } u \end{array}$$

There is an important difference between native abstraction in the lambda calculus and defined abstraction in CL. Native abstraction satisfies the so-called **ξ-law**

$$s \equiv t \;\rightarrow\; \lambda x.s \equiv \lambda x.t$$

while defined abstraction does not. For instance, with

$$\begin{array}{rcl} s & := & Sxyz \\ t & := & xz(yz) \end{array}$$

we have

$$\begin{array}{rcl} {}^x s & = & S(SS(Ky))(Kz) \\ {}^x t & = & S(SI(Kz))(K(yz)) \end{array}$$

Thus $s \equiv t$ and ${}^x s \not\equiv {}^x t$ in CL (since ${}^x s$ and ${}^x t$ are normal and different).

Speaking operationally, the failure of the ξ-law means that we cannot reduce inside defined abstractions in CL. We remark that functional programming languages like ML or Haskell do not provide for reduction inside their native abstractions. So the failure of the ξ-law does not affect the Turing-completeness of CL.

[3] There are many possible abstraction operators for CL. In our definition, the third equation may be omitted and the second equation may be restricted to variables and the constants K and S. Both modifications result in larger terms. The third equation can be seen as an η-reduction.

We can compile the terms of the lambda calculus into terms of CL. However, this does not provide a faithfull implementation of β-reduction. For instance, we have $\lambda x.Kxx \succ^* \lambda x.x$ but ${}^xKxx = SK(SKK)$ and ${}^xx = SKK$ are different normal forms. The problem is that lambda calculus provides for reductions inside abstractions and that the translation with S takes away redexes.

One may consider weak β-reduction that cannot take place within abstractions. However, such a weak lambda calculus is crippled since it does not enjoy the Church-Rosser property. For instance, the term $K(II)$ would have two normal forms in such a calculus ($\lambda x.I$ and $\lambda x.II$). Recall that CL enjoys the Church-Rosser property.

Exercise 12 Prove $({}^xs)t \succ^* s^x_t$ in CL.

1.12 Church Numerals in CL

The canonical translation of Church numerals to CL does not work since $succ\ \overline{0} \succ^3 \overline{1}$ requires two β-reductions inside an abstraction. Thus using the canonical translation to CL, $succ\ \overline{0}$ will not reduce to $\overline{1}$. The problem can be circumvented by using the following definitions in CL.

$$\begin{aligned} zero &:= KI \\ succ &:= SB \\ \hat{n} &:= succ^n zero \end{aligned}$$

Exercise 13 Prove $\hat{n}fx \succ^* f^nx$ in CL.

Exercise 14 Prove the following in lambda calculus using the definitions from CL.

$$\begin{aligned} zero &\succ^* \lambda fx.x \\ succ &\succ^* \lambda nfx.\ f(nfx) \\ \hat{n} &\succ^* \lambda fx.\ f^nx \end{aligned}$$

1.13 Call-by-Value Reduction

We return to the lambda calculus and consider a restricted version of β-reduction known as **call-by-value reduction (CBV)**. In this context a **value** is a term of the form $\lambda x.s$. The inductive definition of CBV is as follows.

$$\frac{t \text{ is a value}}{(\lambda x.s)t \succ_V s^x_t} \qquad \frac{s \succ_V s'}{st \succ_V s't} \qquad \frac{s \text{ is a value} \qquad t \succ_V t'}{st \succ_V st'}$$

Every normal term is CBV-normal, but $(\lambda x.xx)x$ and $\lambda x.II$ are examples of CBV-normal terms that are not normal. CBV-reduction is **deterministic**, that is, if $s \succ_V t$ and $s \succ_V t'$, then $t = t'$.

The lambda calculus with cal-by-value reduction is still Turing-complete. Numbers can be represented with Scott numerals (see Section 1.8):

$$\begin{aligned} \overline{0} &:= \lambda xf.x \\ \overline{Sn} &:= \lambda xf.\,f\overline{n} \end{aligned}$$

The successor and predecessor functions

$$\begin{aligned} \mathit{succ} &:= \lambda nxf.\,fn \\ \mathit{pred} &:= \lambda n.\,n\overline{0}I \end{aligned}$$

still reduce as they should

$$\begin{aligned} \mathit{succ}\,\overline{n} &\succ_V \overline{Sn} \\ \mathit{pred}\,(\mathit{succ}\,\overline{n}) &\succ_V \overline{n} \end{aligned}$$

There is also a combinator providing for recursive functions.

$$\begin{aligned} A_V &:= \lambda xf.\,f(\lambda y.xxfy) \\ \Theta_V &:= A_V A_V \end{aligned}$$

For every abstraction s we have

$$\Theta_V s \;=\; A_V A_V s \;\succ_V^2\; s(\lambda y.\,A_V A_V s y) \;=\; s(\lambda y.\,\Theta_V s y)$$

So Θ_V is a fixed point combinator for abstractions up to η-conversion. This suffices for recursive functions since $\Theta_V s$ and $\lambda y.\,\Theta_V s y$ do the same for all arguments. Combinators like Θ_V are often called **call-by-value fixed point combinators**.

Call-by-value reduction is more complex than unrestricted β-reduction since the elegant interplay between β-reduction and β-equivalence is lost. The reason for considering CBV-reduction is that is used in most execution-oriented programming languages. We will return to CBV-reduction when we consider the typed programming language PCF.

Exercise 15 Compute the normal forms of the following terms in CBV-lambda calculus.

a) $II(II)$

b) $K(\omega(\lambda x.y\Omega))$

1.14 Call-by-Name Reduction

There is also **call-by-name reduction (CBN)**, which is defined as follows.

$$\frac{}{(\lambda x.s)t \succ_N s^x_t} \qquad \frac{s \succ_N s'}{st \succ_N s't}$$

With CBN-reduction, Θ is still a fixed point combinator, that is, $\Theta f \succ_N^* f(\Theta f)$. Moreover, the basic operations on Scott numerals work as expected:

$$\begin{aligned} \overline{0}xf &\succ_N x \\ \overline{Sn}xf &\succ_N f\overline{n} \\ \mathit{succ}\,\overline{n} &\succ_N \overline{Sn} \\ \mathit{pred}\,(\mathit{succ}\,\overline{n}) &\succ_N \overline{n} \end{aligned}$$

However, since argument terms are never evaluated, there is a problem with nested operations:

$$\mathit{succ}\,(\mathit{succ}\,\overline{0}) \succ_N \lambda xf.f(\mathit{succ}\,\overline{0}) \neq \overline{2}$$

We conclude that call-by-name reduction does not make sense for pure lambda calculus.

1.15 Summary

We have considered three different untyped reduction systems:

- $\lambda\beta\eta$, lambda calculus with β- and η-reduction.
- $\lambda\beta$, lambda calculus with β-reduction.
- CL, combinatory logic with K- and S-reduction.

All three systems satisfy the Church-Rosser property, where equivalence in each case is defined as the equivalence closure the reduction relation. Reduction can take at every subterm, which makes equivalence congruent (i.e., compatible with the term structure). This setup provides for equational correctness proofs. Computation reduces a term until a normal form is reached. If always the leftmost-outermost redex is reduced, a normal form will be reached if there exists one. The Church-Rosser property ensures that the reachability of a normal form is preserved by arbitrary reduction steps.

All three systems compute with functions and only with functions. All three systems have a fixed point combinator R such that $Rf \succ^* f(Rf)$. Such every definable function has a fixed point.

All three systems can define functions in equational style:

$$fx_1 \dots x_n \;:=\; s$$

This means that there is an algorithm that giben the equation constructs a term t such that the equivalence

$$tx_1 \dots x_n \;\equiv\; s^f_t$$

holds. In the nonrecursive case

$$t \;:=\; \lambda x_1 \dots x_n.s$$

does the job. In the recursive case

$$t \;:=\; R(\lambda f x_1 \dots x_n.s)$$

does the job where R is some fixed point combinator. In CL, the lambdas used above can be simulated with an abstraction operator.

Constructor-based data structures can be represented with Scott's encoding. For numbers, pairs, and lists we obtain the following.

$$\begin{aligned} zero \;&:=\; \lambda xf.x \\ succ\; a \;&:=\; \lambda xf.fa \\ pair\; a\, b \;&:=\; \lambda f.fab \\ nil \;&:=\; \lambda xf.x \\ cons\; a\, b \;&:=\; \lambda xf.fab \end{aligned}$$

The trick is that a constructor yields the match for the constructed value, where the match is represented as a function taking a "continuation" argument for every constructor. Thus reduction can in fact perform the match. One can also verify that the constructors are injective on normal forms. Moreover, different constructors for the same data structure always yield different results .

From the above it is clear that all three systems are Turing-complete. Nevertheless, the tree systems differ in reduction power, where CL is weaker than $\lambda\beta$, and $\lambda\beta$ is weaker than $\lambda\beta\eta$. For instance, Church's exponentiation works only in $\lambda\beta\eta$, and the canonical translation of Church numerals does not work in CL.

The idea underlying the Church numerals extends to constructor-based data structures in general. Here are the encodings fur numbers, pairs, and lists.

$$\begin{aligned} \mathit{zero} &:= \lambda x f.x \\ \mathit{succ}\ a &:= \lambda x f.f(a x f) \\ \mathit{pair}\ a\, b &:= \lambda f.f a b \\ \mathit{nil} &:= \lambda x f.x \\ \mathit{cons}\ a\, b &:= \lambda x f.f a(b x f) \end{aligned}$$

We have switched the argument order for numbers to reveal the similarity with Scott's encoding. Church's encoding refines Scott's encoding in that "recursive" constructor arguments are committed to the continuations given for the match. This way Church's encoding builds in a particular form of primitive recursion.

2 Inductive Predicates

Inductive definitions of predicates are often informally presented by giving inference rules for atomic propositions obtained with the predicate. In Coq's type theory inductive definitions are a basic feature. In this chapter we present the **intersection model** for inductive predicates, which obtains inductive predicates without using inductive definitions. By studying the intersection model we obtain a deeper understanding of inductive predicates and the accompanying induction principles.

The intersection model for inductive predicates refines the intersection model for the inductive definitions of sets. The standard reference for inductive definitions in set theory is a 1977 handbook chapter by Peter Aczel [?].

There is an accompanying Coq development formalizing the definitions and proofs of this chapter.

2.1 Predicates and Relations

Predicates in type theory correspond to relations in set theory. A predicate is a function

$$X_1 \to \cdots \to X_n \to Prop$$

while a relation is a subset of

$$X_1 \times \cdots \times X_n$$

A main difference between predicates and relations is that relations are extensional while predicates are not. That is, we may have different predicates p and q that are **equivalent**:

$$p \approx q \;\; := \;\; \forall x_1 \dots x_n.\;\; p x_1 \dots x_n \leftrightarrow q x_1 \dots x_n$$

In Coq it is consistent to assume that equivalent predicates are equal, but we will not make this assumption. We define **subsumption** of predicates as follows.

$$p \preccurlyeq q \;\; := \;\; \forall x_1 \dots x_n.\;\; p x_1 \dots x_n \to q x_1 \dots x_n$$

For relations R and S, subsumption is simply set inclusion $R \subseteq S$. Note that equivalence of predicates is an equivalence relation and that subsumption of predicates is a preorder (i.e., a reflexive and transitive predicate).

2.2 Even Numbers

We present the intersection model at the example of an inductive predicate holding for the even natural numbers. We present the definition of the predicate using inference rules.

$$\frac{}{\mathit{even}\ 0} \qquad\qquad \frac{\mathit{even}\ n}{\mathit{even}\ (S(S\,n))}$$

Formally, the definition give us a predicate

$$\mathit{even} : \ \mathit{nat} \to \mathit{Prop}$$

and three **base lemmas**

$$\begin{aligned} B1 &: \ \mathit{even}\ 0 \\ B2 &: \ \forall n.\ \mathit{even}\ n \to \mathit{even}\ (S(S\,n)) \\ BI &: \ \forall p.\ p0 \to (\forall n.\ \mathit{even}\ n \to pn \to p(S(S\,n))) \to \mathit{even} \preceq p \end{aligned}$$

We say that the lemmas *B1* and *B2* are the **introduction principles** and the lemma *BI* is the **induction principle** coming with the definition. We call the lemmas *B1*, *B2*, and *BI* the **base lemmas** for the predicate *even*. As it comes to formal proofs, we make the crucial assumption that all we know about *even* are the base lemmas.

The introduction lemmas *B1* and *B2* allow us to derive propositions *even* n according to the inference rules. The induction lemma *BI* allows us to show that *even* subsumes a given predicate p. To do so, we have to show for each inference rule that p propagates from the recursive premises of the rule to the conclusion of the rule. The second rule for even has a single premise which is recursive. The **propagation condition** for this rule is

$$\forall n.\ \mathit{even}\ n \to pn \to p(S(S\,n))$$

where the premise pn is the so-called **inductive hypothesis**. We get an inductive hypothesis for every recursive premise of a rule.

2.2.1 Inductive Definitions as Specifications

We can see the base lemmas as a **specification**

$$spec : (nat \to Prop) \to Prop$$

of the predicate *even*. This leaves open how the predicate *even* is obtained. We can ask whether the specification has a unique solution (up to equivalence) and whether we can construct a solution of the specification not using a native inductive definition. The answer to both questions is yes.

We formulate the specification for **even** with three **defining predicates** that abstract over the base lemmas.

$$\begin{aligned} D1\; q &:= q0 \\ D2\; q &:= \forall n,\; qn \to q(S(S\,n)) \\ DI\; q &:= \forall p.\, p0 \to (\forall n.\, qn \to pn \to p(S(S\,n))) \to q \preceq p \\ spec\; q &:= D1\; q \land D2\; q \land DI\; q \end{aligned}$$

Showing that the specification has at most one solution (up to equivalence) is straightforward. Moreover, we can show that the **intersection**

$$I \;:=\; \lambda n.\, \forall p.\, D1\; p \to D2\; p \to pn$$

of all predicates satisfying the introduction principles is a solution of the specification. Showing that I satisfies $D1$ and $D2$ is straightforward. To show that I satisfies the induction principle *DI* one first shows that I satisfies the property

$$DL\; q \;:=\; \forall p.\, D1\; p \to D2\; p \to q \preceq p$$

$DL\; I$ is almost the induction principle $DI\; I$ we want to show. In fact, we obtain a proof of $DI\; I$ by applying the proof of $DL\; I$ to the predicate $\lambda n.\, In \land pn$.

That a predicate q satisfies *D1*, *D2*, and *DL* means that q is the least predicate satisfying *D1* and *D2*.

The predicates *DI* and *DL* both formulate induction principles. In fact, we can specify *even* with either *DI* or *DL* since we have

$$DI\; p \to DL\; p$$
$$D1\; p \to D2\; p \to DL\; p \to DI\; p$$

We call *ML* the **pure induction principle** and *MI* the **augmented induction principle**.

We speak of the *intersection model* for inductive predicates since we construct the inductive predicate as the intersection of all predicates satisfying the introduction principles. The intersection model is commonly used in mathematics to obtain the least set satisfying given closure properties.

It is important to understand that the predicates *D1*, *D2*, *DI*, and *DL* can be obtained algorithmically from the defining inference rules.

If we define *even* with an inductive definition in Coq, the value constructors of the definition provide the base lemmas *B1* and *B2*. The remaining base lemma *BI* (the augmented induction principle) will be provided under the name *even_ind*. Coq's induction tactic is a convenient way to apply *BI*. Coq establishes *even_ind* with a proof term using the *fix* and *match* coming with the inductive definition of *even*. If *even* is defined inductively, we can apply the tactics *destruct* and *inversion* to assumptions for propositions obtained with *even*. The uses of *destruct* and *inversion* can always be simulated with the induction principle *BI*.

What we have shown at the example of *even* will carry over to all inductive definitions accepted by Coq.

Exercise 16 Study the intersection model for inductive predicates at the example of the evenness predicate using Coq.

a) Define predicates *D1*, *D2*, and *DI* such that

$$\mathit{spec}\ q \ :=\ \mathit{D1}\ q \land \mathit{D2}\ q \land \mathit{DI}\ q$$

is a specification for evenness predicates.

b) Show that the specification *spec* has at most one solution up to equivalence.

c) Define *DL* and prove

$$\mathit{DI}\ p \to \mathit{DL}\ p$$
$$\mathit{D1}\ p \to \mathit{D2}\ p \to \mathit{DL}\ p \to \mathit{DI}\ p$$

d) Define the predicate

$$\mathit{even}\ n \ :=\ \forall p.\ \mathit{D1}\ p \to \mathit{D2}\ p \to p n$$

and show that it satisfies the specification *spec*.

e) Prove the following facts.

i) $\mathit{even}\ 4$

ii) $\neg \mathit{even}\ 1$

iii) $\mathit{even}\ (S(S\ n)) \to \mathit{even}\ n$

iv) $\mathit{even}\ n \to \neg \mathit{even}\ (Sn)$

Hint: Use the tactic *refine* to apply the induction principle. Note that (ii) and (iv) can be shown with the induction principle L, while (iii) requires the induction principle *BI*.

f) Define an evenness predicate using an inductive definition and prove that it satisfies the specification *spec*.

2.3 Pencil Notation for Inductive Proofs

We prove three claims about the predicate *even* to demonstrate a pencil notation for inductive proofs. Make sure you understand that the proofs only use the induction principle *BI* for *even*.

Proof (1)

	$\neg even\ 1$
	$\forall n.\ even\ n \to n \neq 1$
Induction on *even*	
1. Claim trivial	$0 \neq 1$
2. $A:\ even\ n$	$S(S\,n) \neq 1$
$IH:\ n \neq 1$	
Claim trivial	■

The proof first equivalence transforms the claim so that the induction principle can be used to do case analysis on the derivation of *even 1*. The inductive hypothesis is not used. In Coq the tactic *remember* can be used for the equivalence transformation of the initial claim.

Proof (2)

	$\forall n.\ even\ (S(S\,n)) \to even\ n$
	$\forall k.\ even\ k \to \forall n.\ k = S(S\,n) \to even\ n$
Induction on *even*	
1. Claim trivial	$\forall n.\ 0 = S(S\,n) \to even\ n$
2. $A:\ even\ k$	$S(S\,k) = S(S\,n) \to even\ n$
$IH:\ \forall n.\ k = S(S\,n) \to even\ n$	
Claim follows	■

As before, the proof uses the induction principle to do a case analysis. The inductive hypothesis is not used. We have given the inductive hypotheses in the proofs above so that you know what it is. In the next pencil proof the use of the inductive hypothesis is essential.

Proof $\qquad \forall n.\ \mathit{even}\ n \to \neg \mathit{even}\ (S\ n)$

Induction on *even*

1. Claim follows with (1) $\qquad \neg \mathit{even}\ 1$
2. A : $\mathit{even}\ n$ $\qquad \neg \mathit{even}\ (S(S(S\ n)))$

   IH : $\neg \mathit{even}\ (S\ n)$

   Claim follows with (2) and *IH* ∎

Exercise 17 Simulate the above pencil proofs with Coq.

2.4 Exercises

Exercise 18 (Reachability) Let a type X, a predicate $R : X \to X \to \mathit{Prop}$ and a point $a : X$ be given. We define a predicate "R can reach a from x" inductively:

$$\frac{}{\mathit{reach}\ R\ a\ a} \qquad\qquad \frac{Rxy \qquad \mathit{reach}\ R\ a\ y}{\mathit{reach}\ R\ a\ x}$$

a) Define the predicate *reach R a* with the intersection method in Coq and show that it satisfies the base lemmas coming with the inference rules.

b) Define R^* with a native inductive definition in Coq and prove $R^* \approx \mathit{reach}\ R$.

Exercise 19 (Termination) Let a type X and a predicate $R : X \to X \to \mathit{Prop}$ be given. We define a predicate "R terminates on x" inductively:

$$\frac{Rx \preceq \mathit{ter}\ R}{\mathit{ter}\ R\ x}$$

a) Define the predicate *ter R* with the intersection method in Coq and show that it satisfies the base lemmas coming with the inference rule.

b) Define $\mathit{SN}\ R$ with a native inductive definition in Coq and prove $\mathit{SN}\ R \approx \mathit{ter}\ R$.

Exercise 20 (Knaster-Tarski) Let X be a type and $F : (X \to \mathit{Prop}) \to (X \to \mathit{Prop})$ be a monotone predicate (i.e., $\forall p\ q.\ p \preceq q \to Fp \preceq Fq$). Find a predicate I such that you can prove the following. The intersection $p \sqcap q$ abbreviates the predicate $\lambda x.\ px \wedge qx$.

a) $Fp \preceq p \to I \preceq p$

b) $FI \preceq I$

c) $FI \approx I$

d) $p \preceq I \to Fp \preceq I$

e) $F(I \sqcap p) \preceq I$

f) $F(I \sqcap p) \preceq p \leftrightarrow I \preceq p$

Hint: The problem is a translation of a special case of the Knaster-Tarski fixed point theorem from set theory to type theory. Google to find out more about the theorem and its proof. The proof of the Knaster-Tarski theorem is a classical example for the use of the intersection method in Mathematics.

3 Abstract Reduction Systems

We are aiming at proving the Church-Rosser property of combinatory logic and lambda calculus. The proofs factorize into general results about abstract reduction systems and certain verifications for the concrete systems. An abstract reduction system is a reduction system where the notion of term is kept abstract. The architecture of the confluence proofs can be fully developed at the level of abstract reduction systems, and this will be the topic of this chapter.

We will develop our definitions both in set theory and type theory. The type-theoretic definitions are refinements of the set-theoretic definitions. There is an accompanying Coq development formalizing all definitions and proofs.

3.1 Basic Definitions

Set-theoretically, an **abstract reduction system (ARS)** consists of a set X and a relation $R \subseteq X \times X$. We can see an ARS as a possibly infinite directed graph. Following this view, we call the elements of X **nodes** and the elements of R **edges**. If Rxy, we say that X is a **predecessor** of y, and that y is a **successor** of x.

Type-theoretically, an abstract reduction system consists of a type X and a predicate $R : X \to X \to \mathit{Prop}$.

3.2 Reflexive Transitive Closure

Given an ARS, we want to define the reflexive transitive closure R^* of R. Note that this amounts to an operator mapping a relation R to a relation R^*. There are many different definitions of R^*. While all definitions yield the same relation or equivalent predicates, the particular definition chosen matters since it contributes proof principles. In particular, we want a simple yet powerful induction principle for R^*. We choose the following inductive definition, called **linear definition** in the following.

$$\frac{}{R^*xx} \qquad\qquad \frac{Rxy \qquad R^*yz}{R^*xz}$$

The canonical induction principle for this definition has one case for each rule.

$$\begin{aligned}
&py \to \\
&(\forall xy.\, Rxy \to\ R^*yz \to py \to px) \to \\
&\forall x.\, R^*xy \to px
\end{aligned}$$

The first rule yields a base case. The second rule yields an inductive step with a single inductive hypothesis.

We formalize the definition of R^* as follows in Coq.

```
Variable X : Type.
Implicit Types R : X → X → Prop.
Inductive star R : X → X → Prop :=
| starR x : star R x x
| starC x y z : R x y → star R y z → star R x z.
```

Given this definition, Coq generates an induction principle quantifying over both arguments of R^* (check *star_ind*):

$$\begin{aligned}
&(\forall x.\, pxx) \to \\
&(\forall xyz.\, Rxy \to\ R^*yz \to pyz \to pxz) \to \\
&\forall xy.\, R^*xy \to pxy
\end{aligned}$$

This induction principle is unnecessarily complex. The reason is that the argument y of R^*xy is a uniform parameter mathematically but is not declared as such by our definition. In fact, in Coq we can make y a uniform parameter only if we switch the argument order of R^* (because parameters have to come first). Given this situation, we decide to work with the induction principle Coq generates. It subsumes the canonical induction principle shown above but produces unnecessarily complex inductive hypotheses.

In the above inductive definition of *star* it is possible to make the argument x of *star* $R\ x\ y$ a non-uniform parameter. This leads to more compact matches and destructs but does not change the induction principle generated.

Exercise 21 Establish the canonical induction principle for R^* in Coq.

```
Lemma star_canonical_ind R (p : X → Prop) y :
  p y →
  (∀ x x', R x x' → star R x' y → p x' → p x) →
  ∀ x, star R x y → p x.
```

Exercise 22 Show in Coq that the two induction principles for R^* are equivalent. To do so, assume a type X and two relations R and S in a section. Then formulate the two induction principles (S taking the role of R^*) and prove them equivalent. Interestingly, the equivalence does not depend on the rules (i.e., constructors) defining R^*.

Exercise 23 Given X and R, show that the constructors and the induction principle uniquely determine R^*. To do so, assume X and R and two relations S and S' with corresponding constructors and induction principles and show that S and S' are equivalent.

Exercise 24 Establish the following right-to-left induction principle for R^*.

```
Lemma star_right_ind R (p : X → Prop) x :
  p x →
  (∀ y y', star R x y' → p y' → R y' y → p y) →
  ∀ y, star R x y → p y.
```

3.2.1 Proof of Transitivity

From the definition of R^* it is clear that R^* is reflexive (1st rule) and subsumes R (2nd rule and reflexivity). What is not obvious is that R^* is transitive. In fact, showing the transitivity of R^* requires an inductive proof.

Claim Let R^* be obtained with the linear definition. Then R^* is transitive.

Proof Let R^*xy and R^*yz. We show R^*xz by induction on the derivation of R^*xy. If R^*xy is obtained with the first rule of the definition, we have $x = y$ and the claim follows. If R^*xy is obtained with the second rule, we have Rxx' and $R^*x'y$. The inductive hypothesis applies to $R^*x'z$ and yields $R^*x'z$. (The inductive hypothesis applies since $R^*x'z$ has a shorter derivation than R^*xy.) The claim follows with the second rule of the definition. ■

Here is a more explicit and less verbose presentation of the proof that is easier to verify.

Proof

$A:\ R^*xy$

$B:\ R^*yz$ Show: R^*xz

Induction A

1. $x = y$

   Claim trivial

2. $Rxx' \wedge R^*x'y$

   $IH:\ R^*x'z$

   Claim follows with 2nd rule

■

Exercise 25 Simulate the above transitivity proof with a Coq script. First use the induction tactic and notice that you get a more complex inductive hypothesis. The reason is that the induction tactic is using the automatically generated induction principle *star_ind*, which quantifies over both variables. Then redo the proof using the canonical induction principle (use the tactic *refine* to apply it). This should give you exactly the proof shown above.

Exercise 26 Prove the following properties of R^* in Coq.

a) Monotonicity: $R \preccurlyeq S \to R^* \preccurlyeq S^*$

b) Minimality: If $R \preccurlyeq S$ and S is reflexive and transitive, then $R^* \preccurlyeq S$.

c) Idempotence: $(R^*)^* \approx R^*$

d) Interpolation: $R \preccurlyeq S \preccurlyeq R^* \to R^* \approx S^*$

3.2.2 Binary Definition

Here is a second definition of the reflexive transitive closure called **binary definition** in the following.

$$\frac{}{Rxx} \qquad\qquad \frac{R^\# xy \qquad R^\# yz}{R^\# xz} \qquad\qquad \frac{Rxy}{R^\# xy}$$

This time it is obvious that $R^\#$ is reflexive (1st rule), transitive (2nd rule), and subsumes R (3rd rule). In fact, the inductive definition of $R^\#$ faithfully realizes what the term *reflexive transitive closure* says. This comes at the cost of a complex induction principle with 3 cases, where the second case comes with two inductive hypotheses.

To show that $R^* \approx R^\#$, we have to verify that the rules of one system can simulate the rules of the other system. It is easy to see that the binary system can simulate the two rules of the linear system. For instance, we have to verify

$$Rxy \to R^\# yz \to R^\# xz$$

to show that the binary system can simulate the 2nd rule of the liner system. We also have to verify that the linear system can simulate each rule of the binary system. The first and the second rule are straightforward. That we can simulate the third rule follows by the transitivity lemma for the linear system.

To show $R^* \approx R^\#$ in Coq, we need two inductions. An induction on R^* is needed for $R^* \preccurlyeq R^\#$, and an induction on $R^\#$ is needed for $R^\# \preccurlyeq R^*$.

Exercise 27 Carry out the definition of $R^\#$ in Coq and show the equivalence with R^*.

3.2.3 Power Definition

Here is another set-theoretic definition of R^* that is commonly used in the literature.

$$\begin{aligned} R^* &:= \bigcup_{n\in\mathbb{N}} R^n \\ R^0 &:= \{\,(x,x) \mid x \in X\,\} \\ R^{n+1} &:= R \circ R^n \\ R \circ S &:= \{\,(x,z) \mid \exists y.\, Rxy \wedge Syz\,\} \end{aligned}$$

This definition relies on natural recursion and does not require an understanding of general inductive definitions. The definition inherits the linear induction principle of the natural numbers.

Exercise 28 Carry out the power definition in Coq and prove the equivalence with the linear definition.

3.3 Normal Forms

$$\begin{aligned} \mathit{reducible}\ R\ x &:= \exists y.\, Rxy \\ \mathit{normal}\ R\ x &:= \neg \mathit{reducible}\ R\ x \\ x \Downarrow^R y &:= R^*xy \wedge \mathit{normal}\ R\ y \end{aligned}$$

Exercise 29 Prove $R^*xy \to \mathit{normal}\ R\ x \to x = y$.

3.4 Equivalence Closure

We define the **equivalence closure** $R^{\equiv}$ using the transitive closure.

$$\begin{aligned} R^{\equiv} &:= (R^{\leftrightarrow})^* && \textit{equivalence closure} \\ R^{\leftrightarrow} &:= R \cup R^{-1} && \textit{symmetric closure} \\ R^{-1} &:= \{\,(y,x) \mid (x,y) \in R\,\} && \textit{inverse} \end{aligned}$$

This gives us a linear recursion principle for $R^{\equiv}$. In Coq, we refine the above definition to the inductive definition

```
Inductive ecl R : X → X → Prop :=
| eclR x : ecl R x x
| eclC x y z : R x y → ecl R y z → ecl R x z
| eclS x y z : R y x → ecl R y z → ecl R x z.
```

This yields a convenient linear induction principle building in the case analysis coming with the symmetric closure $R \cup R^{-1}$.

We write $x \equiv_R y$ for $R^{\equiv}xy$.

Exercise 30 Prove the following properties of $R^{\equiv}$ in Coq.

a) Transitivity and symmetry.

b) Monotonicity: $R \preccurlyeq S \to R^{\equiv} \preccurlyeq S^{\equiv}$

c) Minimality: If $R \preccurlyeq S$ and S is an equivalence, then $R^{\equiv} \preccurlyeq S$.

d) Idempotence: $(R^{\equiv})^{\equiv} \approx R^{\equiv}$

e) Interpolation: $R \preccurlyeq S \preccurlyeq R^{\equiv} \to R^{\equiv} \approx S^{\equiv}$

3.5 Church-Rosser-Like Properties

We will use the following definitions for the analysis of the Church-Rosser property.

$$\begin{aligned}
x \downarrow^R y &:= \exists z.\, Rxz \land Ryz && \textit{joinable}\\
R^{\downarrow} &:= \{\, (x,y) \mid x \downarrow^R y \,\}\\
\textit{functional}\ R &:= \forall xyz.\, Rxy \to Rxz \to y = z\\
\textit{diamond}\ R &:= \forall xyz.\, Rxy \to Rxz \to y \downarrow^R z\\
\textit{confluent}\ R &:= \textit{diamond}\ (R^*)\\
\textit{semi-confluent}\ R &:= \forall xyz.\, Rxy \to R^*xz \to y \downarrow^{R^*} z\\
\textit{Church-Rosser}\ R &:= R^{\equiv} \subseteq R^{\downarrow}
\end{aligned}$$

We obtain the following results.

$$\begin{aligned}
\textit{functional}\ R &\to \textit{semi-confluent}\ R\\
\textit{diamond}\ R &\to \textit{semi-confluent}\ R\\
\textit{confluent}\ R &\leftrightarrow \textit{semi-confluent}\ R\\
\textit{Church-Rosser}\ R &\leftrightarrow \textit{semi-confluent}\ R\\
\textit{confluent}\ R &\to \textit{functional}\ (\Downarrow^R)
\end{aligned}$$

All results have illuminating diagram-based proof sketches. The proofs often use induction on R^*. Translating the diagram-based proof sketches into Coq scripts is routine. Often it is necessary to strengthen the inductive claim by reverting assumptions (implicit in the diagram-based sketches). Here is a textual proof of the second result. Setting up the inductive claim as shown is essential.

Claim *diamond* $R \to$ *semi-confluent* R.

Proof

$A:\ \mathit{diamond}\ R$

$B:\ R^*xz$ Show: $\forall y.\, Rxy \to y \downarrow^{R^*} z$

Induction B

1. $x = z$

   Claim follows

2. $Rxx' \wedge R^*x'z$

   $\mathit{IH}:\ \forall y.\, Rx'y \to y \downarrow^{R^*} z$

   Rxy Show: $y \downarrow^{R^*} z$

   $Rx'v \wedge Ryv$ by A

   $R^*vu \wedge R^*zu$ by IH

   R^*yu

   Claim follows ∎

The properties we have defined for reduction predicates are all **extensional**. Extensionality is defined as follows.

$$\mathit{extensional}\ p \ := \ \forall RS.\, R \approx S \to pR \to pS$$

Exercise 31 Simulate the textual proof shown above with a Coq script using the canonical induction principle for R^*.

Exercise 32 Prove that every semi-confluent relation is Church-Rosser. Start with a diagram-based proof sketch, then give the textual proof, and finally write proof scripts in Coq. Do the proof in Coq first with the canonical induction principle so that is simulates your textual proof. Then do the proof with the induction tactic and use automation tactics when convenient.

Exercise 33 Let R be Church-Rosser. Prove the following.

a) $x \equiv_R y \to \mathit{normal}\ R\ y \to x \Downarrow^R y$

b) $x \equiv_R y \to \mathit{normal}\ R\ x \to \mathit{normal}\ R\ y \to x = y$

Exercise 34 Prove that the given definitions of confluence and Church-Rosser are extensional in Coq.

Exercise 35 Explain why $\mathit{sym}\ R := (R = \lambda xy.\, Ryx)$ cannot be used as a definition of symmetry in Coq.

3.6 Reduction Operators

Reduction operators are a tool for proving confluence. A **reduction operator** is a function $\rho : X \to X$. The **triangle property** for reduction operators is defined as follows.

$$\mathit{triangle}\ R\ \rho \ :=\ \forall xy.\, Rxy \to Ry(\rho x)$$

The proof of the following facts is straightforward.

$$\begin{aligned} &\mathit{triangle}\ R\ \rho \ \to\ \mathit{diamond}\ R \\ &R^* = S^* \ \to\ \mathit{triangle}\ S\ \rho \ \to\ \mathit{confluent}\ R \end{aligned}$$

The second fact gives us a method for proving the Church-Rosser property of lambda calculus and combinatory logic. For the given reduction relation R (β-reduction or weak reduction) we construct a reduction relation S (called parallel reduction) and a reduction operator ρ such that $R^* = S^*$ and *triangle* $R\ \rho$. The existence of S and ρ implies the Church-Rosser property of R.

Reduction operators can also be used to compute normal forms with the following algorithm: Given x, iterate ρ on x until a normal term $\rho^n x$ is reached. This simple algorithm is correct if ρ satisfies the following properties.

$$\begin{aligned} \mathit{sound}\ R\ \rho \ &:=\ \forall x.\, R^* x(\rho x) \\ \mathit{cofinal}\ R\ \rho \ &:=\ \forall xy.\, R^* xy \to \exists n.\, R^* y(\rho^n x) \end{aligned}$$

It is not difficult to prove the following facts.

$$\begin{aligned} \mathit{cofinal}\ R\ \rho \ &\to\ \mathit{confluent}\ R \\ \mathit{sound}\ R\ \rho \ &\to\ \mathit{normal}\ R\ x \ \to\ \rho x = x \\ \mathit{sound}\ R\ \rho \ &\to\ \mathit{cofinal}\ R\ \rho \ \to\ (x \Downarrow^R y \leftrightarrow \exists n.\, y = \rho^n x \wedge \mathit{normal}\ R\ y) \\ \mathit{triangle}\ R\ \rho \ &\to\ \mathit{cofinal}\ R\ \rho \\ \mathit{triangle}\ R\ \rho \ &\to\ \mathit{reflexive}\ R \to \mathit{sound}\ R\ \rho \end{aligned}$$

Barendregt [1] calls reduction operators strategies. He shows that left-most outermost reduction yields a sound and cofinal reduction operator for lambda calculus. Barendregt [1] does not say that reduction operators can be used to prove confluence. Takahashi [4] uses reduction operators and the triangle property to prove confluence.

Exercise 36 Let ρ be a reduction operator satisfying the triangle property for R. Prove the following.

a) *reflexive* $R \to$ *sound* $R\ \rho$

b) $Rxy \to R(\rho x)(\rho y)$

c) $Rxy \to R(\rho^n x)(\rho^n y)$

d) *cofinal* $R\ \rho$

Exercise 37 Let ρ be a reduction operator that is sound for R. Prove the following.

a) *normal* $R\ x \ \to\ \rho x = x$

b) *cofinal* $R\ \rho \ \to\ (x \Downarrow^R y \leftrightarrow \exists n.\ y = \rho^n x \wedge$ *normal* $R\ y)$

3.7 Abstract Analysis of Reflexive Transitive Closure

We have already seen several constructions of R^*. All of them used inductive definitions, either directly or through the natural numbers. We will now see how R^* can be specified, constructed, and analyzed without using inductive definitions. We will do the following:

- We give a specification of R^*.
- We show that the specification has at most one solution.
- We construct a solution of the specification not using inductive types.

The techniques we introduce here apply to inductive predicates in general. They are related to the impredicative definitions of the logical connectives.

We say that a relation S is closed under left-composition with a relation R if

$$\mathit{lcomp}\ R\ S \ :=\ \forall xyz.\, Rxy \to Syz \to Sxz$$

We define R^* as the intersection of all relations that are reflexive and closed under left-composition with R.

$$\mathit{star}\ R\ x\ y \ :=\ \forall p,\ \mathit{reflexive}\ p \to \mathit{lcomp}\ R\ p \to pxy$$

Note that this defines a function mapping relations to relations. We refer to this definition of R^* as **impredicative definition**. It is straightforward to show that R^* the least relation that is reflexive and closed under left-composition with R. This provides us with a specification for R^*.

$$\begin{aligned}\mathit{least}\ R\ S \ &:=\ \forall p,\ \mathit{reflexive}\ p \to \mathit{lcomp}\ R\ p \to S \preceq p\\ \mathit{spec}\ R\ S \ &:=\ \mathit{reflexive}\ S \wedge \mathit{lcomp}\ R\ S \wedge \mathit{least}\ R\ S\end{aligned}$$

We already know

$$\mathit{spec}\ R\ (\mathit{star}\ R)$$

It is also easy to show that the specification has at most one solution up to equivalence:

$$\mathit{spec}\ R\ S \to \mathit{spec}\ R\ S' \to S \approx S'$$

If we want to know whether some other definition of R^* is equivalent to the impredicative definition, we just show that the other definition satisfies the specification. This is, for instance, straightforward to do for the linear inductive definition we discussed earlier in this chapter.

We can now ask whether the induction principles we gave before for R^* can be obtained directly from the specification of R^*. This is in fact the case. We define

$$\mathit{ind}\ R\ S \ := \ \forall p.\ \mathit{reflexive}\ p \to (\forall xyz.\ Rxy \to Syz \to pyz \to pxz) \to S \preceq p$$

and show

$$\mathit{spec}\ R\ S \to \mathit{ind}\ R\ S$$

The trick is to use *least* $R\ S$ as an induction principle with the predicate

$$qxy := Sxy \wedge pxy$$

We can take the view that *spec* R specifies R^* as the least relation satisfying two **closure properties**. The closure properties correspond to the rules of the inductive definition, and the requirement that R^* is the least relation satisfying the closure properties yields a primitive but sufficiently powerful induction principle.

From the perspective of proof rules we can see the closure properties as introduction rules and the induction principle as an elimination rule.

3.8 Strong Normalization

A node x of an ARS is **strongly normalizing** if every successor of x is strongly normalizing. This is an interesting inductive definition with a single defining rule. In formal notation the defining rule looks as follows.

$$\frac{\forall y.\ Rxy \to \mathit{SN}\ R\ y}{\mathit{SN}\ R\ x}$$

We have the following fact.

$$\mathit{SN}\ R\ x \ \leftrightarrow\ \forall y.\ Rxy \to \mathit{SN}\ R\ y$$

Make sure you understand the proof of this fact. From left to right one analyses the derivation of the assumption $SN\ R\ x$ (tactic *destruct* in Coq). From right to left one constructs a derivation of the claim $SN\ R\ x$ (tactic *constructor* in Coq). It is easy to show that every normal node is strongly normalizing (since it has no successors).

$$\mathit{normal}\ R\ x \to SN\ R\ x$$

It follows that every node in a finite directed acyclic graph is strongly normalizing. We would expect that a node x with Rxx is not strongly normalizing. Proving this fact requires induction.

Claim $Rxx \to SN\ R\ x \to \bot$

Proof

	Show
$A:\ SN\ R\ x$	$Rxx \to \bot$
Induction A	
$IH:\ \forall y.\, Rxy \to Ryy \to \bot$	
Claim follows by IH	

■

Coq infers the canonical induction principle for SN:

$$\begin{aligned}&(\forall x.\,(\forall y.\,Rxy \to SN\ R\ y) \to (\forall y.\,Rxy \to py) \to px) \to\\ &\forall x.\,SN\ R\ x\ \to\ px\end{aligned}$$

Since there is a single defining rule there is a single inductive step. The first premise of the inductive step is the premise of the defining rule, and the second premise of the inductive step in the inductive hypothesis. Here is the proof term for the induction principle for SN.

```
fun R p step ⇒ fix f x A := match A with SNI g ⇒ step x g (fun y B ⇒ f y (g y B)) end
```

Note the higher-order structural recursion: The function g obtained from A for all arguments yields a derivation structurally smaller than A. This property is an essential feature of Coq's type theory.

$$\mathit{locally\_confluent}\ R\ :=\ \forall xyz.\,Rxy \to Rxz \to\ y \downarrow^{R^*} z$$

Lemma 2 (Newman) A strongly normalizing relation is confluent if it is locally confluent.

Exercise 38 Find a locally confluent relation that is not confluent.

4 Lambda Calculus, Formalization

A major difference between combinatory logic and lambda calculus is that the lambda calculus comes with a variable binder introducing local variables. In fact, the lambda calculus is the prototypical syntactic system with a variable binder. The presence of a variable binder considerable complicates substitution, which due to beta reduction is a basic ingredient of the lambda calculus. We will study the lambda calculus with de Bruijn's term representation where numeric argument references replace the local variables of the standard notation.

4.1 Terms and Substitutions in de Bruijn Representation

Following de Bruijn, we represent the **terms** of the untyped lambda calculus with numeric variable references:

$$s,t \;::=\; n \;\mid\; st \;\mid\; \lambda s \qquad (n \in \mathbb{N})$$

There are no local names. Argument references and free variables are represented with numbers. Here are 2 examples:

$$\begin{array}{ll} \lambda fxy.\, fxy & \lambda\lambda\lambda\hat{2}\hat{1}\hat{0} \\ \lambda f.\, f(\lambda x.\, fx(\lambda y.\, fxy)) & \lambda\hat{0}(\lambda\hat{1}\hat{0}(\lambda\hat{2}\hat{1}\hat{0})) \end{array}$$

A **substitution** is a function $\mathbb{N} \to \mathit{term}$. A **renaming** is a function $\mathbb{N} \to \mathbb{N}$. We consider renamings to be substitutions. We use σ and τ to denote substitutions and ξ and ζ to denote renamings. The application of substitutions to terms is defined with two mutually recursive operators.

$$\begin{aligned} ! &: (\mathbb{N} \to \mathit{term}) \to \mathit{term} \to \mathit{term} \\ \uparrow &: (\mathbb{N} \to \mathit{term}) \to \mathbb{N} \to \mathit{term} \\ \sigma\,!\,n &:= \sigma n \\ \sigma\,!\,st &:= (\sigma\,!\,s)(\sigma\,!\,t) \\ \sigma\,!\,\lambda s &:= \lambda(\uparrow\!\sigma\,!\,s) \\ \uparrow\!\sigma\,0 &:= 0 \\ \uparrow\!\sigma\,(Sn) &:= S\,!\,\sigma n \end{aligned}$$

The **application operator** ! recurses through terms where the substitution to be applied is transformed with the **up-operator** $\uparrow$ when it is pushed below a lambda. The mutual recursion terminates since it goes through the following stages:

$$\sigma\,!\,s \quad\rightsquigarrow\quad \xi\,!\,s \quad\rightsquigarrow\quad \xi\,!\,n$$

4.2 Basic Substitution Laws

Application of the **identity substitution** $id = \lambda n.n$ leaves a term unchanged:

$$id\,!\,s = s$$

For many proofs it is crucial to have a **composition operation** for substitutions such that the **composition law** holds:

$$\sigma\,!\,\tau\,!\,s = \sigma\cdot\tau\,!\,s$$

The definition of the composition operator is straightforward:

$$(\sigma\cdot\tau)n \;:=\; \sigma\,!\,\tau n$$

The proof of the composition law is interesting. By induction on s it reduces to the **distribution law**

$$\uparrow(\sigma\cdot\tau) = \uparrow\sigma\cdot\uparrow\tau$$

The distribution law follows from two instances of the composition law:

$$\sigma\,!\,\xi\,!\,s = \sigma\cdot\xi\,!\,s$$
$$\xi\,!\,\tau\,!\,s = \xi\cdot\tau\,!\,s$$

The instances of the composition laws follow by induction on s and two instances of the distribution law:

$$\uparrow(\sigma\cdot\xi) = \uparrow\sigma\cdot\uparrow\xi$$
$$\uparrow(\xi\cdot\tau) = \uparrow\xi\cdot\uparrow\tau$$

The first instance is obvious. The second instance follows with the first instance of the composition law and the **S-law**, which holds by definitional equality.

$$S\cdot\tau = \uparrow\tau\cdot S$$

With composition substitutions yield an operator monoid satisfying the composition and the identity law.

$$\begin{aligned}\sigma \cdot (\tau \cdot \mu) &= (\sigma \cdot \tau) \cdot \mu \\ id \cdot \sigma &= \sigma = \sigma \cdot id \\ \sigma \,!\, \tau \,!\, s &= \sigma \cdot \tau \,!\, s \\ id \,!\, s &= s\end{aligned}$$

The up-operator is a monomorphism on this monoid.

$$\begin{aligned}\uparrow(\sigma \cdot \tau) &= \uparrow\sigma \cdot \uparrow\tau \\ \uparrow id &= id\end{aligned}$$

4.3 Beta Reduction

Now that we have substitutions we can define beta reduction:

$$\frac{}{(\lambda s)t \succ \beta t \,!\, s} \qquad \frac{s \succ s'}{\lambda s \succ \lambda s'} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

The operator $\beta : term \to \mathbb{N} \to term$ is defined as follows:

$$\begin{aligned}\beta t\, 0 &:= t \\ \beta t\, (Sn) &:= n\end{aligned}$$

By induction on the reduction relation $\succ$ one can show that beta reduction is **substitutive**:

$$s \succ t \to \sigma \,!\, s \succ \sigma \,!\, t$$

The interesting case of the induction follows with the composition law and the equation

$$\beta(\sigma \,!\, t) \cdot \uparrow\sigma = \sigma \cdot \beta t$$

which in turn follows from the equation $\beta t \cdot S = id$.

4.4 Closed Terms

We have not formalized the notion of free variables of terms. We now formalize the notion of closed term. To do so, we define an inductive predicate *dclosed* : $\mathbb{N} \to$ *term* $\to$ *Prop* such that *dclosed* $d\ s$ is derivable if no variable $n \geq d$ is free in s.

$$\frac{n < d}{\mathit{dclosed}\ d\ n} \qquad \frac{\mathit{dclosed}\ d\ s \qquad \mathit{dclosed}\ d\ t}{\mathit{dclosed}\ d\ (st)} \qquad \frac{\mathit{dclosed}\ (S\ d)\ s}{\mathit{dclosed}\ d\ (\lambda s)}$$

4.5 Formalization in Coq

The formalization in Coq has to resolve two complications:

1. Renamings are not substitutions (no subtyping).
2. Substitutions are not extensional (no functional extensionality).

The first complication is resolved by defining the operators $\uparrow$ and $!$ first for renamings and then for substitutions. This eliminates the mutual recursion between $\uparrow$ and $!$ since we now have separate operators for renamings and substitutions. A conversion operator is provided mapping renamings to their corresponding substitutions. One shows that the application operators for renamings and substitutions agree modulo conversion:

$$\xi\,!!\,s = \mathit{sub}\ \xi\,!\,s$$

The lack of functional extensionality can be resolved by working with **equivalence of substitutions** rather than equality.

$$\sigma \approx \tau := \forall n.\ \sigma n = \tau n$$

The application operators for renamings and substitutions satisfy **equivalence laws**

$$\xi \approx \zeta \to \xi\,!!\,s = \zeta\,!!\,s$$
$$\sigma \approx \tau \to \sigma\,!\,s = \tau\,!\,s$$

saying that equivalent substitutions yield identical results. The equivalence laws can be shown by induction on s.

Exercise 39 Show the equivalence laws for renamings and substitutions.

Exercise 40 Prove the composition law.

Exercise 41 Prove that beta reduction is substitutive.

Exercise 42 Prove $\sigma\,!\,s = s$ for every substitution and every closed term. You need two lemmas for the predicate *dclosed*.

Exercise 43 Prove that closedness of terms is a decidable property.

Exercise 44 Prove that a term s is closed if and only if $S\,!\,s = s$. Hint: Show $\uparrow^n S\,!\,s = s \to \mathit{dclosed}\ n\ s$ first.

5 Type Theory

We study a number of typed lambda calculi. We start with a basic calculus STLC (simply typed lambda calculus) and then consider several increasingly more expressive extensions known as T, F, F_ω, CC, and CC_ω. We also consider the extension of CC_ω with inductive types. All systems we consider are strongly normalizing and can be seen as subsystems of the calculus underlying Coq.

The systems we consider are known as type theories. Type theories combine a computational interpretation with a logical interpretation. This double interpretation is known as Curry-Howard correspondence. The logical interpretation is concerned with propositions and proofs. Propositions appear as types and proofs appear as the elements of types. F can express quantified propositional logic as well as primitive recursive functions on number.

In basic type theories like F or CC_ω, the logical and the computational interpretation of the theory are separated in that the theory cannot state and prove theorems about the computational objects it can define. This difficulty can be overcome by extending the theories with inductive types providing the computational objects.

5.1 Simply Typed Lambda Calculus (STLC)

The syntax of STLC consists of types and terms. There is a confluent reduction relation on terms and a typing relation that relates contexts, terms, and types. We consider the de Bruijn version of STLC where abstractions specify argument types.

Types

We assume a set of **type variables** X and define **types** as follows.

$$A, B \ ::= \ X \mid A \to B$$

Terms

We assume a set of **term variables** x and define **terms** as follows.

$$s,t \ ::= \ x \mid st \mid \lambda x{:}A.s$$

A formal representation of terms will represent abstractions without local names using de Bruijn indices. Note that abstractions carry an argument type. This is the de Bruijn version of STLC. There is also a Curry version of STLC where abstractions do not carry argument types. The terms of the Curry version of STLC are identical with the terms of the untyped lambda calculus.

Reduction

Reduction can contract a **β-redex** everywhere.

$$(\lambda x{:}A.s)t \ \succ \ s^x_t$$

Note that reduction ignores the type annotations in abstractions. Reduction in STLC is confluent.

Contexts

A **context**

$$\Gamma = \{x_1 : A_1, \dots, x_n : A_n\}$$

is a finite collection of variable declarations $x : A$ where no variable is declared more than once. The order of the declarations does not matter.

Typing Relation

The **typing relation** $\Gamma \vdash s : A$ is defined inductively.

$$\frac{(x:A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash st : B} \qquad \frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x{:}A.s : A \to B}\ x \notin \Gamma$$

Note that in the rule for abstractions x must not be declared in Γ. This is fine since the local variable x is a notational device that can be renamed. In the de Bruin representation Γ is a stack of types and the variable and abstraction rules look as follows:

$$\frac{}{\Gamma, A \vdash 0 : A} \qquad \frac{\Gamma \vdash n : A}{\Gamma, B \vdash Sn : A} \qquad \frac{\Gamma, A \vdash s : B}{\Gamma \vdash \lambda A.s : A \to B}$$

Properties of Typing

- **Subject reduction** If $s \succ s'$ and $\Gamma \vdash s : A$, then $\Gamma \vdash s' : A$.
- **Canonical form** If $\vdash s : A \to B$ and s is normal, then $s = \lambda x\!:\!A.t$ for some t.
- **Unique type** If $\Gamma \vdash s : A$ and $\Gamma \vdash s : B$, then $A = B$.
- **Strong normalization** If $\Gamma \vdash s : A$, then s is strongly normalizing.
- **Decidability** There is a decision algorithm for $\Gamma \vdash s : A$.

A logic system with simply typed lambda terms was first studied by Church [1940]. A computational system similar to STLC was first studied by Curry and Feys [1958]. They considered a system without type annotations not having the unique type property. One speaks of implicit typing or **Curry-style typing**. The version of STLC with type annotations presented here is known as **de Bruijn version**.

Exercise 45 Formalize *STLC* in Coq.

5.2 Gödel's T

T is a simply typed lambda calculus extended with numbers and primitive recursion. Numbers are accommodated with a base type N and two constructors O and S.

$$\begin{aligned} A, B &::= \mathsf{N} \mid A \to B \\ s, t &::= O \mid Ss \mid \mathsf{prec}\, s\, t\, (xy.u) \mid x \mid st \mid \lambda x\!:\!A.s \end{aligned}$$

Note that there are no type variables. The syntactic form for primitive recursion introduces two local variables x and y in the third constituent. The local variables x and y must be distint and are subject to alpha renaming. A formalization would realize the local variables with de Bruijn indices.

Reduction is defined with the reduction rule for beta redexes and two reduction rules for primitive recursion.

$$\begin{aligned} \mathsf{prec}\, O\, t\, (xy.u) &\succ t \\ \mathsf{prec}\, (S\, s)\, t\, (xy.u) &\succ u^{x\ y}_{s\ \mathsf{prec}\, s\, t\, (xy.u)} \end{aligned}$$

The reduction rules for primitive recursion become more readable if we abbreviate the syntactic form $(xy.u)$ with ρ and express substitution with application.

$$\begin{aligned} \mathsf{prec}\, O\, t\, \rho &\succ t \\ \mathsf{prec}\, (S\, s)\, t\, \rho &\succ \rho\, s\, (\mathsf{prec}\, s\, t\, \rho) \end{aligned}$$

The reduction relation of T is confluent.

The typing rules of T are the typing rules of STLC plus rules for the new syntactic forms.

$$\frac{}{\Gamma \vdash O : \mathsf{N}} \qquad\qquad \frac{\Gamma \vdash s : \mathsf{N}}{\Gamma \vdash S\,s : \mathsf{N}}$$

$$\frac{\Gamma \vdash s : \mathsf{N} \qquad \Gamma \vdash t : A \qquad \Gamma, x : \mathsf{N}, y : A \vdash u : A}{\mathsf{prec}\ s\,t\,(xy.u) : A}\ x, y \notin \Gamma$$

The typing relation of T satisfies the properties we have stated for STLC, where the canonical form property now reads as follows:

- If $\vdash s : \mathsf{N}$ and s is normal, then $s = S^n O$ for some n.
- If $\vdash s : A \to B$ and s is normal, then $s = \lambda x{:}A.t$ for some t.

T can be seen as a definition of the class of higher-order primitive recursive functions. It is straightforward to express Ackermann's function in T. Ackermann's function is not first-order primitive recursive.

Gödel sketched T in a seminal paper in 1958. He showed how proofs in an arithmetic systems could be expressed in T, and argued that the strong normalization of T yields the consistency of the arithmetic system. A strong normalization proof for T was first given by Tait in 1967.

Exercise 46 It is possible to formulate T with simpler terms as follows:

$$s, t \ ::= \ O \mid S \mid \mathsf{prec} \mid x \mid st \mid \lambda x{:}A.s$$

In this version the operator *prec* relies on abstractions obtained with λ. There are advantages and disadvantages to this approach.

a) Express the term $\mathsf{prec}\ s\,t\,(xy.u)$ of the old system in the new system.

b) Express the operator prec of the new system in the old system.

c) Give the typing rules for the terms S and prec in the new system.

d) Give the reduction rules for redexes obtained with prec in the new system.

e) Explain why the unique type property is lost in the new system.

f) State the canonical form property of the new system.

Exercise 47 Formalize T in Coq using de Bruijn indices.

5.3 Girard's F

F extends STLC with polymorphic types $\forall X.A$ whose members are functions mapping a type X to a member of A where A may depend on X. The most straightforward polymorphic type is $\forall X.X \to X$. The single member of this type is the polymorphic identity function $\lambda X.\lambda x{:}X.x$. In Coq, the universe for the type variables of F would be *Prop*.[1] F is also known as the *polymorphic lambda calculus*.

F can express the natural numbers with primitive recursion using Church numerals. In fact, F subsumes T. The type of natural numbers can be expressed in F as follows:[2]

$$nat := \forall X.\ X \to (X \to X) \to X$$

The **types** and **terms** of F are defined as follows:

$$\begin{aligned} A, B \ &::= \ X \mid A \to B \mid \forall X.A \\ s, t \ &::= \ x \mid st \mid \lambda x{:}A.s \mid sA \mid \lambda X.s \end{aligned}$$

Reduction can contract β-redexes everywhere.

$$\begin{aligned} (\lambda x{:}A.s)t \ &\succ \ s^x_t \\ (\lambda X.s)A \ &\succ \ s^X_A \end{aligned}$$

We assume that type variables and term variables are distinct. The reduction relation of F is confluent.

Since F has binders for both term and type variables, contexts must account for both kinds of variables. There are different ways to define contexts for F. We follow Harper [2013] and complement Γ with a separate context Δ for type variables, where Δ is just a finite set of type variables. We first define a **typing relation $\Delta \vdash A$**.

$$\frac{X \in \Delta}{\Delta \vdash X} \qquad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B} \qquad \frac{\Delta, X \vdash A}{\Delta \vdash \forall X.A}\ X \notin \Delta$$

Note that in the rule for abstractions X must not occur in Δ. This is fine since the local variable X is a notational device that can be renamed. Informally, $\Delta \vdash A$ holds if and only if Δ contains all variables that are free in A.

[1] In Coq, the universe *Prop* is impredicative while the universes behind *Type* are predicative.

[2] The arguments of Church numerals are swapped so that the canonical elements of *nat* are exactly the Church numerals.

Next we define the **primary typing relation** $\Delta\;\Gamma \vdash s : A$.

$$\frac{(x : A) \in \Gamma}{\Delta\;\Gamma \vdash x : A}$$

$$\frac{\Delta\;\Gamma \vdash s : A \to B \qquad \Delta\;\Gamma \vdash t : A}{\Delta\;\Gamma \vdash st : B} \qquad\qquad \frac{\Delta \vdash A \qquad \Delta\;\Gamma \vdash s : \forall X.B}{\Delta\;\Gamma \vdash sA : B^X_A}$$

$$\frac{\Delta \vdash A \qquad \Delta\;\Gamma, x : A \vdash s : B}{\Delta\;\Gamma \vdash \lambda x{:}A.s : A \to B}\; x \notin \Gamma \qquad\qquad \frac{\Delta, X\;\Gamma \vdash s : A}{\Delta\;\Gamma \vdash \lambda X.s : \forall X.A}\; X \notin \Delta$$

It is also necessary to define **valid contexts**. A pair $\Delta\;\Gamma$ is **valid** if $\Delta \vdash A$ for every declaration $(x : A) \in \Gamma$. Informally, $\Delta\;\Gamma$ is valid if and only if Δ contains all variables that are free in Γ. For subject reduction to hold we need that $\Delta\;\Gamma \vdash s : A$ is derivable and in addition that $\Delta\;\Gamma$ is valid. Note that the empty context $\emptyset\;\emptyset$ is valid.

For valid contexts, the typing relation of F satisfies the properties we have stated for STLC. The canonical form property for F looks as follows:

- If $\vdash s : A \to B$ and s is normal, then $s = \lambda x{:}A.t$ for some t.
- If $\vdash s : \forall X.A$ and s is normal, then $s = \lambda X.t$ for some t.

Exercise 48 Convince yourself that the typing

$$x : X \vdash (\lambda X.\,(\lambda x{:}X.x)x)(\forall X.X) : \forall X.X$$

is derivable. Subject reduction does not hold for the typing since the term reduces to x. Explain why the context of the typing is not valid. What happens if we employ the valid context $\{X\}\;\{x : X\}$?

Exercise 49 A **canonical element** of a type A is a normal term s such that $\vdash s : A$. One can show that the canonical elements of the type $\forall X.\, X \to (X \to X) \to X$ are exactly the Church numerals with swapped argument order. Show that the type $\forall X.\,(X \to X) \to X \to X$ has the ordinary Church numerals as canonical elements and an extra canonical element representing the number 1.

Exercise 50 Express primitive recursion for *nat* in F. Use Coq to type check your solution. Also use Coq to compute with your solution.

Exercise 51 Define a datatype *list nat* in F. Realize the constructors *nil* and *cons* and a fold function.

Exercise 52 Typing judgements can always be reduced to typing judgements for empty contexts. Let $\Delta = \{X_1, \dots, X_n\}$ and $\Gamma = \{x_1 : A_1, \dots, x_n : A_n\}$ be such that $\Delta\;\Gamma$ is a valid context. Given a term s and a type A, find a term t and a type B such that $\Delta\;\Gamma \vdash s : A$ if and only if $\vdash t : B$.

5.4 Logical Interpretation

The types of STLC and F can be seen as logical propositions. A typing judgement $\Gamma \vdash s : A$ says that s is a proof of the proposition A under the assumptions in Γ. Under this interpretation, the typing rules are the natural deduction rules for implication and universal quantification. If we omit the proof terms, the typing rules of F are the ND rules for implication and universal quantification.

$$\frac{A \in \Gamma}{\Delta\ \Gamma \vdash A}$$

$$\frac{\Delta \vdash A \to B \qquad \Delta\ \Gamma \vdash A}{\Delta\ \Gamma \vdash B} \qquad\qquad \frac{\Delta \vdash A \qquad \Delta\ \Gamma \vdash \forall X.\, B}{\Delta\ \Gamma \vdash B^X_A}$$

$$\frac{\Delta \vdash A \qquad \Delta\ \Gamma, A \vdash B}{\Delta\ \Gamma \vdash A \to B} \qquad\qquad \frac{\Delta, X\ \Gamma \vdash A}{\Delta\ \Gamma \vdash \forall X.\, A}\ X \notin \Delta$$

Note that Δ keeps track of the propositional variables in use, and that Γ collects the propositions for wich assume proofs. We can represent both Δ and Γ as sets. The system is set up such that the wellformedness of contexts is not automatically checked. That is, we need $\Delta \vdash \Gamma$ in addition to $\Delta\ \Gamma \vdash A$.

F can express falsity, conjunction, disjunction, and existential quantification:

$$\begin{aligned}
\bot &:= \forall Z.Z \\
A \wedge B &:= \forall Z.\, (A \to B \to Z) \to Z \\
A \vee B &:= \forall Z.\, (A \to Z) \to (B \to Z) \to Z \\
\exists X.\, A &:= \forall Z.\, (\forall X.\, A \to Z) \to Z
\end{aligned}$$

F is an amazing system. On the one hand, F is a computational system that can express types for numbers, lists, and trees together with a rich class of primitive recursive functions. On the other hand, F constitutes an intuitionistic proof system for quantified propositional logic. There is one thing missing, however: F cannot reason about the computational objects it can express.

From the logical point of view, term reduction in STLC and F simplifies proofs. Subject reduction tells us that proofs are preserved by reduction, and strong normalization tells us that every provable proposition has a normal proof. Showing that $\bot$ has no normal proof is not difficult in F. Together with strong normalization and subject reduction this yields the consistency of F (that is, $\bot$ has no proof).

5.5 F with Single-Sorted Syntax

A formalization of the two-sorted (types and terms) presentation of F with de Bruijn indices is a tedious enterprise since three different application operations for substitutions are needed. Things become easier if we switch to a single-sorted presentation where general terms represent both proper terms and types. We are familiar with a single-sorted syntax from Coq. We employ the following **terms**.

$$s,t,A,B \;=\; x \mid \mathsf{P} \mid A \to B \mid \forall x.\,A \mid st \mid \lambda x{:}A.s$$

The term P serves as **universe** for the types of F.[3] A **context** is a finite set of declarations $x : A$ where no variable is declared more than once. Declarations of the form $x : \mathsf{P}$ introduce type variables. Thus there is no need anymore for a separate context Δ for type variables. The typing relation $\Gamma \vdash s : A$ is defined as follows.

$$\frac{(x:A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash A : \mathsf{P} \qquad \Gamma \vdash B : \mathsf{P}}{\Gamma \vdash A \to B : \mathsf{P}} \qquad \frac{\Gamma, x : \mathsf{P} \vdash A : \mathsf{P}}{\Gamma \vdash \forall x.\,A : \mathsf{P}}\; x \notin \Gamma$$

$$\frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash st : B} \qquad \frac{\Gamma \vdash s : \forall x.\,B \qquad \Gamma \vdash A : \mathsf{P}}{\Gamma \vdash sA : B^x_A}$$

$$\frac{\Gamma \vdash A \to B : \mathsf{P} \qquad \Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x{:}A.s : A \to B}\; x \notin \Gamma \qquad \frac{\Gamma \vdash \forall x.\,A : \mathsf{P} \qquad \Gamma, x : \mathsf{P} \vdash s : A}{\Gamma \vdash \lambda x{:}\mathsf{P}.s : \forall x.\,A}\; x \notin \Gamma$$

Valid contexts are defined as follows:

$$\frac{}{\emptyset \text{ valid}} \qquad \frac{\Gamma \text{ valid}}{\Gamma, x : \mathsf{P} \text{ valid}}\; x \notin \Gamma \qquad \frac{\Gamma \text{ valid} \qquad \Gamma \vdash A : \mathsf{P}}{\Gamma, x : A \text{ valid}}\; x \notin \Gamma$$

The rules ensure that every variable x free in a type A of a declaration $y : A$ in Γ has a declaration $x : \mathsf{P}$ in Γ.

Exercise 53 Explain why $\vdash \mathsf{P} : \mathsf{P}$ cannot be derived.

Exercise 54 Formalize *F* in Coq using single-sorted syntax and de Bruijn indices.

5.6 F with Uniform Syntax

The presentation of F can be further simplified. As is, F comes with two function types that require separate rules for type formation, application, and abstraction. This duplication can be eliminated by working with a single function type

[3] *P* plays the role of *Prop* in Coq.

$\forall x{:}A.B$. If the variable x is not free in B, $\forall x{:}A.B$ represents the ordinary function type $A \to B$. Moreover, $\forall x{:}\mathsf{P}.A$ represents the function type $\forall x.A$. General function types $\forall x{:}A.B$ are called **products**. Products will give us an elegant representation of F that will be the basis for richer systems.[4] We also provide a second universe T that acts as type of the universe P.

The uniform presentation of F employs the following **terms**:

$$s,t,A,B \;=\; x \mid \mathsf{P} \mid \mathsf{T} \mid \forall x{:}A.A \mid st \mid \lambda x{:}A.s \qquad (n \in \mathbb{N})$$

A **context** is a finite set of declarations $x : A$ where no variable is declared twice. We define the **typing relation** $\Gamma \vdash s : A$ as follows.

$$\frac{}{\Gamma \vdash \mathsf{P} : \mathsf{T}}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash A : u \qquad \Gamma, x : A \vdash B : \mathsf{P}}{\Gamma \vdash \forall x{:}A.B : \mathsf{P}}\; x \notin \Gamma$$

$$\frac{\Gamma \vdash s : \forall x{:}A.B \qquad \Gamma \vdash t : A}{\Gamma \vdash st : B^x_t}$$

$$\frac{\Gamma \vdash \forall x{:}A.B : u \qquad \Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x{:}A.s : \forall x{:}A.B}\; x \notin \Gamma$$

The metavariable u ranges over the universes P and T. **Valid contexts** are defined as follows:

$$\frac{}{\emptyset \text{ valid}} \qquad\qquad \frac{\Gamma \text{ valid} \qquad \Gamma \vdash A : u}{\Gamma, x : A \text{ valid}}\; x \notin \Gamma$$

If Γ is a valid context, we have the following:

1. If $\Gamma \vdash s : \mathsf{T}$, then $s = \mathsf{P}$.
2. If $\Gamma \vdash s : \mathsf{P}$, then s corresponds to a type of the two-sorted presentation of F. In particular, if s contains a term $\forall x{:}A.B$ with $A \neq \mathsf{P}$, then x is not free in B. Hence $\forall x{:}A.B$ represents the type $A \to B$.

[4] Calling function types products is common in type theory. There is a name clash with cartesian products $A \times B$ in set theory, whose elements are pairs rather than functions.

5.7 F_ω and CC: Kinds and Conversion

Consider the logical interpretation of F. The abstraction

$$\lambda X{:}\mathsf{P}.\, X \to \bot$$

describes a negation operator. F cannot type this abstraction since it does not provide the type $\mathsf{P} \to \mathsf{P}$ of the abstraction. Types of this form are often called **kinds**. It is straightforward to extend the uniform presentation of F with kinds. We just need an additional product rule providing the function types representing kinds.

$$\frac{\Gamma \vdash A : \mathsf{T} \qquad \Gamma, x : A \vdash B : \mathsf{T}}{\Gamma \vdash \forall x{:}A.\,B : \mathsf{T}}\; x \notin \Gamma$$

We can now type the negation function $\lambda X{:}\mathsf{P}.\, X \to \bot$ with $\mathsf{P} \to \mathsf{P}$. We also have available the kind $(\mathsf{P} \to \mathsf{P}) \to \mathsf{P}$ for the function describing the existential quantifier.

Consider the well-typed application

$$(\lambda X{:}\mathsf{P}.\, X \to \bot)\bot$$

It reduces to the implication $\bot \to \bot$, which is provable. As is, our typing rules do not provide a proof for the equivalent term $(\lambda X{:}\mathsf{P}.\, X \to \bot)\bot$. This can be fixed by adding the **conversion rule**

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash B : \mathsf{P}}{\Gamma \vdash s : B}\; A \equiv_\beta B$$

where $\equiv_\beta$ stands for β-equivalence.

We have now arrived at a system know as F_ω. Reduction in F_ω is β-reduction and is confluent. For valid contexts, F_ω has the properties we have stated for STLC, where types are now unique up to β-equivalence. From confluence and strong normalization it follows that β-equivalence of well-typed terms is decidable. The canonical form property now reads as follows:

- If $\vdash s : \forall x{:}A.\,B$ and s is normal, then $s = \lambda x{:}A.t$ for some t.
- If $\vdash s : \mathsf{P}$ and s is normal, then s has the form $\forall x{:}A.\,B$.
- If $\vdash s : \mathsf{T}$, then s has the form $s ::= \mathsf{P} \mid s \to s$.

F_ω contains F as a subsystem (omit the conversion rule and the product rule for kinds). F in turn contains the STLC as a subsystem: Restrict the product rule of F to

$$\frac{\Gamma \vdash A : \mathsf{P} \qquad \Gamma, x : A \vdash B : \mathsf{P}}{\Gamma \vdash \forall x{:}A.\,B : \mathsf{P}}\; x \notin \Gamma$$

In this formulation of STLC a typing $\Gamma \vdash s : A$ is only derivable if every type variable free in A is declared in Γ.

We observe the following:

- STLC has product types for $\mathsf{P} \times \mathsf{P}$.
- F has product types for $\mathsf{P} \times \mathsf{P}$ and $\mathsf{T} \times \mathsf{P}$.
- F_ω has product types for $\mathsf{P} \times \mathsf{P}$, $\mathsf{T} \times \mathsf{P}$, and $\mathsf{T} \times \mathsf{T}$.

It turns out that F_ω can be extended with the missing products for $\mathsf{P} \times \mathsf{T}$. This yields the **basic calculus of constructions** CC designed by Coquand and Huet in 1985 as the starting point for Coq. Since all four products are allowed, a single product rule suffices for CC (u and v range over the universes P and T):

$$\frac{\Gamma \vdash A : v \qquad \Gamma, x : A \vdash B : u}{\Gamma \vdash \forall x : A.B : u}\; x \notin \Gamma$$

For CC, the conversion rules is generalized so that conversion becomes possible both at P and T.

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash B : u}{\Gamma \vdash s : B}\; A \equiv_\beta B$$

For valid contexts, the typing relation of CC enjoys the properties we have stated for STLC, where the unique types and the canonical forms properties need to be modified. We remark that the strong normalization proof gets harder with each step we climb up the ladder from STLC to CC.

F_ω appeared in Girard's habilitation in 1972. Girard uses a three-sorted syntax distinguishing between terms, types, and kinds and proves strong normalization of F_ω. Pierce's textbook presents F_ω with three-sorted syntax. Single-sorted syntax and general products appeared in the work of de Bruijn and Martin-Löf and are used in Coquand and Huet's initial presentation of the calculus of constructions in 1985. See the handbook chapter of Barendregt and Geuvers [2001] for a comprehensive discussion of CC-like systems called pure type systems.

Exercise 55 Find a term s so that $\vdash s : (\lambda X : \mathsf{P}.\mathsf{P})\bot$ is derivable. Write down the derivation in detail.

Exercise 56 Give functions describing conjunction, disjunction, and existential quantification in F_ω. Start by stating the types for this functions. Check your results with Coq. Prove in Coq that your definitions are equivalent to Coq's predefined versions of conjunction, disjunction, and existential quantification.

5.8 Calculus of Constructions

In CC, the universe T has no type. Consequently, we cannot express function types like $\mathsf{T} \to \mathsf{P}$.

From the naive use of Coq one would expect $\mathsf{T} : \mathsf{T}$. This is, however, not the case. In fact, extending CC with a rule $\mathsf{T} : \mathsf{T}$ yields an inconsistent system where $\bot$ has a proof that has no normal form. A similar result was first shown by Girard [1972] and is know as **Girard's paradox**. A shorter proof of the inconsistency result was given by Hurkens in 1995. The bottom line summarizing these results says that a type theory is inconsistent if it has a universe that contains itself as a member.

The effect of $u : u$ can be obtained to some extend by a hierarchy of infinitely many universes $\mathsf{U}_0 : \mathsf{U}_1 : \mathsf{U}_2 : \cdots$ such that $\mathsf{U}_0 \subseteq \mathsf{U}_1 \subseteq \mathsf{U}_2 \subseteq \cdots$. The idea of an infinite and cumulative hierarchy of universes is known from set theory and was first employed for type theories by Martin-Löf.

We now consider the following **terms**:

$$s, t, A, B \;=\; x \mid \mathsf{U}_n \mid \forall x{:}A.A \mid st \mid \lambda x{:}A.s \qquad (x \in \mathbb{N},\; n \in \mathbb{N})$$

As before, a **context** is a finite set of declarations $x : A$ where no variable is declared twice. We start with the following typing rules (u ranges over the **universes** U_n):

$$\frac{}{\Gamma \vdash \mathsf{U}_n : \mathsf{U}_{n+1}}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash A : u \qquad \Gamma, x : A \vdash B : u}{\Gamma \vdash \forall x{:}A.B : u}\; x \notin \Gamma$$

$$\frac{\Gamma \vdash s : \forall x{:}A.B \qquad \Gamma \vdash t : A}{\Gamma \vdash st : B^x_t}$$

$$\frac{\Gamma \vdash A : u \qquad \Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x{:}A.s : \forall x{:}A.B}\; x \notin \Gamma$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash B : u}{\Gamma \vdash s : B}\; A \equiv_\beta B$$

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash s : B}\; A \preceq B$$

The **subtyping relation** employed by the final **subtyping rule** is defined as follows:

$$\frac{}{A \preceq A} \qquad \frac{}{\mathsf{U}_m \preceq \mathsf{U}_n}\; m < n \qquad \frac{B \preceq B'}{\forall x{:}A.B \preceq \forall x{:}A.B'}$$

Subtyping gives us a cumulative hierarchy $\mathsf{U}_0 \subseteq \mathsf{U}_1 \subseteq \mathsf{U}_2 \subseteq \cdots$. Subtyping is needed so that the product rule works as expected. For instance, $\forall x{:}\mathsf{U}_1.x$ is accommodated as an element of U_2. Without the subtyping rule, $\forall x{:}\mathsf{U}_1.x$ cannot be typed.

Note that the product rule closes every universe under taking products. Moreover, if $A : u$ and $B : v$ for some universes u and v, then the product $\forall x{:}A.B$ is an element of either u or v (the subtyping rule is essential here).

One could hope that the system we have arrived at subsumes F if U_0 is taken as P. However, this is not the case since $\forall x{:}\mathsf{U}_0.x$ is not an element of U_0. This can be fixed by adding a special product rule for U_0:

$$\frac{\Gamma \vdash A : u \qquad \Gamma, x : A \vdash B : \mathsf{U}_0}{\Gamma \vdash \forall x{:}A.B : \mathsf{U}_0}\; x \notin \Gamma$$

One says that this rule makes U_0 **impredicative**. The system we have arrived at is a calculus of constructions known as CC_ω. CC_ω is the basic type theory underlying Coq. An extension of CC_ω is studied in Luo [1994].

It turns out that U_0 is the only universe that can be made impredicative without losing consistency [Harper and Pollak, 1991].

Valid contexts are defined as before:

$$\frac{}{\emptyset \text{ valid}} \qquad \frac{\Gamma \text{ valid} \qquad \Gamma \vdash A : u}{\Gamma, x : A \text{ valid}}\; x \notin \Gamma$$

For valid contexts, the typing relation of CC_ω enjoys the properties we have stated for STLC. The unique types property must be modified to least types unique up to β-equivalence. The canonical forms property needs to be adapted as well. In addition to the properties stated for STLC, CC_ω satisfies the following property.

- **Propagation** If Γ valid and $\Gamma \vdash s : A$, then there exists a universe u such that $\Gamma \vdash A : u$.

A strong normalization proof for CC_ω can be found in Luo [1994].

5.9 Numbers as Inductive Type

The logical and the computational interpretation of CC_ω are separated in that we cannot express and prove propositions about numbers and primitive recursive functions. The situation changes if we accommodate numbers with an inductive type rather than with functions. Recall that we obtained T from STLC by adding the numbers as an inductive type. While T is a purely computational system, the situation changes if we extend CC_ω with inductive numbers since we can now type primitive recursion much more generally than before. It is now possible to express and prove propositions about numbers. It turns out that inductive proofs can be formulated as primitive recursive functions.

Type theories with dependent function types and inductive types were formulated first by Martin-Löf in 1972. In contrast to CC_ω, Martin-Löf's systems restrict all universes to be predicative. The initial versions of Coq did not provide inductive types. Inductive types appeared in Coq around 1991.

From the Coq perspective the eliminator for numbers is the most straightforward combination of match and fix. One may think of the eliminator as a match extended with direct structural recursion.

We extend the **terms** of CC_ω as follows.

$$s, t, A, B \;=\; \cdots \;\mid \mathsf{N} \mid \mathsf{O} \mid \mathsf{S}\,s \mid \mathsf{elim}_A\, s\, t_1\, t_2$$

We say that N, O, and S are **constructors** and that elim is an **eliminator**. The eliminator gives us primitive recursion and inductive proofs. The recursion is obtained with two **reduction rules**.

$$\begin{aligned} \mathsf{elim}_A\, \mathsf{O}\, t_1\, t_2 &\succ t_1 \\ \mathsf{elim}_A\, (\mathsf{S}\,s)\, t_1\, t_2 &\succ t_2\, s\, (\mathsf{elim}_A\, s\, t_1\, t_2) \end{aligned}$$

Coming from Coq, it is helpful to think of the term $\mathsf{elim}_A\, s\, t_1\, t_2$ as a recursive match for $s : nat$. The terms t_1 and t_2 represent the two branches of the match (t_1 is the branch for O and t_2 is the branch for S). The second argument of t_2 takes the result of the direct structural recursion. The term A represents the so-called **return type function** of the match. This function is present in Coq but is usually inferred automatically.

Here are the **typing rules** for the new constructs.

$$\frac{}{\Gamma \vdash \mathsf{N} : \mathsf{U}_1} \qquad \frac{}{\Gamma \vdash \mathsf{O} : \mathsf{N}} \qquad \frac{\Gamma \vdash s : \mathsf{N}}{\Gamma \vdash \mathsf{S}\,s : \mathsf{N}}$$

$$\frac{\Gamma \vdash A : \mathsf{N} \to u \qquad \Gamma \vdash s : \mathsf{N} \qquad \Gamma \vdash t_1 : A\mathsf{O} \qquad \Gamma \vdash t_2 : \forall n{:}\mathsf{N}.\, An \to A(\mathsf{S}n)}{\Gamma \vdash \mathsf{elim}_A\, s\, t_1\, t_2 : As}$$

Note that the instance $u = \mathsf{U}_0$ of the typing rule for the eliminator yields the induction principle for natural numbers.

The extension of CC_ω with inductive numbers preserves the important properties of the typing relation.

Given an inductive type definition, Coq automatically derives a function representing the eliminator for the type. For the inductive definition of *nat*, the elimination function is declared with the name *nat_rect*.

The theories underlying early versions of Coq reduced match and fix to eliminators. So match and fix appeared as syntactic convenience for eliminators. Current versions of Coq provide for more permissive uses of match and fix. So far a fully satisfying foundation for the permissive use of match and fix is missing.

5.10 Dependent Pair Types

The elements of a function type $\forall x : A.\, B$ are functions that take an element x of A to an element of B, where the type B may depend on x. Dependent pair types $\Sigma x : A.\, B$ apply this typing idea to pairs: The elements of $\Sigma x : A.\, B$ are pairs (x, y) such that x is an element of A and y is an element of B, where B may depend on x. Following the notation (due to Martin-Löf), dependent pair types are often called **Σ-types**.

We now extend CC_ω with dependent pair types. The usual eliminators for pair types are the projections π_1 (first component) and π_2 (second component). We will also consider a more expressive eliminator, which represents the match coming with the inductive definition of dependent pair types in Coq.

We extend the **terms** of CC_ω as follows:

$$s, t, A, B \;=\; \cdots \;\mid\; \Sigma x : A.B \;\mid\; (s, t)_A \;\mid\; \pi_1\, s \;\mid\; \pi_2\, s \;\mid\; \mathsf{elim}_A\, s\, t$$

The **reduction rules** for the new eliminators are as follows.

$$\begin{aligned} \pi_1\,(s_1, s_2)_A &\succ s_1 \\ \pi_2\,(s_1, s_2)_A &\succ s_2 \\ \mathsf{elim}_A\,(s_1, s_2)_B\; t &\succ t\, s_1\, s_2 \end{aligned}$$

There is an additional **subtyping rule** for pair types.

$$\frac{A \preceq A' \qquad B \preceq B'}{\Sigma x : A.\, B \preceq \Sigma x : A'.\, B'}$$

Here are the **typing rules** for the new constructors and eliminators.

$$\frac{\Gamma \vdash A : u \qquad \Gamma, x : A \vdash B : u}{\Gamma \vdash \Sigma x{:}A.\,B : u}\; x \notin \Gamma,\; u \neq \mathsf{U}_0$$

$$\frac{\Gamma, x : A \vdash B : u \qquad \Gamma \vdash s : A \qquad \Gamma \vdash t : B^x_s}{\Gamma \vdash (s,t)_{\Sigma x:A.B} : \Sigma x{:}A.\,B}\; x \notin \Gamma,\; u \neq \mathsf{U}_0$$

$$\frac{\Gamma \vdash s : \Sigma x{:}A.\,B}{\Gamma \vdash \pi_1\, s : A} \qquad\qquad \frac{\Gamma \vdash s : \Sigma x{:}A.\,B}{\Gamma \vdash \pi_2\, s : B^x_{\pi_1 s}}$$

$$\frac{\begin{array}{l}\Gamma \vdash C : (\Sigma x{:}A.\,B) \to u \\ \Gamma \vdash s : \Sigma x{:}A.\,B \qquad\qquad \Gamma \vdash t : \forall x{:}A.\,\forall y{:}B.\; C\,(x,y)_{\Sigma x:A.B}\end{array}}{\Gamma \vdash \mathsf{elim}_C\, s\, t : C s}\; x \neq y$$

The first rule asserts that all universes but U_0 are closed under taking dependent pair types. The impredicative universe U_0 is excluded to avoid inconsistency. The rules for pairing and projection express basic intuitions. Pairs are annotated with their types so that the decidability of the typing relation is preserved. The rule for the eliminator elim corresponds to the typing rule for the matches Coq provides if Σ-types are defined inductively. The argument C of elim represents the **return type function** of the match.

It is not difficult to express the projections π_1 and π_2 with the eliminator elim. On the other hand, expressing the eliminator elim with the projections π_1 and π_2 seems impossible.

The extension of CC_ω with dependent pair types preserves the important properties of the typing relation. Proofs can be found in Luo [1994].

In the logical interpretation, function types appear as implications and universal quantifications. Likewise, pair types appear as conjunctions and existential quantifications.

Existential quantifications can be expressed with function types if the universe of propositions is impredicative. For pair types in predicative universes the coding does not work since the coded pair types appear as members of the next higher universe.

Exercise 57 Study dependent pair types in Coq.

a) Define dependent pair types inductively.

b) Define a function *elim* that models the eliminator for dependent pair types.

c) Define the projections π_1 and π_2 using *elim*.

d) Prove the η-law $s = (\pi_1\, s, \pi_2\, s)$ using *elim*.

e) Convince yourself that you cannot prove the η-law using the projections.

f) Convince yourself that you cannot define *elim* using the projections.

Do not use matches except when you define *elim*.

5.11 Recommended Reading

- The first chapter of the homotopy type theory (HOTT) book [2013] gives an informal presentation of type theory relating it to set theory. There is also a discussion of dependent pair types.
- Luo's book [1994]. Comprehensive presentation of CC_ω with pair types. Complete proofs. Philosophical background.
- Handbook chapter of Barendregt and Geuvers [2001]. Covers pure type systems generalizing CC.
- Girard's book [1990]. Presents the story from STLC to F_ω.
- Martin-Löf's *An Intuionistic Theory of Types*. Revised version of a paper from 1972 at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.926. Official references to Martin-Löf's papers appear in the HOTT book.

6 PCF

PCF is a Turing complete system with general recursion. We present PCF with call-by-value reduction (CBV).

Bibliography

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax an Semantics*. North-Holland, 1984. Revised Edition.

[2] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators, an Introduction*. Cambridge University Press, 2008.

[3] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[4] Masako Takahashi. Parallel reductions in lambda calculus. *Inf. Comput.*, 118(1):120–127, 1995.