Lambda Calculus: A Case for Inductive Definitions

Ralph Matthes
Institut für Informatik der Universität München
Oettingenstraße 67, 80538 München
matthes@informatik.uni-muenchen.de

July 8, 2000

Abstract

These lecture notes intend to introduce to the subject of lambda calculus and types. A special focus is on the use of inductive definitions. The ultimate goal of the course is an advanced treatment of inductive types.

Contents

1 Overview

Typed λ-calculi are especially simple typed functional programming languages. The study of those basic formalisms has applications to the design and the development of programming languages, program logics and influences mathematical logic (especially structural proof theory) from which it originated.

Starting with the inductive definitions of (untyped) λ-terms and β-reduction, the normal terms, the weakly normalizing terms and the strongly normalizing terms are characterized inductively. This means, the properties of not having a reduct or not having an infinite reduction sequence, are turned into positive statements. Moreover, with strongly normalizing terms it is done in a way facilitating a proof of strong normalization of intersection-typed terms enormously (which would even cover η-rules). Concerning the weakly normalizing terms, an inductively defined relation is introduced for the proof of standardization (dealt with in the exercises). The technique of superdelopments is used to establish confluence. The system of universal types (system F à la Curry) is proved to be strongly normalizing, and system F à la Church is introduced at length. Again confluence and strong normalization are established which later requires but a small modification to cover the extensions of system F by monotone inductive types.

After having used induction on inductively defined sets so successfully, the means of induction are added to the lambda calculus itself: By a constructively-minded inspection of Tarski's fixed-point theorem the most general formulation of inductive types is gained (via the Curry-Howard isomorphism). This gives a lot more insight into the capability of system F for modelling abstract data types and into usual formulations of inductive types found in the literature. The idea of interpolation will elucidate why those systems give equal computational power. Unfortunately, the material concerning inductive types is not available in the present version of these notes.

In essence, this course is a presentation (of a logician's view) of several of the most important results on the syntax (and operational semantics) by help of general induction (in principle known to everybody who knows about context-free grammars), and then a reflection of the principle via λ-calculi with inductive types.

Description of the "course's philosophy": In some sense, the presentation will be more elementary than in standard textbooks because everything is accomplished in a strictly constructive fashion. On the other hand, it requires some level of mathematical sophistication to fully understand induction on inductively defined sets although the concept will be introduced carefully and applied to a wide range of examples (the first part of which could as well be treated by ordinary induction). Concerning content, most of the course will stick to results found in standard textbooks such as [Bar84, Bar93, GLT89, Hin97, Kri93, Mit96]. The reader may easily gain insight by comparing the quite different styles of presentation.

To sum up, the material is self-contained and is intended to convey several of the central insights of λ-calculus in a way which should also be interesting for those who already know the results.

Citations are quite rare in these notes. This does not indicate that I consider the results to be original. Credits are given in my research papers. In a future version, I might add many more citations to enhance fairness.

Acknowledgements to my colleagues Thorsten Altenkirch and Felix Joachimski for helpful comments on drafts of this work.

Section 2 deals with an extended example of simultaneous inductive definitions: context-free grammars. Two grammars for the same language are proven to be equal. The proofs are done so carefully that a functional program can be read off immediately. Its purpose is the transformation of the parse trees from one grammar into the other. But the main purpose is the illustration of induction.

In section 3 the concept of binding is motivated from calculus. Untyped λ-calculus is introduced: terms, substitution, β-equality, Curch numerals, β-reduction.

Confluence of β-reduction is defined and proved in section 4. Friends of diagrams may enjoy the proof of local confluence which is given beforehand since it is much more perspicuous. However, the method of establishing confluence in the spirit of M. Takahashi with superdevelopments à la F. von Raamsdonk, is explained in great detail in order to make it as conceivable as the local confluence proof.

Section 5 provides the notions of normalizability. Weak and strong normalizability are both characterized in a syntax-directed way—useful for proofs of normalization in later sections.

Types enter the scene in section 6: Simple and intersection types are motivated and studied. The technical development is confined to intersection types since it is more demanding as well as more informative: type assignment matters! The results are closure under substitution and reverse substitution, Inversion and Subject Reduction, and the main result the well-known fact that exactly the strongly normalizing terms are typable with intersection types.

In section 7 universal types prevail. First, the λ-terms get a richer typing system (system of universal types=system F à la Curry) which requires the candidate method for the proof of strong normalization. After a glance at the undecidability of type checking, the explicit form of system F is introduced with a richer term structure. Several technical complications occur (e. g., local confluence fails without typability restriction). A first set of examples demonstrates F's ability to encode datatypes. Strong normalization is proved as a preparation for the system with fixed-point types.

Section 8 is incomplete: After an illuminating ML program a discussion of the virtues of inductive types and the limitations of F's encodings therof can be found. It is intended to make this quite large a section in later versions of these lecture notes.

2 Introduction to Inductive Definitions

In theoretical computer science, inductive definitions are ubiquitous. Mostly, they appear in the disguise of formal grammars which are idealizations of natural language grammars. Therefore, this introduction to inductive definitions concentrates on examples of (even context-free) grammars. In [ASU86, Example 4.8] two equivalent grammars for arithmetic expressions are studied for the purpose of illustrating the elimination of left recursion. The first one is (with **id** a set of identifiers):

$$\begin{array}{lcl} \mathsf{E} & \to & \mathsf{E}+\mathsf{T} \mid \mathsf{T} \\ \mathsf{T} & \to & \mathsf{T}*\mathsf{F} \mid \mathsf{F} \\ \mathsf{F} & \to & (\mathsf{E}) \mid \mathbf{id} \end{array}$$

The intuition of the non-terminals is given by:

- E: expression = a sum of terms
- T: term = a product of factors
- F: factor = a parenthesized expression or an identifier

Thus we model that in arithmetic expressions, $*$ has a higher precedence than $+$, and that parentheses enforce grouping. The grammar is left recursive and therefore cannot be treated by top-down parsing methods: One runs into the loop $\mathsf{E} \to \mathsf{E}+\mathsf{T} \to \mathsf{E}+\mathsf{T}+\mathsf{T} \to \mathsf{E}+\mathsf{T}+\mathsf{T}+\mathsf{T} \to \dots$

The second grammar is as follows (with empty word ϵ):

$$\begin{array}{lcl} \mathsf{E} & \to & \mathsf{TH} \\ \mathsf{H} & \to & +\mathsf{TH} \mid \epsilon \\ \mathsf{T} & \to & \mathsf{FK} \\ \mathsf{K} & \to & *\mathsf{FK} \mid \epsilon \\ \mathsf{F} & \to & (\mathsf{E}) \mid \mathbf{id} \end{array}$$

The idea is to decompose an expression into a term and a string of the form $+\mathsf{T}+\mathsf{T}+\dots+\mathsf{T}$ (expressed by the auxiliary non-terminal H) while in the first grammar, the decomposition would be into a list of T's with $+$'s inbetween. The same idea is used for the terms.

The claim is that both grammars are equivalent, i. e., that the same expressions can be derived from the non-terminal E. How can we prove that with full mathematical rigour? Answer: By induction. More concretely: By induction on the generation of the expressions according to these grammars. This generation process may be made precise as follows: We simultaneously define the sets $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$ of strings derivable in the first grammar from the non-terminals E, T and F, respectively. This will be done by means of an inductive definition to be expressed by the following rules:

(1) If $e \in \mathcal{E}$ and $t \in \mathcal{T}$ then $e + t \in \mathcal{E}$.

(2) If $t \in \mathcal{T}$ then $t \in \mathcal{E}$.

(3) If $t \in \mathcal{T}$ and $f \in \mathcal{F}$ then $t * f \in \mathcal{T}$.

(4) If $f \in \mathcal{F}$ then $f \in \mathcal{T}$.

(5) If $e \in \mathcal{E}$ then $(e) \in \mathcal{F}$.

(6) If $i \in \mathbf{id}$ then $i \in \mathcal{F}$.

These rules have to be interpreted as describing the generation of the sets $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$, hence the three sets will have all the properties expressed by the rules, and an element only enters one of the three sets if this is possible by one of the six rules. E. g., $e + t$ only enters $\mathcal{E}$ by the first rule, if already $e \in \mathcal{E}$ and $t \in \mathcal{T}$.

Clearly, we are only interested in the set $\mathcal{E}$ of expressions, but we have to define sets for every non-terminal in the grammar.

Analogously, we inductively define five sets $\mathcal{E}', \mathcal{H}', \mathcal{T}', \mathcal{K}', \mathcal{F}'$, corresponding to the five non-terminals of the second grammar:

(a) If $t \in \mathcal{T}'$ and $h \in \mathcal{H}'$ then $th \in \mathcal{E}'$.

(b) If $t \in \mathcal{T}'$ and $h \in \mathcal{H}'$ then $+th \in \mathcal{H}'$.

(c) $\epsilon \in \mathcal{H}'$.

(d) If $f \in \mathcal{F}'$ and $k \in \mathcal{K}'$ then $fk \in \mathcal{T}'$.

(e) If $f \in \mathcal{F}'$ and $k \in \mathcal{K}'$ then $*fk \in \mathcal{K}'$.

(f) $\epsilon \in \mathcal{K}'$.

(g) If $e \in \mathcal{E}'$ then $(e) \in \mathcal{F}'$.

(h) If $i \in \mathbf{id}$ then $i \in \mathcal{F}'$.

Lemma 1 (Equivalence) $\mathcal{E} = \mathcal{E}'$.

Proof It is obvious that we cannot prove $\mathcal{E} = \mathcal{E}'$ in isolation. We first prove $\mathcal{E} \subseteq \mathcal{E}'$ and simultaneously prove $\mathcal{T} \subseteq \mathcal{T}'$ and $\mathcal{F} \subseteq \mathcal{F}'$ by induction on the simultaneous inductive definition of $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$. Proving by induction means arguing on the generation of all the strings in $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$ by help of the rules (1) to (6). Since the rules specify which strings have to be in $\mathcal{E}$, $\mathcal{T}$ or $\mathcal{F}$ before, we may assume that for those strings we already have that they are in $\mathcal{E}'$, $\mathcal{T}'$ and $\mathcal{F}'$, respectively. This is always called the induction hypothesis.

We go through (1) to (6):

(1) If $e + t \in \mathcal{E}$ has been concluded from $e \in \mathcal{E}$ and $t \in \mathcal{T}$, we have to show that $e + t \in \mathcal{E}'$. By induction hypothesis, $e \in \mathcal{E}'$ and $t \in \mathcal{T}'$. Since there is only one rule for $\mathcal{E}'$, we conclude that $e = t'h$ with $t' \in \mathcal{T}'$ and $h \in \mathcal{H}'$. Now, we first prove an auxiliary statement: If $h \in \mathcal{H}'$ and $t \in \mathcal{T}'$ then $h + t \in \mathcal{H}'$. This is proved by induction on $\mathcal{H}'$: If $h \in \mathcal{H}'$ due to rule (b)

then $h = +t'h'$ with $t' \in \mathcal{T}'$ and $h' \in \mathcal{H}'$ which entered $\mathcal{H}'$ before h. By induction hypothesis, $h' + t \in \mathcal{H}'$, hence $h + t = +t'h' + t \in \mathcal{H}'$ by rule (b) again. If $h \in \mathcal{H}'$ due to rule (c) then $h = \epsilon$. By rules (c) and (b), $h + t = \epsilon + t = +t = +t\epsilon \in \mathcal{H}'$. By help of this statement, we infer that $h + t \in \mathcal{H}'$, hence $e + t = t'h + t \in \mathcal{E}'$ by rule (a).

(2) If $t \in \mathcal{E}$ has been concluded from $t \in \mathcal{T}$, we have to show that $t \in \mathcal{E}'$, and the induction hypothesis is $t \in \mathcal{T}'$. We are done by rules (c) and (a).

(3) We need another auxiliary lemma: If $k \in \mathcal{K}'$ and $f \in \mathcal{F}'$ then $k * f \in \mathcal{K}'$. The proof and the whole case are similar to (1).

(4) Similar to (2).

(5) If $(e) \in \mathcal{F}$ has been concluded from $e \in \mathcal{E}$, we have to show that $(e) \in \mathcal{F}'$, and the induction hypothesis is $e \in \mathcal{E}'$. So were are done by rule (g).

(6) If $i \in \mathcal{F}$ has been concluded from $i \in \mathbf{id}$, we have to show that $i \in \mathcal{F}'$. This holds by rule (h).

Let us prove $\mathcal{E}' \subseteq \mathcal{E}$. Since the definitions of $\mathcal{E}'$, $\mathcal{H}'$, $\mathcal{T}'$, $\mathcal{K}'$ and $\mathcal{F}'$ are entangled, we need to prove something for all of those. Therefore, we inductively define two auxiliary sets $\mathcal{H}$ and $\mathcal{K}$. $\mathcal{H}$ is defined by mimicking rules (b) and (c):

(b+) If $t \in \mathcal{T}$ and $h \in \mathcal{H}$ then $+th \in \mathcal{H}$.

(c+) $\epsilon \in \mathcal{H}$.

Analogously, $\mathcal{K}$ is defined after the model of (e) and (f):

(e+) If $f \in \mathcal{F}$ and $k \in \mathcal{K}$ then $*fk \in \mathcal{K}$.

(f+) $\epsilon \in \mathcal{K}$.

Now, we can simultaneously prove $\mathcal{E}' \subseteq \mathcal{E}$, $\mathcal{H}' \subseteq \mathcal{H}$, $\mathcal{T}' \subseteq \mathcal{T}$, $\mathcal{K}' \subseteq \mathcal{K}$ and $\mathcal{F}' \subseteq \mathcal{F}$ by induction on the simultaneous inductive definition of $\mathcal{E}'$, $\mathcal{H}'$, $\mathcal{T}'$, $\mathcal{K}'$ and $\mathcal{F}'$. In order to do this, we first prove that (a) to (h) hold when all the primed entities $\mathcal{X}'$ are replaced by the $\mathcal{X}$, for $\mathcal{X} \in \{\mathcal{E}, \mathcal{H}, \mathcal{T}, \mathcal{K}, \mathcal{F}\}$.

(a) Show that if $t \in \mathcal{T}$ and $h \in \mathcal{H}$ then $th \in \mathcal{E}$. We need an auxiliary lemma: If $e \in \mathcal{E}$ and $e' \in \mathcal{E}$ then $e + e' \in \mathcal{E}$. It is proved by induction on the generation of $e' \in \mathcal{E}$: If $e' \in \mathcal{E}$ has been concluded by rule (1) then $e' = e'' + t$ for some previously found $e'' \in \mathcal{E}$ and some $t \in \mathcal{T}$. By induction hypothesis, $e + e'' \in \mathcal{E}$, hence by rule (1), $e + e' = e + e'' + t \in \mathcal{E}$. If $e' \in \mathcal{E}$ has been concluded by rule (2) from $e' \in \mathcal{T}$ then $e + e' \in \mathcal{E}$ by (1).

Now, the claim is proved for arbitrary $t \in \mathcal{T}$ by induction on $h \in \mathcal{H}$: If $h \in \mathcal{H}$ stems from rule (b+) then $h = +t'h'$ for some $t \in \mathcal{T}$ and some previously generated $h' \in \mathcal{H}$. By induction hypothesis, $t'h' \in \mathcal{E}$. Since $t \in \mathcal{E}$, the auxiliary lemma yields $th = t + t'h' \in \mathcal{E}$. If $h \in \mathcal{H}$ by rule (c+) then $h = \epsilon$ and therefore $th = t \in \mathcal{E}$.

(b) By rule (b+): If $t \in \mathcal{T}$ and $h \in \mathcal{H}$ then $+th \in \mathcal{H}$.

(c) $\epsilon \in \mathcal{H}$ by rule (c+).

(d) Show that if $f \in \mathcal{F}$ and $k \in \mathcal{K}$ then $fk \in \mathcal{T}$. In analogy to (a), we first prove $t, t' \in \mathcal{T} \Rightarrow t * t' \in \mathcal{T}$ by induction on $t' \in \mathcal{T}$ and then do induction on $k \in \mathcal{K}$.

(e) By rule (e+): If $f \in \mathcal{F}$ and $k \in \mathcal{K}$ then $*fk \in \mathcal{K}$.

(f) $\epsilon \in \mathcal{K}$ by rule (f+).

(g) By rule (5): If $e \in \mathcal{E}$ then $(e) \in \mathcal{F}$.

(h) If $i \in \mathbf{id}$ then $i \in \mathcal{F}$ by (6).

Therefore, the unprimed entities fulfill the defining clauses of the primed entities. Since the primed entities are assumed to be the smallest sets with those properties (elements only enter if one of the rules applies), we conclude that the primed entities are all included in the unprimed ones, respectively.

Readers not being familiar with this order-theoretic arguments may simply do an inductive proof of $x \in \mathcal{X}' \Rightarrow x \in \mathcal{X}$ for $\mathcal{X} \in \{\mathcal{E}, \mathcal{H}, \mathcal{T}, \mathcal{K}, \mathcal{F}\}$ in traditional style. We only consider this for rule (a): Assume that $th \in \mathcal{E}'$ has been concluded from $t \in \mathcal{T}'$ and $h \in \mathcal{H}'$. By induction hypothesis, $t \in \mathcal{T}$ and $h \in \mathcal{H}$. Therefore, by the validity of (a) with $\mathcal{T}'$, $\mathcal{H}'$ and $\mathcal{E}'$ replaced by $\mathcal{T}$, $\mathcal{H}$ and $\mathcal{E}$, respectively, we get $th \in \mathcal{E}$ and are done. $\square$

Since our proof only contained constructive arguments, it is straightforward to write a program by which the parse trees for expressions according to the first and the second grammar can be transformed into each other. The programming language we chose is Standard ML of New Jersey (Version 110.0.6 of October 1999).

The readers are not expected to be familiar with SML. Nevertheless, it should be possible to understand the program text without explaining the syntax. Only the intentions are given and the output of the system when fed with the program lines shown so far.

```
datatype
    'id expr = Pnode (* P for plus *) of 'id expr * 'id term |
               Tnode (* T for term *) of 'id term
and
    'id term = Mnode (* M for multiply *) of 'id term * 'id factor |
               Fnode (* F for factor *) of 'id factor
and
    'id factor = Bnode (* B for bracket *) of 'id expr |
                 Inode (* I for identifier *) of 'id;
```

Three inductive datatypes are introduced which are parameterized by `'id` representing the set of identifiers (in our examples this will always be the built-in type `string` of strings). They model the set of derivation trees for $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$. The rules (1) to (6) are represented by constructors (which are nothing but tags). So, e. g., Pnode stands for (1) and thus requires some `'id expr` and some 'id term (* denotes pairing). Note that only the derivation trees are modeled and not the expressions themselves. The system's answer (with its preferred type variable name 'a):

```
datatype 'a expr = Pnode of 'a expr * 'a term | Tnode of 'a term
datatype 'a term = Fnode of 'a factor | Mnode of 'a term * 'a factor
datatype 'a factor = Bnode of 'a expr | Inode of 'a
```

```
val f1 = Bnode(Pnode(Tnode(Fnode(Inode("7"))),Fnode(Inode("3"))));

val t1 = Fnode(Bnode(Tnode(Mnode(Fnode(Inode("8")),f1))));

val e1 = Pnode(Tnode(t1),Mnode(Fnode(Inode("6")),Inode("5")));

fun itplus(expr,n) =
    if n <= 0 then expr
    else Pnode(expr,Fnode(Bnode(itplus(expr,n-1))));
    (* for the formation of big terms *)

val e2 = itplus(e1,50);
val e3 = itplus(e1,1000);
val e4 = itplus(e1,100000);
```

Some examples are provided, leading to the output

```
val f1 = Bnode (Pnode (Tnode (Fnode #),Fnode (Inode #))) : string factor
val t1 = Fnode (Bnode (Tnode (Mnode (#,#)))) : string term
val e1 = Pnode (Tnode (Fnode (Bnode #)),Mnode (Fnode #,Inode #)) : string expr
val itplus = fn : 'a expr * int -> 'a expr
val e2 = Pnode (Pnode (Tnode #,Mnode #),Fnode (Bnode (Pnode #))) : string expr
val e3 = Pnode (Pnode (Tnode #,Mnode #),Fnode (Bnode (Pnode #))) : string expr
val e4 = Pnode (Pnode (Tnode #,Mnode #),Fnode (Bnode (Pnode #))) : string expr
```

Note that # indicates truncation of output. By recursion on the derivation trees we define the derived elements of $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$ as lists of symbols:

```
fun ppe(Pnode(expr,term)) = ppe(expr)@["+"]@ppt(term) |
    ppe(Tnode(term)) = ppt(term)

and
    ppt(Mnode(term,factor)) = ppt(term)@["*"]@ppf(factor) |
    ppt(Fnode(factor)) = ppf(factor)

and
    ppf(Bnode(expr)) = ["("]@ppe(expr)@[")"] |
    ppf(Inode(ident)) = [ident];

val str1 = ppf f1;
val str2 = ppt t1;
val str3 = ppe e1;
```

Note that @ denotes concatenation of lists. We get

```
val ppe = fn : string expr -> string list
val ppt = fn : string term -> string list
val ppf = fn : string factor -> string list
val str1 = ["(","7","+","3",")"] : string list
val str2 = ["(","8","*","(","7","+","3",")",")"] : string list
val str3 = ["(","8","*","(","7","+","3",")",")","+","6","*",...] : string list
```

Note that ... denotes truncation of the list (although in this example ... stands for nothing but "5" which can be checked by entering the line
`(str3=["(","8","*","(","7","+","3",")",")","+","6","*","5"]);`
the response to which should be `val it = true : bool`).

The second grammar is dealt with similarly:

```
datatype
    'id ee = THnode of 'id tt * 'id hh
and
    'id hh = PPnode of 'id tt * 'id hh |
             EmptyH
and
    'id tt = FKnode of 'id ff * 'id kk
and
    'id kk = MMnode of 'id ff * 'id kk |
             EmptyK
and
    'id ff = BBnode of 'id ee |
             IInode of 'id;

(* examples *)
val hh1 = PPnode(FKnode(IInode("6"),MMnode(IInode("5"),EmptyK)),EmptyH);

val kk1 = MMnode(BBnode(THnode(FKnode(IInode("7"),EmptyK),
                PPnode(FKnode(IInode("3"),EmptyK),EmptyH))),EmptyK);

val ee1 = THnode(FKnode(BBnode(THnode(FKnode(IInode("8"),kk1),
                                     EmptyH)),EmptyK),hh1);

(* string representation *)
fun ppee(THnode(tt,hh)) = pptt(tt)@pphh(hh)
and
    pphh(PPnode(tt,hh)) = ["+"]@pptt(tt)@pphh(hh) |
    pphh(EmptyH) = nil
and
    pptt(FKnode(ff,kk)) = ppff(ff)@ppkk(kk)
and
    ppkk(MMnode(ff,kk)) = ["*"]@ppff(ff)@ppkk(kk) |
    ppkk(EmptyK) = nil
and
    ppff(BBnode(ee)) = ["("]@ppee(ee)@[")"] |
    ppff(IInode(ident)) = [ident];
```

This yields

```
datatype 'a ee = THnode of 'a tt * 'a hh
```

```
datatype 'a hh = EmptyH | PPnode of 'a tt * 'a hh
datatype 'a tt = FKnode of 'a ff * 'a kk
datatype 'a kk = EmptyK | MMnode of 'a ff * 'a kk
datatype 'a ff = BBnode of 'a ee | IInode of 'a
val hh1 = PPnode (FKnode (IInode #,MMnode #),EmptyH) : string hh
val kk1 = MMnode (BBnode (THnode (#,#)),EmptyK) : string kk
val ee1 = THnode (FKnode (BBnode #,EmptyK),PPnode (FKnode #,EmptyH))
  : string ee
val ppee = fn : string ee -> string list
val pphh = fn : string hh -> string list
val pptt = fn : string tt -> string list
val ppkk = fn : string kk -> string list
val ppff = fn : string ff -> string list
```

```
val str4 = ppee(ee1);
val ok1 = (str3 = str4);
```

leads to

```
val str4 = ["(","8","*","(","7","+","3",")",")","+","6","*",...] : string list
val ok1 = true : bool
```

and shows that ee1 is the derivation in the second grammar of the same expression as that derived by e1 in the first one.

Now we come to the conversion from the first grammar to the second grammar reflecting the first part of the proof of the lemma. The auxiliary statement for (1), namely if $h \in \mathcal{H}'$ and $t \in \mathcal{T}'$ then $h + t \in \mathcal{H}'$, is turned into a function plus which takes an argument in 'a hh and one in 'a tt to produce that in 'a hh reflecting their addition. Similarly, we define mult to reflect the auxiliary lemma for (3). Moreover, we exploit that there are only one rule for $\mathcal{E}'$ and $\mathcal{T}'$ each which allows to decompose the derivations into two pieces.

```
fun plus(PPnode(tt',hh),tt) = PPnode(tt',plus(hh,tt)) |
    plus(EmptyH,tt) = PPnode(tt,EmptyH);

fun mult(MMnode(ff',kk),ff) = MMnode(ff',mult(kk,ff)) |
    mult(EmptyK,ff) = MMnode(ff,EmptyK);

fun eetott(THnode(tt,hh)) = tt;

fun eetohh(THnode(tt,hh)) = hh;

fun tttoff(FKnode(ff,kk)) = ff;

fun tttokk(FKnode(ff,kk)) = kk;
```

```
val plus = fn : 'a hh * 'a tt -> 'a hh
val mult = fn : 'a kk * 'a ff -> 'a kk
val eetott = fn : 'a ee -> 'a tt
val eetohh = fn : 'a ee -> 'a hh
val tttoff = fn : 'a tt -> 'a ff
val tttokk = fn : 'a tt -> 'a kk
```

The following definitions of the transforming functions are straightforwardly read off the proofs of (1) to (6). Clearly, they have to be defined simultaneously, and every application of the induction hypothesis now becomes some recursive call to one of the three functions about to be defined.

```
fun cve(Pnode(expr,term)) =
    let val ee = cve(expr) in
        THnode(eetott(ee),plus(eetohh(ee),cvt(term)))
    end |
    cve(Tnode(term)) = THnode(cvt(term),EmptyH)

and
    cvt(Mnode(term,factor)) =
    let val tt = cvt(term) in
        FKnode(tttoff(tt),mult(tttokk(tt),cvf(factor)))
    end |
    cvt(Fnode(factor)) = FKnode(cvf(factor),EmptyK)

and
    cvf(Bnode(expr)) = BBnode(cve(expr))|
    cvf(Inode(ident)) = IInode(ident);

val ok2=(cve(e1)=ee1);
```

Notice that the `let` construction provides for abbreviations used locally.

```
val cve = fn : 'a expr -> 'a ee
val cvt = fn : 'a term -> 'a tt
val cvf = fn : 'a factor -> 'a ff
val ok2 = true : bool
```

We turn to the other direction corresponding to the proof of $\mathcal{E}' \subseteq \mathcal{E}$. The two auxiliary sets $\mathcal{H}$ and $\mathcal{K}$ are represented by two additional inductive datatypes:

```
datatype    'id auxh = PPPnode of 'id term * 'id auxh |
    EmptyAuxH;

datatype    'id auxk = MMMnode of 'id factor * 'id auxk |
    EmptyAuxK;
```

```
datatype 'a auxh = EmptyAuxH | PPPnode of 'a term * 'a auxh
datatype 'a auxk = EmptyAuxK | MMMnode of 'a factor * 'a auxk
```

We again need auxiliary functions reflecting the lemmas

- If $e \in \mathcal{E}$ and $e' \in \mathcal{E}$ then $e + e' \in \mathcal{E}$.
- If $t \in \mathcal{T}$ and $h \in \mathcal{H}$ then $th \in \mathcal{E}$.
- If $t \in \mathcal{T}$ and $t' \in \mathcal{T}$ then $t * t' \in \mathcal{T}$.
- If $f \in \mathcal{F}$ and $k \in \mathcal{K}$ then $fk \in \mathcal{T}$.

embedded into the proof. The operations on the derivations are given in the order of those lemmas:

```
fun pluse(expr,Pnode(expr',term)) = Pnode(pluse(expr,expr'),term) |
    pluse(expr,Tnode(term)) = Pnode(expr,term);

fun concth(term,EmptyAuxH) = Tnode(term) |
    concth(term,PPPnode(term',auxh)) =
          pluse(Tnode(term),concth(term',auxh));


fun multt(term,Mnode(term',factor)) = Mnode(multt(term,term'),factor) |
    multt(term,Fnode(factor)) = Mnode(term,factor);

fun concfk(factor,EmptyAuxK) = Fnode(factor) |
    concfk(factor,MMMnode(factor',auxk)) =
          multt(Fnode(factor),concfk(factor',auxk));
```

```
val pluse = fn : 'a expr * 'a expr -> 'a expr
val concth = fn : 'a term * 'a auxh -> 'a expr
val multt = fn : 'a term * 'a term -> 'a term
val concfk = fn : 'a factor * 'a auxk -> 'a term
```

As before, no intuition is needed to produce the transforming functions from the proofs of (a) to (h). However, note that since we only showed them for the rules where the primed entities $\mathcal{X}'$ have been replaced by the $\mathcal{X}$, the applications of the induction hypotheses are hidden, but nevertheless the recursive calls have to be made as for the other direction.

```
fun cvee(THnode(tt,hh)) = concth(cvtt(tt),cvhh(hh))

and
    cvhh(PPnode(tt,hh)) = PPPnode(cvtt(tt),cvhh(hh)) |
    cvhh(EmptyH) = EmptyAuxH
and
    cvtt(FKnode(ff,kk)) = concfk(cvff(ff),cvkk(kk))
and
    cvkk(MMnode(ff,kk)) = MMMnode(cvff(ff),cvkk(kk)) |
    cvkk(EmptyK) = EmptyAuxK
and
    cvff(BBnode(ee)) = Bnode(cvee(ee)) |
    cvff(IInode(ident)) = Inode(ident);

val ok3 = (cvee(ee1) = e1);

val ok4 = (cvee(cve(e2)) = e2);

val ok5 = (cvee(cve(e3)) = e3);

val ok6 = (cvee(cve(e4)) = e4); (* takes some time *)
```

```
val cvee = fn : 'a ee -> 'a expr
val cvhh = fn : 'a hh -> 'a auxh
val cvtt = fn : 'a tt -> 'a term
val cvkk = fn : 'a kk -> 'a auxk
val cvff = fn : 'a ff -> 'a factor
val ok3 = true : bool
val ok4 = true : bool
val ok5 = true : bool
val ok6 = true : bool
val it = () : unit
```

Note that the last line of output stems from the execution of the whole program loaded into SML by the command `use("expr.sml")` where `expr.sml` is the name of the source file.

3 Lambda Calculus

We introduce λ-calculus in its simplest form: There is only λ-abstraction and application, and no typing whatsoever. Nevertheless, in the motivating examples, a richer signature will freely be used.

3.1 Motivation

Imagine how in mathematical texts you will express that the function f is the squaring function. The easiest way to do that is by saying that $f(x) = x^2$ for all x. Others prefer to write $f := (\cdot)^2$, using the anonymous dot instead of the variable name x. But how would we denote the two-place function g which forms the sum of the squares of the arguments? We could write $g(x, y) := x^2 + y^2$ with the variables x and y. Is $g := (\cdot)^2 + (*)^2$ acceptable? It does not indicate which is the first and which is the second argument. Hence we prefer to have variable names.

If we want to speak about the sum of two squares, why should we first introduce some name g for that, instead of directly writing a mathematical description? In λ-calculus, we would write $\lambda x \lambda y. x^2 + y^2$.

How would we express the first derivative $\partial_2 g$ of $g := \lambda x \lambda y. x^2 + y^2$ w. r. t. the second argument (often written $\frac{\partial g}{\partial y}$)? It is again a function of two arguments:

$$\partial_2 g := \lambda x \lambda y. \lim_{\to 0} \lambda h. \frac{g(x, y + h) - g(x, y)}{h}.$$

In order to use λ-notation as much as possible, we even have written

$$\lim_{\to 0} \lambda h. \frac{g(x, y + h) - g(x, y)}{h}$$

instead of the more usual

$$\lim_{h \to 0} \frac{g(x, y + h) - g(x, y)}{h}.$$

Since the construction does not depend on the concrete definition of g, we may again do an abstraction and define ∂_2 as an operation which takes a two-place function g and returns $\partial_2 g$, i. e., we define

$$\partial_2 := \lambda g \lambda x \lambda y. \lim_{\to 0} \lambda h. \frac{g(x, y+h) - g(x, y)}{h}.$$

Note that the limit need not exist for every g and that therefore, strictly speaking, ∂_2 is not well-defined. So, do not take this example too serious. A much more important issue: We do not need to have the name ∂_2 at hand in order to be able to express our concept of forming the derivative. We may simply say that it is given by

$$\lambda g \lambda x \lambda y. \lim_{\to 0} \lambda h. \frac{g(x, y+h) - g(x, y)}{h}.$$

Let us now calculate $\lambda x. \partial_2 (\lambda x \lambda y. x^2 + y^2)\, x\, 2$, i. e., the function taking any argument x to the partial derivative of our previous g w. r. t. y, evaluated at the point $(x, 2)$. By expanding the abbreviation, we get

$$\lambda x. \Big(\lambda g \lambda x \lambda y. \lim_{\to 0} \lambda h. \frac{g(x, y+h) - g(x, y)}{h} \Big) (\lambda x \lambda y. x^2 + y^2)\, x\, 2.$$

Clearly, we now want to replace the formal parameters g, x, y of ∂_2 by the expressions $\lambda x \lambda y. x^2 + y^2, x, 2$, respectively. This yields

$$\lambda x. \lim_{\to 0} \lambda h. \frac{(\lambda x \lambda y. x^2 + y^2)(x, 2+h) - (\lambda x \lambda y. x^2 + y^2)(x, 2)}{h}.$$

Again we replace the formal parameters x, y of g by the actual arguments. (For this example, we do not distinguish between two subsequent arguments and a pair of arguments.) We get

$$\lambda x. \lim_{\to 0} \lambda h. \frac{(x^2 + (2+h)^2) - (x^2 + 2^2)}{h}.$$

It is now a matter of algebra to see that the numerator of the fraction has the same value as $4h + h^2$. And $\lim_{\to 0} \lambda h. 4 + h$ will certainly be 4 since it is sufficient for that to evaluate $\lambda h. 4 + h$ at the argument 0, hence replacing the formal parameter h by 0 in $4 + h$, yielding $4 + 0$ which, by algebra, has the same value as 4. Hence, we have transformed $\lambda x. \partial_2 (\lambda x \lambda y. x^2 + y^2)\, x\, 2$ into $\lambda x. 4$ which does not allow any further simplification. So we may say that the constant function returning 4 on any input, is the result of our calculation.

In pure λ-calculus, we only model the bare bones of this example: There will be neither a reference to algebraic manipulations nor even to the concept of a limit. So, there will be no squares, no sums, no fractions, no subtractions. There are even no pairs (no tuples). But we may freely λ-abstract “formal” variables and may always apply one expression to another with the intuition that the first expression represents a function and the second one an argument to it. And the only mechanism for “calculation” will be the replacement of formal parameters by actual arguments.

3.2 Pure Untyped Lambda Calculus

Let an infinite set $\mathcal{V}$ of identifiers be given. The identifiers serve as names for variables. We usually denote elements of $\mathcal{V}$ by x, y, z. The identifiers themselves do not matter and will never appear in the presentation. The possibility of having different sets $\mathcal{V}$ is not exploited. The most basic λ-calculus only models functionals (since those functions may take as well functions as arguments, it is preferred to refer to them as functionals). This is done by giving a simple grammar for them—more precisely, by the following inductive definition of terms:

Definition 1 (Terms) *The set $\mathcal{T}$ of terms is inductively given by:*

- *If $x \in \mathcal{V}$ then $x \in \mathcal{T}$.*
- *If $x \in \mathcal{V}$ and $r \in \mathcal{T}$ then $\lambda x r \in \mathcal{T}$.*
- *If $r \in \mathcal{T}$ and $s \in \mathcal{T}$ then $(rs) \in \mathcal{T}$.*

The intention is that $\lambda x r$ models the function $x \mapsto r(x)$ in general mathematical language, where $r(x)$ is just r with the dependency on x indicated (hence only a metasyntactically blown up notation for r). In mathematics one would perhaps prefer to write $r(s)$ instead of the λ-calculus notation (rs) for an application of r to s.

By definition, terms are strings consisting of identifiers in $\mathcal{V}$, parentheses and the greek letter λ. Since we do not aim at studying parsing issues, we will view terms as trees, i. e., we identify a term with its inductive generation by the above definition. And as long as it is clear which tree is meant, we leave out parentheses. We also assume that application associates to the left and use the dot notation: A dot hides a pair of parentheses, which opens at the dot and closes as far to the right as is syntactically possible.

Examples 1 xx *is* (xx). xxx *is* $((xx)x)$. $\omega := \lambda x.xx$ *is* $\lambda x(xx)$. $\Omega := \omega\omega$ *is* $(\lambda x(xx)\lambda x(xx))$.

Note that all of the examples are quite counterintuitive since x is applied to x itself, hence x is on one hand viewed as a function and on the other as an argument to that function. By typing restrictions to be introduced later, those bizarre terms will be ruled out. Nevertheless, the theorems on pure λ-calculus always also hold for them.

To give at least some intuition for Ω, think of functions as predicates, i. e., functions with boolean range. Then fx means that f holds true of the argument x. Moreover, $\lambda x r$ is the predicate which holds true of x iff $r(x)$ is true. In this way, λ-abstraction becomes set comprehension. In a set-theoretic notation, ω would then correspond to $M := \{x \mid x \in x\}$ and Ω to the assertion that $M \in M$.[1]

[1] Bertrand Russell used the set $M := \{x \mid x \notin x\}$ and the assertion $M \in M$ for his famous set-theoretic paradox. As a remedy he proposed typed systems. The cure for unpleasant behaviour of λ-terms will also be types, i. e., the restriction to terms which follow some typing discipline.

Definition 2 (Free variables) *Define the set $FV(r)$ of variables occurring free in r by recursion on r:*

- $FV(x) := \{x\}$.
- $FV(\lambda x r) := FV(r) \setminus \{x\}$.
- $FV(rs) := FV(r) \cup FV(s)$.

Obviously, $FV(r)$ is always a finite subset of $\mathcal{V}$. As long as a term has free variables (a term without free variables is *closed*), its intended meaning depends on the assignment of values for the free variables. Hence, it is important to know the names of the variables. On the contrary, the variables occurring in r which are not free (the formal parameters or *bound variables*) are only a means of pointing to the places where to bind the actual argument to the formal parameter. Hence, there is no difference at all between $\lambda x x$ and $\lambda y y$. More generally, we will never make a distinction between terms differing only in the names of their bound variables as long as the internal references are the same.[2] So for us, $(\lambda x.xy)x \equiv (\lambda z.zy)x$ where $\equiv$ denotes syntactic equality. Surely, $(\lambda z.zy)x \not\equiv (\lambda y.yy)x$.

A word of caution: In the passage from $\lambda x r$ to r, x may become a free variable, and consequently its name matters. Hence in arguments by induction on terms, the case of an abstraction typically reads as follows: "Case $\lambda x r$. By renaming of the bound variable x, we may assume that x does not appear in the set M [given beforehand]. By induction hypothesis for r, ...". This means that although an arbitrary $\lambda x r$ has to be studied, we feel free to rename x in it. We even assume this has already been done and led to the choice x. Then we fix the variable name and break up $\lambda x r$ to yield r with possible free occurrences of x. We may now apply the induction hypothesis to r or rename other bound variables in r, etc.[3]

Definition 3 (Substitution) *Define the result $r[x := s]$ of replacing every free occurrence of the variable x in r by the term s recursively as follows:*

- $x[x := s] := s$
- $y[x := s] := y$ *for* $y \neq x$.
- $(\lambda y r)[x := s] := \lambda y.r[x := s]$ *where we may assume by renaming of the bound variable y that $y \notin \{x\} \cup FV(s)$.*
- $(rt)[x := s] := r[x := s]t[x := s]$.

[2] In the literature, this will often be called a variable convention or that terms are considered up to renaming of bound variables or modulo α-equivalence. A mathematically sound justification of this identification process is not as trivial as one might expect it to be. Moreover, for any operation on terms such as substitution defined below, the independence of the chosen representative of the term has to be checked.

[3] In a rigorous treatment, induction on the term structure would not even be available. Instead, one would have to argue by induction on the height of a term since the height is the same for α-equivalent terms.

Note in the case of an abstraction that $y = x$ would forbid any replacement since then x would not be free in $\lambda x r$. If we allowed $y \in FV(s)$, this free variable of s would be *captured* by the outer λ-abstraction although there has been no functional dependency beforehand. This would be counterintuitive and also make substitution incompatible with renaming of bound variables.

Lemma 2 *$r[x := s][y := t] = r[y := t][x := s[y := t]]$ for $x \notin \{y\} \cup FV(t)$.*

Proof Induction on r. $\square$

With the notion of substitution at hand, we are now able to define which terms are considered to be computationally equal if our computations are restricted to the replacement of formal parameters by arguments, i. e., when replacing $(\lambda x r)s$ by $r[x := s]$ in any part of the expressions.

Definition 4 (β-equality) *Let $=_\beta$ be the congruence relation generated from $(\lambda x r)s =_\beta r[x := s]$, i. e., $=_\beta$ is defined inductively by:*

(β) $(\lambda x r)s =_\beta r[x := s]$ (outer β-equality).

(ξ) $r =_\beta r' \Rightarrow \lambda x r =_\beta \lambda x r'$ (β-equality under an abstraction).

(a) $r =_\beta r' \wedge s =_\beta s' \Rightarrow rs =_\beta r's'$ (application).

(r) $r =_\beta r$ (reflexivity).

(t) $r =_\beta s \wedge s =_\beta t \Rightarrow r =_\beta t$ (transitivity).

(s) $r =_\beta s \Rightarrow s =_\beta r$ (symmetry).

(ξ)[4] and (a) are the rules of compatibility with the term formation rules, the rules (r), (t) and (s) express that $=_\beta$ is an equivalence relation.

Clearly, it would suffice to restrict the reflexivity rule (r) to $x =_\beta x$ since (ξ) and (a) are present in the system. Note that the names β and ξ are standard notation.

Example 2 *$\Omega = (\lambda x.xx)\omega =_\beta (xx)[x := \omega] = \omega\omega = \Omega$ by rule (β). This does not sound interesting since reflexivity would also prove $\Omega =_\beta \Omega$.*

Example 3 (Church numerals) *Define $\underline{n} := \lambda f \lambda x. \underbrace{f(\ldots(f}_{n \text{ times}} x)\ldots)$. The term $\underline{n}$ is called the n-th Church numeral. It will be convenient to introduce the abbreviation $r^n s := \underbrace{r(\ldots(r}_{n \text{ times}} s)\ldots)$ such that $\underline{n} = \lambda f \lambda x. f^n x$. The composition of terms is defined as $r \circ s := \lambda x.r(sx)$ (for some $x \notin FV(r) \cup FV(s)$). Then for every natural numbers m and n and terms r we have that $(\underline{m}r) \circ (\underline{n}r) =_\beta \underline{m+n}r$. Thus addition is represented within pure λ-calculus. The proof is only sketched: By choosing the names of the bound variables appropriately,*

[4]The names β and ξ are standard notation.

the left-hand side becomes $\lambda z.\Big((\lambda f\lambda x.f^m x)r\Big)\Big(\Big((\lambda g\lambda y.g^n y)r\Big)z\Big)$. *By applying* (β) *twice, compatibility several times, and associativity once, we get the* β*-equal term* $\lambda z.(\lambda x.r^m x)\Big((\lambda y.r^n y)z\Big) =_\beta \lambda z.(\lambda x.r^m x)(r^n z) =_\beta \lambda z.r^m(r^n z) = \lambda z.r^{m+n}z =_\beta (\lambda f\lambda z.f^{m+n}z)r$. *Why is this a proof sketch? Because we tacitly use several properties which need to be proved by induction on natural numbers.*

Exercise 1 *Define* $\mathsf{K} := \lambda x\lambda y x$ *and* $\mathsf{S} := \lambda x\lambda y\lambda z.xz(yz)$ *with* x, y, z *different. (Recall that* $xz(yz) = ((xz)(yz))$*.) Produce some term* t *such that* $\mathsf{SKK} =_\beta t$ *and* t *has no subterm of the form* $(\lambda x r)s$ *(hence* t *will later be called* β*-normal).*

Exercise 2 *Show for every* m *and* n *that* $\underline{m} \circ \underline{n} =_\beta \underline{m \cdot n}$ *and for* $m \neq 0$ *and every* n*:* $\underline{m}\ \underline{n} =_\beta \underline{n^m}$*. Hence, also multiplication and exponentiation are represented within pure* λ*-calculus.*[5]

Exercise 3 (Predecessor) *(difficult) Define some closed term* P *representing the predecessor function: For every* $n \geq 1$, $\mathsf{P}\underline{n} =_\beta \underline{n-1}$ *and* $\mathsf{P}0 =_\beta 0$*. Show that* P *meets its specification.*

One possible solution[6] *is as follows: Numeral* $\underline{n}$ *applies its first argument* n *times to its second argument. Hence, iteration is already present in the system. Full primitive recursion may be derived from iteration by means of pairing. Pairs may be defined very easily (a term* $\langle r, s\rangle$ *and terms* $\pi_1 r$ *and* $\pi_2 r$ *for any terms* r *and* s *such that* $\pi_1\langle r, s\rangle =_\beta r$ *and* $\pi_2\langle r, s\rangle =_\beta s$*).*

Problem 1 *How can we argue that* $\lambda x\lambda y x \neq_\beta \lambda x\lambda y y$*? Clearly, we want them to be different since they represent procedures taking two arguments and returning the first and the second argument, respectively.*[7] *But how do we know that we cannot use (*β*) together with (s) in order to produce terms* $(\lambda x r)s$ *out of* $r[x := s]$ *in course of the hypothetical derivation of* $\lambda x\lambda y x =_\beta \lambda x\lambda y y$*?*

By leaving out the rules of equivalence relations, and by adjusting the application rule properly, we arrive at the definition of β-*reduction*.

Definition 5 (β-reduction) *Inductively define the relation* $\to_\beta$ *as follows:*

- *(*β*)* $(\lambda x r)s \to_\beta r[x := s]$ *(outer* β*-reduction).*
- *(*ξ*)* $r \to_\beta r' \Rightarrow \lambda x r \to_\beta \lambda x r'$ *(*β*-reduction under an abstraction).*
- *(r)* $r \to_\beta r' \Rightarrow rs \to_\beta r's$ *(right application).*
- *(l)* $r \to_\beta r' \Rightarrow sr \to_\beta sr'$ *(left application).*

[5] In fact, every partial recursive function can be modeled within λ-calculus by taking the Church numerals as numbers.

[6] There are much easier solutions and much more bizarre ones.

[7] If they were β-equal, all the terms would be β-equal.

If $r \to_\beta s$ we say that r reduces by one β-*reduction step* to s.

Example 4 *$\Omega = (\lambda x.xx)\omega \to_\beta (xx)[x := \omega] = \omega\omega = \Omega$ by rule (β). This* is *interesting since reflexivity has been excluded by passing from $=_\beta$ to $\to_\beta$. In fact, this example is a major nuisance which will later be removed by typing restrictions.*

Lemma 3 *$r \to_\beta r' \Rightarrow FV(r') \subseteq FV(r)$.*

Proof Induction on $\to_\beta$. $\square$

For every binary relation $\to$, the *transitive reflexive closure* of $\to$ (i.,e., the least transitive and reflexive relation containing $\to$) is denoted by $\to^*$. Equivalently, $r \to^* s$ iff there are $n \in \mathbb{N}_0$ and $r_0, r_1, \ldots, r_n$ such that $r = r_0$, $\forall i \in \{1, \ldots, n\}. r_{i-1} \to r_i$, and $r_n = s$.

Corollary 4 *$r \to_\beta^* r' \Rightarrow FV(r') \subseteq FV(r)$.* $\square$

It is easy to see that $\to_\beta^*$ has all the defining properties of $=_\beta$ except symmetry. But the absence of symmetry makes life much easier as will be the theme of the next section. Problem 1 will be solved.

4 Confluence

Although λ-calculus even enjoys confluence, a proof of local confluence is shown first.

Definition 6 (Local confluence) *A binary relation $\to \subseteq M \times M$ is locally confluent iff*

$$\forall r \in M \forall r' \in M \forall r'' \in M.\, r \to r' \land r \to r'' \Rightarrow \exists t \in M.\, r' \to^* t \land r'' \to^* t.$$

$$\begin{tikzcd} & r \arrow[dl] \arrow[dr] & \\ r' \arrow[dr, dashed, "*"] & \textit{local confluence} & r'' \arrow[dl, dashed, "*"'] \\ & t & \end{tikzcd}$$

Lemma 5 *$\to_\beta$ is locally confluent.*

Proof Let $r \to_\beta r'$ and $r \to_\beta r''$. By induction on r show that there is a term t such that $r' \to_\beta^* t$ and $r'' \to_\beta^* t$. We distinguish sixteen cases according to the four cases in the generation of $r \to_\beta r'$ and $r \to_\beta r''$, respectively. This will be indicated by a pair of names of rules taken from the definition of β-reduction.

β/β. The situation is trivial:

$$\begin{tikzcd}
& (\lambda x r)s \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
r[x := s] \arrow[dr, equal] & & r[x := s] \arrow[dl, equal] \\
& r[x := s] &
\end{tikzcd}$$

ξ/ξ. We have

$$\begin{tikzcd}
& \lambda x r \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
\lambda x r' & & \lambda x r''
\end{tikzcd}$$

due to

$$\begin{tikzcd}
& r \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
r' & & r''
\end{tikzcd}.$$

By induction hypothesis, we get a term t with

$$\begin{tikzcd}
r' \arrow[dr, "\beta^*"'] & & r'' \arrow[dl, "\beta^*"] \\
& t &
\end{tikzcd}.$$

Therefore, also

$$\begin{tikzcd}
\lambda x r' \arrow[dr, "\beta^*"'] & & \lambda x r'' \arrow[dl, "\beta^*"] \\
& \lambda x t &
\end{tikzcd}.$$

r/r and l/l. Similarly.

r/l. We have

$$\begin{tikzcd}
& rs \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
r's & & rs'
\end{tikzcd}$$

with $r \to_\beta r'$ and $s \to_\beta s'$.

Obviously,

$$\begin{tikzcd}
r's \arrow[dr, "\beta"'] & & rs' \arrow[dl, "\beta"] \\
& r's' &
\end{tikzcd}.$$

l/r. Symmetric to the preceding case.

r/β. We have

$$\begin{tikzcd}
& (\lambda x r)s \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
ts & & r[x := s]
\end{tikzcd}$$

with $\lambda x r \to_\beta t$.

Hence, $t = \lambda x r'$ with $r \to_\beta r'$. Since this implies $r[x := s] \to_\beta r'[x := s]$ (see the substitutivity lemma below),
we arrive at

$$\begin{tikzcd}
ts \arrow[d, equal] & & r[x := s] \arrow[ddl, "\beta"] \\
(\lambda x r')s \arrow[dr, "\beta"'] & & \\
& r'[x := s] &
\end{tikzcd}.$$

β/r. Symmetric to the preceding case.

l/β. We have
$$\begin{tikzcd}[column sep=small] & (\lambda x r)s \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\ (\lambda x r)s' & & r[x := s] \end{tikzcd}$$
with $s \to_\beta s'$.

By the compatibility lemma below, this implies $r[x := s] \to_\beta^* r[x := s']$ (with as many steps as free occurrences of x in r).

Hence,
$$\begin{tikzcd}[column sep=small] (\lambda x r)s' \arrow[dr, "\beta"] & & r[x := s] \arrow[dl, "*"', "\beta"] \\ & r[x := s'] & \end{tikzcd}$$

Notice that it is essential to put $\to_\beta^*$ instead of $\to_\beta$ in this diagram.

r/β. Symmetric to the preceding case.

ξ with other rules. Those six cases are impossible since the other rules need application terms. □

Two properties of $\to_\beta$ have been used in this proof which are now stated more prominently:

Lemma 6 (Substitutivity) *If* $r \to_\beta r'$ *then* $r[x := s] \to_\beta r'[x := s]$.

Proof Induction on $\to_\beta$. □

Lemma 7 (Compatibility) *If* $s \to_\beta s'$ *then* $r[x := s] \to_\beta^* r[x := s']$, *and* $r[x := s] \to_\beta r[x := s']$ *if* x *occurs exactly once free in* r.

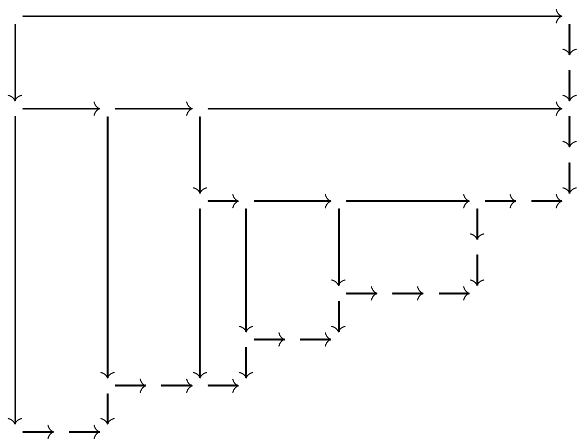
Proof Prove by induction on r that $r[x := s] \to_\beta^* r[x := s']$ with as many β-reduction steps as the number of free occurrences of x in r. □

Definition 7 (Confluence) *A binary relation* $\to \subseteq M \times M$ *is confluent iff*

$$\forall r \in M \forall r' \in M \forall r'' \in M.\, r \to^* r' \land r \to^* r'' \Rightarrow \exists t \in M.\, r' \to^* t \land r'' \to^* t.$$

$$\begin{tikzcd}[column sep=small] & r \arrow[dl, "*"'] \arrow[dr, "*"] & \\ r' \arrow[dr, dashed, "*"'] \arrow[rr, phantom, "\textit{confluence}"] & & r'' \arrow[dl, dashed, "*"] \\ & t & \end{tikzcd}$$

Problem 2 *Can we get confluence of* $\to$ *out of its local confluence? A direct proof fails:*



This could go on with ever increasing complexity. Moreover, there is a binary relation $\to$ which is locally confluent and not confluent: It is given by the following graph on four items: $\longleftarrow \;\rightleftarrows\; \longrightarrow$ *.*

Later we will see a condition for deriving confluence from local confluence.

Theorem 1 $\to_\beta$ *is confluent.*

The proof will occupy the rest of this section.

Definition 8 (Diamond Property) *A binary relation* $\to\; \subseteq \mathrm{M} \times \mathrm{M}$ *has the diamond property iff*

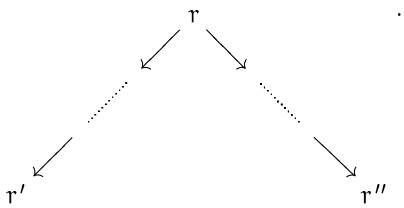
$$\forall r \in M \forall r' \in M \forall r'' \in M.\, r \to r' \wedge r \to r'' \Rightarrow \exists t \in M.\, r' \to t \wedge r'' \to t.$$

$$\begin{tikzcd}
 & r \arrow[dl] \arrow[dr] & \\
r' \arrow[dr, dashed] & \textit{diamond property} & r'' \arrow[dl, dashed] \\
 & t &
\end{tikzcd}$$

Notice that confluence is a derived concept: $\to$ is confluent iff $\to^*$ has the diamond property. Also note that the case l/β of the proof of local confluence immediately lets us find a counterexample to the diamond property for $\to_\beta$, e. g., by starting with $(\lambda x.yxx)((\lambda zr)s)$.

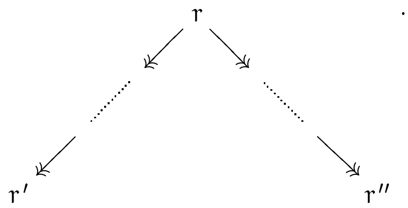
The idea to prove confluence of some $\to$ is as follows: Find a binary relation $\twoheadrightarrow$ (to be read as *parallel reduction*) such that $\to\; \subseteq\; \twoheadrightarrow\; \subseteq\; \to^*$ and $\twoheadrightarrow$ has the diamond property. Confluence of $\to$ follows easily: Assume
$$\begin{tikzcd}
 & r \arrow[dl, "*"'] \arrow[dr, "*"] & \\
r' & & r''
\end{tikzcd}$$
,

i. e.,



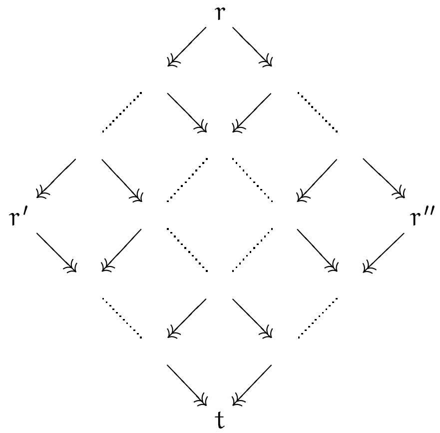
.

Since $\to \subseteq \twoheadrightarrow$, we have



.

Multiple uses of the diamond property for $\twoheadrightarrow$ give:



Hence, by $\twoheadrightarrow \subseteq \to^*$, $r' \to^* \ldots \to^* t$ and $r'' \to^* \ldots \to^* t$. Since $\to^*$ is transitive by definition, we finally get $r' \to^* t$ and $r'' \to^* t$. (Of course, this proof with dots could be made more precise by some inductive argument.)

How do we define such a notion $\twoheadrightarrow_\beta$ for $\to_\beta$? Reconsider the crucial case $1/\beta$ in the proof of local confluence. If we want to satisfy $\to_\beta \subseteq \twoheadrightarrow_\beta$ and the diamond property for $\twoheadrightarrow_\beta$, we have to solve

$$\begin{tikzcd}
 & (\lambda x r)s \arrow[dl, twoheadrightarrow, "\beta"'] \arrow[dr, twoheadrightarrow, "\beta"] & \\
(\lambda x r)s' \arrow[dr, twoheadrightarrow, "\beta"'] & & r[x := s] \arrow[dl, twoheadrightarrow, "\beta"] \\
 & ? &
\end{tikzcd}.$$

(Recall that $s \to_\beta s'$ is assumed in $1/\beta$.) Clearly, we want to have $r[x := s']$ as the common reduct. Therefore, our $\twoheadrightarrow_\beta$ must fulfill

$$s \to_\beta s' \Rightarrow r[x := s] \twoheadrightarrow_\beta r[x := s'].$$

This intuitively means that in $r[x := s] \twoheadrightarrow_\beta r[x := s']$, the β-reduction step from s to s' has to be carried out in parallel for each free occurrence of x in r. We will now try to find out which β-reductions on a term can be performed in one pass through it, and will later define $\twoheadrightarrow_\beta$ such that $r \twoheadrightarrow_\beta s$ iff s is the result when performing β-reductions on r in one pass through r. First we define the optimal result r^* of such an action on r.

Definition 9 (Complete superdevelopment) *By recursion on terms r define the term r^* as follows:*

$$\begin{aligned} x^* &:= x, \\ (\lambda x r)^* &:= \lambda x r^*, \\ (rs)^* &:= \begin{cases} t[x := s^*] & \text{if } r^* = \lambda x t, \\ r^* s^* & \text{otherwise.} \end{cases} \end{aligned}$$

In case of a variable, there is nothing to do. Everything we can do with an abstraction $\lambda x r$, takes place in its kernel r. Concerning an application rs, we first look what can be done with r and s. If the result r^* happens to be an abstraction, we even carry out the outer β-reduction step $(\lambda x t)s^* \to_\beta t[x := s^*]$, otherwise we simply apply r^* to s^*. (Further possibilities for β-reduction steps cannot be grasped uniformly by one pass through the term.)

Examples 5 *Consider $r := (\lambda x \lambda y.xyy)st$. In order to calculate r^* we need to know $((\lambda x \lambda y.xyy)s)^*$ which hin turn calls for $(\lambda x \lambda y.xyy)^*$. $(\lambda x \lambda y.xyy)^* = \lambda x \lambda y.xyy$ is plain. Therefore, $((\lambda x \lambda y.xyy)s)^* = (\lambda y.xyy)[x := s^*] = \lambda y.s^* yy$. Since we may assume that $y \notin FV(s)$, we finally get $r^* = (s^* yy)[y := t^*] = s^* t^* t^*$. We see that the complete superdevelopment is capable of replacing a list of formal parameters (here x and y) by the actual arguments (here the results s^* and t^*).*

The complete superdevelopment may also eliminate intervening identities, e. g., $((\lambda xx)(\lambda yr)s)^* = r^*[y := s^*]$. Notice that $(\lambda xx)(\lambda yr)s$ has hidden parentheses, as shown in $((\lambda xx)(\lambda yr))s$, preventing the β-reduction of $(\lambda yr)s$. Nevertheless, $(\lambda xx)(\lambda yr)s$ and $(\lambda yr)s$ have the same complete superdevelopment.

Finally, complete superdevelopments cannot remove every possibility for β-reduction, e. g., for $x \notin FV(s)$, we get $((\lambda x.xs)(\lambda yr))^* = (\lambda yr^*)s^*$ (use that $FV(s^*) \subseteq FV(s)$ shown below) which can further be β-reduced to $r^*[y := s^*]$ (which of course may again be β-reducible depending on r and s).

Lemma 8 *$r \to^* r^*$.*

Proof Induction on r. $\square$

As a corollary, we get $\mathrm{FV}(r^*) \subseteq \mathrm{FV}(r)$.

Now, we want to see how $\twoheadrightarrow_\beta$ should be defined to ensure that $r \twoheadrightarrow_\beta r^*$: Since $x^* = x$, we have to require $x \twoheadrightarrow_\beta x$. Concerning $(\lambda x r)^* = \lambda x r^*$, we will already know that $r \twoheadrightarrow_\beta r^*$ and have to show that $\lambda x r \twoheadrightarrow_\beta \lambda x r^*$. This suggests to require, more generally, $r \twoheadrightarrow_\beta r' \Rightarrow \lambda x r \twoheadrightarrow_\beta \lambda x r'$. Similarly—we now treat the application—in case $r^* = \lambda x t$, we will already know that $r \twoheadrightarrow_\beta \lambda x t$ and $s \twoheadrightarrow_\beta s^*$ and have to show that $rs \twoheadrightarrow_\beta t[x := s^*]$. This suggest to require

$r \twoheadrightarrow_\beta \lambda x t \wedge s \twoheadrightarrow_\beta s' \Rightarrow rs \twoheadrightarrow_\beta t[x := s']$. Finally, if we already know $r \twoheadrightarrow_\beta r^*$ and $s \twoheadrightarrow_\beta s^*$, we want to conclude $rs \twoheadrightarrow_\beta r^* s^*$, if r^* fails to be an abstraction. This suggests to require $r \twoheadrightarrow_\beta r' \wedge s \twoheadrightarrow_\beta s' \Rightarrow rs \twoheadrightarrow_\beta r's'$, which seems more reasonable if we also allow r' to be an abstraction. These four requirements will be the definition of $\twoheadrightarrow_\beta$.

Definition 10 (Parallel β-reduction) *Define the binary relation $\twoheadrightarrow_\beta$ inductively as follows:*

(β) $r \twoheadrightarrow_\beta \lambda x t \wedge s \twoheadrightarrow_\beta s' \Rightarrow rs \twoheadrightarrow_\beta t[x := s']$.

(ξ) $r \twoheadrightarrow_\beta r' \Rightarrow \lambda x r \twoheadrightarrow_\beta \lambda x r'$.

(a) $r \twoheadrightarrow_\beta r' \wedge s \twoheadrightarrow_\beta s' \Rightarrow rs \twoheadrightarrow_\beta r's'$.

(v) $x \twoheadrightarrow_\beta x$.

Notice that for given r, there may be several terms r' such that $r \twoheadrightarrow_\beta r'$, e. g., in case $r \twoheadrightarrow_\beta \lambda x t$ and $s \twoheadrightarrow_\beta s'$, we have that $rs \twoheadrightarrow_\beta t[x := s']$ by (β) and $rs \twoheadrightarrow_\beta (\lambda x t)s'$ by (a). (Recall that there are possibilities that those terms coincide: $t = xx$ and $s' = \omega$.)

Lemma 9 $r \twoheadrightarrow_\beta r^*$.

Proof By induction on r we verify that $\twoheadrightarrow_\beta$ indeed has the property which led us to the definition. □

Lemma 10 $\twoheadrightarrow_\beta$ *is reflexive.*

Proof By induction on r show $r \twoheadrightarrow_\beta r$. This does not need rule (β). □

Corollary 11 $\to_\beta \subseteq \twoheadrightarrow_\beta$.

Proof By induction on $\to_\beta$. Since $\to_\beta$ is the smallest set with its defining properties, we simply have to show those properties for $\twoheadrightarrow_\beta$, i. e., we have to show:

(β) $(\lambda x r)s \twoheadrightarrow_\beta r[x := s]$.

(ξ) $r \twoheadrightarrow_\beta r' \Rightarrow \lambda x r \twoheadrightarrow_\beta \lambda x r'$.

(r) $r \twoheadrightarrow_\beta r' \Rightarrow rs \twoheadrightarrow_\beta r's$.

(l) $r \twoheadrightarrow_\beta r' \Rightarrow sr \twoheadrightarrow_\beta sr'$.

This uses reflexivity of $\twoheadrightarrow_\beta$ at least four times. □

Lemma 12 $\twoheadrightarrow_\beta \subseteq \to_\beta^*$.

Proof By induction on $\twoheadrightarrow_\beta$. Since $\twoheadrightarrow_\beta$ is the smallest set with its defining properties, we simply have to show those properties for $\to_\beta^*$, i. e., we have to show:

(β) $r \to_\beta^* \lambda x t \wedge s \to_\beta^* s' \Rightarrow rs \to_\beta^* t[x := s']$.

(ξ) $r \to_\beta^* r' \Rightarrow \lambda x r \to_\beta^* \lambda x r'$.

(a) $r \to_\beta^* r' \wedge s \to_\beta^* s' \Rightarrow rs \to_\beta^* r's'$.

(v) $x \to_\beta^* x$.

They clearly hold. $\square$

The only remaining task is to prove the diamond property for $\twoheadrightarrow_\beta$. A crucial case (with $r \twoheadrightarrow_\beta r'$ and $s \twoheadrightarrow_\beta s'$) will be

$$\begin{tikzcd}
 & (\lambda x r)s \arrow[dl, two heads, "\beta"'] \arrow[dr, two heads, "\beta"] & \\
(\lambda x r')s' \arrow[dr, two heads, "\beta"'] & & r[x := s] \arrow[dl, two heads, "\beta"] \\
 & ? &
\end{tikzcd}$$

Clearly, we want to have $r'[x := s']$ as the common reduct. This calls for the following

Lemma 13 $r \twoheadrightarrow_\beta r' \wedge s \twoheadrightarrow_\beta s' \Rightarrow r[x := s] \twoheadrightarrow_\beta r'[x := s']$.

Proof Induction on $r \twoheadrightarrow_\beta r'$. We only consider the most technical case (β): Assume that $ru \twoheadrightarrow_\beta t[y := u']$ has been derived from $r \twoheadrightarrow_\beta \lambda y t$ and $u \twoheadrightarrow_\beta u'$. By the inductive hypothesis, $r[x := s] \twoheadrightarrow_\beta (\lambda y t)[x := s'] = \lambda y.t[x := s']$ (since we may assume that $y \notin \{x\} \cup FV(s')$), and also $u[x := s] \twoheadrightarrow_\beta u'[x := s']$. Hence,

$$r[x := s]u[x := s] \twoheadrightarrow_\beta t[x := s'][y := u'[x := s']] = t[y := u'][x := s'],$$

by Lemma 2. Therefore, $(ru)[x := s] \twoheadrightarrow_\beta t[y := u'][x := s']$, as required. $\square$

Note that this lemma comprises substitutivity and compatibility (cf. Lemma 6 and Lemma 7 for $\to_\beta$).

Lemma 14 (Maximality) *If* $r \twoheadrightarrow_\beta r'$ *then* $r' \twoheadrightarrow_\beta r^*$.

$$\begin{tikzcd}
 & r \arrow[dl, two heads, "\beta"'] \\
r' \arrow[dr, dashed, two heads, "\beta"'] & \textit{maximality of } (.)^* \\
 & r^*
\end{tikzcd}$$

Thus r^* is the optimum what can be done by β-reduction in one pass, and $\twoheadrightarrow_\beta$ gives all the possibilities one has in one pass (including doing nothing since $\twoheadrightarrow_\beta$ is reflexive), and if in the passage from r to r', the optimal way has not been chosen (hence $r' \neq r^*$), r^* can be reached in another pass through r'.

Proof Induction on $r \twoheadrightarrow_\beta r'$. We only look at the interesting cases.

(β). Case $rs \twoheadrightarrow_\beta t[x := s']$ thanks to $r \twoheadrightarrow_\beta \lambda xt$ and $s \twoheadrightarrow_\beta s'$. By induction hypothesis, $\lambda xt \twoheadrightarrow_\beta r^*$ and $s' \twoheadrightarrow_\beta s^*$. By inspection of the rules of $\twoheadrightarrow_\beta$, it is clear that $\lambda xt \twoheadrightarrow_\beta r^*$ can only be derived by rule (ξ). Therefore, r^* has to be a λ-abstraction $r^* = \lambda xt'$ and $t \twoheadrightarrow_\beta t'$. Hence, $(rs)^* = t'[x := s^*]$. By the preceding lemma, $t[x := s'] \twoheadrightarrow_\beta t'[x := s^*]$.

(α). By induction hypothesis, $r' \twoheadrightarrow_\beta r^*$ and $s' \twoheadrightarrow_\beta s^*$. We have to show that $r's' \twoheadrightarrow_\beta (rs)^*$. If $r^* = \lambda xt$ then, by rule (β), $r's' \twoheadrightarrow_\beta t[x := s^*] = (rs)^*$. Otherwise, by rule ($\alpha$), $r's' \twoheadrightarrow_\beta r^*s^* = (rs)^*$. □

Notice that Lemma 9 is merely the special case with $r' = r$, hence its proof is superfluous since we did not use it to prove maximality.

Corollary 15 *$\twoheadrightarrow_\beta$ has the diamond property.*

Proof Given r, we already know the term t which for any r' and r'' such that $r \twoheadrightarrow_\beta r'$ and $r \twoheadrightarrow_\beta r''$ fulfills $r' \twoheadrightarrow_\beta t$ and $r'' \twoheadrightarrow_\beta t$: It is r^*. □

To conclude, we have found a binary relation $\twoheadrightarrow_\beta$ with $\to_\beta \subseteq \twoheadrightarrow_\beta \subseteq \to_\beta^*$ and the diamond property. Hence, $\to_\beta$ is confluent.

Exercise 4 *Show that $\to_\beta$ has the Church-Rosser property: For every terms r and s, $r =_\beta s$ implies that there is some term t such that $r \to_\beta^* t$ and $s \to_\beta^* t$.*

Exercise 5 *Solve Problem 1.*

Exercise 6 *(a sequel to Exercise 2) Show that for $m \neq 0$ and any n, we have $\underline{m}\,\underline{n} \to_\beta^* \underline{n^m}$.*

Exercise 7 *A simple lemma says $\forall s \exists r. sr =_\beta r$. The idea: For given s, set $r := (\lambda x.s(xx))(\lambda x.s(xx))$ with $x \notin FV(s)$. Verify that $sr =_\beta s$. (Hence, r is a fixed point of s viewed as a function.)*

In the intended solution to Exercise 3, one needs some successor on Church numerals, hence some closed terms S such that $\forall n \in \mathbb{N}. \mathsf{S}\underline{n} =_\beta \underline{n+1}$. Show that $\mathsf{S} := \lambda z \lambda f \lambda x. f(zfx)$ is a possible choice.

By the little lemma, there is a term r such that $\mathsf{S}r =_\beta r$. The term r in the proof is no Church numeral, i. e., not of the form $\underline{n}$ with some $n \in \mathbb{N}$. Show that (this is no accident:) no Church numeral is fixed point of S.

5 Weak and Strong Normalization

Definition 11 *Given a binary relation $\to \subseteq M \times M$, an element $r \in M$ is* in normal form *iff there is no $s \in M$ such that $r \to s$.*

Lemma 16 (β-normal forms) *The set nf of terms in normal form w. r. t. $\to_\beta$ equals the set NF, defined by induction as follows:*

- *If $\vec{r} \subset \mathsf{NF}$ then $x\vec{r} \in \mathsf{NF}$.*

- *If* $r \in \mathsf{NF}$ *then* $\lambda x r \in \mathsf{NF}$.

The notation with vectors deserves a short explanation: $\vec{r}$ denotes a finite list of terms. This is not part of the syntax of λ-calculus but a metasyntactic device to communicate terms. In fact, we have infinitely many rules of the form: $r_1, \ldots, r_n \in \mathsf{NF} \Rightarrow x r_1 \ldots r_n \in \mathsf{NF}$ for every $n \in \mathbb{N}_0$.[8]

Proof $\mathsf{nf} \subseteq \mathsf{NF}$ is proved by induction on terms: $x \in \mathsf{NF}$. Let rs be in nf, hence also $r, s \in \mathsf{nf}$ since a reduction in r or s gives rise to a reduction in rs. By induction hypothesis $r, s \in \mathsf{NF}$. If r were of the form $\lambda x t$, then certainly $rs \notin \mathsf{nf}$. Hence, $r = x\vec{s}$ for $\vec{s} \subset \mathsf{NF}$. We conclude $rs = x\vec{s}s \in \mathsf{NF}$. Let $\lambda x r$ be in nf. Hence, also $r \in \mathsf{nf}$. By induction hypothesis, $r \in \mathsf{NF}$ which yields $\lambda x r \in \mathsf{NF}$.

$\mathsf{NF} \subseteq \mathsf{nf}$ is proved by induction on the definition of NF. Let $x\vec{r}$ be in NF thanks to $\vec{r} \subset \mathsf{NF}$. By induction hypothesis, $\vec{r} \subset \mathsf{nf}$, hence also $x\vec{r} \in \mathsf{nf}$ since any possible reduction of $x\vec{r}$ has to happen in one of the $\vec{r}$. Let $\lambda x r$ be in NF due to $r \in \mathsf{NF}$. By induction hypothesis, $r \in \mathsf{nf}$. Since reductions on $\lambda x r$ can only happen by help of the rule ξ, also $\lambda x r \in \mathsf{nf}$. $\square$

Examples 6 $\omega = \lambda x.xx \in \mathsf{NF}$. $\Omega = \omega\omega \notin \mathsf{NF}$.

Definition 12 *Given a binary relation* $\to \subseteq M \times M$, *an element* $r \in M$ *is* weakly normalizing *iff there is an* $s \in M$ *in normal form such that* $r \to^* s$.

Let wn be the set of terms which are weakly normalizing w.r.t. $\to_\beta$.

Examples 7 $\mathsf{nf} \subset \mathsf{wn}$. $\Omega \notin \mathsf{wn}$ *since the only term* t *such that* $\Omega \to_\beta^* t$ *is* Ω *itself.* $(\lambda x \lambda y.y)\Omega \in \mathsf{wn}$ *since its reduct* λyy *is in* nf.

Exercise 8 *Show that* $r \in \mathsf{wn}$ *and* $r \to_\beta^* r'$ *imply* $r' \in \mathsf{wn}$.

Definition 13 *Given a binary relation* $\to \subseteq M \times M$, *an element* $r \in M$ *is* strongly normalizing *iff there is no infinite reduction sequence starting from* r, *i. e., iff there are no* $r_0, r_1, \ldots$ *such that* $r_0 = r$ *and* $r_i \to r_{i+1}$ *for every* i.

This definition has a major drawback: It is formulated as a negative statement, namely the non-existence of an infinite reduction sequence. This can be avoided and even the reference to infinity be removed in the following inductive characterization.

Definition 14 *Given a binary relation* $\to \subseteq M \times M$, *its* accessible part acc *is inductively defined by:*

$$\forall r \in M.(\forall s \in M.r \to s \Rightarrow s \in \mathsf{acc}) \Rightarrow r \in \mathsf{acc}.$$

In other words: An object $r \in M$ qualifies for acc if every one-step reduct s (any object $s \in M$ such that $r \to s$) already qualified for acc.

Clearly, if r is in normal form, then there is nothing to check, and r enters acc immediately.

[8]The infinity could be avoided by introducing the concept of neutral term simultaneously with NF: Variables are neutral, and if r is neutral and $s \in \mathsf{NF}$, then rs neutral. Moreover, all neutral terms are in NF.

Lemma 17 *The set* acc *is the set of strongly normalizing elements of* M.

Proof How can we prove that acc only contains strongly normalizing objects? By induction on the inductive definition of acc! Let r be in acc thanks to $s \in$ acc for every s such that $r \to s$. Assume, we had an infinite reduction sequence starting from r, i. e., some $s_{-1}, s_0, s_1, \dots$ such that $r = s_{-1}$ and for all $i \geq -1$: $s_i \to s_{i+1}$. Setting $s := s_0$, we have $r \to s$, and an infinite reduction sequence starting from s. This contradicts the induction hypothesis which tells us that s is already strongly normalizing (since s has entered acc before r entered it).

We now show that an $r \in M \setminus \mathsf{acc}$ is not strongly normalizing by constructing $r_0, r_1, \dots$ such that $r_0 = r$ and for all i, $r_i \to r_{i+1}$ and $r_i \in M \setminus \mathsf{acc}$. Set $r_0 := r$. Assume that $r_0, \dots, r_i$ have already been constructed. Since $r_i \notin \mathsf{acc}$, it did not qualify for acc, hence there has to be some one-step reduct r_{i+1} of r_i which also did not qualify for acc, i. e., some $r_{i+1} \in M$ such that $r_i \to r_{i+1}$ and $r_{i+1} \notin \mathsf{acc}$. □

Note that the second part of the proof uses some form of the axiom of choice and is not constructively justified.[9]

Define $\mathsf{sn} := \mathsf{acc}_{\to_\beta}$. Hence, sn is the set of λ-terms which do not have an infinite sequence of β-reductions.

Example 8 $(\lambda x \lambda y.y)\Omega \notin \mathsf{sn}$ *since we get a constant infinite reduction sequence by* β*-reductions of* Ω.

Exercise 9 *Show that for a strongly normalizing binary relation* $\to$*, i. e., every object is strongly normalizing w. r. t.* $\to$*, local confluence implies confluence.*

Hint: By induction on r *being in the accessible part of* $\to$ *show that every diagram for confluence originating in* r *may be closed.*

Lemma 18 (Characterization of the strongly normalizing terms) *The set* sn *equals the set* SN*, defined by induction as follows:*

- *If* $\vec{r} \subset \mathsf{SN}$ *then* $x\vec{r} \in \mathsf{SN}$.
- *If* $r \in \mathsf{SN}$ *then* $\lambda x r \in \mathsf{SN}$.
- *If* $r[x := s]\vec{s} \in \mathsf{SN}$ *and* $s \in \mathsf{SN}$ *then* $(\lambda x r)s\vec{s} \in \mathsf{SN}$.

Proof The more important part is proving that $\mathsf{SN} \subseteq \mathsf{sn}$ which may also be called the soundness of SN w. r. t. strong normalization of $\to_\beta$ (every element of SN is indeed strongly normalizing). Clearly, this has to be done by induction on the inductive definition of SN. We will profit from a deeper understanding of induction: Proving by induction on an inductively defined set I, that every inhabitant of I has some property expressed by a set M, amounts to showing that this set M has each of the defining properties of I (where—of course—the

[9] A constructivist would simply say that the accessible part is the notion of strong normalization and would probably never encounter any necessity for the exclusion of infinite reduction sequences.

reference to I has to be replaced by M). Why is that so? The very idea of an inductive definition of I is that the rules describing its definition give all the possibilities. No element may enter I without the application of one of the rules. Hence, I becomes the smallest set with those closure properties, and if M is an arbitrary set fulfilling the closure properties, then $I \subseteq M$, which was our goal to prove.

Therefore, it suffices to prove

- If $\vec{r} \subset \mathsf{sn}$ then $x\vec{r} \in \mathsf{sn}$.
- If $r \in \mathsf{sn}$ then $\lambda x r \in \mathsf{sn}$.
- If $r[x := s]\vec{s} \in \mathsf{sn}$ and $s \in \mathsf{sn}$ then $(\lambda x r)s\vec{s} \in \mathsf{sn}$.

We first give a proof which does not argue by induction on the accessible part but by the non-existence of infinite reduction sequences. Imagine an infinite reduction sequence starting from $x\vec{r}$. In every step there has to be a reduction in one of the $\vec{r}$. Since there are only finitely many terms in $\vec{r}$, there has to be an r_i in $\vec{r}$ which faces infinitely many reduction steps. Contradiction. Assume an infinite reduction sequence with first term $\lambda x r$. Since the reductions can only take place in r, we get an infinite reduction sequence starting from r. Contradiction. Imagine an infinite reduction sequence starting from $(\lambda x r)s\vec{s}$. Since $r \to_\beta r'$ implies $r[x := s] \to_\beta r'[x := s]$, and consequently $r[x := s]\vec{s} \to_\beta r'[x := s]\vec{s}$, there can neither be an infinite reduction sequence starting from r nor from s nor from any of the $\vec{s}$. Hence, there has to be a reduct of $(\lambda x r)s\vec{s}$ in the infinite reduction sequence which does no longer have the shape $(\lambda x r')s'\vec{s}\,'$ with $r \to_\beta^* r'$, $s \to_\beta^* s'$, $s_i \to_\beta^* s_i'$. Assume this were the last successive term of this shape. The next term then has to be $r'[x := s']\vec{s}\,'$, still followed by an infinite reduction sequence. From Lemma 6 and Lemma 7, we get $r[x := s]\vec{s} \to_\beta^* r'[x := s']\vec{s}\,'$. Therefore, we also get an infinite reduction sequence starting with $r[x := s]\vec{s}$. Contradiction.

Now we redo the proof for the last clause and completely avoid Lemma 17: Show that $(\lambda x r)s\vec{s} \in \mathsf{sn}$ by main induction on $s \in \mathsf{sn}$ and side induction on $r[x := s]\vec{s} \in \mathsf{sn}$. Hence, prove $t \in \mathsf{sn}$ for every t such that $(\lambda x r)s\vec{s} \to_\beta t$. The following reductions are possible.

$(\lambda x r)s\vec{s} \to_\beta (\lambda x r')s\vec{s}$. Then $r[x := s]\vec{s} \to_\beta r'[x := s]\vec{s}$ by substitutivity, hence by side induction hypothesis $(\lambda x r')s\vec{s} \in \mathsf{sn}$.

$(\lambda x r)s\vec{s} \to_\beta (\lambda x r)s'\vec{s}$. Then $r[x := s]\vec{s} \to_\beta^* r[x := s']\vec{s}$ by compatibility, hence also $r[x := s']\vec{s} \in \mathsf{sn}$. The main induction hypothesis yields $(\lambda x r)s'\vec{s} \in \mathsf{sn}$.

$(\lambda x r)s\vec{s} \to_\beta (\lambda x r)s\vec{s}\,'$. Then $r[x := s]\vec{s} \to_\beta r[x := s]\vec{s}\,'$, hence by side induction hypothesis $(\lambda x r)s\vec{s}\,' \in \mathsf{sn}$.

$(\lambda x r)s\vec{s} \to_\beta r[x := s]\vec{s} \in \mathsf{sn}$ by assumption.

In order to clarify the intricate structure of the argument we prove this once again and spell out the nesting of inductions precisely. Main induction on $s \in \mathsf{sn}$

amounts to showing the defining property of sn for the set

$$M := \{s \mid \forall r, \vec{s}. r[x := s]\vec{s} \in \mathsf{sn} \;\Rightarrow\; (\lambda x r) s \vec{s} \in \mathsf{sn}\}.$$

So assume that any one-step reduct of s is in M. For $s \in M$ we do side induction on $r[x := s]\vec{s} \in \mathsf{sn}$, i.e., we show the defining property of sn for the set

$$N := \{t' \mid t' \in \mathsf{sn} \wedge \forall r, \vec{s}. t' = r[x := s]\vec{s} \Rightarrow (\lambda x r) s \vec{s} \in \mathsf{sn}\}.$$

So assume t' is a term and each immediate reduct is in N. We have to show $t' \in N$. $t' \in \mathsf{sn}$ follows from the definition of sn, since $N \subseteq \mathsf{sn}$. If t' has the form $r[x := s]\vec{s}$ we have to show $(\lambda x r) s \vec{s} \in \mathsf{sn}$, so we prove $t \in \mathsf{sn}$ for every one-step reduct t. Henceforward, each clause of the proof in the above proof can be recast using the pending assumptions.

For the other direction $\mathsf{sn} \subseteq \mathsf{SN}$ (also called completeness of SN), we do induction on sn. Hence, we have to show $\forall r. (\forall r'. r \to_\beta r' \Rightarrow r' \in \mathsf{SN}) \Rightarrow r \in \mathsf{SN}$. This is done by induction on the term r. For this to work, note that every term has exactly one of the following shapes (to be proved by induction on terms): $x\vec{r}, \lambda x r, (\lambda x r) s \vec{s}$. And because we do induction on terms, we may use the induction hypothesis for r_i in the case $x\vec{r}$, for r in the case $\lambda x r$ (no surprise!) and for s in the case $(\lambda x r) s \vec{s}$. The rest is routine verification. (Another proof would first show that strongly normalizing terms are also strongly normalizing when the binary relation $\to_\beta$ is extended by $\rhd$ defined by $r \rhd s$ iff s is a subterm of r. Then the result follows by induction on $\mathsf{acc}_{\to_\beta \cup \rhd}$.) □

Exercise 10 *(another proof with induction on natural numbers) Let* $\mathsf{sn}(k)$ *be the set of terms where all* β*-reduction sequences have at most length* k*. We start counting such that* $\mathsf{sn}(0)$ *is the set of* β*-normal terms. Show that* $\mathsf{SN} \subseteq \bigcup_{k \in \mathbb{N}} \mathsf{sn}(k)$*. This shall be done constructively. Clearly by induction, but as follows: Define* n*-place functions* f_n *with* $n \in \mathbb{N}$*, a one-place function* g *and a two-place function* h *such that*

- *If* $r_i \in \mathsf{sn}(k_i)$ *for* $i \in \{1, \ldots, n\}$ *then* $x\vec{r} \in \mathsf{sn}(f_n(k_1, \ldots, k_n))$.
- *If* $r \in \mathsf{sn}(k)$ *then* $\lambda x r \in \mathsf{sn}(g(k))$.
- *If* $r[x := s]\vec{s} \in \mathsf{sn}(k)$ *and* $s \in \mathsf{sn}(\ell)$ *then* $(\lambda x r) s \vec{s} \in \mathsf{sn}(h(k, \ell))$.

Prove these properties.

As a final remark: SN is syntax-directed in the sense that from the shape of every term r it can be read off which single rule of SN could prove that r belongs to SN. This may be exploited for the following naive normalization algorithm working for every term $r \in \mathsf{SN}$ and defined by recursion on r:

$$\begin{aligned} \mathsf{nf}(x\vec{r}) &:= x\, \mathsf{nf}(r_1) \ldots \mathsf{nf}(r_n) \\ \mathsf{nf}(\lambda x r) &:= \lambda x\, \mathsf{nf}(r) \\ \mathsf{nf}((\lambda x r) s \vec{s}) &:= \mathsf{nf}(r[x := s]\vec{s}) \end{aligned}$$

Clearly, we do not make use of $s \in \mathsf{SN}$ in the last clause. Therefore, the algorithm would also work if this requirement were dropped in the definition of SN. The following exercise shows that one even gets an inductive characterization of the weakly normalizing terms wn by this modification.

Exercise 11 *Inductively define the set* WN*:*

- *If* $\vec{r} \subset \mathsf{WN}$ *then* $x\vec{r} \in \mathsf{WN}$.
- *If* $r \in \mathsf{WN}$ *then* $\lambda x r \in \mathsf{WN}$.
- *If* $r[x := s]\vec{s} \in \mathsf{WN}$ *then* $(\lambda x r)s\vec{s} \in \mathsf{WN}$.

As mentioned above, the only difference with the definition of SN *is the omission of "*$s \in \mathsf{SN}$*" in the last clause.*

Define the binary relation $\rightsquigarrow$ *on terms inductively (the vector notation is understood elementwise, implicitly assuming that both vectors are of the same length)*

- $\vec{r} \rightsquigarrow \vec{r}\,' \Rightarrow x\vec{r} \rightsquigarrow x\vec{r}\,'$
- $r \rightsquigarrow r' \wedge \vec{s} \rightsquigarrow \vec{s}\,' \Rightarrow (\lambda x r)\vec{s} \rightsquigarrow (\lambda x r')\vec{s}\,'$
- $r[x := s]\vec{s} \rightsquigarrow t \Rightarrow (\lambda x r)s\vec{s} \rightsquigarrow t$

$\rightsquigarrow$ *is reflexive (3rd rule is not needed).*
Show ***Lemma 1****:* $r \rightsquigarrow r' \wedge s \rightsquigarrow s' \Rightarrow rs \rightsquigarrow r's'$.

Hint: Induction on the derivation of $r \rightsquigarrow r'$.

Show ***Lemma 2****:* $\to_\beta \subseteq \rightsquigarrow \subseteq \to_\beta^*$.

Hint: For every inclusion do induction on the derivation of the relation to be proved smaller than the other.

Show ***Lemma 3****:* $r \rightsquigarrow r' \wedge s \rightsquigarrow s' \Rightarrow r[x := s] \rightsquigarrow r'[x := s']$.

Hint: Induction on the derivation of $r \rightsquigarrow r'$.

Show ***Lemma 4****:* $r \rightsquigarrow r' \to_\beta r'' \Rightarrow r \rightsquigarrow r''$.

Hint: Induction on the derivation of $r \rightsquigarrow r'$ *and analysis which possibilities arise for* $r' \to_\beta r''$*. This is the major work of this exercise and needs Lemma 1 and Lemma 3.*

Show the ***Corollary****:* $r \to_\beta^* r' \Rightarrow r \rightsquigarrow r'$.

Remark: Thus we have learnt $\to_\beta^* = \rightsquigarrow$*, hence a new induction principle for* $\to_\beta^*$*, namely induction on the derivation of* $\rightsquigarrow$*. Up to now we only had the opportunity to argue by induction on the number of steps in* $r \to_\beta^* r'$.

Define $\rightsquigarrow^-$ *by omitting* $\vec{s}$ *from the second clause of the definition of* $\rightsquigarrow$ *(hence* $\rightsquigarrow^- \subseteq \rightsquigarrow$*). The inductive definition reads:*

- $\vec{r} \rightsquigarrow^- \vec{r}\,' \Rightarrow x\vec{r} \rightsquigarrow^- x\vec{r}\,'$
- $r \rightsquigarrow^- r' \Rightarrow \lambda x r \rightsquigarrow^- \lambda x r'$
- $r[x := s]\vec{s} \rightsquigarrow^- t \Rightarrow (\lambda x r)s\vec{s} \rightsquigarrow^- t$

Show ***Lemma 5****:* $r \rightsquigarrow^- r' \Rightarrow r \in \mathsf{WN} \wedge r' \in \mathsf{NF}$.

Hint: Induction on the derivation of $r \rightsquigarrow^- r'$.

Show ***Lemma 6****:* $r \in \mathsf{WN} \Rightarrow \exists r'.r \rightsquigarrow^- r'$.
Show ***Lemma 7****:* $r \rightsquigarrow r' \in \mathsf{NF} \Rightarrow r \rightsquigarrow^- r'$.

Hint: Induction on $\rightsquigarrow$*. (We even see that the derivation of* $r \rightsquigarrow r'$ *only uses rules which are also present in* $\rightsquigarrow^-$*, hence there is no need for a transformation of the derivation.)*

Theorem WN *is the set* wn *of weakly normalizing terms.*

Proof: "'WN $\subseteq$ wn"' By L. 6 we have $r \rightsquigarrow^- r'$ for some r'. By L. 5, $r' \in$ NF. Since $\rightsquigarrow^- \subseteq \to_\beta^*$, we conclude $r \in$ wn.

"'wn $\subseteq$ WN"' Let $r \to_\beta^* r' \in$ NF. From the Corollary we get $r \rightsquigarrow r' \in$ NF. L. 7 shows $r \rightsquigarrow^- r'$, and finally, by L. 5, $r \in$ WN.

Final question: Which of the two parts of the statement needs the witty approach taken in this exercise? (The Corollary is called the Standardization Theorem, its proof followed [Loa98].)

6 Simple and Intersection Types

We have seen that there are terms which are not strongly normalizing and even terms which fail to be weakly normalizing. The idea was to use the unintuitive term $\omega = \lambda x.xx$ and apply it to itself. The first examples in section 3.1 dealing with derivatives of functions on the reals were modeled without using the information on the domains and ranges of the functions at hand. E. g., we never expressed that we considered the squaring function

$$f : \begin{cases} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & x^2 \end{cases}$$

This more specific information may be expressed by $\lambda x.x^2 : \mathbb{R} \to \mathbb{R}$, meaning that $\lambda x.x^2$ is considered as a function from the reals to the reals. That our $\lambda x.x^2$ belongs to the function space $\mathbb{R} \to \mathbb{R}$ is called type information. If we again ignore the fact that limits do not always exist, we might type $\lim_{\to 0}$ with $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$, reflecting that $\lim_{\to 0}$ takes a function from $\mathbb{R}$ to $\mathbb{R}$ and returns a real. What would be the type of ∂_2? Again by ignoring undefinedness problems, we would give it the type $(\mathbb{R} \to \mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R}$: The first argument is a two-place real-valued function on $\mathbb{R}$, the second and third arguments are reals again, and the result is again in $\mathbb{R}$.

Our semantic intuition is a world of functionals, i. e., functions taking functions and even functionals as arguments. In that *simply-typed* world, there is no place for $\omega = \lambda x.xx$. Since ω is already in normal form, it is very well-behaved w. r. t. β-reduction. So, in some sense, we have to give up too much when restricting to the simply-typed world.

If we loosen the typing concept to include *intersection types* invented in [CDC78], we also cover ω: Simply allow intersections of already formed types. If some x "lives" in $\mathbb{N} \to \mathbb{N}$ as well as in $\mathbb{N}$, there is no hesitation to assert that xx "lives" in $\mathbb{N}$. Hence, we would type ω with $((\mathbb{N} \to \mathbb{N}) \cap \mathbb{N}) \to \mathbb{N}$. Are there objects which can been seen as numbers and as number-theoretic functions? Not in the set-theoretic model we were appealing to, but in recursion-theoretic models (where recursive functions are coded by numbers). But this is not the point. We only need the soundness of the implications $x : (\mathbb{N} \to \mathbb{N}) \cap \mathbb{N} \Rightarrow x : \mathbb{N} \to \mathbb{N} \wedge x : \mathbb{N}$ and $x : \mathbb{N} \to \mathbb{N} \wedge x : \mathbb{N} \Rightarrow xx : \mathbb{N}$ to conclude a correct type for ω.

6.1 Simply-Typed Lambda Calculus

Assume a set $\mathcal{V}_T$ of identifiers.[10] These serve as names for atomic types. Their generic name will be ι. Typical examples would be nat and real, representing $\mathbb{N}$ and $\mathbb{R}$ by mere syntax.

Definition 15 (Simple types) *The set $\mathcal{T}_s$ of simple types is inductively given by:*

- *If $\iota \in \mathcal{V}_T$ then $\iota \in \mathcal{T}_s$.*
- *If $\rho \in \mathcal{T}_s$ and $\sigma \in \mathcal{T}_s$ then $(\rho \to \sigma) \in \mathcal{T}_s$.*

Hence, the simple types are nothing but strings of symbols built up from the given symbols in $\mathcal{V}_T$ by help of the binary $\to$, syntactically representing the construction of function spaces. As for the terms, we avoid parentheses as much as possible, and therefore assume $\to$ to be right-associative (application was meant to be left-associative).

Examples 9 $\rho \to \sigma$ *is* $(\rho \to \sigma)$. $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ *is* $(\mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat}))$. *The second example represents the space of number-theoretic functions of two arguments and explains why $\to$ has been declared right-associative. More complicated things need parentheses:* $(\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat}$ *represents the functionals taking number-theoretic functions and returning values in $\mathbb{N}$. Here, we only suppressed the outer parentheses.*

As a further abbreviation, set $\rho_1, \ldots, \rho_n \to \sigma := \rho_1 \to \ldots \to \rho_n \to \sigma$ and also use this with $\vec{\rho} := \rho_1, \ldots, \rho_n$, hence $\vec{\rho} \to \sigma$ is a well-defined type. (For $n = 0$ we set $\emptyset \to \sigma := \sigma$.) By induction on types, it is easy to verify that every type uniquely decomposes into $\vec{\rho} \to \iota$ for some $\iota \in \mathcal{V}_T$ and arbitrary types $\vec{\rho}$.

Definition 16 (Simple typing) *The relation $\Gamma \vdash r : \rho$ (term r has type ρ in context Γ) is inductively defined by the following rules:*

$$\frac{}{\Gamma, x : \rho \vdash x : \rho}(V) \qquad \frac{\Gamma, x : \rho \vdash r : \sigma}{\Gamma \vdash \lambda x r : \rho \to \sigma}(\to_I) \qquad \frac{\Gamma \vdash r : \rho \to \sigma \qquad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}(\to_E)$$

From these clauses, it is clear that only finite lists of variable names with types (each pair separated by a colon) may occur on the left-hand side of $\vdash$. We moreover restrict those contexts *to lists with pairwise disjoint variable names. Hence, by writing $\Gamma, x : \rho$, it is implicit that x does not occur in Γ, written $x \notin \Gamma$. We also syntactically identify contexts which are permutations of each other. This amounts to using the following exchange rule without including it into the typing system: If Δ is a permutation of the pairs in Γ and $\Gamma \vdash r : \rho$ then $\Delta \vdash r : \rho$ (Exchange 1).*

[10] Generally, we assume that $\mathcal{V}_T$ and $\mathcal{V}$ are disjoint sets, and even that terms and types are always disjoint—in contrast to the systems of dependent types not covered by this course.

Note that our notation always suggests that we have lists on the left-hand side of $\vdash$. In fact, we consider the contexts to be sets of variable declarations without inconsistencies arising from multiple declarations for some variable. The empty context will be denoted by $\emptyset$, hence $\vdash r : \rho$ and $\emptyset \vdash r : \rho$ mean the same assertion.

Some comments on the rules: Rule (V) is nothing but lookup in the context, rule $(\to_I)$, the $\to$-introduction rule, follows our intuition: If r gets type σ under the assumption that the possibly free variable x is of type ρ, then the abstraction $\lambda x r$ gets the function type $\rho \to \sigma$. The $\to$-elimination rule $(\to_E)$ requires that the argument s to r of type $\rho \to \sigma$ has to have the domain type ρ, and states that the result of the application has the range type σ.

Lemma 19 *There are no Γ and ρ such that $\Gamma \vdash \omega : \rho$.*

Proof If there were Γ and ρ, then the rule $(\to_I)$ would have been applied last. Hence, we would have $\rho = \rho_1 \to \rho_2$ and $\Gamma, x : \rho_1 \vdash xx : \rho_2$. This can only be derived by help of $(\to_E)$. Hence, there is a type σ such that $\Gamma, x : \rho_1 \vdash x : \sigma \to \rho_2$ and $\Gamma, x : \rho_1 \vdash x : \sigma$. These can only be found by applying (V), hence $\sigma \to \rho_2 = \rho_1 = \sigma$. Contradiction (the equality is the equality of strings). $\square$

Exercise 12 *Let ρ be any type. Show that for the n-th Curch numeral $\underline{n}$, we have $\vdash \underline{n} : (\rho \to \rho) \to \rho \to \rho$. Which types are possible for $\underline{m} \circ \underline{n} := \lambda x.\underline{m}(\underline{n}x)$ and $\underline{m}\ \underline{n}$ in the empty context?*

Exercise 13 *Show that every term in NF receiving type $(\iota \to \iota) \to \iota \to \iota$ (according to our convention, ι is an atomic type) in the empty context is either a Church numeral or $\lambda f f$. (What is the difference between $\underline{1}$ and $\lambda f f$?)*

Exercise 14 *Show that for typable terms in normal form we can always reconstruct the types ρ which have been used in the abstraction rule, i. e., in the rule $\Gamma, x : \rho \vdash r : \sigma \Rightarrow \Gamma \vdash \lambda x r : \rho \to \sigma$. Show by example that this is not the case for arbitrary typable terms.*

Exercise 15 *We want to prove weak normalization of simply-typed terms with nearly no overhead. Therefore we abandon the typing assignments and simply assume that every term we consider is simply-typed. More precisely, we inductively define the simply-typed terms with their types as follows:*

- *If x is a variable and ρ is a type, then x^ρ is a term of type ρ.*
- *If x is a variable, ρ is a type and r is a term of type σ then $\lambda x^\rho r$ is a term of type $\rho \to \sigma$.*
- *If r is a term of type $\rho \to \sigma$ and s is a term of type ρ then rs is a term of type σ.*

r *is a term of type* ρ *will be expressed by* $r:\rho$ *or* r^ρ, *e. g.,* $\lambda f^{\rho\to\rho}\lambda x^\rho.f(f(fx)) : (\rho\to\rho)\to\rho\to\rho$. *For simplification of notation, the types of the bound variables are not written if there is no ambiguity. Of course, e. g.,* $\lambda x^\rho\lambda x^\sigma.x$ *is ambiguous for* $\rho\neq\sigma$ *and therefore illegal. But even after disambiguation, those terms should not be used in explicit considerations.*

The set NF *now has to obey the additional proviso of well-formedness of* $x\vec{r}$. *This will not be made explicit.*

Define $r\downarrow :\Leftrightarrow \exists t\in\mathsf{NF}.r\to^*_\beta t$. *In principle* $\{r\mid r\downarrow\}$ *equals* wn, *but* wn *is made up of untyped terms.*

(a) Show the following lemma: If $r\in\mathsf{NF}$ *and* $s^\rho\in\mathsf{NF}$, *then (i)* $rs\downarrow$ *if* rs *is a typed term, and (i)* $r[x^\rho:=s]\downarrow$.

Hint: Main induction on the type ρ, *side induction on* $r\in\mathsf{NF}$. *A somewhat more involved proof first shows only (ii) (and uses* $r^{\rho\to\sigma}\in\mathsf{NF}\Rightarrow rx^\rho\downarrow$ *to be proved beforehand) and infers (i).*

(b) Show that every typed term r *satisfies* $r\downarrow$.

6.2 Lambda Calculus with Intersection Types

Definition 17 (Intersection types) *The set* $\mathcal{T}_i$ *of intersection types is inductively given by:*

- *If* $\iota\in\mathcal{V}_T$ *then* $\iota\in\mathcal{T}_i$.
- *If* $\rho\in\mathcal{T}_i$ *and* $\sigma\in\mathcal{T}_i$ *then* $(\rho\to\sigma)\in\mathcal{T}_i$.
- *If* $\rho\in\mathcal{T}_i$ *and* $\sigma\in\mathcal{T}_i$ *then* $(\rho\cap\sigma)\in\mathcal{T}_i$.

Since we have more types, i. e., $\mathcal{T}_s\subset\mathcal{T}_i$, we also have additional rules for typing. Nevertheless, we will use the same symbol $\vdash$ for both simple typing and intersection typing.

Definition 18 (Intersection typing) *The relation* $\Gamma\vdash r:\rho$ *(term* r *has type* ρ *in context* Γ*) is inductively defined by the following rules:*

$$\frac{}{\Gamma,x:\rho\vdash x:\rho}(V)\qquad \frac{\Gamma,x:\rho\vdash r:\sigma}{\Gamma\vdash\lambda xr:\rho\to\sigma}(\to_I)\qquad \frac{\Gamma\vdash r:\rho\to\sigma\qquad \Gamma\vdash s:\rho}{\Gamma\vdash rs:\sigma}(\to_E)$$

$$\frac{\Gamma\vdash r:\rho\qquad \Gamma\vdash r:\sigma}{\Gamma\vdash r:\rho\cap\sigma}(\cap_I)\qquad \frac{\Gamma\vdash r:\rho_1\cap\rho_2\qquad i\in\{1,2\}}{\Gamma\vdash r:\rho_i}(\cap_E)$$

The conventions concerning contexts are as for simple typing.

Note that the rules of $\cap$-introduction $(\cap_I)$ and $\cap$-elimination $(\cap_E)$ capture the intuition that the intersection type construct models intersection.

Examples 10 $x:(\rho\to\rho)\cap\rho\vdash x:\rho\to\rho$ *and* $x:(\rho\to\rho)\cap\rho\vdash x:\rho$, *hence* $x:(\rho\to\rho)\cap\rho\vdash xx:\rho$ *and finally* $\vdash\omega:((\rho\to\rho)\cap\rho)\to\rho$. *In Lemma 24, we will see that for no* Γ *and* ρ, $\Gamma\vdash\Omega:\rho$ *holds, i. e.,* Ω *is not typable with intersection types.*

Lemma 20 (Derived rules)

- *If* $\Gamma \vdash r : \rho$ *then* $\Gamma, x : \sigma \vdash r : \rho$ *(Weakening 1).*
- *If* $\Gamma, x : \sigma \vdash r : \rho$ *then* $\Gamma, x : \sigma \cap \tau \vdash r : \rho$ *(Weakening 2).*
- *If* $\Gamma, x : \sigma \vdash r : \rho$ *and* $x \notin FV(r)$ *then* $\Gamma \vdash r : \rho$ *(Strengthening).*
- *If* $\Gamma, x : \rho \cap \sigma \vdash r : \tau$ *then* $\Gamma, x : \sigma \cap \rho \vdash r : \tau$ *(Exchange 2).*
- *If* $\Gamma, x : (\rho_1 \cap \rho_2) \cap \rho_3 \vdash r : \tau$ *then* $\Gamma, x : \rho_1 \cap (\rho_2 \cap \rho_3) \vdash r : \tau$ *(Exchange 3).*

Proof By rule induction on $\vdash$. □

Lemma 21 (Substitution)

(a) *If* $\Gamma, x : \rho \vdash r : \tau$ *and* $\Gamma \vdash s : \rho$ *then* $\Gamma \vdash r[x := s] : \tau$.

(b) *If* $\Gamma \vdash r[x := s] : \tau$, $x \notin FV(s) \cup \Gamma$ *and* $(x \notin FV(r) \Rightarrow \Gamma \vdash s : \rho_0)$ *then there is a type* ρ *such that* $\Gamma, x : \rho \vdash r : \tau$ *and* $\Gamma \vdash s : \rho$.

Proof (a) has a completely straightforward proof by induction on the derivation of $\Gamma, x : \rho \vdash r : \tau$ and does not reveal anything specific to intersection types. (b) is typical of intersection types (compare with Proposition 3 in [Kri93, p. 51] which is unfortunately too weak since it always requires $\Gamma \vdash s : \rho_0$ for some type ρ_0) and is proved by induction on the derivation of $\Gamma \vdash r[x := s] : \tau$. The case $r = x$ is trivial. Let now $r \neq x$.

(V) Let $\Gamma \vdash r[x := s] = y : \tau$ due to $y : \tau \in \Gamma$. Since $r \neq x$, $r = y \neq x$. Therefore, $x \notin \mathrm{FV}(r)$, and we set $\rho := \rho_0$ and apply rule (V).

($\to_I$) Let $\Gamma \vdash r[x := s] = \lambda y t : \tau = \tau_1 \to \tau_2$ due to $\Gamma, y : \tau_1 \vdash t : \tau_2$. Since $r \neq x$, $r = \lambda y r_0$ and $t = r_0[x := s]$. We may assume that $y \notin \{x\} \cup \mathrm{FV}(s)$. Since $x \notin \mathrm{FV}(r_0)$ implies $x \notin \mathrm{FV}(r)$ implies $\Gamma \vdash s : \rho_0$ implies $\Gamma, y : \tau_1 \vdash s : \rho_0$, we may apply the induction hypothesis and get a type ρ such that $\Gamma, y : \tau_1, x : \rho \vdash r_0 : \tau_2$ and $\Gamma, y : \tau_1 \vdash s : \rho$. Strengthening yields $\Gamma \vdash s : \rho$. Finally, $\Gamma, x : \rho \vdash \lambda y r_0 : \tau_1 \to \tau_2$.

($\to_E$) Let $\Gamma \vdash r[x := s] = t_1 t_2 : \tau$ be derived from $\Gamma \vdash t_1 : \sigma \to \tau$ and $\Gamma \vdash t_2 : \sigma$. Since $r \neq x$, $r = r_1 r_2$ and $t_i = r_i[x := s]$ $(i = 1, 2)$. If $x \in \mathrm{FV}(r)$ then $x \in \mathrm{FV}(r_1)$ or $x \in \mathrm{FV}(r_2)$. In both cases the induction hypothesis provides a type ρ_0 such that $\Gamma \vdash s : \rho_0$. Hence, from the assumption of (b) we always have some ρ_0 with $\Gamma \vdash s : \rho_0$. By induction hypothesis, there are types ρ_1 and ρ_2 such that $\Gamma, x : \rho_1 \vdash r_1 : \sigma \to \tau$, $\Gamma \vdash s : \rho_1$, $\Gamma, x : \rho_2 \vdash r_2 : \sigma$ and $\Gamma \vdash s : \rho_2$. By Weakening 2 and Exchange 2, we get $\Gamma, x : \rho_1 \cap \rho_2 \vdash r_1 : \sigma \to \tau$ and $\Gamma, x : \rho_1 \cap \rho_2 \vdash r_2 : \sigma$, hence $\Gamma, x : \rho_1 \cap \rho_2 \vdash r_1 r_2 : \tau$. Finally, $\Gamma \vdash s : \rho_1 \cap \rho_2$.

($\cap_I$) Intersect the two types given by the induction hypothesis as in the preceding case.

($\cap_E$) Trivial from the induction hypothesis. □

The next lemma will always be needed if a given type assignment is analyzed. For its statement we need the concepts of prime types and prime factors [Kri93, p. 50]: A *prime type* is a type which is not of the form $\rho \cap \sigma$. Let $\mathcal{P}$ be the set of prime types. Every type can be written in the form $\rho_1 \cap \ldots \cap \rho_n$ with $n \geq 1$ and $\rho_i \in \mathcal{P}$ for $i \in \{1, \ldots, n\}$. Note that we do not care about parentheses since they do not change typability (neither in contexts due to Exchange 3 nor in the derived type due to $(\cap_I)$ and $(\cap_E)$). The ρ_i are called *prime factors* of ρ. We now present Lemma 1 from [Kri93, p. 50]. In the sequel the notation $\cap\vec{\rho}$ will be used for $\rho_1 \cap \ldots \cap \rho_n$ with the implicit assumption that the ρ_i are prime types. $\rho \in \vec{\rho}$ clearly means that $\rho = \rho_i$ for some i.

Lemma 22 (Inversion) *Let* $\Gamma \vdash r : \rho$ *with* $\rho \in \mathcal{P}$.

1. *If* $r = x$ *then* $x : \cap\vec{\rho} \in \Gamma$ *with* $\rho \in \vec{\rho}$.
2. *If* $r = \lambda x t$ *then* $\rho = \rho_1 \to \rho_2$ *and* $\Gamma, x : \rho_1 \vdash t : \rho_2$.[11]
3. *If* $r = ts$ *then* $\Gamma \vdash t : \sigma \to \cap\vec{\rho}$ *and* $\Gamma \vdash s : \sigma$ *for some types* σ *and* $\vec{\rho}$ *such that* $\rho \in \vec{\rho}$.

Proof Consider in the derivation of $\Gamma \vdash r : \rho$ an uppermost occurrence of some $\Gamma \vdash r : \cap\vec{\rho}$ with $\rho \in \vec{\rho}$. The rule by which this is achieved cannot be $(\cap_I)$ or $(\cap_E)$ since they would require an earlier occurrence of the described form. Therefore, in the first, second and third case, the rule is (V), $(\to_I)$ and $(\to_E)$, respectively. □

Lemma 23 (Subject Reduction) *If* $\Gamma \vdash r : \rho$ *and* $r \to_\beta r'$ *then* $\Gamma \vdash r' : \rho$.

Proof By induction on $\Gamma \vdash r : \rho$. Only the case $(\to_E)$ is non-trivial: We have $\Gamma \vdash rs : \rho$ due to $\Gamma \vdash r : \sigma \to \rho$ and $\Gamma \vdash s : \sigma$, and $rs \to_\beta t$. Show that $\Gamma \vdash t : \tau$.

- If $t = r's'$ by one reduction step altogether from r to r' and s to s' then we are done by the induction hypothesis.
- In the case of an outer β-reduction $r = \lambda x r_0$ and $t = r_0[x := s]$. By the preceding lemma, $\Gamma, x : \sigma \vdash r_0 : \rho$. By Lemma 21(a), $\Gamma \vdash r_0[x := s] : \rho$. □

As a further application of Inversion, we show that Ω is not typable.

Lemma 24 *Let* $\vec{\sigma} \subset \mathcal{P}$. *The following is impossible:*

$$\Gamma \vdash \omega : \cap\vec{\sigma} \to \rho \textit{ and } \Gamma \vdash \omega : \cap\vec{\sigma}.$$

Proof Induction on the number of different types in $\vec{\sigma}$. Assume both typings. By Inversion, $\Gamma, x : \cap\vec{\sigma} \vdash xx : \rho$. $\rho = \cap\vec{\rho}$. Therefore, $\Gamma, x : \cap\vec{\sigma} \vdash xx : \rho_1$ (the first element of $\vec{\rho}$). Again by Inversion, $\Gamma, x : \cap\vec{\sigma} \vdash x : \tau \to \cap\vec{\mu}$ and $\Gamma, x : \cap\vec{\sigma} \vdash x : \tau$ with $\rho_1 \in \vec{\mu}$. τ decomposes into prime factors as $\tau = \tau_1 \cap \ldots \cap \tau_n$, hence for every $k \in \{1, \ldots, n\}$: $\Gamma, x : \cap\vec{\sigma} \vdash x : \tau_k$. Again by Inversion, $\tau_k \in \vec{\sigma}$ for every $k \in \{1, \ldots, n\}$. Therefore, $\tau = \cap\vec{\sigma}\,'$ for some finite list $\vec{\sigma}\,'$ composed only of

[11] We may assume that $x \notin \Gamma$.

elements of $\vec{\sigma}$. Once more by Inversion, applied to $\Gamma, x : \cap\vec{\sigma} \vdash x : \cap\vec{\sigma}\,' \to \cap\vec{\mu}$, there is a σ_j in $\vec{\sigma}$ such that $\sigma_j = \cap\vec{\sigma}\,' \to \cap\vec{\mu}$. Hence $\sigma_j \notin \vec{\sigma}\,'$, and consequently $\vec{\sigma}\,'$ has fewer different types than $\vec{\sigma}$. From $\Gamma \vdash \omega : \cap\vec{\sigma}$ we now immediately get $\Gamma \vdash \omega : \cap\vec{\sigma}\,' \to \cap\vec{\mu}$ and $\Gamma \vdash \omega : \cap\vec{\sigma}\,'$. Contradiction by induction hypothesis. $\square$

Corollary 25 $\Gamma \nvdash \Omega : \rho$.

Proof If $\Gamma \vdash \Omega : \rho$ then also for some $\rho \in \mathcal{P}$. By Inversion, there is $\vec{\rho}$ and σ such that $\rho \in \vec{\rho}$ and $\Gamma \vdash \omega : \sigma \to \cap\vec{\rho}$ and $\Gamma \vdash \omega : \sigma$. Since $\sigma = \cap\vec{\sigma}$ with $\vec{\sigma} \subset \mathcal{P}$, the previous lemma applies. $\square$

Exercise 16 *Produce a weakly normalizing term which cannot be typed with intersection types.*

6.3 Strong Normalization of Typable Terms

The following proof is in the spirit of [JM99] which dealt with permutative conversions instead of intersection types.

Lemma 26 *If* $r \in \mathsf{SN}$ *and* $x \notin FV(r)$ *then* $rx \in \mathsf{SN}$.

Proof Induction on SN. $\square$

Note that $x \notin \mathrm{FV}(r)$ is a superfluous assumption. Nevertheless, we do not need a stronger statement, and therefore only consider what can be proved so easily.

Lemma 27 *If* $r, s \in \mathsf{SN}$ *and* $\Gamma \vdash s : \rho$ *and* $\Gamma, x : \rho \vdash r : \sigma$ *then* $r[x := s] \in \mathsf{SN}$.

Proof We first define a measure $h(\rho) \in \mathbb{N}_0$ for every type ρ by recursion on ρ as follows:

$$\begin{aligned} h(\iota) &:= 0 \\ h(\rho \to \sigma) &:= 1 + \max(h(\rho), h(\sigma)) \\ h(\rho \cap \sigma) &:= \max(h(\rho), h(\sigma)) \end{aligned}$$

The proof is by main induction on $h(\rho)$, side induction on $r \in \mathsf{SN}$ and case distinction according to $r \in \mathsf{SN}$. (Observe that we also have $\Gamma \vdash r[x := s] : \sigma$ by Lemma 21(a).)

Case $y\vec{r}$. This has been derived from $\vec{r} \subset \mathsf{SN}$. By multiple Inversion, we get $\Gamma, x : \rho \vdash \vec{r} : \vec{\mu}$ for suitable $\vec{\mu}$. By side induction hypothesis, $\vec{r}[x := s] \subset \mathsf{SN}$. Therefore, $y\vec{r}[x := s] \in \mathsf{SN}$. This finishes with the case $y \neq x$. So assume $y = x$. If $\vec{r}$ is empty, the claim is trivial. Otherwise, $\vec{r} = t, \vec{t}$, hence $\Gamma, x : \rho \vdash xt\vec{t} : \sigma$. $xt\vec{t} = (z\vec{t})[z := xt]$ for some "new" variable z. By Lemma 21(b), there is a type τ such that $\Gamma, x : \rho, z : \tau \vdash z\vec{t} : \sigma$ and $\Gamma, x : \rho \vdash xt : \tau$. $\tau = \cap\vec{\tau}$. Consider the element τ_k of $\vec{\tau}$. $\Gamma, x : \rho \vdash xt : \tau_k$. By Inversion, there is a type ρ'_k and types $\vec{\nu}_k$ with $\tau_k \in \vec{\nu}_k$ such that $\Gamma, x : \rho \vdash x : \rho'_k \to \cap\vec{\nu}_k$ and $\Gamma, x : \rho \vdash t : \rho'_k$. With $\rho = \cap\vec{\rho}$ and Inversion, we get $\rho'_k \to \cap\vec{\nu}_k \in \vec{\rho}$. Therefore, $h(\tau_k) \leq h(\cap\vec{\nu}_k) < h(\rho'_k \to \cap\vec{\nu}_k) \leq h(\rho)$. Since this holds for every k, also $h(\tau) < h(\rho)$. We continue with $k := 1$,

$\rho' := \rho'_1$ and $\vec{v} := \vec{v}_1$. Clearly, $\Gamma \vdash s : \rho' \to \cap\vec{v}$. By the previous lemma, $sz' \in \mathsf{SN}$ with a "new" variable z'. By Weakening 1, $\Gamma, z' : \rho' \vdash sz' : \cap\vec{v}$. By Lemma 21(a): $\Gamma \vdash t[x := s] : \rho'$. Since $h(\rho') < h(\rho' \to \cap\vec{v}) \leq h(\rho)$ we may apply the main induction hypothesis for type ρ' and get $st[x := s] \in \mathsf{SN}$. By Lemma 21(a), $\Gamma, z : \tau \vdash z\vec{t}[x := s] : \sigma$ and $\Gamma \vdash st[x := s] : \tau$. Clearly, $z\vec{t}[x := s] \in \mathsf{SN}$. Hence, by the main induction hypothesis for type τ, we finally get $(x\vec{r})[x := s] = st[x := s]\vec{t}[x := s] \in \mathsf{SN}$.

Case $\lambda y r$. This comes from $r \in \mathsf{SN}$. We have $\Gamma, x : \rho \vdash \lambda y r : \sigma$. Show $(\lambda y r)[x := s] \in \mathsf{SN}$. We may assume that $y \notin \{x\} \cup \mathrm{FV}(s)$. Therefore $(\lambda y r)[x := s] = \lambda y.r[x := s]$, and it suffices to show $r[x := s] \in \mathsf{SN}$. $\sigma = \cap\vec{\sigma}$ and $\Gamma, x : \rho \vdash \lambda y r : \sigma_1$ (the first element of $\vec{\sigma}$). By Inversion, $\sigma_1 = \tau \to \sigma'$ and $\Gamma, x : \rho, y : \tau \vdash r : \sigma'$. By Weakening 1, $\Gamma, y : \tau \vdash s : \rho$. The side induction hypothesis yields $r[x := s] \in \mathsf{SN}$.

Case $(\lambda y r)s\vec{s}$. This is derived from $r[y := t]\vec{t} \in \mathsf{SN}$ and $t \in \mathsf{SN}$. We have $\Gamma, x : \rho \vdash (\lambda y r)t\vec{t} : \sigma$ and have to show that (we assume that $y \notin \{x\} \cup \mathrm{FV}(s)$) $((\lambda y r)t\vec{t})[x := s] = (\lambda y.r[x := s])t[x := s]\vec{t}[x := s] \in \mathsf{SN}$. For this we need $(r[x := s])[y := t[x := s]]\vec{t}[x := s] \in \mathsf{SN}$ and $t[x := s] \in \mathsf{SN}$. By multiple Inversion, we get $\Gamma, x : \rho \vdash t : \mu$ for some type μ. The side induction hypothesis yields $t[x := s] \in \mathsf{SN}$. By Subject Reduction, $\Gamma, x : \rho \vdash r[y := t]\vec{t} : \sigma$. Hence, again by side induction hypothesis, $(r[y := t]\vec{t})[x := s] \in \mathsf{SN}$. We are done by Lemma 2 which tells us that $(r[y := t]\vec{t})[x := s] = (r[x := s])[y := t[x := s]]\vec{t}[x := s]$. □

Corollary 28 (Main Theorem) *If* $\Gamma \vdash r : \rho$ *then* $r \in \mathsf{SN}$.

Proof By induction on $\vdash$. In the case $(\to_E)$ use the induction hypothesis, $rs = (rx)[x := s]$ and the preceding two lemmas. □

Since SN is the set of strongly normalizing terms (here we only need soundness of SN), every typable term is strongly normalizing. Since typability with simple types is more restrictive than that with intersection types, simple typability also implies strong normalization.

Note also that Corollary 25 is a trivial consequence of our normalization result.

Exercise 17 *Let* ι *and* κ *be different atomic types. Show that*

$$\nvdash r : ((\iota \to \kappa) \to \iota) \to \iota$$

for every term r.

Hint: It is advisable to solve the problem first for simple types. This case is well-known as the underivability of the Peirce formula in minimal logic.

6.4 Typability of Strongly Normalizing Terms

Let us write $\Gamma + \Delta$ for the context where the requirements on the variables in Γ and Δ are added. More formally, if $\Gamma = \vec{x} : \vec{\rho}, \vec{y} : \vec{\sigma}$ and $\Delta = \vec{x} : \vec{\rho}\,', \vec{z} : \vec{\tau}$ with $\vec{x}$, $\vec{y}$ and $\vec{z}$ pairwise disjoint and $\vec{x}$ of length n, then

$$\Gamma + \Delta := x_1 : \rho_1 \cap \rho_1', \ldots, x_n : \rho_n \cap \rho_n', \vec{y} : \vec{\sigma}, \vec{z} : \vec{\tau}.$$

From Weakening 1 and Weakening 2, it follows that $\Gamma \vdash r : \rho$ implies $\Gamma + \Delta \vdash r : \rho$. We will also form $\Gamma_1 + \ldots + \Gamma_n$ in a similar way without caring about parentheses.

Theorem 2 (Completeness) *If $r \in \mathsf{SN}$ then there are Γ and ρ such that $\Gamma \vdash r : \rho$.*

Proof Induction on $r \in \mathsf{SN}$.

$x\vec{r}$. By induction hypothesis, $\Gamma_i \vdash r_i : \rho_i$ for every i. Therefore

$$\Gamma_1 + \ldots + \Gamma_n + x : (\rho_1 \to \ldots \to \rho_n \to \rho) \vdash x\vec{r} : \rho$$

for any type ρ.

$\lambda x r$. Let $\lambda x r \in \mathsf{SN}$ due to $r \in \mathsf{SN}$. By induction hypothesis, $\Gamma \vdash r : \rho$. Possibly by Weakening 1, we may assume that $\Gamma = \Delta, x : \sigma$. This yields $\Delta \vdash \lambda x r : \sigma \to \rho$.

$(\lambda x r)s\vec{s}$. Assume $(\lambda x r)s\vec{s} \in \mathsf{SN}$ has been derived from $r[x := s]\vec{s} \in \mathsf{SN}$ and $s \in \mathsf{SN}$. By induction hypothesis, $\Gamma \vdash r[x := s]\vec{s} : \tau$ and $\Gamma' \vdash s : \sigma$. Setting $\Delta := \Gamma + \Gamma'$, we get $\Delta \vdash r[x := s]\vec{s} : \tau$ and $\Delta \vdash s : \sigma$. Writing $r[x := s]\vec{s} = (y\vec{s})[y := r[x := s]]$ and applying Lemma 21(b), we find a type ρ such that $\Delta \vdash y\vec{s} : \tau$ and $\Delta \vdash r[x := s] : \rho$. Again by Lemma 21(b), there is a type σ' such that $\Delta, x : \sigma' \vdash r : \rho$ and $\Delta \vdash s : \sigma'$. Hence, $\Delta \vdash (\lambda x r)s : \rho$ and by Lemma 21(a), $\Delta \vdash (\lambda x r)s\vec{s} : \tau$. □

Remark: One can also characterize the weakly normalizing terms via intersection types. For this, one has to add an atomic type (typically called Ω) which is inhabited by every term (hence the typing rules have to be extended by this simple rule). Then the weakly normalizing terms are exactly those which are typable in the extended system with a type and a context where in both of them the special atomic type does not occur. (However, it may appear in the typing derivation!)

7 Parametric Polymorphism

Although the situation with intersection types is quite satisfying since the associated typing system exactly captures the strongly normalizing terms, there is interest for other typing systems which type only strongly normalizing terms but fail to type all of them.

Exercise 18 *In [Urz96] it is shown that* $(\lambda f \lambda x.f(fx))(\lambda f \lambda x.f(fx))(\lambda x \lambda y.x)$ *is not typable in the system of universal types presented below. Show that it is strongly normalizing.*

Why is there an interest? By intersection typing, one can model that a term has finitely many types in parallel. But very often, terms have infinitely many types all being instances of some pattern. The easiest example is the identity λxx which may receive any type of the form $\rho \to \rho$ in the empty context. It is a natural idea to allow universal quantification over type identifiers (the elements of $\mathcal{V}_T$). In this example, we would assume $\alpha \in \mathcal{V}_T$ and give the type $\forall\alpha.\alpha \to \alpha$ to λxx (in the empty context). And we may instantiate this for any type ρ instead of α to get $\vdash \lambda xx : \rho \to \rho$. The universal quantifier thus expresses the parametric polymorphism of the identity: For every type ρ (which is the parameter) the identity acts as an element of the function space $\rho \to \rho$, hence belongs in some sense to many different spaces, but in a uniform fashion, namely in spaces described uniformly by a type expression depending on the parameter ρ. In the following, this intuition will be made precise. And since we want to study the idea of parametric polymorphism in isolation, we abandon the ad hoc polymorphism stemming from intersection typing.

Definition 19 (Universal types) *The set* $\mathcal{T}_u$ *of universal types is inductively given by (we will always assume that* $\mathcal{V}_T$ *is an infinite set):*

- *If* $\alpha \in \mathcal{V}_T$ *then* $\alpha \in \mathcal{T}_u$*.*
- *If* $\rho \in \mathcal{T}_u$ *and* $\sigma \in \mathcal{T}_u$ *then* $(\rho \to \sigma) \in \mathcal{T}_u$*.*
- *If* $\alpha \in \mathcal{V}_T$ *and* $\rho \in \mathcal{T}_u$ *then* $\forall\alpha\rho \in \mathcal{T}_u$*.*

Note that the usual name for type identifiers is now changed from ι to α (β and γ will also be used) which emphasizes their variable nature.

Definition 20 (Free type variables) *Define the set* $FV(\rho)$ *of type variables occurring free in* ρ *by recursion on* ρ*:*

- $FV(\alpha) := \{\alpha\}$.
- $FV(\rho \to \sigma) := FV(\rho) \cup FV(\sigma)$.
- $FV(\forall\alpha\rho) := FV(\rho) \setminus \{\alpha\}$.

As is indicated by the preceding definition, the universal quantifier $\forall$ is considered as a binder like the λ in terms. We will follow the same conventions concerning the irrelevance of the given variable name, e.g., we syntactically identify the types $\forall\alpha.\alpha \to \alpha$ and $\forall\beta.\beta \to \beta$ and also use (like in the previous examples) the dot notation for "invisible parentheses".

Definition 21 (Type substitution) *Define the result* $\rho[\alpha := \sigma]$ *of replacing every free occurrence of the variable* α *in* ρ *by the type* σ *recursively as follows:*

- $\alpha[\alpha := \sigma] := \sigma$.
- $\beta[\alpha := \sigma] := \beta$ *for* $\beta \neq \alpha$.
- $(\rho \to \tau)[\alpha := \sigma] := \rho[\alpha := \sigma] \to \tau[\alpha := \sigma]$.
- $(\forall\beta\rho)[\alpha := \sigma] := \forall\beta.\rho[\alpha := \sigma]$ *where we may assume by renaming of the bound type variable* α *that* $\alpha \notin \{\beta\} \cup FV(\sigma)$.

Lemma 29 *If* $\alpha \notin FV(\rho)$ *then* $\rho[\alpha := \sigma] = \rho$.

Proof Induction on ρ. $\square$

Lemma 30 $FV(\rho[\alpha := \sigma]) \subseteq FV(\forall\alpha\rho) \cup FV(\sigma)$.

Proof Induction on ρ. $\square$

Let us describe the typing system known under the name "system F in Curry-style [Bar93]", and overload the symbol $\vdash$ once more.

Definition 22 (Universal typing) *The relation* $\Gamma \vdash r : \rho$ *(term* r *has type* ρ *in context* Γ*) is inductively defined by the following rules:*

$$\frac{}{\Gamma, x : \rho \vdash x : \rho}(\mathrm{V}) \qquad \frac{\Gamma, x : \rho \vdash r : \sigma}{\Gamma \vdash \lambda x r : \rho \to \sigma}(\to_{\mathrm{I}}) \qquad \frac{\Gamma \vdash r : \rho \to \sigma \qquad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}(\to_{\mathrm{E}})$$

$$\frac{\Gamma \vdash r : \rho \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash r : \forall\alpha\rho}(\forall_{\mathrm{I}}) \qquad \frac{\Gamma \vdash r : \forall\alpha\rho \qquad \sigma \in \mathcal{T}_u}{\Gamma \vdash r : \rho[\alpha := \sigma]}(\forall_{\mathrm{E}})$$

The conventions concerning contexts are as before, and since contexts are lists of declarations for term variables, $\alpha \notin FV(\Gamma)$ *shall mean that* $\alpha \notin FV(\rho)$ *for* ρ *being one of the assigned types in* Γ.

The rule $(\forall_{\mathrm{I}})$ of $\forall$-introduction has the proviso called "eigenvariable condition" that nothing depending on α may have been assumed on the free variables in r. Otherwise, we would derive $x : \alpha \vdash x : \forall\alpha\alpha$, then $x : \alpha \vdash x : \rho$ for any type ρ, and finally $\vdash \lambda xx : \alpha \to \rho$ for any ρ which we never had in mind. The $\forall$-elimination rule $(\forall_{\mathrm{E}})$ expresses that every instance of ρ (with ρ replaced by any type σ) is also a type of r if it received the type $\forall\alpha\rho$. Note that the type σ may again involve universal quantifiers, e. g., we have that

$$\vdash \lambda xx : (\forall\alpha.\alpha \to \beta) \to (\forall\alpha.\alpha \to \beta)$$

and hence $\vdash \lambda xx : \forall\beta.(\forall\alpha.\alpha \to \beta) \to (\forall\alpha.\alpha \to \beta)$. We may also type ω: E. g., $\vdash \omega : \forall\alpha\alpha \to \forall\alpha\alpha$.

We again have Exchange 1 (p. 34) as part of our understanding of contexts. It is quite easy to establish Weakening 1, Strengthening (see Lemma 20) and Lemma 21(a). Subject Reduction (see Lemma 23) also holds but requires a more intricate Inversion Lemma (see e. g. [Bar93, p. 174]).

Exercise 19 *[Bar93, p. 165] Show that our favourite example term* ω *receives the types* $\forall\beta.\forall\alpha\alpha \to \beta$*,* $\forall\beta.\forall\alpha\alpha \to \beta \to \beta$ *and* $\forall\alpha\alpha \to \forall\alpha\alpha$ *in the empty context.*

7.1 Strong Normalization of Typable Terms

We again want to establish strong normalization, i. e., that whenever $\Gamma \vdash r : \rho$ then $r \in \mathsf{SN}$. For proof-theoretic reasons which require the strength of the metatheory exceeding that of second-order arithmetic[12], our direct proof for intersection typing in section 6.3 that SN is closed under typed substitution cannot work. Therefore, the powerful and versatile candidate method (also called computability predicates method) is introduced by which a stronger statement than $r \in \mathsf{SN}$ for every typable term r is proved. For this we need the concept of saturated sets which are subsets of SN with good closure properties. First we define them, then explain the method, and finally apply the method to our setting.

7.1.1 Saturated Sets

The following concept was first used by Tait [Tai75].

Definition 23 (Saturated set) *A set $\mathcal{M}$ of terms is* saturated *if the following conditions are met:*

1. *If $r \in \mathcal{M}$ then $r \in \mathsf{SN}$.*
2. *If $\vec{r} \subset \mathsf{SN}$ then $x\vec{r} \in \mathcal{M}$.*
3. *If $r[x := s]\vec{s} \in \mathcal{M}$ and $s \in \mathsf{SN}$ then $(\lambda x r)s\vec{s} \in \mathcal{M}$.*

This definition is nearly the same as that in [Bar93, p. 177].[13] Let SAT be the set of saturated sets. Trivially, $\mathsf{SN} \in \mathsf{SAT}$.

The transformation of the definition of SN into that of saturatedness is well motivated by typing considerations: It would be possible to relativize the definition to τ-saturatedness for a type τ by restricting to terms of type τ (in a given context) only. For this to work it is essential to have no requirement in the rules that a term of a different type shall be in $\mathcal{M}$ in order to conclude that some term belongs to $\mathcal{M}$. Therefore we omit the rule which changes the type (hence leave out the abstraction rule), and in the other rules replace every occurrence of SN by $\mathcal{M}$ in the conclusion and for the terms in the premisses receiving the same type. In general, the other terms do not get the same type and therefore can only be required to belong to SN.

The candidate method goes as follows: By means of saturated sets we define (by recursion on types) predicates of strong computability with respect to an assignment of saturated sets for type variables (a candidate assignment) and finally show (by induction on typings) that every typable term is strongly computable under substitution. Hence every typable term is contained in a saturated set (the computability predicate) which only consists of strongly normalizing terms (due to $\mathsf{SN} \subseteq \mathsf{sn}$).

[12] a consequence of Gödel's second incompleteness theorem, see chapter 15 in [GLT89]

[13] Note, however, that in our definition SN stands for the syntax-directed definition, whereas [Bar93] considers strong normalization which is intensionally different (and extensionally the same).

This all works if there are constructions for saturated sets corresponding to the type constructs of the system for which the introduction rules and the elimination rules are sound. (This is presented at length in [Mat98].)

7.1.2 Calculating with Saturated Sets

It is always possible to produce a saturated set from any set M of terms by the *saturated closure* $\mathsf{cl}(\mathsf{M})$, defined by induction as follows:

- If $r \in \mathsf{M} \cap \mathsf{SN}$ then $r \in \mathsf{cl}(\mathsf{M})$.
- If $\vec{r} \subset \mathsf{SN}$ then $x\vec{r} \in \mathsf{cl}(\mathsf{M})$.
- If $r[x := s]\vec{s} \in \mathsf{cl}(\mathsf{M})$ and $s \in \mathsf{SN}$ then $(\lambda x r)s\vec{s} \in \mathsf{cl}(\mathsf{M})$.

Since $\mathsf{cl}(\mathsf{M}) \subseteq \mathsf{SN}$ (proved by induction on the definition), it is the least saturated set containing $\mathsf{M} \cap \mathsf{SN}$. In the remainder of the normalization proof, let $\mathcal{M}$ and $\mathcal{N}$ denote saturated sets.

We want to construct a saturated set $\mathcal{M} \to \mathcal{N}$ which will later serve to define strong computability for function types. Define

$$\mathsf{S}_x(\mathcal{M},\mathcal{N}) := \{r \mid \forall s \in \mathcal{M}\; r[x := s] \in \mathcal{N}\},$$

$$\mathsf{I}(\mathcal{M},\mathcal{N}) := \{\lambda x r \mid x \in \mathcal{V} \text{ and } r \in \mathsf{S}_x(\mathcal{M},\mathcal{N})\} \quad \text{and}$$

$$\mathsf{E}(\mathcal{M},\mathcal{N}) := \{r \mid \forall s \in \mathcal{M}, rs \in \mathcal{N}\}.$$

We get the introduction-based definition $\mathcal{M} \to_{\mathrm{I}} \mathcal{N}$ and the elimination-based definition $\mathcal{M} \to_{\mathrm{E}} \mathcal{N}$ of saturated sets:

$$\mathcal{M} \to_{\mathrm{I}} \mathcal{N} := \mathsf{cl}(\mathsf{I}(\mathcal{M},\mathcal{N})) \text{ and } \mathcal{M} \to_{\mathrm{E}} \mathcal{N} := \mathsf{cl}(\mathsf{E}(\mathcal{M},\mathcal{N})).$$

Lemma 31 $\mathsf{I}(\mathcal{M},\mathcal{N}) \subseteq \mathsf{SN}$, $\mathsf{E}(\mathcal{M},\mathcal{N}) \cap \mathsf{SN} \in \mathsf{SAT}$, *and* $\mathsf{I}(\mathcal{M},\mathcal{N}) \subseteq \mathsf{E}(\mathcal{M},\mathcal{N})$.

Proof

(1) Let $r \in \mathsf{S}_x(\mathcal{M},\mathcal{N})$. Then for $s := x \in \mathcal{M}$, we get $r = r[x := s] \in \mathcal{N} \subseteq \mathsf{SN}$, hence also $\lambda x r \in \mathsf{SN}$.

(2) Check the conditions of saturatedness for $\mathsf{E}(\mathcal{M},\mathcal{N}) \cap \mathsf{SN}$:

   1. Trivial.
   2. Let $\vec{r} \subset \mathsf{SN}$ and $s \in \mathcal{M}$. Since $s \in \mathsf{SN}$ and $\mathcal{N} \in \mathsf{SAT}$, $x\vec{r}s \in \mathcal{N}$.
   3. Simply append s and use saturatedness of $\mathcal{N}$.

(3) Let $r \in \mathsf{S}_x(\mathcal{M},\mathcal{N})$ and $s \in \mathcal{M}$. Show that $(\lambda x r)s \in \mathcal{N}$. Since $\mathcal{N} \in \mathsf{SAT}$, it suffices to show $r[x := s] \in \mathcal{N}$ and $s \in \mathsf{SN}$ which follow by definition of $\mathsf{S}_x(\mathcal{M},\mathcal{N})$ and $\mathcal{M} \subseteq \mathsf{SN}$. □

From the lemma, we get $\mathsf{I}(\mathcal{M},\mathcal{N}) \subseteq \mathcal{M} \to_{\mathrm{I}} \mathcal{N}$, $\mathcal{M} \to_{\mathrm{E}} \mathcal{N} = \mathsf{E}(\mathcal{M},\mathcal{N}) \cap \mathsf{SN}$ and, due to monotonicity of cl, $\mathcal{M} \to_{\mathrm{I}} \mathcal{N} \subseteq \mathcal{M} \to_{\mathrm{E}} \mathcal{N}$.

Define $\mathcal{M} \to \mathcal{N} := \mathcal{M} \to_X \mathcal{N}$ with $X \in \{\mathrm{I}, \mathrm{E}\}$. We never use any property depending on this choice but only the following three properties which are valid for both choices and follow immediately from the preceding remarks:

(SAT) $\mathcal{M} \to \mathcal{N} \in \mathsf{SAT}$.

$(\to_{\mathrm{I}})$ If $r \in \mathsf{S}_x(\mathcal{M}, \mathcal{N})$ then $\lambda x r \in \mathcal{M} \to \mathcal{N}$.

$(\to_{\mathrm{E}})$ $r \in \mathcal{M} \to \mathcal{N} \wedge s \in \mathcal{M} \Rightarrow rs \in \mathcal{N}$.

Exercise 20 *Show that $\mathcal{M} \to_{\mathrm{I}} \mathcal{N} \neq \mathcal{M} \to_{\mathrm{E}} \mathcal{N}$ is possible by considering $\mathcal{M} := \mathsf{SN}$ and $\mathcal{N} := \mathsf{cl}(\{(\lambda x x)t \mid t \in \mathsf{SN}\})$.*

Hint: Study how abstractions may enter the saturated closure and apply this to the identity.

7.1.3 Strong Computability

In order to specify strong computability for universally quantified types $\forall \alpha \rho$, we have to define strong computability for the type ρ, but have to provide an arbitrary candidate for the strong computability of its argument α. Our candidates are the saturated sets, and the candidate assignments provide the relativization needed to put the proof through.

Definition 24 (Candidate assignment) *Any finite set of pairs (written $\alpha : \mathcal{M}$), consisting of a type variable and a saturated set, such that no type variable occurs twice.*

Candidate assigments are the counterpart to contexts. We will again use the letter Γ to denote a candidate assignment and write $\Gamma, \alpha : \mathcal{M}$ for the extended candidate assignment (with the implicit proviso that α does not occur in Γ).

Definition 25 (Strong computability) *Define the saturated set $\mathsf{SC}^{\rho}[\Gamma]$ of* strongly computable terms *w. r. t. type ρ and the candidate assigment Γ by recursion on ρ:*

- $\mathsf{SC}^{\alpha}[\Gamma] := \begin{cases} \mathcal{M} & \text{if } \alpha : \mathcal{M} \in \Gamma, \\ \mathsf{SN} & \text{otherwise.} \end{cases}$
- $\mathsf{SC}^{\rho \to \sigma}[\Gamma] := \mathsf{SC}^{\rho}[\Gamma] \to \mathsf{SC}^{\sigma}[\Gamma]$.
- $\mathsf{SC}^{\forall \alpha \rho}[\Gamma] := \bigcap_{\mathcal{M} \in \mathsf{SAT}} \mathsf{SC}^{\rho}[\Gamma, \alpha : \mathcal{M}]$ *(with set-theoretic intersection that clearly does not lead outside* SAT*; note that we may assume that α does not occur in Γ).*

The definition of $\mathsf{SC}^{\rho \to \sigma}[\Gamma]$ is a variant of the computability predicate definition in the famous [Tai67], its relativization to a candidate assignment and the big intersection in the definition of $\mathsf{SC}^{\forall \alpha \rho}[\Gamma]$ have been invented in [Gir72] and only today seem to be the straightforward extension of Tait's ideas. It has to be stressed that exactly this big intersection shows the impredicativity of the system of universal types: We need the intersection over any saturated set $\mathcal{M}$ in order to define a specific saturated set, namely $\mathsf{SC}^{\forall \alpha \rho}[\Gamma]$. This definition cannot be dealt with by second-order arithmetic, and, as remarked above, strong normalization also cannot be established by other means taken from second-order arithmetic. Finally note that we could easily reprove strong normalization for the system of intersection types by setting $\mathsf{SC}^{\rho \cap \sigma} := \mathsf{SC}^{\rho} \cap \mathsf{SC}^{\sigma}$ and by abandoning the notion of candidate assignment altogether.

Lemma 32 (Coincidence) *If* $\alpha \notin FV(\rho)$ *then* $\mathsf{SC}^\rho[\Gamma, \alpha : \mathcal{M}] = \mathsf{SC}^\rho[\Gamma]$.

Proof Induction on ρ.[14] $\square$

Lemma 33 (Substitution) $\mathsf{SC}^{\rho[\alpha:=\sigma]}[\Gamma] = \mathsf{SC}^\rho[\Gamma, \alpha : \mathsf{SC}^\sigma[\Gamma]]$.

Proof Induction on ρ, using the previous lemma. $\square$

We want to show that every typable term is strongly normalizing. By using the inductive characterization, we only need to show that they are in SN. Since saturated sets are contained in SN, it suffices to show that $r \in \mathsf{SC}^\rho[\emptyset]$ whenever r gets type ρ. This is achieved by applying the following lemma to the identity substitution. Unfortunately, we first have to extend the notion of substitution $r[x := s]$ to the simultaneous substitution $r[\vec{x} := \vec{s}]$ of all occurrences of x_i by s_i (for every i, with different variables x_i) in r which may be defined by recursion on r like ordinary substitution. We will also use the notation $\vec{x} : \vec{\rho}$ for $x_1 : \rho_1, \ldots, x_n : \rho_n$ and $\vec{s} \in \mathsf{SC}^{\vec{\rho}}[\Gamma]$ for $s_1 \in \mathsf{SC}^{\rho_1}[\Gamma], \ldots, s_n \in \mathsf{SC}^{\rho_n}[\Gamma]$.

Lemma 34 *If* $\vec{x} : \vec{\rho} \vdash r : \rho$ *and* $\vec{s} \in \mathsf{SC}^{\vec{\rho}}[\Gamma]$ *then* $r[\vec{x} := \vec{s}] \in \mathsf{SC}^\rho[\Gamma]$.

Proof By induction on $\vec{x} : \vec{\rho} \vdash r : \rho$ simultaneously for every candidate assignment Γ.

- (V) $r : \rho = x_i : \rho_i$. Obvious.
- $(\to_I)$ Let $\vec{x} : \vec{\rho} \vdash \lambda x r : \rho \to \sigma$ thanks to $\vec{x} : \vec{\rho}, x : \rho \vdash r : \sigma$. We have to show that $(\lambda x r)[\vec{x} := \vec{s}] \in \mathsf{SC}^\rho[\Gamma] \to \mathsf{SC}^\sigma[\Gamma]$. We may assume that $x \notin \vec{x} \cup \mathrm{FV}(\vec{s})$, and hence $(\lambda x r)[\vec{x} := \vec{s}] = \lambda x.r[\vec{x} := \vec{s}]$. It suffices to show that $r[\vec{x} := \vec{s}] \in \mathsf{S}_x(\mathsf{SC}^\rho[\Gamma], \mathsf{SC}^\sigma[\Gamma])$. So assume $s \in \mathsf{SC}^\rho[\Gamma]$ and show $r[\vec{x} := \vec{s}][x := s] \in \mathsf{SC}^\sigma[\Gamma]$. This follows from the induction hypothesis since $r[\vec{x} := \vec{s}][x := s] = r[\vec{x}, x := \vec{s}, s]$ by our assumption.
- $(\to_E)$ This is an immediate consequence of the induction hypothesis and the rule $(\to_E)$ for saturated sets.
- $(\forall_I)$ Let $\vec{x} : \vec{\rho} \vdash r : \forall \alpha \rho$ thanks to $\vec{x} : \vec{\rho} \vdash r : \rho$ and $\alpha \notin \mathrm{FV}(\vec{\rho})$. Let $\mathcal{M} \in \mathsf{SAT}$. We have to show that $r[\vec{x} := \vec{s}] \in \mathsf{SC}^\rho[\Gamma, \alpha : \mathcal{M}]$. Since $\alpha \notin \mathrm{FV}(\vec{\rho})$, we may apply the Coincidence Lemma and get $\vec{s} \in \mathsf{SC}^{\vec{\rho}}[\Gamma, \alpha : \mathcal{M}]$. Hence, we are done by the induction hypothesis.
- $(\forall_E)$ Let $\vec{x} : \vec{\rho} \vdash r : \rho[\alpha := \sigma]$ thanks to $\vec{x} : \vec{\rho} \vdash r : \forall \alpha \rho$. By induction hypothesis, $r[\vec{x} := \vec{s}] \in \mathsf{SC}^{\forall \alpha \rho}[\Gamma]$. Set $\mathcal{M} := \mathsf{SC}^\sigma[\Gamma]$. Then $r[\vec{x} := \vec{s}] \in \mathsf{SC}^\rho[\Gamma, \alpha : \mathcal{M}] = \mathsf{SC}^{\rho[\alpha:=\sigma]}[\Gamma]$ by the Substitution Lemma. $\square$

By setting $s_i := x_i \in \mathsf{SC}^{\rho_i}[\emptyset]$ for $\Gamma = \vec{x} : \vec{\rho}$, and by using $\mathsf{SC}^\rho[\emptyset] \subseteq \mathsf{SN}$, we get the following

Theorem 3 (Strong normalization) *If* $\Gamma \vdash r : \rho$ *then* $r \in \mathsf{SN}$. $\square$

[14] Clearly, our choice whether $\mathcal{M} \to \mathcal{N}$ equals $\mathcal{M} \to_I \mathcal{N}$ or $\mathcal{M} \to_E \mathcal{N}$ has to be made consistently. To be on the safe side, we assume that it has been made once and for all.

Note that by help of the interactive theorem-proving environment LEGO a variant to this proof has been produced [Alt93] which demonstrates the capability of those systems to deal with essentially complicated mathematical theorems.

Exercise 21 *Show that there is no term* r *such that* $\vdash r : \forall\alpha\alpha$.

7.2 Undecidability of Type Checking

The problem of type checking is to find out whether $\Gamma \vdash r : \rho$ holds for given Γ, r and ρ. The problem of typability is to find out if there is a type ρ for given Γ and r, such that $\Gamma \vdash r : \rho$. For both of these problems it was an open question whether they may be solved algorithmically [Bar93, p. 183]. It was generally believed that they are both undecidable. Nevertheless, the result was an achievement much applauded at the 1994 LICS[15] conference.

Theorem 4 ([Wel94]) *Type checking and typability are undecidable for the system of universal types.*

Proof See the 46 pages paper [Wel99] which rests on the undecidability of semi-unification. □

Therefore, in the sequel we will study a variant of the pure calculus with universal typing to be called system F. It has type information inside the term system making type checking decidable again.

7.3 An Explicit System of Parametric Polymorphism

This time, we do not alter the type system but the term system.

Definition 26 (Terms of system F) *The set* $\mathcal{T}_\mathsf{F}$ *of terms is inductively given by:*

- *If* $x \in \mathcal{V}$ *then* $x \in \mathcal{T}_\mathsf{F}$.
- *If* $x \in \mathcal{V}$, $\rho \in \mathcal{T}_u$ *and* $r \in \mathcal{T}_\mathsf{F}$ *then* $\lambda x^\rho r \in \mathcal{T}_\mathsf{F}$.
- *If* $r \in \mathcal{T}_\mathsf{F}$ *and* $s \in \mathcal{T}_\mathsf{F}$ *then* $(rs) \in \mathcal{T}_\mathsf{F}$.
- *If* $r \in \mathcal{T}_\mathsf{F}$ *and* $\alpha \in \mathcal{V}_\mathsf{T}$ *then* $\Lambda\alpha r \in \mathcal{T}_\mathsf{F}$.
- *If* $r \in \mathcal{T}_\mathsf{F}$ *and* $\sigma \in \mathcal{T}_u$ *then* $(r\sigma) \in \mathcal{T}_\mathsf{F}$.

The idea is to add type information to the terms. $\lambda x^\rho r$ is $\lambda x r$ but with an indication which was the type in the extended context for the typing of r (see the typing rules below). $\Lambda\alpha r$ is r but with its polymorphism in the parameter α made explicit. $(r\sigma)$ is r but with a declaration that it is used with type σ.

Again parentheses are omitted as much as possible (with applications associating to the left).

[15] Annual IEEE Symposium on Logic in Computer Science

Definition 27 (Free variables) *Define the set $FV(r)$ of variables occurring free in r by recursion on r:*

- $FV(x) := \{x\}$.
- $FV(\lambda x^\rho r) := FV(r) \setminus \{x\}$.
- $FV(rs) := FV(r) \cup FV(s)$.
- $FV(\Lambda \alpha r) := FV(r)$.
- $FV(r\sigma) := FV(r)$.

As before, λx^ρ binds the free occurrences of x in r, and we syntactically identify terms which only differ in the names of their bound variables.

Since types may form parts of a term, we now have an additional concept of free type variables of a term:

Definition 28 (Free type variables of a term) *Define the set $FTV(r)$ of type variables occurring free in r by recursion on r:*

- $FTV(x) := \emptyset$.
- $FTV(\lambda x^\rho r) := FV(\rho) \cup FTV(r)$.
- $FTV(rs) := FTV(r) \cup FTV(s)$.
- $FTV(\Lambda \alpha r) := FTV(r) \setminus \{\alpha\}$.
- $FTV(r\sigma) := FTV(r) \cup FV(\sigma)$.

Clearly, $\Lambda\alpha$ binds the free occurrences of α in r. We also identify terms which differ only in the names of their bound type variables.

It is straightforward to redefine substitution of terms for term variables in terms:

Definition 29 (Substitution) *Define the result $r[x := s]$ of replacing every free occurrence of the variable x in r by the term s recursively as follows:*

- $x[x := s] := s$
- $y[x := s] := y$ *for* $y \neq x$.
- $(\lambda y^\rho r)[x := s] := \lambda y^\rho . r[x := s]$ *where we may assume as usual that* $y \notin \{x\} \cup FV(s)$.
- $(rt)[x := s] := r[x := s]t[x := s]$.
- $(\Lambda \alpha r)[x := s] := \Lambda \alpha . r[x := s]$ *where we assume that* $\alpha \notin FTV(s)$.
- $(r\sigma)[x := s] := r[x := s]\sigma$.

But we also have substitution of types for type variables in terms:

Definition 30 (Type substitution) *Define the result* $r[\alpha := \sigma]$ *of replacing every free occurrence of the type variable* α *in* r *by the type* σ *recursively as follows:*

- $x[\alpha := \sigma] := x$.
- $(\lambda y^\rho r)[\alpha := \sigma] := \lambda y^{\rho[\alpha:=\sigma]}.r[\alpha := \sigma]$.
- $(rt)[\alpha := \sigma] := r[\alpha := \sigma]t[\alpha := \sigma]$.
- $(\Lambda\beta r)[\alpha := \sigma] := \Lambda\beta.r[\alpha := \sigma]$ *where we assume that* $\beta \notin \{\alpha\} \cup FV(\sigma)$.
- $(r\tau)[\alpha := \sigma] := r[\alpha := \sigma]\tau[\alpha := \sigma]$.

Lemma 35 *If* $x \notin FV(r)$ *then* $r[x := s] = r$. *If* $\alpha \notin FTV(r)$ *then* $r[\alpha := \sigma] = r$.

Proof Induction on r. □

The richer term syntax allows a new reduction in the spirit of β-reduction, namely $(\Lambda\alpha r)\sigma \to r[\alpha := \sigma]$ which perfectly fits with our intuition of Λ-abstraction and type application. For coding purposes, this is yet not enough (as will be clear later). Therefore, we also include the following η-reduction rules: $\lambda x^\rho.rx \to r$ if $x \notin \mathrm{FV}(r)$, and $\Lambda\alpha.r\alpha \to r$ if $\alpha \notin \mathrm{FTV}(r)$. Clearly, we could have added the first rule (without the type superscript) to the pure untyped λ-calculus. Unfortunately, the addition of $\lambda x.rx \to_\beta r$ for $x \notin \mathrm{FV}(r)$ would have destroyed Subject Reduction: Take three different type variables α, β, γ and apply rule $(\to_I)$ to $z : \alpha \to \forall\gamma\beta, x : \alpha \vdash zx : \beta$ or to $z : \alpha \to \beta, x : \alpha \vdash zx : \forall\gamma\beta$. Note that neither $z : \alpha \to \forall\gamma\beta \vdash z : \alpha \to \beta$ nor $z : \alpha \to \beta \vdash z : \alpha \to \forall\gamma\beta$ holds.

Definition 31 ($\beta\eta$-reduction) *Inductively define the relation* $\to_{\beta\eta}$ *as follows:*

(β) $(\lambda x^\rho r)s \to_{\beta\eta} r[x := s]$ *(outer β-reduction).*

(η) $\lambda x^\rho.rx \to_{\beta\eta} r$ *if* $x \notin FV(r)$ *(outer η-reduction).*

(β_F) $(\Lambda\alpha r)\sigma \to_{\beta\eta} r[\alpha := \sigma]$ *(outer type-β-reduction).*

(η_F) $\Lambda\alpha.r\alpha \to_{\beta\eta} r$ *if* $\alpha \notin FTV(r)$ *(outer type-η-reduction).*

(ξ) $r \to_{\beta\eta} r' \Rightarrow \lambda x^\rho r \to_{\beta\eta} \lambda x^\rho r'$ *(reduction under a λ-abstraction).*

(ξ_F) $r \to_{\beta\eta} r' \Rightarrow \Lambda\alpha r \to_{\beta\eta} \Lambda\alpha r'$ *(reduction under a Λ-abstraction).*

(r) $r \to_{\beta\eta} r' \Rightarrow rs \to_{\beta\eta} r's$ *(right application).*

(l) $r \to_{\beta\eta} r' \Rightarrow sr \to_{\beta\eta} sr'$ *(left application).*

(t) $r \to_{\beta\eta} r' \Rightarrow r\sigma \to_{\beta\eta} r'\sigma$ *(type application).*

If $r \to_{\beta\eta} s$ we say that r reduces by one $\beta\eta$-*reduction step* to s.

Lemma 36 *If* $r \to_{\beta\eta} r'$ *then* $FV(r') \subseteq FV(r)$ *and* $FTV(r') \subseteq FTV(r)$.

Proof Induction on $\to_{\beta\eta}$. For (β), one first has to prove $\mathrm{FV}(r[x := s]) \subseteq (\mathrm{FV}(r) \setminus \{x\}) \cup \mathrm{FV}(s)$ and $\mathrm{FTV}(r[x := s]) \subseteq \mathrm{FTV}(r) \cup \mathrm{FTV}(s)$. For (β_{F}), we need $\mathrm{FV}(r[\alpha := \sigma]) = \mathrm{FV}(r)$ and $\mathrm{FTV}(r[\alpha := \sigma]) \subseteq (\mathrm{FTV}(r) \setminus \{\alpha\}) \cup \mathrm{FV}(\sigma)$. $\square$

Lemma 6 and Lemma 7 also hold for the extended syntax. Moreover, we get substitutivity w.r.t. type substitution:

Lemma 37 (Substitutivity and compatibility)

> *If* $r \to_{\beta\eta} r'$ *then* $r[x := s] \to_{\beta\eta} r'[x := s]$ *and* $r[\alpha := \sigma] \to_{\beta\eta} r'[\alpha := \sigma]$.
>
> *If* $s \to_{\beta\eta} s'$ *then* $r[x := s] \to^*_{\beta\eta} r[x := s']$, *and* $r[x := s] \to_{\beta\eta} r[x := s']$ *if* x *occurs exactly once free in* r.

Proof Substitutivity is proved by induction on $r \to_{\beta\eta} r'$, compatibility is proved by induction on r. $\square$

Unfortunately, $\to_{\beta\eta}$ is *not* locally confluent: Consider

$$\begin{tikzcd}
 & \lambda x^{\rho}.(\lambda y^{\sigma} r)x \arrow[dl, "\beta\eta"'] \arrow[dr, "\beta\eta"] & \\
\lambda x^{\rho}.r[y := x] \arrow[d, equal] & & \lambda y^{\sigma} r \\
\lambda y^{\rho} r & &
\end{tikzcd}$$

with $\rho \neq \sigma$ and $x \notin \mathrm{FV}(\lambda y r)$. This problem will be overcome by typing.

Definition 32 (Typing for system F) *The relation* $\Gamma \vdash r : \rho$ *(term* r *has type* ρ *in context* Γ*) is inductively defined by the following rules:*

$$\frac{}{\Gamma, x : \rho \vdash x : \rho}(\mathrm{V}) \qquad \frac{\Gamma, x : \rho \vdash r : \sigma}{\Gamma \vdash \lambda x^{\rho} r : \rho \to \sigma}(\to_{\mathrm{I}}) \qquad \frac{\Gamma \vdash r : \rho \to \sigma \qquad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}(\to_{\mathrm{E}})$$

$$\frac{\Gamma \vdash r : \rho \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda\alpha r : \forall\alpha\rho}(\forall_{\mathrm{I}}) \qquad \frac{\Gamma \vdash r : \forall\alpha\rho \qquad \sigma \in \mathcal{T}_u}{\Gamma \vdash r\sigma : \rho[\alpha := \sigma]}(\forall_{\mathrm{E}})$$

The usual conventions concerning contexts apply.

Note how we restored the property that we always know which typing rule has been applied last (like with simple typing and unlike intersection typing or even universal typing).

Lemma 38 *If* $\Gamma \vdash r : \rho$ *then* $FV(\rho) \subseteq FV(\Gamma) \cup FTV(r)$.

Proof Induction on $\vdash$. Note that in case $(\to_{\mathrm{I}})$ we need that the information on the type of the abstracted variable is included in the syntax: If $\alpha \in \mathrm{FV}(\rho \to \sigma)$ then $\alpha \in \mathrm{FV}(\rho)$ or $\alpha \in \mathrm{FV}(\sigma)$. In the first case, $\alpha \in \mathrm{FTV}(\lambda x^{\rho} r)$. In the second case, the induction hypothesis gives that $\alpha \in \mathrm{FV}(\Gamma) \cup \mathrm{FV}(\rho) \cup \mathrm{FTV}(r) = \mathrm{FV}(\Gamma) \cup \mathrm{FTV}(\lambda x^{\rho} r)$. The case $(\forall_{\mathrm{E}})$ uses Lemma 30. $\square$

Lemma 39 *Every typable term, i. e., every term* r *such that there are* Γ *and* ρ *with* $\Gamma \vdash r : \rho$*, has exactly one of the following forms:*

$$x\vec{S} \qquad \lambda x^\rho r \qquad (\lambda x^\rho r)s\vec{S} \qquad \Lambda\alpha r \qquad (\Lambda\alpha r)\sigma\vec{S}$$

where $\vec{S}$ *shall denote a finite list of* terms and types *(we will later also use* $\vec{R}$ *for such a list).*

Proof It is clear that *every* term has exactly one of the following forms:

$$x\vec{S} \qquad \lambda x^\rho r \qquad (\lambda x^\rho r)s\vec{S} \qquad (\lambda x^\rho r)\sigma\vec{S} \qquad \Lambda\alpha r \qquad (\Lambda\alpha r)s\vec{S} \qquad (\Lambda\alpha r)\sigma\vec{S}.$$

Typability rules out the fourth and sixth possibility (note that the typability of $r\vec{S}$ implies that of r). □

Once again we have Exchange 1 (p. 34) as part of our understanding of contexts, and it is again quite easy to establish Weakening 1, Strengthening (see Lemma 20) and Lemma 21(a). Moreover, we have a version of Lemma 21(a) pertaining to type substitution:

Lemma 40 *If* $\Gamma \vdash r : \rho$ *then* $\Gamma[\alpha := \sigma] \vdash r[\alpha := \sigma] : \rho[\alpha := \sigma]$.

Proof $\Gamma[\alpha := \sigma]$ clearly denotes the context where all the types of the variables are substituted. The proof is by induction on $\vdash$. □

Lemma 41 (Subject reduction) *If* $r \to_{\beta\eta} r'$ *and* $\Gamma \vdash r : \rho$ *then* $\Gamma \vdash r' : \rho$.

Proof Induction on $\to_{\beta\eta}$. We only consider the initial cases.

(β) Let $\Gamma \vdash (\lambda x^\rho r)s : \sigma$. Then $\Gamma \vdash \lambda x^\rho r : \rho \to \sigma$ and $\Gamma \vdash s : \rho$. Hence $\Gamma, x : \rho \vdash r : \sigma$, and by the analogue of Lemma 21(a), $\Gamma \vdash r[x := s] : \sigma$.

(η) Let $\Gamma \vdash \lambda x^\rho.rx : \rho \to \sigma$. Hence, $\Gamma, x : \rho \vdash rx : \sigma$, and therefore, $\Gamma, x : \rho \vdash r : \rho \to \sigma$. By Strengthening, $\Gamma \vdash r : \rho \to \sigma$.

(β_F) Let $\Gamma \vdash (\Lambda\alpha r)\sigma : \tau$. Then $\Gamma \vdash \Lambda\alpha r : \forall\alpha\rho$ and $\rho[\alpha := \sigma] = \tau$. Consequently, $\Gamma \vdash r : \rho$ and $\alpha \notin \mathrm{FV}(\Gamma)$. Hence $\Gamma[\alpha := \sigma] = \Gamma$ and by the preceding lemma $\Gamma \vdash r[\alpha := \sigma] : \rho[\alpha := \sigma] = \tau$.

(η_F) Let $\Gamma \vdash \Lambda\alpha.r\alpha : \forall\alpha\rho$ with $\alpha \notin \mathrm{FTV}(r)$. Then $\Gamma \vdash r\alpha : \rho$ and $\alpha \notin \mathrm{FV}(\Gamma)$. Hence, $\Gamma \vdash r : \forall\beta\sigma$ and $\sigma[\beta := \alpha] = \rho$. If $\alpha = \beta$, then we are done. Otherwise, by Lemma 38, $\alpha \notin \mathrm{FV}(\sigma)$, hence $\forall\alpha\rho = \forall\beta\sigma$ by renaming of the bound variable. □

Note that the examples on page 50 are no longer critical: The first one yields $z : \alpha \to \forall\gamma\beta \vdash \lambda x^\alpha.zx\gamma : \alpha \to \beta$, the second $z : \alpha \to \beta \vdash \lambda x^\alpha\Lambda\gamma.zx : \alpha \to \forall\gamma\beta$. In both cases, we cannot apply an outer η-reduction.

Before studying confluence and strong normalization of the typable terms, we consider several examples showing the expressivity of system F.

Examples 11 *1. Set* $0 := \forall\alpha\alpha$*. Then* $\Gamma \vdash r : 0$ *implies* $\Gamma \vdash r\rho : \rho$.

2. *Set* $1 := \forall\alpha.\alpha \to \alpha$ *and* $\mathsf{IN1} := \Lambda\alpha\lambda x^\alpha x$. *Then* $\vdash \mathsf{IN1} : 1$.

3. *Set* $\rho \times \sigma := \forall\alpha.(\rho \to \sigma \to \alpha) \to \alpha$ *for some* $\alpha \notin FV(\rho) \cup FV(\sigma)$. *Set* $\langle r, s\rangle_{\rho,\sigma} := \Lambda\alpha\lambda z^{\rho\to\sigma\to\alpha}.zrs$ *for some* $z \notin FV(r) \cup FV(s)$ *and assume that* $\alpha \notin FTV(r) \cup FTV(s)$. *Hence, if* $\Gamma \vdash r : \rho$ *and* $\Gamma \vdash s : \sigma$ *then* $\Gamma \vdash \langle r, s\rangle_{\rho,\sigma} : \rho \times \sigma$. *Set* $r\mathsf{L}_{\rho,\sigma} := r\rho(\lambda x^\rho\lambda y^\sigma.x)$ *and* $r\mathsf{R}_{\rho,\sigma} := r\sigma(\lambda x^\rho\lambda y^\sigma.y)$. *Hence, if* $\Gamma \vdash r : \rho \times \sigma$ *then* $\Gamma \vdash r\mathsf{L}_{\rho,\sigma} : \rho$ *and* $\Gamma \vdash r\mathsf{R}_{\rho,\sigma} : \sigma$. *Moreover,* $\langle r, s\rangle_{\rho,\sigma}\mathsf{L}_{\rho,\sigma} \to^*_{\beta\eta} r$ *and* $\langle r, s\rangle_{\rho,\sigma}\mathsf{R}_{\rho,\sigma} \to^*_{\beta\eta} s$.

4. *Set* $\rho + \sigma := \forall\alpha.(\rho \to \alpha) \to (\sigma \to \alpha) \to \alpha$ *for some* $\alpha \notin FV(\rho) \cup FV(\sigma)$. *Set* $\mathsf{INL}_{\rho,\sigma}r := \Lambda\alpha\lambda x^{\rho\to\alpha}\lambda y^{\sigma\to\alpha}.xr$ *and* $\mathsf{INR}_{\rho,\sigma}r := \Lambda\alpha\lambda x^{\rho\to\alpha}\lambda y^{\sigma\to\alpha}.yr$ *for some* $x, y \notin FV(r)$ *(we assume that* $\alpha \notin FTV(r)$*). Hence, if* $\Gamma \vdash r : \rho$ *then* $\Gamma \vdash \mathsf{INL}_{\rho,\sigma}r : \rho + \sigma$, *and if* $\Gamma \vdash r : \sigma$ *then* $\Gamma \vdash \mathsf{INR}_{\rho,\sigma}r : \rho + \sigma$. *Moreover, if* $\Gamma \vdash r : \rho + \sigma$, $\Gamma \vdash s : \rho \to \tau$ *and* $\Gamma \vdash t : \sigma \to \tau$ *then* $\Gamma \vdash r\tau st : \tau$ *which gives a construct for case distinction as follows:* $\mathsf{INL}_{\rho,\sigma}r\tau st \to^*_{\beta\eta} sr$ *and* $\mathsf{INR}_{\rho,\sigma}r\tau st \to^*_{\beta\eta} tr$.

5. *Set* $\exists\alpha\rho := \forall\beta.(\forall\alpha.\rho \to \beta) \to \beta$ *for some* $\beta \notin \{\alpha\} \cup FV(\rho)$. *Set* $\mathsf{C}_{\exists\alpha\rho,\tau}r := \Lambda\beta\lambda x^{\forall\alpha.\rho\to\beta}.x\tau r$ *for some* $x \notin FV(r)$ *and* $\beta \notin FTV(r) \cup FV(\tau)$. *Hence, if* $\Gamma \vdash r : \rho[\alpha := \tau]$ *then* $\Gamma \vdash \mathsf{C}_{\exists\alpha\rho,\tau}r : \exists\alpha\rho$. *Also, if* $\Gamma \vdash r : \exists\alpha\rho$ *and* $\Gamma \vdash s : \forall\alpha.\rho \to \sigma$ *with* $\alpha \notin FV(\sigma)$, *then* $\Gamma \vdash r\sigma s : \sigma$. *Moreover,* $\mathsf{C}_{\exists\alpha\rho,\tau}r\sigma s \to^*_{\beta\eta} s\tau r$.

6. *Set* $\mathsf{nat} := \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$. *Set* $0 := \Lambda\alpha\lambda x^{\alpha\to\alpha}\lambda y^\alpha y$ *(this should not be confused with the type* $0 = \forall\alpha\alpha$*). Then* $\vdash 0 : \mathsf{nat}$. *Set* $\mathsf{S}r := \Lambda\alpha\lambda x^{\alpha\to\alpha}\lambda y^\alpha.x(r\alpha xy)$ *for some* $x, y \notin FV(r)$ *and* $\alpha \notin FTV(r)$. *Hence, if* $\Gamma \vdash r : \mathsf{nat}$ *then* $\Gamma \vdash \mathsf{S}r : \mathsf{nat}$. *This gives back iteration on naturals as follows: If* $\Gamma \vdash r : \mathsf{nat}$, $\Gamma \vdash s : \rho \to \rho$ *and* $\Gamma \vdash t : \rho$ *then* $\Gamma \vdash r\rho st : \rho$. *Moreover,* $0\rho st \to^*_{\beta\eta} t$ *and* $(\mathsf{S}r)\rho st \to^*_{\beta\eta} s(r\rho st)$, *hence* t *is the initial term of the iteration, and* s *is the step term.*

Note that the examples with exception of the last one may all be seen as intuitionistic variants of classical encodings. If instead of the universally quantified α, we only had the falsum $\perp$, we came to the following classical identities (writing $\neg\rho$ for $\rho \to \perp$, $\perp$ for 0, $\top$ for 1, $\rho \wedge \sigma$ for $\rho \times \sigma$ and $\rho \vee \sigma$ for $\rho + \sigma$): $\perp = \perp$, $\top = \perp \to \perp$, $\rho \wedge \sigma = \neg(\rho \to \neg\sigma)$, $\rho \vee \sigma = \neg\rho \to \neg\neg\sigma$, $\exists\alpha\rho = \neg\forall\alpha\neg\rho$.

7.4 Strong Normalization and Typed Confluence of F

We will see that local confluence holds for typable terms and that every typable term is strongly normalizing and finally conclude that typable terms even enjoy confluence.

Lemma 42 (Typed local confluence) *If* $\Gamma \vdash r : \rho$, $r \to_{\beta\eta} r'$ *and* $r \to_{\beta\eta} r''$ *then there is a term* t *such that* $r' \to_{\beta\eta} t$ *and* $r'' \to_{\beta\eta} t$.

Proof Induction on r, case distinction according to the last rule of $\to_{\beta\eta}$ used to establish $r \to_{\beta\eta} r'$ and $r \to_{\beta\eta} r''$. Hence, we have to distinguish 81 cases.

The 9 cases in which the same rule is applied in both reductions either hold trivially (in the initial cases) or are immediate by the induction hypothesis. The other 72 cases come in 36 pairs of symmetric situations. We only consider the pairs where the first rule comes first in the list of rules. Note that the following 30 cases are syntactically impossible:

- β with η, β_F, η_F, ξ, ξ_F and t
- η with β_F, η_F, ξ_F, r, l and t
- β_F with η_F, ξ, ξ_F, r and l
- η_F with ξ, r, l and t
- ξ with ξ_F, r, l and t
- ξ_F with r, l and t
- r with t
- l with t

Hence, only 6 cases have to be considered:

β/r. We have

$$\begin{tikzcd} & (\lambda x^\rho r)s \arrow[dl, "\beta\eta"'] \arrow[dr, "\beta\eta"] & \\ r[x := s] & & ts \end{tikzcd}$$

with $\lambda x^\rho r \to_{\beta\eta} t$.

If $t = \lambda x^\rho r'$ with $r \to_{\beta\eta} r'$ then by substitutivity $r[x := s] \to_{\beta\eta} r'[x := s]$, hence we get

$$\begin{tikzcd} r[x := s] \arrow[ddr, "\beta\eta"'] & & ts \arrow[d, equal] \\ & & (\lambda x^\rho r')s \arrow[dl, "\beta\eta"] \\ & r'[x := s] & \end{tikzcd}.$$

Otherwise, $r = tx$ with $x \notin FV(t)$. Then $r[x := s] = t[x := s]s = ts$.

β/l. Use compatibility as in the case l/β in the proof of Lemma 5.

η/ξ. We have

$$\begin{tikzcd} & \lambda x^\rho .rx \arrow[dl, "\beta\eta"'] \arrow[dr, "\beta\eta"] & \\ r & & \lambda x^\rho t \end{tikzcd}$$

with $rx \to_{\beta\eta} t$.

If $t = r'x$ with $r \to_{\beta\eta} r'$, then also $x \notin FV(r')$, hence $\lambda x^\rho t \to_{\beta\eta} r'$, yielding

$$\begin{tikzcd} r \arrow[dr, "\beta\eta"'] & & \lambda x^\rho t \arrow[dl, "\beta\eta"] \\ & r' & \end{tikzcd}.$$

Otherwise, $r = \lambda y^\sigma s$ and $t = s[y := x]$. Since $\lambda x^\rho.rx$ is typable[16], there are Γ and τ such that $\Gamma \vdash \lambda x^\rho.(\lambda y^\sigma s)x : \rho \to \tau$. This comes from $\Gamma, x : \rho \vdash (\lambda y^\sigma s)x : \tau$, hence $\Gamma, x : \rho \vdash \lambda y^\sigma s : \rho \to \tau$. We conclude $\rho = \sigma$. Finally, since $x \notin \mathrm{FV}(r)$, $\lambda x^\rho t = \lambda x^\sigma.s[y := x] = \lambda y^\sigma s = r$.

β_{F}/t. We have
$$\begin{tikzcd}
& (\Lambda\alpha r)\sigma \arrow[dl, "\beta\eta"'] \arrow[dr, "\beta\eta"] & \\
r[\alpha := \sigma] & & t\sigma
\end{tikzcd}$$
with $\Lambda\alpha r \to_{\beta\eta} t$.

If $t = \Lambda\alpha r'$ with $r \to_{\beta\eta} r'$ then by Lemma 37, $r[\alpha := \sigma] \to_{\beta\eta} r'[\alpha := \sigma]$, hence we get
$$\begin{tikzcd}
r[\alpha := \sigma] \arrow[ddr, "\beta\eta"'] & & t\sigma \arrow[d, equal] \\
& & (\Lambda\alpha r')s \arrow[dl, "\beta\eta"] \\
& r'[\alpha := \sigma] &
\end{tikzcd}$$
.

Otherwise, $r = t\alpha$ with $\alpha \notin \mathrm{FTV}(t)$, hence $r[\alpha := \sigma] = t[\alpha := \sigma]\sigma = t\sigma$.

$\eta_{\mathsf{F}}/\xi_{\mathsf{F}}$. We have
$$\begin{tikzcd}
& \Lambda\alpha.r\alpha \arrow[dl, "\beta\eta"'] \arrow[dr, "\beta\eta"] & \\
r & & \Lambda\alpha t
\end{tikzcd}$$
with $r\alpha \to_{\beta\eta} t$.

If $t = r'\alpha$ with $r \to_{\beta\eta} r'$ then also $\alpha \notin \mathrm{FTV}(r')$, hence $\Lambda\alpha t \to_{\beta\eta} r'$, yielding
$$\begin{tikzcd}
r \arrow[dr, "\beta\eta"'] & & \Lambda\alpha t \arrow[dl, "\beta\eta"] \\
& r' &
\end{tikzcd}$$
.

Otherwise, $r = \Lambda\beta s$ and $t = s[\beta := \alpha]$. If $\alpha = \beta$, then we are done. Otherwise, $\alpha \notin \mathrm{FTV}(s)$ and hence also $\Lambda\alpha t = \Lambda\beta s$ by renaming of the bound type variable.

r/l. See the same case in the proof of Lemma 5. $\square$

In order to prove that even typed confluence holds, we first establish strong normalization: First we redefine the sets SN and SAT, and then show essentially the same results for SN and SAT as for the system of universal types.[17]

Definition 33 *Define the set* SN *inductively by:*

- *If the terms among* $\vec{R}$ *are in* SN *then* $x\vec{R} \in \mathsf{SN}$.
- *If* $r \in \mathsf{SN}$ *then* $\lambda x^\rho r \in \mathsf{SN}$.
- *If* $r \in \mathsf{SN}$ *then* $\Lambda\alpha r \in \mathsf{SN}$.

[16] This is the only place where we need the typability assumption.

[17] One could also derive strong normalization from that of the system of universal types. We prefer the direct proof, since it will be extended to the system of fixed-point types.

- *If* $r[x := s]\vec{S} \in \mathsf{SN}$ *and* $s \in \mathsf{SN}$ *then* $(\lambda x^\rho r)s\vec{S} \in \mathsf{SN}$.
- *If* $r[\alpha := \sigma]\vec{S} \in \mathsf{SN}$ *then* $(\Lambda\alpha r)\sigma\vec{S} \in \mathsf{SN}$.

Definition 34 *A set* $\mathcal{M}$ *of terms of system* F *is saturated if the following holds:*

1. *If* $r \in \mathcal{M}$ *then* $r \in \mathsf{SN}$.
2. *If the terms among* $\vec{R}$ *are in* SN *then* $x\vec{R} \in \mathcal{M}$.
3. *If* $r[x := s]\vec{S} \in \mathcal{M}$ *and* $s \in \mathsf{SN}$ *then* $(\lambda x r)s\vec{S} \in \mathcal{M}$.
4. *If* $r[\alpha := \sigma]\vec{S} \in \mathcal{M}$ *then* $(\Lambda\alpha r)\sigma\vec{S} \in \mathcal{M}$.

Let again SAT be the set of saturated sets. Again, $\mathsf{SN} \in \mathsf{SAT}$.

Lemma 43 $\mathsf{SN} \subseteq \mathsf{sn} := \mathsf{acc}_{\to_{\beta\eta}}$.

Proof We have to show that sn has all the defining properties of SN, i. e., we have to show that

- If the terms among $\vec{R}$ are in sn then $x\vec{R} \in \mathsf{sn}$.
- If $r \in \mathsf{sn}$ then $\lambda x^\rho r \in \mathsf{sn}$.
- If $r \in \mathsf{sn}$ then $\Lambda\alpha r \in \mathsf{sn}$.
- If $r[x := s]\vec{S} \in \mathsf{sn}$ and $s \in \mathsf{sn}$ then $(\lambda x^\rho r)s\vec{S} \in \mathsf{sn}$.
- If $r[\alpha := \sigma]\vec{S} \in \mathsf{sn}$ then $(\Lambda\alpha r)\sigma\vec{S} \in \mathsf{sn}$.

The case with the variable in the head is obvious since every reduction in $x\vec{R}$ takes place in one of the $\vec{R}$.

The abstraction case is more complicated: Do induction on $r \in \mathsf{sn}$. Assume $\lambda x^\rho r \to_{\beta\eta} t$. Either $t = \lambda x^\rho r'$ with $r \to_{\beta\eta} r'$, and $t \in \mathsf{sn}$ by induction hypothesis, or $r = tx$ and $t \in \mathsf{sn}$ since it is a subterm of $r \in \mathsf{sn}$. A similar argument is needed for the Λ-abstraction.

The case of $(\lambda x^\rho r)s\vec{S}$ is as for untyped lambda calculus (see the proof of Lemma 18): By main induction on $s \in \mathsf{sn}$ and side induction on $r[x := s]\vec{S} \in \mathsf{sn}$ we prove that $(\lambda x^\rho r)s\vec{S} \in \mathsf{sn}$. Therefore, we have to show for every t with $(\lambda x^\rho r)s\vec{S} \to_{\beta\eta} t$ that $t \in \mathsf{sn}$. The only new case compared to the treatment of untyped lambda calculus is $r = r'x$ with $x \notin \mathrm{FV}(r')$ and an η-reduction applied to $\lambda x^\rho.r'x$, leading to $r's\vec{S}$. However, this is not really a new case, since β-reduction of $(\lambda x^\rho.r'x)s$ also yields $(\lambda x^\rho r)s\vec{S} \to_{\beta\eta} r'[x := s]s\vec{S} = r's\vec{S}$.

Finally consider $(\Lambda\alpha r)\sigma\vec{S}$. Show that this term is in sn by induction on $r[\alpha := \sigma]\vec{S} \in \mathsf{sn}$. Assume that $(\Lambda\alpha r)\sigma\vec{S} \to_{\beta\eta} t$. Show that $t \in \mathsf{sn}$. Either $t = (\Lambda\alpha r)\sigma\vec{S}$ with $r \to_{\beta\eta} r'$, hence $r[\alpha := \sigma]\vec{S} \to_{\beta\eta} r'[\alpha := \sigma]\vec{S}$, and we are done by the induction hypothesis. Or $t = (\Lambda\alpha r)\sigma\vec{S}'$ with reduction of one of the terms in $\vec{S}$. Then $r[\alpha := \sigma]\vec{S} \to_{\beta\eta} r[\alpha := \sigma]\vec{S}'$, and again the induction hypothesis applies. Or $t = r[\alpha := \sigma]\vec{S} \in \mathsf{sn}$. Or, finally, $r = r'\alpha$ with $\alpha \notin \mathrm{FTV}(r')$ and $t = r'\sigma\vec{S}$. But $t = (r'x)[\alpha := \sigma]\vec{S} \in \mathsf{sn}$ by assumption. $\square$

Note that $\mathsf{SN} = \mathsf{sn}$ does not hold (see Lemma 39).

Again, it is always possible to produce a saturated set from an arbitrary set M of terms by the saturated closure $\mathsf{cl}(M)$, defined by induction as follows:

- If $r \in M \cap \mathsf{SN}$ then $r \in \mathsf{cl}(M)$.
- If the terms among $\vec{R}$ are in SN then $x\vec{R} \in \mathsf{cl}(M)$.
- If $r[x := s]\vec{S} \in \mathsf{cl}(M)$ and $s \in \mathsf{SN}$ then $(\lambda x^\rho r)s\vec{S} \in \mathsf{cl}(M)$.
- If $r[\alpha := \sigma]\vec{S} \in \mathsf{cl}(M)$ then $(\Lambda\alpha r)\sigma\vec{S} \in \mathsf{cl}(M)$.

Since, again, $\mathsf{cl}(M) \subseteq \mathsf{SN}$ (proved by induction on the definition), it is the least saturated set containing $M \cap \mathsf{SN}$.

Given a saturated set $\mathcal{M}$ and a saturated set $\mathcal{N}$, we construct a saturated set $\mathcal{M} \to \mathcal{N}$: Define

$$\mathsf{S}_x(\mathcal{M}, \mathcal{N}) := \{r \mid \forall s \in \mathcal{M}\; r[x := s] \in \mathcal{N}\},$$

$$\mathsf{I}(\mathcal{M}, \mathcal{N}) := \{\lambda x^\rho r \mid x \in \mathcal{V},\ \rho \in \mathcal{T}_u \text{ and } r \in \mathsf{S}_x(\mathcal{M}, \mathcal{N})\} \quad \text{and}$$

$$\mathsf{E}(\mathcal{M}, \mathcal{N}) := \{r \mid \forall s \in \mathcal{M}, rs \in \mathcal{N}\}.$$

We get the introduction-based definition $\mathcal{M} \to_{\mathrm{I}} \mathcal{N}$ and the elimination-based definition $\mathcal{M} \to_{\mathrm{E}} \mathcal{N}$ of saturated sets:

$$\mathcal{M} \to_{\mathrm{I}} \mathcal{N} := \mathsf{cl}(\mathsf{I}(\mathcal{M}, \mathcal{N})) \text{ and } \mathcal{M} \to_{\mathrm{E}} \mathcal{N} := \mathsf{cl}(\mathsf{E}(\mathcal{M}, \mathcal{N})).$$

We get the same properties as before (compare Lemma 31).

Lemma 44 $\mathsf{I}(\mathcal{M}, \mathcal{N}) \subseteq \mathsf{SN}$, $\mathsf{E}(\mathcal{M}, \mathcal{N}) \cap \mathsf{SN} \in \mathsf{SAT}$, *and* $\mathsf{I}(\mathcal{M}, \mathcal{N}) \subseteq \mathsf{E}(\mathcal{M}, \mathcal{N})$.

Proof

(1) See the proof of Lemma 31.

(2) Check the conditions of saturatedness for $\mathsf{E}(\mathcal{M}, \mathcal{N}) \cap \mathsf{SN}$:

1. Trivial.
2. Let the terms among $\vec{R}$ be in SN and $s \in \mathcal{M}$. Since $s \in \mathsf{SN}$ and $\mathcal{N} \in \mathsf{SAT}$, $x\vec{R}s \in \mathcal{N}$.

3./4. Simply append s and use saturatedness of $\mathcal{N}$.

(3) See the proof of Lemma 31. □

From the lemma, we get $\mathsf{I}(\mathcal{M}, \mathcal{N}) \subseteq \mathcal{M} \to_{\mathrm{I}} \mathcal{N}$, $\mathcal{M} \to_{\mathrm{E}} \mathcal{N} = \mathsf{E}(\mathcal{M}, \mathcal{N}) \cap \mathsf{SN}$ and, due to monotonicity of cl, $\mathcal{M} \to_{\mathrm{I}} \mathcal{N} \subseteq \mathcal{M} \to_{\mathrm{E}} \mathcal{N}$.

Define $\mathcal{M} \to \mathcal{N} := \mathcal{M} \to_{\mathrm{X}} \mathcal{N}$ with $\mathrm{X} \in \{\mathrm{I}, \mathrm{E}\}$. As before, we never use any property depending on this choice but only the following three properties which are valid for both choices and follow immediately from the preceding remarks:

(SAT) $\mathcal{M} \to \mathcal{N} \in \mathsf{SAT}$.

($\to_I$) If $r \in \mathsf{S}_x(\mathcal{M}, \mathcal{N})$ then $\lambda x^\rho r \in \mathcal{M} \to \mathcal{N}$.

($\to_E$) $r \in \mathcal{M} \to \mathcal{N} \wedge s \in \mathcal{M} \Rightarrow rs \in \mathcal{N}$.

Since we now have an explicit $\forall$-introduction and $\forall$-elimination in the term system, we also have to provide an explicit construction of universal quantification on saturated sets. Given a function Φ from SAT to SAT, we define

$$\mathsf{S}_\alpha(\Phi) := \{r \mid \forall \sigma \in \mathcal{T}_u \forall \mathcal{M} \in \mathsf{SAT}\; r[\alpha := \sigma] \in \Phi(\mathcal{M})\},$$

$$\mathsf{I}(\Phi) := \{\Lambda \alpha r \mid \alpha \in \mathcal{V}_T \text{ and } r \in \mathsf{S}_\alpha(\Phi)\} \quad \text{and}$$

$$\mathsf{E}(\Phi) := \{r \mid \forall \sigma \in \mathcal{T}_u \forall \mathcal{M} \in \mathsf{SAT}\; r\sigma \in \Phi(\mathcal{M})\}.$$

We get the introduction-based definition $\forall_I \Phi$ and the elimination-based definition $\forall_E \Phi$ of saturated sets:

$\forall_I \Phi := \mathsf{cl}(\mathsf{I}(\Phi))$ and $\forall_E \Phi := \mathsf{cl}(\mathsf{E}(\Phi))$.

We get similar properties:

Lemma 45 $\mathsf{I}(\Phi) \subseteq \mathsf{SN}$, $\mathsf{E}(\Phi) \cap \mathsf{SN} \in \mathsf{SAT}$, *and* $\mathsf{I}(\Phi) \subseteq \mathsf{E}(\Phi)$.

Proof

(1) Let $r \in \mathsf{S}_\alpha(\Phi)$. Then for $\sigma := \alpha$ and $\mathcal{M} := \mathsf{SN}$, we get $r = r[\alpha := \sigma] \in \Phi(\mathcal{M}) \subseteq \mathsf{SN}$, hence also $\Lambda \alpha r \in \mathsf{SN}$.

(2) Check the conditions of saturatedness for $\mathsf{E}(\Phi) \cap \mathsf{SN}$:

1. Trivial.
2. Let the terms among $\vec{R}$ be in SN, $\sigma \in \mathcal{T}_u$ and $\mathcal{M} \in \mathsf{SAT}$. Since $\Phi(\mathcal{M}) \in \mathsf{SAT}$, $x\vec{R}\sigma \in \Phi(\mathcal{M})$.

3./4. Append σ and use saturatedness of $\Phi(\mathcal{M})$.

(3) Let $r \in \mathsf{S}_\alpha(\Phi)$, $\sigma \in \mathcal{T}_u$ and $\mathcal{M} \in \mathsf{SAT}$. Show that $(\Lambda \alpha r)\sigma \in \Phi(\mathcal{M})$. Since $\Phi(\mathcal{M}) \in \mathsf{SAT}$, if suffices to show $r[\alpha := \sigma] \in \Phi(\mathcal{M})$ which follows by definition of $\mathsf{S}_\alpha(\Phi)$. □

From the lemma, we get $\mathsf{I}(\Phi) \subseteq \forall_I \Phi$, $\forall_E \Phi = \mathsf{E}(\Phi) \cap \mathsf{SN}$ and $\forall_I \Phi \subseteq \forall_E \Phi$. Define $\forall \Phi := \forall_X \Phi$ with $X \in \{I, E\}$. As usual, we never use any property depending on this choice but only:

(SAT) $\forall \Phi \in \mathsf{SAT}$.

($\forall_I$) If $r \in \mathsf{S}_\alpha(\Phi)$ then $\Lambda \alpha r \in \forall \Phi$.

($\forall_E$) $r \in \forall \Phi \wedge \sigma \in \mathcal{T}_u \wedge \mathcal{M} \in \mathsf{SAT} \Rightarrow r\sigma \in \Phi(\mathcal{M})$.

The notion of candidate assignment remains unchanged:

Definition 35 (Candidate assignment) *Any finite set of pairs (written $\alpha : \mathcal{M}$), consisting of a type variable and a saturated set, such that no type variable occurs twice.*

Definition 36 (Strong computability) *Define the saturated set* $\mathsf{SC}^\rho[\Gamma]$ *of* strongly computable terms *w. r. t. type* ρ *and the candidate assigment* Γ *by recursion on* ρ*:*

- $\mathsf{SC}^\alpha[\Gamma] := \begin{cases} \mathcal{M} & \text{if } \alpha : \mathcal{M} \in \Gamma, \\ \mathsf{SN} & \text{otherwise.} \end{cases}$
- $\mathsf{SC}^{\rho\to\sigma}[\Gamma] := \mathsf{SC}^\rho[\Gamma] \to \mathsf{SC}^\sigma[\Gamma]$.
- *Define* $\Phi : \mathsf{SAT} \to \mathsf{SAT}$ *by setting* $\Phi(\mathcal{M}) := \mathsf{SC}^\rho[\Gamma, \alpha : \mathcal{M}]$. *Set* $\mathsf{SC}^{\forall\alpha\rho}[\Gamma] := \forall\Phi$.

Lemma 46 (Coincidence) *If* $\alpha \notin FV(\rho)$ *then* $\mathsf{SC}^\rho[\Gamma, \alpha : \mathcal{M}] = \mathsf{SC}^\rho[\Gamma]$.

Proof Induction on ρ. □

Lemma 47 (Substitution) $\mathsf{SC}^{\rho[\alpha:=\sigma]}[\Gamma] = \mathsf{SC}^\rho[\Gamma, \alpha : \mathsf{SC}^\sigma[\Gamma]]$.

Proof Induction on ρ, using the previous lemma. □

As in Lemma 34, we make use of the simultaneous substitution $r[\vec{x} := \vec{s}]$ of all occurrences of x_i by s_i (for every i, with different variables x_i) in r and moreover of the simultaneous substitution $r[\vec{\alpha} := \vec{\sigma}]$ of all occurrences of α_i by σ_i (for every i, with different variables α_i) in r which both may be defined by recursion on r like the ordinary notions of substitution. We will again use the notation $\vec{x} : \vec{\rho}$ for $x_1 : \rho_1, \ldots, x_n : \rho_n$, $\vec{s} \in \mathsf{SC}^{\vec{\rho}}[\Gamma]$ for $s_1 \in \mathsf{SC}^{\rho_1}[\Gamma], \ldots, s_n \in \mathsf{SC}^{\rho_n}[\Gamma]$, and moreover, $\vec{\alpha} : \vec{\mathcal{M}}$ for $\alpha_1 : \mathcal{M}_1, \ldots, \alpha_m : \mathcal{M}_m$.

Lemma 48 *If* $\vec{x} : \vec{\rho} \vdash r : \rho$*,* $\vec{s} \in \mathsf{SC}^{\vec{\rho}}[\vec{\alpha} : \vec{\mathcal{M}}]$ *and* $\vec{\sigma}$ *is a list of types corresponding to* $\vec{\alpha}$ *then* $r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}] \in \mathsf{SC}^\rho[\vec{\alpha} : \vec{\mathcal{M}}]$.

Proof By induction on $\vec{x} : \vec{\rho} \vdash r : \rho$ simultaneously for every candidate assignment.

(V) $r : \rho = x_i : \rho_i$. Obvious.

($\to_I$) Let $\vec{x} : \vec{\rho} \vdash \lambda x^\rho r : \rho \to \sigma$ thanks to $\vec{x} : \vec{\rho}, x : \rho \vdash r : \sigma$. We have to show that $(\lambda x^\rho r)[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}] \in \mathsf{SC}^\rho[\vec{\alpha} : \vec{\mathcal{M}}] \to \mathsf{SC}^\sigma[\vec{\alpha} : \vec{\mathcal{M}}]$. We may assume that $x \notin \vec{x} \cup \mathrm{FV}(\vec{s})$, and hence

$$(\lambda x^\rho r)[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}] = \lambda x^{\rho[\vec{\alpha}:=\vec{\sigma}]}.r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}].$$

It suffices to show that $r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}] \in \mathsf{S}_x(\mathsf{SC}^\rho[\vec{\alpha} : \vec{\mathcal{M}}], \mathsf{SC}^\sigma[\vec{\alpha} : \vec{\mathcal{M}}])$. So assume $s \in \mathsf{SC}^\rho[\vec{\alpha} : \vec{\mathcal{M}}]$ and show $r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}][x := s] \in \mathsf{SC}^\sigma[\vec{\alpha} : \vec{\mathcal{M}}]$. This follows from the induction hypothesis since, by our assumption,

$$r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}][x := s] = r[\vec{\alpha} := \vec{\sigma}][\vec{x}, x := \vec{s}, s].$$

($\to_E$) This is an immediate consequence of the induction hypothesis and the rule ($\to_E$) for saturated sets.

($\forall_I$) Let $\vec{x} : \vec{\rho} \vdash \Lambda\alpha r : \forall\alpha\rho$ thanks to $\vec{x} : \vec{\rho} \vdash r : \rho$ and $\alpha \notin \mathrm{FV}(\vec{\rho})$. We may assume that $\alpha \notin \vec{\alpha} \cup \mathrm{FV}(\vec{\sigma}) \cup \mathrm{FTV}(\vec{s})$. Therefore, $(\Lambda\alpha r)[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}] = \Lambda\alpha.r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}]$. Let $\sigma \in \mathcal{T}_u$ and $\mathcal{M} \in \mathsf{SAT}$. We have to show that $r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}][\alpha := \sigma] \in \mathsf{SC}^{\rho}[\vec{\alpha} : \vec{\mathcal{M}}, \alpha : \mathcal{M}]$. Since $\alpha \notin \mathrm{FV}(\vec{\rho})$, we may apply the Coincidence Lemma and get $\vec{s} \in \mathsf{SC}^{\vec{\rho}}[\vec{\alpha} : \vec{\mathcal{M}}, \alpha : \mathcal{M}]$. We are done by the induction hypothesis since

$$r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}][\alpha := \sigma] = r[\vec{\alpha}, \alpha := \vec{\sigma}, \sigma][\vec{x} := \vec{s}].$$

($\forall_E$) Let $\vec{x} : \vec{\rho} \vdash r\sigma : \rho[\alpha := \sigma]$ thanks to $\vec{x} : \vec{\rho} \vdash r : \forall\alpha\rho$. By induction hypothesis, $r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}] \in \mathsf{SC}^{\forall\alpha\rho}[\vec{\alpha} : \vec{\mathcal{M}}]$. Set $\mathcal{M} := \mathsf{SC}^{\sigma}[\vec{\alpha} : \vec{\mathcal{M}}]$. Then $r[\vec{\alpha} := \vec{\sigma}][\vec{x} := \vec{s}]\sigma[\vec{\alpha} := \vec{\sigma}] \in \mathsf{SC}^{\rho}[\vec{\alpha} : \vec{\mathcal{M}}, \alpha : \mathcal{M}] = \mathsf{SC}^{\rho[\alpha:=\sigma]}[\vec{\alpha} : \vec{\mathcal{M}}]$ by the Substitution Lemma. □

By setting $s_i := x_i \in \mathsf{SC}^{\rho_i}[\emptyset]$ for $\Gamma = \vec{x} : \vec{\rho}$, and by using $\mathsf{SC}^{\rho}[\emptyset] \subseteq \mathsf{SN}$, we get the following

Theorem 5 (Strong normalization of F) *If* $\Gamma \vdash r : \rho$ *then* $r \in \mathsf{SN}$. □

As an application, we prove:

Lemma 49 (Typed confluence) *If* $\Gamma \vdash r : \rho$, $r \to_{\beta\eta}^* r'$ *and* $r \to_{\beta\eta}^* r''$ *then there is a term* t *such that* $r' \to_{\beta\eta} t$ *and* $r'' \to_{\beta\eta} t$.

Proof Induction on $r \in \mathsf{sn}$.[18] If $r = r'$ or $r = r''$ then the claim is trivial (set $t = r''$ or $t = r'$, respectively). Otherwise, $r \to_{\beta\eta} r_0' \to_{\beta\eta}^* r'$ and $r \to_{\beta\eta} r_0'' \to_{\beta\eta}^* r''$ for some terms r_0' and r_0''. By typed local confluence, there is a term s such that $r_0' \to_{\beta\eta}^* s$ and $r_0'' \to_{\beta\eta}^* s$. By Subject Reduction, $\Gamma \vdash r_0' : \rho$ and $\Gamma \vdash r_0'' : \rho$. Hence, by induction hypothesis for r_0', there is s' such that $r' \to_{\beta\eta}^* s'$ and $s \to_{\beta\eta}^* s'$, hence $r_0'' \to_{\beta\eta}^* s'$. We now apply the induction hypothesis to r_0'' and get the term t with and $r'' \to_{\beta\eta}^* t$ and $s' \to_{\beta\eta}^* t$, hence also $r' \to_{\beta\eta}^* t$. □

Exercise 22 *Show that there is no term* r *such that* $\vdash r : \forall\alpha\alpha$.

8 Monotone Inductive Types

The expressiveness of system F is highlighted by the fact that least pre-fixed points of monotone operators can be represented—even with respect to reduction behaviour. Its main practical consequence arises in the field of program extraction: The computational content of intuitionistic proofs with inductive definitions consists of terms of system F whose normalization yields the objects whose existence has been proved.[19] Later we will see that one also needs to model fixed-points (not only pre-fixed points) in order to get primitive recursion (not only iteration), and those fixed-points are not available in system F as is generally believed and greatly supported by [SU99].

[18] This proof is in essence the proof of Newman's Lemma saying that a locally confluent and strongly normalizing binary relation is confluent.

[19] Unfortunately, because of lack of space, this claim cannot be substantiated in these notes.

8.1 The Example of Continuations

The guiding example for the treatment of inductive types will be the one in [Hof95] treating a classical problem in algorithm design by a non-strictly positive inductive type: The labels of a finite labelled binary tree shall be put into a list breadth-first, i. e., first the root label, then the labels of its children, then the labels of the next layer, ...

The following SML program will be studied in great detail:

```
datatype nat = O | S of nat;
(* natural numbers *)

val one= S O;
val two = S one;
val three = S two;
val four = S three;
val five = S four;
val six = S five;
val seven = S six;
val eight = S seven;
val nine = S eight;
(* example numbers *)

datatype btree = L of nat | N of nat*btree*btree;
(* binary trees *)

val extree = N(one,N(two,L seven,N(three,L five,L four)),N(four,
                                     N(six,L two,L nine),L eight));
(* the example tree *)

datatype list = nil | cons of nat*list;
(* lists of natural numbers *)

datatype cont = D | C of (cont -> list) -> list;
(* non-strictly positive !! *)

fun apply(D,g) = g D |
    apply(C f,g) = f g;
(* definition without recursion but with inversion *)

fun breadth(L x,k) = C(fn(g)=>cons(x,apply(k,g))) |
    breadth(N(x,s,t),k) = C(fn(g)=>cons(x,(apply(k,
                          fn(m)=>g(breadth(s,(breadth(t,m))))))));
(* iteration on the tree argument
   fn(g)=> is the notation for lambda-abstraction of g *)

fun ex(D) = nil | ex(C f) = f ex;
```

```
(* iteration on the datatype cont !! *)

fun breadthfirst t = ex(breadth(t,D));

val result = breadthfirst(extree);

val exlist= cons(one, cons(two, cons(four, cons(seven, cons(three,
            cons(six, cons(eight, cons(five, cons(four, cons(two,
            cons(nine,nil)))))))))));

val ok=(result=exlist);
```

This leads to the following output:

```
Standard ML of New Jersey,
Version 110.0.6, October 31, 1999 [CM; autoload enabled]
-  use("hofmann.sml");
[opening hofmann.sml]
datatype nat = O | S of nat
val one = S O : nat
val two = S (S O) : nat
val three = S (S (S O)) : nat
val four = S (S (S (S O))) : nat
val five = S (S (S (S (S #)))) : nat
val six = S (S (S (S (S #)))) : nat
val seven = S (S (S (S (S #)))) : nat
val eight = S (S (S (S (S #)))) : nat
val nine = S (S (S (S (S #)))) : nat
datatype btree = L of nat | N of nat * btree * btree
val extree = N (S O,N (S #,L #,N #),N (S #,N #,L #)) : btree
datatype list = cons of nat * list | nil
datatype cont = C of (cont -> list) -> list | D
val apply = fn : cont * (cont -> list) -> list
val breadth = fn : btree * cont -> cont
val ex = fn : cont -> list
val breadthfirst = fn : btree -> list
val result = cons (S O,cons (S #,cons #)) : list
val exlist = cons (S O,cons (S #,cons #)) : list
val ok = true : bool
val it = () : unit
-
```

The first questions to raise:

1. Are the results always correct?
2. Does the program terminate for every input tree?

3. Is it efficient? (Compare it with the state-based implementation with a queue.)

4. How can we understand a definition of a function `ex` having the form `ex(C f) = f ex`? The function about to be defined is passed over as an argument in the recursive call!

There are pleasing answers:

1. The program is correct, and can be shown so by suitable inductive arguments.

2. The program terminates since it can be expressed in an extension of system F by fixed-point types, and every term typable in that system is strongly normalizing, to be shown by a straightforward extension of the proof for system F which has been designed so as to facilitate this extension.

3. It is running in linear time like the implementation with a queue. Encodings in system F are extremely unlikely to give linear time. The problem is with the definition of `apply` where `apply(C f,g) = f g` needs to isolate `f` out of `C f`, and which is an instance of inversion.

4. The recursive call can be understood quite well: It is indeed an instance of iteration which can be modeled inside system F. And even much more demanding inductive types can be treated: every monotone inductive type. The proof of monotonicity provides the iteration principle.

The theoretical understanding goes further: The embedding of iteration on monotone inductive types into system F can be read off a careful proof of Tarski's fixed-point theorem stating that a monotone operator on a complete lattice has a least fixed-point (see the explanation in [Mat99b]). Moreover, we only need non-interleaving non-strictly positive fixed-point types to derive full primitive recursion on monotone inductive types because of system F's impredicative capabilities [Mat99a].

Unfortunately, this all can only be addressed in a future version of these lecture notes. The plan is to enlarge this section until it consumes about one third of the total time of the lecture course on lambda calculus, a case for inductive definitions.

References

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In Bezem and Groote [BG93], pages 13–28.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North–Holland, Amsterdam, second revised edition, 1984.

[Bar93] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Tom S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1993.

[BG93] Marc Bezem and J.F. Groote, editors. *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[CDC78] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for λ-terms. *Archive for Mathematical Logic*, 19:139–156, 1978.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII, 1972.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[Hof95] Martin Hofmann. Approaches to recursive datatypes—a case study. 5 pages. Unpublished, April 1995.

[JM99] Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. Submitted to the Archive for Mathematical Logic, 1999.

[Kri93] Jean-Louis Krivine. *Lambda-calculus, types and models*. Masson, Paris and Ellis Horwood, Hemel Hempstead, 1993. English translation of Lambda-calcul, types et modèles, Masson, 1990.

[Loa98] Ralph Loader. Notes on simply typed lambda calculus. Reports of the Laboratory for Foundations of Computer Science ECS-LFCS-98-381, University of Edinburgh, 1998.

[Mat98] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. Doktorarbeit (PhD thesis), University of Munich, 1998. Available via the homepage `http://www.tcs.informatik.uni-muenchen.de/~matthes/`.

[Mat99a] Ralph Matthes. Monotone (co)inductive types and positive fixed-point types. *Theoretical Informatics and Applications*, 33(4/5):309–328, 1999.

[Mat99b] Ralph Matthes. Tarski's fixed-point theorem and lambda calculi with monotone inductive types. To appear in: Benedikt Löwe and Florian Rudolph, Foundations of the Formal Sciences, Refereed Papers of a Research Colloquium, Humboldt-Universität zu Berlin, May 7-9, 1999.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. The MIT Press, 1996.

[SU99] Zdzisław Spławski and Paweł Urzyczyn. Type Fixpoints: Iteration vs. Recursion. *SIGPLAN Notices*, 34(9):102–113, 1999. Proceedings of the 1999 International Conference on Functional Programming (ICFP), Paris, France.

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Tai75] William W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloquium Boston 1971/72*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer Verlag, 1975.

[Urz96] Paweł Urzyczyn. Positive recursive type assignment. *Fundamenta Informaticae*, 28(1–2):197–209, 1996.

[Wel94] Joe B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science*, pages 176–185. IEEE Computer Society Press, 1994.

[Wel99] Joe B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.