Thesis for the degree of Doctor of Philosophy

Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion

David Wahlstedt

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden, 2007

Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion
David Wahlstedt

ISBN 978-91-7291-979-2
ISSN 0346-718X

Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 2660

Technical report 34D
Department of Computer Science and Engineering
Research group: Programming Logic

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Telephone +46 (0)31-772 1000

Typeset by the author using LaTeX, and Paul Taylor's `proof.sty`
Printed at the Department of Computer Science and Engineering
Göteborg, 2007

Abstract

We present a variation of Martin-Löf's logical framework with $\beta\iota$-equality, extended with first-order parameterized algebraic data types and recursive pattern-matching definitions. Our contribution is a proof of normalization for the proposed system, from which we obtain decidable type-correctness. Our result holds under the assumption that the call relation of the recursive definitions is well-founded.

Recursive definitions can be read as intuitive specifications, which makes it easier to understand their intended meaning, compared to definitions that use only pre-defined recursion operators, called elimination rules in type theory. Pattern-matching definitions can be seen as the underlying mechanism with which we describe elimination rules. The arguments used to justify recursively defined elimination rules are essentially the same as those justifying pattern-matching definitions. The use of pattern-matching takes the proof system closer to the look and feel of a programming language like Haskell (Peyton Jones, 2003) or ML (Milner *et al.*, 1990; Leroy *et al.*, 2004).

We use the *Size-Change Principle for Program Termination* (C.S. Lee, N.D. Jones, A. Ben-Amram 2001) to establish that the recursive definitions have a well-founded call relation. The size-change criterion subsumes many known characterizations of terminating recursions, including primitive recursion and lexicographical ordering, but it also deals transparently with permuted arguments and mutual recursion. When instantiating the size-change relation for this criterion to be constructor decomposition, it corresponds closely to what we could call *well-founded structural recursion*.

Keywords: type theory, dependent types, lambda-calculus, logical framework, new constants, stratification, type checking, decidability, normalization, reducibility, well-founded recursion, size-change termination, pattern-matching, term rewriting.

Acknowledgments

First of all, I would like to thank my supervisor Thierry Coquand. His help was essential for overcoming some of the most difficult technical problems in this dissertation. I also want to thank Thierry for his open-mindedness and his inspiring, thoughtful and curious mentality. Our discussions have been a great source of inspiration and insight.

I want to thank my examiner, Bengt Nordström, who has been supporting me constantly during these years. Discussions with Bengt also contributed to the simplification of the type-checking relation presented here. I want to thank my opponent Ralph Matthes for his effort, having conducted an extremely careful reading of my dissertation. His comments have improved the quality of this work substantially. For inspiration and discussions,[1] I want to thank Per Martin-Löf and Neil Jones. For fruitful technical discussions and feedback, I would like to thank Ulf Norell, Andreas Abel and Michael Hedberg. For comments on the language and writing in general, I want to thank Björn Bjurling, Andy Fugard, Peter Gammie, Thomas Hallgren and David Sands.

I would like to thank the members of my PhD advisory committee, David Sands and Aarne Ranta, and our vice-prefect Reiner Hähnle, who have both encouraged me and helped me with feedback on my study plan during my years as a PhD student. I want to thank my former office mates Josef Svenningsson, Kristofer Johannisson, Nils Anders Danielsson and Fredrik Lindblad, for having got to know you, as well as all the employees at Chalmers Department of Computer Science and Engineering, for providing a stimulating atmosphere.

Many thanks to Agneta Flock, Per Carlsson, Ida Strindberg, Vera Strindberg, Karin Hårding, Tero Herkönen, Aare Mällo and Ing-Marie Andrén for their help with our children. I also want to thank Urban Wahlstedt, Ethel Tillnert, Maggie Strindberg, Lina Persdotter, Ingrid Flock, May Wahlstedt, and my friends, who have not been mentioned here, for their support.

I want to thank Anneli Pihlgren for all her sacrificed working time, support and patience. I thank Jonatan Pihlgren and Ossian Wahlstedt, for being such wonderful children.

[1] Even if only for a few occasions, this has been very important.

Contents

Chapter 1

Introduction

Constructive type theory combines two concepts: Firstly, it is a formalism in which we can express mathematical notions, assert propositions and construct proofs. Secondly, it can be seen as a dependently typed[1] functional programming language with a syntax similar to Haskell (Peyton Jones, 2003) and ML (Milner *et al.*, 1990; Leroy *et al.*, 2004). In Haskell and ML, the strong static typing helps to eliminate run-time errors of a kind that frequently occur in programs written by humans. This makes programs more reliable, and also saves time from debugging. With dependent types we have also the ability, if we desire, to express semantic properties about the programs. In type theory, one can check mechanically if a program fulfills a given specification or not. From the mathematical point of view, one can check the correctness of proofs.

A recent example of a type theoretic correctness proof of a complex piece of software can be found in Leroy (2006) and Blazy *et al.* (2006), who proved formally the correctness of an optimizing compiler for a C-like programming language. The proof is based on the Coq (The Coq Development Team, 2006) proof assistant. Another important result that shows the applicability of type theory is the proof of the four colour theorem by Gonthier (2004). The problem was conjectured in 1852 by Francis Guthrie, and many failed attempts were made until it was proved by Appel & Haken (1976), assisted by computer. However, the proof involves a *billion* case distinctions which each had to be verified. A computer program written in assembly code checked the cases, but the program itself was not formally verified. Moreover, thousands of cases also had to be verified manually. Therefore the reliability of the proof have been questioned. As opposed to Appel and Haken's work, Gonthier's proof has been completely formalized in type theory, giving it a level of reliability far beyond what could be obtained by human inspection. An interesting point was made concerning their proof—the success was largely due to the fact that they, quoting Gonthier, *approached the Four Colour Theorem mainly as a programming problem, rather than a formalization problem.*

[1] The notion of dependent type is explained in the technical background, Section 1.2.5, page 11.

In this dissertation we will present a system influenced by Martin-Löf's type theory (1972; 1984), where we emphasize the programming style using terminating recursive definitions by pattern-matching on algebraic data types to represent proofs by induction and computation. We give sufficient conditions for the correctness and decidability of type-correctness in this system. This approach makes the development of type theoretic definitions look and feel like functional programming.

1.1 Historical background

We present a brief historical overview of the developments preceding constructive type theory. We will do so focusing on a few, particularly important concepts that we want to emphasize.

1.1.1 Brief history of recursion schemes

An early example of recursive computation can be found in *The Sand Reckoner* by Archimedes c. 250 - 212 BC (Cf. Newman, 1956) who, before the notion of exponentiation was established, showed the existence of a number bigger than the number of sand grains that would be needed to fill the universe, as known from that time, approximated as a sphere centered in the sun, spanning the fixed stars. He calculated this number to be roughly 10^{64}. To exceed this, he constructed a notation system for numbers as a double recursion scheme:[2]

$$\begin{aligned} h_0(x) &= 1 \\ h_{n+1}(0) &= h_n(a) \\ h_{n+1}(x+1) &= a \cdot h_{n+1}(x) \end{aligned}$$

where $a = 10.000^2$, or "a myriad myriads", the largest named number at that time. From the scheme he constructed the number $h_a(a)$, which equals $10^{(8 \cdot 10^{16})}$, or ten to the power of eighty quadrillions !

Recursive functions were first systematically studied in modern mathematics by Dedekind, Hilbert, Herbrand (1931) and others. Dedekind (1888), took an important step towards today's notion of primitive recursion. The following theorem can be traced back to Dedekind's result:[3]

Assume $a_0 \in A$ *and* $F : \mathbb{N} \times A \to A$.
Then there is a unique $H : \mathbb{N} \to A$ *such that*

$$\begin{aligned} H(0) &= a_0 \\ H(n+1) &= F(n, H(n)) \text{ for all } n \in \mathbb{N} \end{aligned}$$

[2] In the original version a verbal presentation was given, whereas here we give a translation into modern notation taken from Odifreddi (Fall 2006).

[3] This example is taken from Aczel & Rathjen (1997). Dedekind's theorem, which was rather an iteration theorem, was close to this one, but there F had only one parameter.

The above system of equations is called a *schema* for primitive recursion. If A is $\mathbb{N}^k$, the schema characterizes the class of primitive recursive functions on natural numbers. The theorem can be used to justify recursive definitions of new functions—once the defining equations obey the schema, one can be sure that there exists one and only one such function.

Hilbert (1925) defined *functionals* over natural numbers using primitive recursion—functions that may take functions as arguments and may return a function, what we call higher-order functions in functional programming today. There he formulated the following examples defining the functionals[4] ι and φ

$$\begin{aligned} \iota(f,a,1) &= a \\ \iota(f,a,n+1) &= f(a,\iota(f,a,n)) \end{aligned}$$

$$\begin{aligned} \varphi_1(a,b) &= a+b \\ \varphi_{n+1}(a,b) &= \iota(\varphi_n,a,b) \end{aligned}$$

as instances of the general schema

$$\begin{aligned} \rho(\mathfrak{g},\mathfrak{a},0) &= \mathfrak{a} \\ \rho(\mathfrak{g},\mathfrak{a},n+1) &= \mathfrak{g}(\rho(\mathfrak{g},\mathfrak{a},n),n) \end{aligned}$$

The first functional, in its applied form $\iota(f,a,n)$, takes some function f and iterates it n times on a. The second schema, defining $\varphi_n(a,b)$, gives rise to a sequence of functions of two variables $\varphi_1,\ \varphi_2,\ \varphi_3,\ \ldots,$ where the first one is addition, the second one multiplication, the third one exponentiation, the fourth one is the so-called *tower function*, which is the b-fold exponentiation of a:

$$\varphi_4(a,b) = a^{(a^{(a^{\cdot^{\cdot^{a}}})})}$$

iterated b times. Ackermann (1928) proved that $\varphi_a(a,a)$ could not be defined with primitive recursion, by showing that it grows faster than any primitive recursive function.

Herbrand (1931) presented schemata defining first-order recursive functions as systems of first-order recursive equations. His formulation of recursive functions did not restrict the functions to be primitive recursive. Herbrand's version of Hilbert's and Ackermann's function, was written as follows:

$$\begin{aligned} \varphi(n+1,a,b) &= \varphi(n,a,\varphi(n+1,a,b-1)) \\ \varphi(n,a,1) &= a \\ \varphi(0,a,b) &= a+b \end{aligned}$$

In Herbrand's scheme one could introduce any number of new functions $f_i(x_1,x_2,\ \ldots\ ,x_{n_i})$, using functions already defined, and possible occurrences of f_i in the right-hand sides of the defining equations, provided, quoting Herbrand:

[4]The notation is taken from the translation of Hilbert's article in van Heijenoort (1977).

they make the actual computation of the $f_i(x_1, x_2, \ldots, x_{n_i})$ *possible for every given set of numbers, and it is possible to prove intuitionistically that we obtain a well-determined result.*

Gödel (cf. Feferman, 1986) continued to study recursive functions, influenced by Herbrand and Hilbert. In Gödel's version of Herbrand's equations, the restriction mentioned above about the obligation to prove intuitionistically the totality of the function to be defined was dropped. An improvement of Herbrand's equations can be found in McCarthy (1963a), who imposed restrictions to enforce a unique solution.

Kleene (1938) introduced the notion of *partial* functions. Church (1941) introduced the lambda-calculus as a model for computations. The fact that the Herbrand/Gödel recursive functions were extensionally equivalent to Church's lambda-calculus—what we now call general recursive functions—was discovered later, but we will not use the notion of general recursive functions in our work.

1.1.2 Reasoning about inductively defined objects

McCarthy (1962) pioneered reasoning with programs as mathematical objects and initiated the investigation of automatic proof-checking. Curry (Curry & Feys, 1958) and McCarthy (1963b), introduced the notion of abstract syntax. Landin (1964, 1966) suggested how to use inductive data types to represent programming languages and computing machines. Burstall (1969), suggested how to use structural induction to prove properties about syntactical objects, and he also introduced the case-notation. Their proofs had the shape of recursive programs, but proofs were not considered as objects in their formalism.

1.1.3 Martin-Löf's type theory

Martin-Löf's intuitionistic type theory (Martin-Löf, 1972, 1984) has brought together the concepts we presented in the previous sections. Martin-Löf's type theory has been primarily intended for "mathematical logic as foundations (or philosophy) of mathematics" (Martin-Löf, 1984). However, as described in Nordström, Petersson and Smith (1990), Martin-Löf's type theory also provides a formalism in which one can reason about programs and their specifications. One can state and prove properties about both recursive data types and recursive functions, and in the system, proofs are objects represented in the formalism.

Sets (also known as *algebraic data types*) are specified using construction schemes, called *introduction rules*, together with computation rules, called *elimination rules*, specified as structural primitive recursion schemes for the corresponding data types. The structure of a proof in Martin-Löf's type theory follows that of Gentzen's system of natural deduction (Prawitz, 1965; Gentzen, 1969), built up of introduction rules and elimination rules. The ideas of Brouwer, Heyting and Kolmogorov (cf. Curry & Feys, 1958) and later Curry and Howard, in 1969 (Howard, 1980) of how to interpret propositions as types and proofs as programs have found their realization in Martin-Löf's type theory. This was

made possible through the use of a dependently typed lambda-calculus (as opposed to the simply typed lambda-calculus, which is sufficient for representing propositional logic). Previous use of dependent types can be found in de Bruijn (1968), in the AUTOMATH proof system. Dependent types also appeared in Curry's work already in 1956 (cf. Seldin, 2002).

1.2 Technical background

We will here give a short introduction to some basic concepts of what we could call a formal system for Martin-Löf-style type theory, following the syntax of the logical framework, which can be found in Nordström *et al.* (1990). Henceforth we will refer to "type theory" as the system we present here, although a wide range of type theories actually exists. Type theory has many similarities to strongly statically typed functional programming languages, for instance Haskell (Peyton Jones, 2003), and we will show how concepts in type theory correlate to concepts in functional programming.[5]

1.2.1 Types, elements and judgements

The syntactical expressions of type theory are thought of as trees, rather than strings of symbols. There are two categories of expressions: types and elements. The notion of *type* in type theory plays the role of sets in set theory. A type is a collection of objects–its *elements*. An important difference to set theory is in the interpretation of functions. Instead of explaining a function $f : A \to B$ as a set of pairs taken from $A \times B$, one considers f as an *algorithm* that for every object a in A, uniquely *computes* an object $f\ a$ in B.[6]

Typing judgements

We can make *judgements* involving types and their elements. Judgements should not be confused with propositions: we can make the judgement "A is a type", when we have the reason why A should qualify as a valid type expression. We will write this as $\vdash A$. Having established that A is a type, we can make the judgement "a is an element of A", which will be written $\vdash a : A$. When we introduce new types, we introduce simultaneously the rules of how to form their objects. When we refer to objects, we also have to mention which types they belong to.

[5] It is therefore an advantage if the reader is familiar to a similar language.

[6] Henceforth we will use juxtaposition notation (introduced by Schönfinkel (1924), and now used in functional programming) for function application, writing $f\ x$ instead of $f(x)$, which could be confused with the conception of f as an object having x as a free variable occurring in it. However, if the argument itself would be an application $g\ x$, we will write $f(g\ x)$, when necessary. Ideally, brackets should only serve as a notation for grouping objects together.

Inference rules and derivations

There are *inference rules* governing what conclusions one may make from previously established judgements, presented in the form

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

where $J_1, \ \ldots \ , J_n$ are the *premises* and J is the *conclusion* of the rule. If $n = 0$, the list of premises is empty, and the rule is seen as an *axiom*. Inference rules can be nested in each other to form *derivations* D, of the form

$$\frac{D_1 \quad \ldots \quad D_n}{J}$$

where a valid derivation must be built exclusively from the given inference rules, eventually ending with axioms.

1.2.2 Ground types and computations

The basic types considered in type theory are called *ground types* (also called *sets*, *base types*, *small types* or *canonical types*). They correspond to *algebraic data types* in functional programming. Examples are booleans, natural numbers, lists, pairs and sums.

Introduction rules

For a type expression A to qualify as a ground type, we must give an inductive definition consisting of a set of rules called *introduction rules*, that specify how the objects in the type A are built up syntactically.[7] The introduction rules correspond to the constructors of the corresponding data type in functional programming.

Elimination rules

Having given the introduction rules of A, one must give a set of rules called *elimination rules* of A, one for each of the given introduction rules. These play a two-fold role: in the proof system point of view, they serve as *induction principles*, prescribing how to prove properties about arbitrary objects in A, and from the programming point of view, they serve as generic recursion operators, similar to *folds* in functional programming.

To simplify the presentation, we will not explain the elimination rules until later. We will assume that computation rules can be defined directly by giving their recursion schemes. In Martin-Löf's presentations of type theory, one has to give the elimination rules first, and then one may introduce other computation schemes, provided that they can be interpreted in terms of elimination rules.

[7] The construction of objects is similar to the description of data structures of Landin (1964).

Lambda terms

In addition to the elimination rules, there is a built-in notion of computation based on substitution. The mechanisms to perform these computations are based on the reduction rules of *lambda-calculus*. An *abstraction* is a term of the form $\lambda x.t$, sometimes written $(x)t$. It denotes a nameless function, corresponding to the mathematical notation $x \mapsto t$, mapping its argument x to the term t, in which x may occur free. The actual value, u, that x is assigned to is substituted for x in t, the result is written $t[u/x]$. For instance $\lambda x.x + x$ is the function that doubles its argument, and so $(\lambda x.x + x)\ 5$ reduces to $(x + x)[5/x]$, which is the same as $5 + 5$, which is eventually computed to 10.

Canonical and non-canonical forms

For objects in ground types, we distinguish between their *canonical* and *non-canonical* forms. A canonical object in A must have the shape as it appears in the conclusion in one of the introduction rules of A. In functional programming this corresponds to that the object is in head constructor form. A non-canonical object may be a variable or an object that is not yet computed to canonical form. For instance from the previous example, $(\lambda x.x + x)\ 5$ is non-canonical, whereas the result 10 is canonical.

From the computation rules we can also extract *equality* rules, prescribing when two arbitrary objects in the given type are computed to the same canonical form.

1.2.3 Introducing new sets

In the logical framework presentation of type theory, the type Set is the type of ground types. That A is an object of Set is expressed by making the judgement $\vdash A : \text{Set}$. The objects A can be thought of as codes for ground types. Attached with the type Set, there is also an operation El, that converts a set-code into a type. This is expressed by the inference rule

$$\frac{\vdash A : \text{Set}}{\vdash \text{El}\ A}$$

We can think of Set as a universe of sets, restricted to the sets we have described so far.

Booleans

Let us introduce the set of booleans, Bool. Having introduced Bool as a set-constructor, we can infer that El Bool is a well-formed type:

$$\frac{\vdash \text{Bool} : \text{Set}}{\vdash \text{El Bool}}$$

Elements of this type have canonical forms 'true' and 'false', and so we give the introduction rules:

$$\frac{}{\vdash \text{true} : \text{El Bool}} \qquad \frac{}{\vdash \text{false} : \text{El Bool}}$$

We can define boolean functions, conjunction and negation, for instance,

$$\begin{aligned}
&\text{and} : \text{El Bool} \to \text{El Bool} \to \text{El Bool}\\
&\text{and true } x = x\\
&\text{and false } x = \text{false}\\
&\\
&\text{not } : \text{El Bool} \to \text{El Bool}\\
&\text{not true} = \text{false}\\
&\text{not false} = \text{true}
\end{aligned}$$

Using the inference rule

$$\frac{\vdash f : A \to B \quad \vdash a : A}{\vdash f\ a : B}$$

we can derive that, for instance, 'and true false' is a boolean expression.

$$\dfrac{\dfrac{\vdash \text{and} : \text{El Bool} \to \text{El Bool} \to \text{El Bool} \quad \vdash \text{true} : \text{El Bool}}{\vdash \text{and true} : \text{El Bool} \to \text{El Bool}} \qquad \vdash \text{false} : \text{El Bool}}{\vdash \text{and true false} : \text{El Bool}}$$

Notation 1.2.1. In the following examples we will drop applications of El, in order to make the presentation lighter, since they can be inferred from their context.

Natural numbers

Here follows an example of an *inductively defined* data type, the set of natural numbers:

$$\frac{}{\vdash \text{Nat}} \qquad \frac{}{\vdash 0 : \text{Nat}} \qquad \frac{}{\vdash \text{s} : \text{Nat} \to \text{Nat}}$$

In the second introduction rule, the constructor s : Nat → Nat can be seen as a function, that given an object n in Nat returns a new object s n in Nat. We can define addition, and introduce the constant $(+)$.[8]

$$\begin{aligned}
&(+) : \text{Nat} \to \text{Nat} \to \text{Nat}\\
&x + 0 = x\\
&x + (\text{s } y) = \text{s } (x + y)
\end{aligned}$$

We can combine the sets we have introduced to define a boolean function on natural numbers, for instance

$$\begin{aligned}
&\text{odd} : \text{Nat} \to \text{Bool}\\
&\text{odd } 0 = \text{false}\\
&\text{odd } (\text{s } 0) = \text{true}\\
&\text{odd } (\text{s } (\text{s } x)) = \text{odd } x
\end{aligned}$$

[8] We will use brackets in declarations of infix operators.

Parameterized data types

Now consider an inductively defined *parameterized* data type, the set of lists of a given set.

$$\begin{aligned}&\text{List} \ : \text{Set} \rightarrow \text{Set}\\ &[] : \text{List}\ A\\ &(::) : A \rightarrow \text{List}\ A \rightarrow \text{List}\ A\end{aligned}$$

Note that the constructor is polymorphic in its type argument A, and this is inferred from the context. We can infer that 1::2::3::[] belongs to the type List Nat, and true::false::true::[] belongs to the type List Bool. We can implement the polymorphic length function on lists as follows:

$$\begin{aligned}&\text{length} \ : (A : \text{Set}) \rightarrow \text{List}\ A \rightarrow \text{Nat}\\ &\text{length}\ A\ [] = 0\\ &\text{length}\ A\ (a{::}as) = \text{s}\ (\text{length}\ A\ as)\end{aligned}$$

This is a simple example of a polymorphic function using type parameters.

1.2.4 Propositions as types

A key concept in Martin-Löf's type theory is the *propositions-as-types* interpretation, as we have mentioned previously. We identify a proposition with the set of its proofs, so that an object in a set will correspond to a proof of a proposition. The fact that A is a proposition, corresponds to the fact that the ground type A is well-formed. To prove *directly* that A is a true proposition corresponds to building a canonical object in the type A. An *indirect* proof of A, corresponds to a non-canonical object in the type A.

A proposition A is represented by a type whose structure depends on the form of A. For instance, the false proposition is represented by the empty type $\bot$, the proposition with no proofs. A direct proof of a conjunction $A \wedge B$ is a pair (a, b) such that a is an object in the type representing A and b is an object in the type representing B. A proof of an implication $A \supset B$ is represented by a function $f : A \rightarrow B$, that given an object a of the type representing A produces a proof $f\ a$ that can be computed to an object in the type representing B.

The proof a of A is called a *proof term*. Each proof term can be computed to a unique *normal form*, an expression that cannot be further computed. From a normal term and a type it is possible to reconstruct the derivation. This derivation corresponds to a natural deduction proof, as we will exemplify below.

Atomic propositions

As we mentioned above, since a proposition is interpreted as the set of its proofs, consequently we interpret the false proposition as the empty set, *without introduction rules*.

$$\frac{}{\vdash \bot}$$

thus, there will be no way to give a canonical proof of the absurdity. We interpret the trivially true proposition by the singleton type, with one introduction rule:

$$\frac{}{\vdash \top} \qquad \frac{}{\vdash \text{unit} : \top}$$

The negation of A, written $\neg A$, is usually interpreted in type theory as a shortcut of the function type $A \to \bot$, so that asserting A would give us a method to prove the absurdity.

Logical connectives

Using parameterized data types we can encode parameterized propositions, or in other words, connectives. For instance we can consider the type of pairs as the type theoretic interpretation of conjunction:

$$\frac{}{\vdash (\times) : \text{Set} \to \text{Set} \to \text{Set}} \qquad \frac{}{\vdash (,) : A \to B \to A \times B}$$

We define its projection functions (corresponding to elimination rules in Gentzen's natural deduction):

$$\begin{aligned} &\text{fst} \ : (A : \text{Set}) \to (B : \text{Set}) \to A \times B \to A \\ &\text{fst} \ A \ B \ (a, b) = a \end{aligned}$$

$$\begin{aligned} &\text{snd} \ : (A : \text{Set}) \to (B : \text{Set}) \to A \times B \to B \\ &\text{snd} \ A \ B \ (a, b) = b \end{aligned}$$

We will show a proof in Gentzen's system of natural deduction of the propositional tautology $A \wedge B \to B \wedge A$, and then show how this corresponds to the type derivation of a proof term in type theory.

$$\dfrac{\dfrac{\dfrac{[A \wedge B]}{B} \quad \dfrac{[A \wedge B]}{A}}{B \wedge A}}{A \wedge B \to B \wedge A}$$

The corresponding derivation in type theory will look as follows:[9]

$$\dfrac{\dfrac{\dfrac{x : A \times B \vdash x : A \times B}{x : A \times B \vdash \text{snd} \ x : B} \quad \dfrac{x : A \times B \vdash x : A \times B}{x : A \times B \vdash \text{fst} \ x : A}}{x : A \times B \vdash (\text{snd} \ x, \ \text{fst} \ x) : B \times A}}{\vdash \lambda x.(\text{snd} \ x, \ \text{fst} \ x) : A \times B \to B \times A}$$

In this example $x : A \times B$ occurred as an assumption to the left of the turnstile, and this was discharged in leaves of the tree.

[9] We omit the type arguments A : Set and B : Set, and skip some derivation steps of function application to make the derivations match. In Martin-Löf's polymorphic type theory the correspondence is more direct, cf. the first part of Nordström *et al.* (1990).

1.2.5 Dependent function types

The concept of dependent types is a natural extension of simple types in order to adapt the propositions-as-types correspondence in the presence of quantifiers. That a type of a function is *dependent*, means that the result type of the function is determined by the *value* of its argument. The dependent function type is often written $(\Pi\ x \in A)B(x)$, analogous to the set theoretical notation $\Pi_{x \in A} B_x$ for the Cartesian product of a family of sets B_x indexed by elements x of A. Alternatively it can be written with arrow notation $(x : A) \to B$, or juxtaposition notation $(x : A)B$. The expression B denotes a type-valued function that specifies how the range depends on the argument. For each $a : A$, the range is obtained by computing $B\ a$.

To give a concrete example of a dependent type, we introduce the propositional function T, that assigns a proposition for a given boolean argument.

$$\begin{aligned} &\mathrm{T} : \mathrm{Bool} \to \mathrm{Set} \\ &\mathrm{T\ true} = \top \\ &\mathrm{T\ false} = \bot \end{aligned}$$

We can make assertions about boolean objects, for instance 'T(odd 5)' will be a true proposition, since the boolean function will return 'true' in this case, and the proof object will be 'unit'. On the contrary, the assertion 'T(odd 0)' will be a false proposition, with no proof.

Quantification

By analogy with Church's higher-order logic (cf. Andrews, Fall 2006) which has a polymorphic constant $\forall : (\alpha \to o) \to o$, one introduces in type theory a constant

$$\Pi : (A : \mathrm{Set}) \to (A \to \mathrm{Set}) \to \mathrm{Set}$$

The type theoretical counterpart of $\forall x P(x)$ is then interpreted as the dependent function space $\Pi\ A\ F$, with $A : \mathrm{Set}$ and $F : A \to \mathrm{Set}$. A proof of such a proposition is a function $f : \Pi\ A\ F$. The application $f\ a$ for $a : A$ corresponds to "forall elimination" in Gentzen's system, and F is interpreted as a propositional function, so $f\ a$ is a proof of the proposition $F\ a$.

A very simple example[10] involving quantification over booleans is

$$\Pi\ \mathrm{Bool}\ (\lambda x.\mathrm{T}\ (\mathrm{not}\ (\mathrm{and}\ x\ (\mathrm{not}\ x))))$$

where the proof object is 'λx.unit'. An arbitrary boolean will cause the expression T (not (and x (not x))) to evaluate to $\top$.

Analogous to the interpretation of Cartesian product, for the set theoretical notion of disjoint union of the family $\{B(x)\}_{x \in A}$, written $\Sigma x : A.B(x)$, one introduces the dependent sum using the constant

$$\Sigma : (A : \mathrm{Set}) \to (A \to \mathrm{Set}) \to \mathrm{Set}.$$

[10] Recall the previously given computation rules of boolean negation and conjunction, Section1.2.3, page 8.

The type theoretical counterpart of $\exists x P(x)$ is interpreted as the type denoted by $\Sigma\ A\ F$ of dependent *pairs*. A canonical proof of such a proposition is a pair $(a, b) : \Sigma\ A\ F$, such that $a : A$ and $b : F\ a$ holds.

Another very simple example, claiming the existence of a non-odd number,

$$\Sigma\ \text{Nat}\ (\lambda x.\text{T}\ (\text{not}\ (\text{odd}\ x)))$$

has a proof object $(0, \text{unit})$, because the expression 'T (not (odd 0))' evaluates to '$\top$'.

This is an instance of the important *existence property* of constructive mathematics, that from a proof of Σ Nat P, where P is a property of natural numbers, we can effectively compute a number n such that $P\ n$ holds. This idea can be traced back to the constructive interpretation of existence by Kolmogorov (1932). We compute the proof object $p : \Sigma$ Nat P to the canonical form (n, q) giving us the so-called *witness* n and the proof q, of $P\ n$.

Here follows a more involved example, that strong existence implies weak existence.

$$\Sigma\ A\ P \to \neg(\Pi\ A\ (\lambda x.\neg(P\ x)))$$

The proof term in this case would be $\lambda h.\lambda f.f(\text{fst}\ h)(\text{snd}\ h)$. Given the first argument $h : \Sigma\ A\ P$, it expects the next argument $f : \Pi\ A\ (\lambda x.\neg(P\ x))$, and then the task is to return an element in the empty type. This can be done under the assumptions made about the given arguments. We can compute h to canonical form of a pair (a, p), where $a : A$ and $p : P\ a$ will be obtained from 'fst h' and 'snd h' respectively. The function f is applied to a, yielding an object g in $\neg(P\ a)$. Then since g is a function in $P\ a \to \bot$, we can apply it to p and obtain the desired element in the empty set, the contradiction.

Elimination rules

An elimination rule for type A, is a generic method, that given a set-valued function F, produces an object in $F\ a$ for an arbitrary object a in A. In the special case when F is a propositional function, the elimination rule serves as an induction principle. By means of case distinction, constructor decomposition and recursion, the computation scheme can exhaust all the possible constructions of objects in the given type.

As a first minimal example, we give an elimination rule for Bool, by introducing the constant

$$\text{boolElim} : (F : \text{Bool} \to \text{Set}) \to F\ \text{true} \to F\ \text{false} \to (b : \text{Bool}) \to F\ b$$

with the defining equations

$$\begin{aligned} \text{boolElim}\ F\ p_1\ p_2\ \text{true} &= p_1 \\ \text{boolElim}\ F\ p_1\ p_2\ \text{false} &= p_2 \end{aligned}$$

This can be used to implement the usual "if-then-else" construction, but here with dependent types, enabling the two branches to have different types if desired.

We give also an example of elimination constant for an inductive type, the natural numbers:

$$\begin{aligned}
&\text{natrec} : (F : \text{Nat} \to \text{Set}) \to \\
&\qquad (z : F\ 0) \to \\
&\qquad (f : (n : \text{Nat}) \to F\ n \to F\ (\text{s}\ n)) \to \\
&\qquad (n : \text{Nat}) \\
&\qquad\ \to F\ n \\
&\text{natrec}\ F\ z\ f\ 0 = z \\
&\text{natrec}\ F\ z\ f\ (\text{s}\ m) = f\ m\ (\text{natrec}\ F\ z\ f\ m)
\end{aligned}$$

The base case is z, and the step function is f. We can use natrec to define equality on numbers, as well as computations like addition and multiplication. Addition by recursion on the second argument can be implemented by the following term:

$$\text{add} =_{def} \lambda x.\lambda y.\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ y$$

1.3 Adding new constants in type theory

1.3.1 Towards a general formulation

A concept often stressed in Martin-Löf's type theory, is that the system is *open*, that one may extend the theory gradually as new ideas evolve. New types and constants can be added as long as they can be justified and understood in a way that complies with the fundamental principles of type theory. A natural direction of investigation is then to try formulating a scheme prescribing how to introduce these extensions systematically, preserving the desired meta-theoretic properties.

In Martin-Löf's intuitionistic theory of iterated inductive definitions (1971), a general formulation of inductive definitions was given. This system was a predecessor of type theory, and did not have dependent types. In Martin-Löf (1971) and in his later formulations, fixed theories have been given, along with statements made that the system can be extended if desired. A scheme for inductive definitions in extensional type theory was presented by Constable & Mendler (1985). For intensional type theory, a general formulation of inductive types for Coquand's Calculus of Constructions (1985) has been given by Pfenning & Paulin-Mohring (1990) using impredicative encodings. This formulation needed no extra constants to be introduced, but a drawback was that the recursion operators were computationally inefficient. In 1988 (Coquand & Paulin-Mohring, 1990) gave an extension formulated for the Calculus of Constructions with inductive types, influenced by Martin-Löf's type theory. General formulations of predicative intensional Martin-Löf type theory were given implicitly by examples in Nordström *et al.* (1990), and there Martin-Löf's logical framework was also presented. It was shown how new constants could be declared in the framework, along with derivations about them. Dybjer (1994, 2000) gave a general

formulation, of *inductive-recursive* definitions in Martin-Löf's type theory, influenced by a general formulation of inductive definitions of Backhouse (1986); Backhouse & Chisholm (1989). Dybjer's notion of induction-recursion made explicit the mutual dependency between induction and recursion in definitions, that was inherent in Martin-Löf's type theory.

Considering a computer based proof system for type theory with decidable proof correctness, we need a systematic way of extending a theory. In Section 5.2 we will show a decision procedure to ensure valid extensions of our system.

1.3.2 A method using elimination constants

As we have mentioned, for each new ground type that we introduce, we also introduce an elimination rule defined by primitive recursion on the given type. Thus a new constant, the elimination constant, is obtained each time a new type is given. Other constants are then defined in terms of already existing elimination constants.

If we want to justify the addition of a constant f specified by recursive pattern-matching equations $f\ \vec{p}_1 = t_1,\ \ldots\ , f\ \vec{p}_n = t_n$, a methodology due to Martin-Löf[11] is to find an expression g defined in terms of elimination constants, such that for each equation $f\ \vec{p}_i = t_i$, the expressions $g\ \vec{p}_i$ and $t_i[g/f]$ are equal by convertibility.

For the previously given example with addition, the equalities to prove are

$$\text{add}\ x\ 0 \quad = \quad x \tag{1.1}$$

$$\text{add}\ x\ (\text{s}\ y) \quad = \quad \text{s}\ (\text{add}\ x\ y) \tag{1.2}$$

where equality is convertibility. This is easy to verify:
(1.1) holds because of the reduction sequence (reductions will be written $\rightsquigarrow$)

$$\begin{array}{l}
\text{add}\ x\ 0\ \rightsquigarrow \\
(\lambda x.\lambda y.\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ y)\ x\ 0\ \rightsquigarrow \\
(\lambda y.\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ y)\ 0\ \rightsquigarrow \\
\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ 0\ \rightsquigarrow \\
x
\end{array}$$

and (1.2) holds because of the reduction sequences

$$\begin{array}{l}
\text{add}\ x\ (\text{s}\ y)\ \rightsquigarrow \\
(\lambda x.\lambda y.\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ y)\ x\ (\text{s}\ y)\ \rightsquigarrow \\
(\lambda y.\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ y)\ (\text{s}\ y)\ \rightsquigarrow \\
\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ (\text{s}\ y)\ \rightsquigarrow \\
(\lambda m.\lambda n.\text{s}\ n)\ y\ (\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ y)\ \rightsquigarrow \\
(\lambda n.\text{s}\ n)\ (\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ y)\ \rightsquigarrow \\
\text{s}\ (\text{natrec}\ (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s}\ n)\ y)
\end{array}$$

[11] The author was not able to access literature that explicitly mentions or describes this methodology. What we present here evolves from personal communication with Per Martin-Löf, Johan Granström and Thierry Coquand.

and

$$\begin{array}{l}\text{s (add } x\ y) \rightsquigarrow \\ \text{s } ((\lambda x.\lambda y.\text{natrec } (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s } n)\ y)\ x\ y) \rightsquigarrow \\ \text{s } ((\lambda y.\text{natrec } (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s } n)\ y)\ y) \rightsquigarrow \\ \text{s (natrec } (\lambda k.\text{Nat})\ x\ (\lambda m.\lambda n.\text{s } n)\ y)\end{array}$$

First-order / higher-order recursion Certain functions, for instance the minimum function specified by the equations

$$\begin{array}{llll}\text{min}: & \text{Nat} \to & \text{Nat} & \to \text{Nat} \\ \text{min} & 0 & n & = 0 \\ \text{min} & (\text{s } m) & 0 & = 0 \\ \text{min} & (\text{s } m) & (\text{s } n) & = \text{s (min } m\ n)\end{array}$$

cannot be expressed as a first-order primitive recursive scheme (cf. Colson, 1989) with the same computational behaviour as the original program. However, it should be pointed out that one can still implement non-primitive recursive functions using elimination constants of *higher* types. For this example, using the elimination constant natrec for the natural numbers, we can define the minimum function as follows:

$$\text{min} = \text{natrec}(\lambda x.\text{Nat} \to \text{Nat}, \lambda n.0, \lambda m.\lambda h.\text{natrec}(\lambda y.\text{Nat}, 0, \lambda n.\lambda u.\text{s } (h\ n)))$$

which is then primitive recursive, with higher-order parameters. This definition satisfies the system of equations specifying 'min' above, in the same sense as in the example for addition given above. Another example of a "truly" non-primitive recursive program is the Ackermann function from Section 1.1.1 above, which Hilbert gave as a higher-order primitive recursive scheme. Thus, the exclusive use of primitive recursive schemes is not that limiting as one may think, at first glance.

1.3.3 Elimination constants versus pattern-matching

Readability Programming using only elimination constants may be difficult, and programs tend to be hard to understand and analyse by humans. For instance, the pattern-matching version of the minimum function given above can be thought of as an intuitive specification, and it is certainly easier to understand than the version using elimination constants. If we only allow definitions of constants to be given in terms of elimination rules, it might be hard to be convinced that it has the intended meaning.

Foundations If we consider type theory as a system constructed to lay the foundations of mathematics, we want our constructions to be built up by self-evident principles. Primitive recursion is somehow the smallest mental leap required to understand computations on arbitrary objects of inductive types, such as natural numbers and lists. To justify more general schemes of recursive equations is far more difficult, and in this sense it does not seem reasonable

to consider such schemes as primitive building blocks of computation. On the other hand, the freedom letting us define computation by recursive equations lets us abstract away from the termination proof. Once we have a justification of termination, the definition in question can be trusted.

It should be noted that primitive recursive schemes also require subtle arguments in order to be justified. For instance, Martin-Löf (1984), used well-founded induction in the justification of the recursion operator[12] for natural numbers. Exactly the same justification applies to definitions like the addition function that we showed previously in Section 1.3.2, page 14.

Non-sequential patterns If we consider an arbitrary system of recursive equations, that we are convinced is a correct specification of some recursive function, it is not always the case that it can be interpreted in terms of elimination constants, preserving the computational behaviour of the specified program.

As an example let us consider Berry's majority function defined as follows:

maj :	Bool →	Bool →	Bool	→	Bool
maj	true	true	true	=	true
maj	x	true	false	=	x
maj	false	y	true	=	y
maj	true	false	z	=	z
maj	false	false	false	=	false

The function returns the boolean value that is the most frequent among the three input values. Instead of exhausting all eight possible case distinctions, only five equations are required. Intuitively, the three arguments have to be evaluated in parallel. The algorithm is *stable* but not *sequential* (cf. Berry, 1978, 1979). However, a program constructed exclusively by elimination constants is always sequential, and will not behave like the above specification.

Pattern-matching definitions versus case expressions The above example illustrates not only a limitation with elimination constants, but also a potential disadvantage with pattern-matching. Specifying a program only by pattern-matching equations leaves the order in which the arguments have to be evaluated unspecified. In this sense elimination rules have no ambiguity. However, using case-notation instead of pattern-matching dissolves this ambiguity. Using case notation will restrict us to sequential programs. The problem of how to translate automatically between "sequentially valid" pattern-matching equations and case notation was investigated by Augustsson (1985). A disadvantage with case-notation when it comes to type-checking is that equality becomes less clear than it would be with pattern-matching. A reasonable compromise is to consider sequential pattern-matching definitions.

[12] See the example 'natrec' in end of section 1.2.5.

Permuted arguments A related question is to what extent equations that involve permuted arguments can be interpreted using elimination constants. For instance, let us give an alternative specification of the addition function:

$$\begin{aligned}&\text{addSwap} : \text{Nat} \to \text{Nat} \to \text{Nat}\\ &\text{addSwap}\ x\ 0 = 0\\ &\text{addSwap}\ x\ (\text{s}\ y) = \text{s}\ (\text{addSwap}\ y\ x)\end{aligned}$$

There is no doubt that this program is total, but it is likely to be difficult, if at all possible to define this intensionally using the elimination constant natrec, satisfying the given equations.

1.4 Related work

Several quite different approaches have been investigated to enable programs to be specified by recursive equations in type theory. The core of the problem is to ensure that the new extension preserves termination, and so decidability of convertibility and type-correctness. We will give a rough guide to the different methodologies that exist.

1.4.1 Translation into elimination constants

Smith (1983) showed how programs given by recursion equations could be translated into higher-order primitive recursive schemes. As an example, quick-sort was shown to be derivable. Techniques to translate recursive equations into elimination constants automatically have been developed by Cornes (1997), now available in Coq (The Coq Development Team, 2006), and by McBride (2000) for Epigram (cf. McBride & McKinna, 2004). This technique follows the ideas presented in Section 1.3.2 above, and so it is faithful to the core concepts of type theory.

1.4.2 Domain predicates

Bove (2002); Bove & Capretta (2005) presented a methodology for representing partiality and general recursive functions *within* type theory. Given a recursive definition of a constant[13] $f : A \to B$, one introduces an inductive family $D : A \to \text{Set}$ that expresses for which values $a : A$ the application $f\ a$ is defined. The formal definition of f has now the type $(x : A) \to D\ x \to B$. Intuitively, we can see D as the *domain* of f. Establishing that f is total amounts to proving that $D\ a$ holds for each argument $a : A$. In the type-theoretic definition of f an extra argument is added in the recursive equations, which is a proof object inhabiting the inductive family in question. The so obtained program is then structurally recursive on this extra argument.

A program obtained using this method has a strict semantics when it comes to the recursive arguments and hence, may have a different behavior from the

[13] We simplify by considering only one argument.

"intended" program that was given as a set of recursive equations. In case one wants to extract programs from proofs, and erase computationally irrelevant parts, the extra argument may in some cases be safely removed from the program. An advantage of this methodology is that it does not require external tools, such as termination checkers.

1.4.3 Term based approaches

Pattern-matching with dependent types Coquand (1992) investigated how to introduce definitions by pattern-matching in Martin-Löf's logical framework. Two sufficient conditions for correct definitions were identified: well-founded recursive definitions and covering (exhaustiveness of pattern-matching). Using this approach the usual elimination rules of type theory could be expressed in this framework.

Giménez (1995) gave a schema for the definition of constants by primitive recursion and case analysis, with a precise proof of normalization. This is now a possible way to define constants in the Coq proof assistant (The Coq Development Team, 2006).

Lexicographical orderings Abel (1999) and Abel & Altenkirch (2002), presented a combination of simply typed lambda-calculus, functional programs and algebraic data-types. They analyze the call structure of the program, the *call graph*, and for each recursive call a *call matrix* is constructed, that relates formal parameters with actual parameters of the corresponding call. Then they compute a closure of matrix multiplications for compatible matrices in the call graph, and perform a search for lexicographical orderings of all the loops of the program. Their analysis covers mutual recursion, but not permuted argument positions.

Term rewriting Allowing recursive equations to be given as term rewriting systems has many advantages in terms of both syntactic flexibility and computational efficiency. Termination of untyped term rewriting systems has been extensively studied, and an example of a tool that integrates a wide range of known methods in this direction is the AProVE system, (Giesl *et al.*, 2004). In 1988, Breazu-Tannen proved confluence for a combination of simply typed lambda-calculus and confluent rewrite-rules. Jouannaud & Okada (1997, 1991) proved strong normalization for a combination of the simply typed lambda calculus and first-order rewrite rules over algebraic data types, obeying a generalization of primitive recursion called the general schema. Blanqui (2005) proved strong normalization for a system called the Calculus of Algebraic Constructions: a combination of the general schema of Jouannaud and Okada with the Calculus of Inductive Constructions (Coquand & Paulin-Mohring, 1990).

1.4.4 Type based termination

Another technique dealing with recursive equations in typed lambda calculi, is the so-called *type based termination*. In Mendler (1987), a type system is presented in which programs specified by recursion equations are well-typed only if they are equivalent to constructions based on certain elimination constants. More recent work with connection to this can be found in Matthes (1998) and Uustalu & Vene (2002). Hughes *et al.* (1996), developed a type system in which size constraints of expressions in a program could be expressed in the type system, applicable for instance to obtain compile-time bounds for memory allocation in embedded ML, a functional programming language for embedded systems. See also Pareto (2000) and Björk (2000).

Giménez (1998) suggested an extension of the Calculus of Constructions where recursive data structures have size information, that is used to ensure size decrease in recursive calls. Their approach enables termination conditions to be abstracted from details in the syntactic structure of programs, and it also has the advantage that function types can express preservation of size bounds. Abel (2004) and Blanqui (2004), have developed these ideas further, influenced by Hughes, Pareto and Giménez. An algorithm has been developed to infer size-annotations automatically by Barthe *et al.* (2006). Type based termination has also been investigated for co-inductive types in Barthe *et al.* (2004) and Abel (2006c,b). Using type based termination one can represent recursive functions directly as systems of recursive equations as we suggested above. An advantage with this approach is that even non-structurally terminating programs can be justified. Quick-sort, for instance, is justified, but also highly non-trivial termination problems can be dealt with in an elegant way (Abel, 2006a,d). However, the type system is modified, and the type theory so obtained is both different from the original one and more complex. To our knowledge it is not clear if this technique applies to programs with permuted arguments, though one could speculate that it could be combined with the size-change principle (explained below) by Lee, Jones, Ben-Amram (2001) to solve this problem.

1.5 Methodology and contribution

1.5.1 Objective

Our objective is to build a constructive type-theoretic formalism for proofs and programs, in which one should be able to recognize mechanically when a given argument is a proof or not. One should be able to add new constants to the system using recursive pattern-matching definitions. This approach is close in spirit to Herbrand's, as discussed at page 3, letting us separate the obligation to prove termination from the task of specifying the input/output relation.

1.5.2 Decidable type-checking

Due to the propositions-as-types property of type theory, we have that proof checking can be reduced to type checking. To check decidability of type correctness, it is sufficient to have decidable convertibility of terms, which is the case if our reduction rules are normalizing and all terms have a normal form which is unique. However, since convertibility in general has to be checked under assumptions, we have to establish normalization for *open* terms.

1.5.3 Reducibility

A well-known method for proving normalization is the *reducibility method* stemming from Gödel 1941 (Feferman, 1986), Tait (1967), adapted for dependent types in Girard (1971) and Martin-Löf (1971, 1972). A more recent presentation was given by C. Coquand (1998), which is close to the approach of Tait (1975), which stresses the similarity between the reducibility method and the realizability interpretation of Kleene (1945).

Reducibility is a semantic property of normalizing *open* terms, and resembles Martin-Löf's notion of meaning explanations (Martin-Löf, 1996). For *simple types* we may specify what it means to be a reducible term t in type A, recursively on A as follows:

- when A is a ground type, then t reduces to a normal canonical form prescribed by the introduction rules of A, or to an application $x\ t_1\ \ldots\ t_n$, where t_i are normal.
- when A is a function type $B \to C$, for all b reducible in B, then $t\ b$ is reducible in C.

To prove normalization of well-typed terms, one proves that well-typed terms are reducible in their respective types, and then that reducibility implies normalization.

For dependent types, the notion of reducibility has to be modified, since types in general depend on terms. Before we can establish that an object a is reducible in type A, we must know what it means for A to be reducible. We specify first what is a reducible set, and then for each reducible set, what is a reducible element in such a set. When one specifies this for the Cartesian product of a family of sets, the set predicate and the element predicate have to be mutually defined, thus giving rise to an *inductive-recursive* definition (cf. Dybjer, 1994, 2000). This inductive-recursive nature of the reducibility predicates is present already in the normalisation proofs of Martin-Löf (1971, 1972) and Coquand (1998), as well as in Martin-Löf's meaning explanations.

1.5.4 A semantic criterion for new constants

If we decide to give a general formal scheme prescribing how to extend our system, what constants should then be allowed ? We suggest that reducibility is a good choice as a semantic property, as it is sufficient for decidable

type-correctness. Below we will give examples of reducible and non-reducible constants.

Reduction of open terms

Assume we are about to add the constant zero function, using the predecessor function as follows:

$$\begin{aligned}&\text{isZero} : \text{Nat} \to \text{Bool}\\ &\text{isZero } 0 \;=\; \text{true}\\ &\text{isZero } (\text{s } x) = \text{false}\end{aligned}$$

$$\begin{aligned}&\text{pred} : \text{Nat} \to \text{Nat}\\ &\text{pred } 0 \;=\; 0\\ &\text{pred } (\text{s } x) = x\end{aligned}$$

$$\begin{aligned}&\text{reduceToZero} : \text{Nat} \to \text{Nat}\\ &\text{reduceToZero } x = \text{ if (isZero } x) \text{ then 0 else reduceToZero (pred } x)\end{aligned}$$

This program has a normal form for all closed instances of x, but not for open terms, for instance x itself. As implemented above, the constant 'reduceToZero' is not reducible. But we can use Vogel's trick (cf. Vogel, 1976)[14] to transform this program into a reducible one, by replacing the if-then-else construction with a call to an auxiliary definition, such that if the evaluation of the boolean argument gets blocked, the whole computation will be blocked:

$$\begin{aligned}&\text{aux} : \text{Nat} \to \text{Bool} \to \text{Nat}\\ &\text{aux } x \text{ true} = \; 0\\ &\text{aux } x \text{ false} = \text{reduceToZero (pred } x)\end{aligned}$$

$$\text{reduceToZero } x = \text{ aux } x \text{ (isZero } x)$$

Now, when providing x as input, the computation will stop when trying to evaluate the expression 'aux x (isZero x)'.

A non-reducible looping program

A more intricate example is the following constant 'mp' to encode the witness of Markov's principle, stating that if an infinite boolean sequence f does not have the value 'false' at every position, then there is a position n such that $f\ n = \text{true}$, and moreover that we can find this position by searching through

[14] Cf. Berger (2005) for an application of this technique.

the positions until we find the desired index.

$$\begin{array}{l} \mathrm{F} : \mathrm{Bool} \to \mathrm{Set} \\ \mathrm{F\ true} = \bot \\ \mathrm{F\ false} = \top \end{array}$$

$$\begin{array}{l} \mathrm{mp}' : (f : \mathrm{Nat} \to \mathrm{Bool}) \to (((n : \mathrm{Nat}) \to F(f\ n)) \to \bot) \to \mathrm{Nat} \to \mathrm{Nat} \\ \mathrm{mp}'\ f\ p\ k = \mathrm{if}\ f\ k\ \mathrm{then}\ k\ \mathrm{else}\ \mathrm{mp}'\ f\ p\ (\mathrm{s}\ k) \end{array}$$

$$\begin{array}{l} \mathrm{mp} : (f : \mathrm{Nat} \to \mathrm{Bool}) \to (((n : \mathrm{Nat}) \to F(f\ n)) \to \bot) \to \mathrm{Nat} \\ \mathrm{mp}\ f\ p = \mathrm{mp}'\ f\ p\ 0 \end{array}$$

If one accepts Markov's principle, mp is total for closed arguments, but 'mp f p' has no normal form when f and p are variables. Let us try Vogel's trick again, redefining mp′ as follows:

$$\begin{array}{l} \mathrm{aux}\ f\ p\ k\ \mathrm{true} = k \\ \mathrm{aux}\ f\ p\ k\ \mathrm{false} = \mathrm{mp}'\ f\ p\ (\mathrm{s}\ k) \end{array}$$

$$\begin{array}{l} \mathrm{mp}' : (f : \mathrm{Nat} \to \mathrm{Bool}) \to (((n : \mathrm{Nat}) \to F(f\ n)) \to \bot) \to \mathrm{Nat} \to \mathrm{Nat} \\ \mathrm{mp}'\ f\ p\ k = \mathrm{aux}\ f\ p\ k\ (f\ k) \end{array}$$

Is the new definition reducible ? —The answer is no: we can find reducible arguments for which the result is not reducible. Define

$$\begin{array}{l} \mathrm{returnFalse} : \mathrm{Nat} \to \mathrm{Bool} \\ \mathrm{returnFalse}\ x = \mathrm{false} \end{array}$$

The term 'mp′ returnFalse y 0' has no normal form, but we have 'returnFalse' reducible in Nat → Bool, and the variable y, the assumed "proof", is reducible in the type $((n : \mathrm{Nat}) \to F(\mathrm{returnFalse}\ n)) \to \bot$. For this reason, one might suspect that Markov's principle cannot be encoded by a reducible term.

1.5.5 The size-change principle

The solution investigated in this dissertation follows the approach presented in Coquand (1992), that we mentioned in Section 1.4.3 above. We use here the *Size-change principle for program termination* by Lee, Jones, Ben-Amram (2001), for justifying our recursive equations. It covers many forms of recursion,[15] such as lexicographical argument ordering, permuted parameters and mutual recursion. Their analysis has many similarities with Abel and Altenkirch's method, but the size-change analysis needs no special treatment to find lexicographical argument orderings.

[15] The class of functions computable by first-order size-change terminating programs is the same as the class of multiply recursive functions described by Rózsa Péter (1967), as shown by Ben-Amram (2002).

While the general halting problem is undecidable, the size-change principle provides a decision procedure for "structurally terminating" computation, if we consider structural size-changes between left-hand sides and arguments in recursive calls. Such a criterion is still close to the way elimination constants are informally justified in type theory, based on the fact that constructor decomposition is a well-founded relation.

As in Abel and Altenkirch's work, the call graph is annotated by call matrices (called *size-change graphs*), and the closure of multiplications (compositions of size-change graphs) of adjacent matrices is calculated. The criterion is then simple to formulate: every infinite path in the call graph contains an infinite *thread* of non-increasing transitions containing infinitely many decreasing transitions. This makes an infinite execution path impossible due to the fact that then some parameter value must decrease infinitely, which is impossible for a well-founded object. What is surprising is that the size-change termination criterion, with respect to a given call graph with its call matrices, is decidable. Moreover, the decision procedure can be implemented by a small program (cf. Wahlstedt, 2000).

1.5.6 Obtaining reducibility from well-founded recursion

To prove the reducibility of a recursively specified constant, we proceed in two steps. We define a relation of *instance of recursive call*[16], which we prove to be well-founded. Having established this, we can prove that the constant in question is reducible.

We proceed by proving that the size-change criterion implies that the call-instance relation is well-founded. In this way we get a reasonably large fragment of structurally terminating programs that can be recognized automatically. However, there are programs that are well-known to terminate, that have a simple formulation, but are not structurally terminating. Let us consider the naive quick-sort program:

$$\begin{aligned}&\text{quickSort } [] = [] \\ &\text{quickSort } (x : xs) = \\ &\qquad \text{quickSort } (\text{filter } (< x)\ xs) +\!\!+ x : \text{quickSort } (\text{filter } (\geq x)\ xs)\end{aligned}$$

where $(+\!\!+)$ is the list append operator, and the calls of 'filter' partition the list into the strictly smaller elements and the larger elements respectively. The constant 'quickSort' *is* reducible, and hence it can be added to our type theory. We realize that the lists in the recursive calls must be smaller than the input list, but not *structurally*. For this example our structural restriction of the size-change principle is not enough for establishing a well-founded call-instance relation. However, if we can prove this by other means, we can proceed from this point and establish reducibility using the result we present in this dissertation. Thus our approach is modular, and should allow further improvements to be considered for termination detection.

[16] See Definition 3.6.3, page 80 in the proof text.

The work order is to design a system of dependent types as simple as possible, with sufficient syntactic conditions to ensure decidable type correctness. Having achieved this we may consider further extensions.

1.5.7 Contribution

Our contributions are the following:

- A type theoretic formal system.
  We have given a general formulation of a predicative constructive intensional Martin-Löf-style type theory, based on untyped lambda-calculus and pattern-matching definitions of functional programs on first-order parameterized algebraic data types.

- Normalization.
  We present a detailed proof of normalization for the proposed system. For recursive definitions we assume a well-founded call relation. In this way we are not restricted to a specific schema for the termination of recursive definitions. This is the main contribution.

- A decision procedure for type-checking a theory.
  Given a set of data type definitions and a sequence of blocks of mutually recursive definitions having a well-founded call relation, we can check if this corresponds to a valid stratification.

- Size-change termination.
  We have shown that the call relation for the recursive definitions in our system is well-founded, provided that it obeys the size-change principle of Lee, Jones and Ben-Amram (2001).

Chapter 2

Syntax

In this chapter we present a language of *raw terms*, which is the untyped lambda-calculus extended with constants. The structure, called *signature*, in which a theory is built is then described. For this language we give rules of untyped substitution and reduction, followed by a proof of the Church-Rosser property, establishing uniqueness of normal form and that convertibility is an equivalence relation. We give rules of inference, called *typing* rules, of how to form syntactically correct terms, and conditions for what is a syntactically correct signature. We prove a number of properties about the type system, including the subject reduction property, which states that typing is preserved under computation. Then we define a relation of type correctness for β-normal terms, which is the fragment of the language for which the typing relation is decidable. We show this definition sound and complete with respect to the typing relation.

2.1 A language of raw terms

2.1.1 Syntactical categories

Rather than defining a few syntactical categories of general form such as the bare lambda-calculus with constants, we give our language extensions in several syntactical sub-categories. Instead of using explicit predicates on terms, we can refer to our sub-categories in order to avoid lengthy conjunctions of predicates.

Instead of following the Barendregt-style notation with M, N for terms, we follow the notation by Girard *et al.* (1989) using t, u, v for terms, and we reserve upper case notations T, U, V, etc. for sets and types.

Definition 2.1.1 (Raw terms, types and contexts).
We define inductively the following syntactic categories:

x, y, z			Variables
f, g			Defined constants
c			First-order element constructors
d			First-order set constructors
t, u, v	$::=$	$a \mid \lambda x.t \mid t\ u$	Terms
a	$::=$	$x \mid f \mid d \mid c$	Atoms - Names
	$\mid$	$\Pi \mid \text{fun} \mid \text{El} \mid \text{Set} \mid \text{Fun}$	Atoms - Primitive constants
T, U, V	$::=$	$\text{Set} \mid \text{El}\ t \mid \text{Fun}\ T\ (\lambda x.U)$	Types
Γ, Δ	$::=$	$() \mid \Gamma, x : T$	Contexts

Notation 2.1.2 *(Iterated application).* Application associates to the left, and thus the iterated application $(\ldots(t\ t_1)\ldots)\ t_n$ is written $t\ t_1\ \ldots\ t_n$.

Definition 2.1.3 (Set Patterns). The syntactic sub-category of terms called *set patterns* is inductively defined by the grammar:
$e \quad ::= \quad x \mid d\ e_1\ \ldots\ e_n$

Definition 2.1.4 (Constructor Patterns). The syntactic sub-category of terms called *constructor patterns* is inductively defined by the grammar:
$p, q \quad ::= \quad x \mid c\ p_1\ \ldots\ p_n \mid \text{fun}\ x$

In the latter production, 'fun' is the constructor of our only higher-order data type former Π. An expression of the form $\Pi\ U\ (\lambda x.V)$ is a set-level counterpart of the dependent function type $\text{Fun}\ U\ (\lambda x.V)$. See section 2.3 for its typing rules.

Notation 2.1.5 *(Vector notation).*
Let $\vec{t}$ be a shorthand of the sequence $t_1, \ldots, t_n$ or the tuple $(t_1, \ldots, t_n)$. We write $t, \vec{t}$ for $(t, t_1, \ldots, t_n)$ and $\vec{t}, \vec{u}$ for $(t_1, \ldots, t_n, u_1, \ldots, u_m)$. We write $t\ \vec{t}$ as a shorthand for the iterated application $t\ t_1\ \ldots\ t_n$. We write $\lambda\vec{x}.t$ as a shorthand for the iterated abstraction $\lambda x_1.\lambda x_2 \ldots \lambda x_n.t$. The length of $\vec{t}$ is denoted by $|\vec{t}|$.

Raw types

As shown in Definition 2.1.1 the syntactical category of types is a sub-category of terms. Therefore general definitions involving terms will also apply to types. For instance, the free variables of a type is just the free variables of a term that represents the type in question. Accordingly, a closed type is a closed term. When we later define reduction and substitution, these notions will automatically apply to terms as well as types.

2.1.2 Context notations and operations

Definition 2.1.6 (Context lookup).

$$\begin{cases} (\Gamma, x : T)(x) = T \\ (\Gamma, y : T)(x) = \Gamma(x), \quad \text{if } y \neq x \end{cases}$$

Definition 2.1.7 (Context support). Let $\Gamma = (x_1 : T_1, \ \ldots \ , x_n : T_n)$. The *support* of Γ, written $supp(\Gamma)$, and the *support vector* for Γ, written $\overrightarrow{supp}(\Gamma)$ are defined as follows:

$$\begin{cases} supp(\Gamma) = \{x_1, \ \ldots \ , x_n\} \\ \overrightarrow{supp}(\Gamma) = (x_1, \ \ldots \ , x_n) \end{cases}$$

Notation 2.1.8 *(Disjoint context).* The context $(x_1 : T_1, \ \ldots \ , x_n : T_n)$ is *disjoint* iff all of x_i are distinct.

Notation 2.1.9 *(Closed context).* The context $(x_1 : T_1, \ \ldots \ , x_n : T_n)$ is *closed* if $FV(T_i) \subseteq \{x_1, \ \ldots \ , x_{i-1}\}$

Notation 2.1.10 *(Extended context).* Let $\Gamma = (x_1 : T_1, \ \ldots \ , x_n : T_n)$.
Let $\Delta = (y_1 : U_1, \ \ldots \ , y_m : U_m)$. Then $\Theta = \Gamma, \Delta$ is the context
$(x_1 : T_1, \ \ldots \ , x_n : T_n, \ y_1 : U_1, \ \ldots \ , y_m : U_m)$. We write that Θ *extends* Γ.

2.1.3 Function type notations

Notation 2.1.11 *(Arrow notation).* We may write $(x : U) \to V$ as a shorthand for Fun U $(\lambda x.V)$. The arrow associates to the right.

Note that in some cases it is hard to translate from Fun-notation into arrow-notation. For instance a type denoted by Fun U $((\lambda x.V)\ \gamma)$ would be cumbersome to translate into arrow notation, because it would involve constraints to prevent name capturing of the bound variables. See Section 2.2.1 on substitution properties.

Notation 2.1.12 *(Telescope notation).*
We will write $(x_1 : T_1, \ \ldots \ , x_n : T_n) \to T$ or even shorter $\Gamma \to T$ where $\Gamma = (x_1 : T_1, \ \ldots \ , x_n : T_n)$ as a shorthand for $(x_1 : T_1) \to \ldots \to (x_n : T_n) \to T$.

Remark 2.1.13. If $\Gamma \to T$ is closed, then Γ is closed.[1]

Remark 2.1.14. The function types $(x_1 : T_1, \ \ldots \ , x_{n-1} : T_{n-1}, \ x_n : T_n) \to T$ and $(x_1 : T_1, \ \ldots \ , x_{n-1} : T_{n-1}) \to ((x_n : T_n) \to T)$ are the same.

Notation 2.1.15 *(Independent function types).* Function types
$(x : U) \to V$ with $x \notin FV(V)$ are called *independent*, and we write $U \to V$, both in order to emphasize this fact and to get a lighter notation. Thus a function type $(x_1 : T_1, \ \ldots \ , x_k : T_k, \ \ldots \ , x_n : T_n) \to T$ where $(x_k : T_k, \ \ldots \ , x_n : T_n) \to T$ is independent, can be written $(x_1 : T_1, \ \ldots \ , T_k, \ \ldots \ , x_n : T_n) \to T$ to emphasize the independence of x_k.

Notation 2.1.16. We write $T^n \to U$ as a shorthand for $(T_1, \ \ldots \ , T_n) \to U$ with all of T_i being syntactically equal.

[1] Recall that a closed type is just a closed term in the sub-category of types.

2.1.4 Signature

Definition 2.1.17 (Data type specification). A data type specification is written $(\mathcal{D}, \mathcal{C})$. Each set constructor d has a uniquely associated type $\mathcal{D}(d)$ of the form $\mathrm{Set}^n \to \mathrm{Set}$. Each value constructor c has a uniquely associated independent function type $\mathcal{C}(c)$ of the form $(\mathrm{El}\ e_1,\ \ldots\ , \mathrm{El}\ e_n) \to \mathrm{El}\ (d\ x_1\ \ldots\ x_k)$ such that $FV(e_i) \subseteq \{x_1,\ \ldots\ , x_k\}$ and $\mathcal{D}(d) = \mathrm{Set}^k \to \mathrm{Set}$.[2]

Note that the types of our constructors are independent. For instance we cannot encode the type of lists of a specified length directly as a data type. Furthermore the target types of our constructors are restricted to $\mathrm{El}\ (d\ \vec{x}\,)$ (see above). However one can get around some of these limitations by defining set-valued functions, as shown later in section 2.3.5, page 42.

Definition 2.1.18 (Typing specification of defined constants). $\mathcal{F}$ is a collection of typing specifications of defined constants. When there is a type associated in $\mathcal{F}$ for f, we write $f \in \mathcal{F}$, and in that case we write $\mathcal{F}(f)$ for this type. When $f \in \mathcal{F}$ then $\mathcal{F}(f)$ is a closed expression of the form S. The empty typing specification is written $\emptyset$. The concatenation of $\mathcal{F}_1$ and $\mathcal{F}_2$ is written $\mathcal{F}_1\mathcal{F}_2$, where $\mathcal{F}_1$ and $\mathcal{F}_2$ are assumed to be disjoint.

Definition 2.1.19 (Arity). If h is a constant, then h has a uniquely associated natural number n (ranging from zero), called the *arity* of h, written $ar(h)$. The following properties hold:

- When $f \in \mathcal{F}$ and $\mathcal{F}(f) = (x_1 : T_1,\ \ldots\ , x_n : T_n) \to T$ with T of the form Set or El t, we have $ar(f) = n$.
- When $\mathcal{C}(c) = (\mathrm{El}\ e_1,\ \ldots\ , \mathrm{El}\ e_n) \to \mathrm{El}\ (d\ x_1\ \ldots\ x_k)$, we have $ar(c) = n$.
- When $\mathcal{D}(d) = \mathrm{Set}^n \to \mathrm{Set}$, we have $ar(d) = n$.
- We have $ar(\Pi) = 2$ and $ar(\mathrm{fun}) = 1$.

The following definition characterizes the terms allowed in bodies of defined constants, β-normal[3] terms. We require full applications of constants, a technical restriction used in the proofs of Corollary 2.5.6 (soundness of type inhabitation checking), page 64, and Lemma 3.6.6 (Key lemma), page 82. The restriction does not limit the expressibility, since an incomplete application can always be represented by an η-expanded term.

Definition 2.1.20 (β-normal terms, types and contexts). The syntactical sub-categories s, S and Ξ are inductively defined by the grammar:

s	::=	$x\ s_1\ \ldots\ s_n \mid h\ s_1\ \ldots\ s_{ar(h)} \mid \lambda x.s$	β-normal terms
h	::=	$f \mid c \mid d \mid \Pi \mid \mathrm{fun}$	heads
S	::=	$\mathrm{Set} \mid \mathrm{El}\ s \mid \mathrm{Fun}\ S_1\ (\lambda x.S_2)$	β-normal types
Ξ	::=	$() \mid \Xi, x : S$	β-normal contexts

[2] Note that n and / or k may be zero.

[3] See Definition 2.2.35, page 37.

Definition 2.1.21 (Signature). A signature Σ is a quadruple $(\mathcal{D}, \mathcal{C}, \mathcal{F}, \mathcal{R})$, where $(\mathcal{D}, \mathcal{C})$ is a data type specification and $\mathcal{F}$ is a typing specification of defined constants. $\mathcal{R}$ is a set of rules of the form $f\ \vec{p} = s$, with $FV(s) \subseteq FV(\vec{p})$ and $|\vec{p}\,| = ar(f)$. The empty set of rules is written $\emptyset$. The union of $\mathcal{R}$ and $\mathcal{R}'$ is written $\mathcal{R}\mathcal{R}'$.

Note that the above definition allows that $\mathcal{R}$ have rules for constants that are not given in $\mathcal{F}$. This reflects the fact that our reduction system is a priori untyped, and we may consider a rule without having to think about its type.

Restriction 2.1.22. From now on we assume a given signature Σ. All further references to any of $\mathcal{D}, \mathcal{C}, \mathcal{F}, \mathcal{R}$ implicitly refers to Σ, unless otherwise stated, and we consider only constants d, c, f with respect to Σ.

2.2 Substitution and reduction

2.2.1 Substitution

A substitution, denoted $\gamma, \delta, \rho, \sigma$, is a function from variables to terms, used with prefix notation $\gamma(x)$, defined as follows:

Definition 2.2.1 (Parallel substitution on variables).

$$[](x) = x$$
$$[\gamma, u/x](y) = \begin{cases} u & \text{if } x = y, \\ \gamma(y) & \text{otherwise.} \end{cases}$$

Definition 2.2.2 (Fresh variable). A variable y is *fresh* for t, x, γ iff

$$y \notin \bigcup_{z \in FV(t) - \{x\}} FV(\gamma(z))$$

Notation 2.2.3 *(Substitution in terms).* We use post fix notation $t\ \gamma$, denoting the term obtained by simultaneously replacing all the free variables x in t by $\gamma(x)$.

Definition 2.2.4 (Substitution).

$$x\ \gamma = \gamma(x) \qquad (u\ v)\ \gamma = (u\ \gamma)(v\ \gamma)$$

$$(\lambda x.t)\ \gamma = \lambda y.(t[\gamma, y/x]), \text{ if } y \text{ is fresh for } t, x, \gamma$$

Remark 2.2.5 *(α-equivalence).* Terms that differ only in the names of their bound variables are identified. Because of this, Definition 2.2.4 is deterministic, up to α-conversion.

Definition 2.2.6 (Composition of substitutions).
The composition of γ and δ is written $\gamma\delta$, where $(\gamma\delta)(x)$ is defined as $(\gamma(x))\delta$.

Proposition 2.2.7 (Properties of substitution).

$$t(\gamma\delta) = (t\ \gamma)\delta \qquad t[u/x]\gamma = t[\gamma, (u\ \gamma)/x]$$

$$t[\gamma, u/x] = t\ \gamma, \quad \textit{if } x \notin FV(t) \qquad t\ \gamma = t, \quad \textit{if } t \textit{ is closed}$$

$$(\lambda x.t)\ \gamma = \lambda x.(t\ \gamma), \ \textit{if } x \textit{ is fresh for } t, x, \gamma \textit{ and } x = \gamma(x)$$

$$t\gamma[u\ \gamma/x] = t[u/x]\gamma \ \textit{if } x \textit{ is fresh for } t, x, \gamma \textit{ and } x = \gamma(x)$$

Notation 2.2.8 *(Substitution in vectors).*
Let $\vec{t}\gamma$ be a shorthand for $(t_1\gamma, \ \ldots\ , t_n\gamma)$.

Notation 2.2.9 *(Substitution in contexts).*
Let $\Gamma = (x_1 : T_1, \ \ldots\ , x_n : T_n)$.
Then $\Gamma\ \gamma$ denotes $(x_1 : T_1\gamma, \ \ldots\ , x_n : T_n\gamma)$.

Notation 2.2.10 *(Substitution restricted by context[4]).*
Let $\Gamma = (x_1 : T_1, \ \ldots\ , x_n : T_n)$.
Then $\gamma_{|\Gamma}$ is the substitution $[\gamma(x_1)/x_1, \ \ldots\ , \gamma(x_n)/x_n]$.

Notation 2.2.11 *(Short for multiple substitution).* We may use $[u_1, \ \ldots\ , u_n]$ or even shorter $[\vec{u}]$ as a shorthand for the substitution $[u_1/x_1, \ \ldots\ , u_n/x_n]$, when there is no confusion about x_i.

2.2.2 Reduction rules and equality

Definition 2.2.12 (Reduction). The relations $t \rightsquigarrow_\beta u$, $t \rightsquigarrow_\iota u$ and $t \rightsquigarrow u$ are inductively defined as follows:

$$\frac{}{(\lambda x.t)\ u \rightsquigarrow_\beta t[u/x]} \qquad \frac{}{(f\ p_1\ \ldots\ p_n)\ \gamma \rightsquigarrow_\iota (\lambda\vec{x}.s)\ \gamma} \begin{cases} f\ p_1\ \ldots\ p_n\ \vec{x} = s\ \in \mathcal{R} \\ p_n \text{ is not a variable} \end{cases}$$

$$\frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'} \qquad \frac{t \rightsquigarrow t'}{t\ u \rightsquigarrow t'\ u} \qquad \frac{u \rightsquigarrow u'}{t\ u \rightsquigarrow t\ u'} \qquad \frac{t \rightsquigarrow_\beta t'}{t \rightsquigarrow t'} \qquad \frac{t \rightsquigarrow_\iota t'}{t \rightsquigarrow t'}$$

Notation 2.2.13 *(Multi-step reduction).* We write $\rightsquigarrow^*$ for the reflexive transitive closure of $\rightsquigarrow$.

The choice of ι-rule The ι-rule given above is constructed so that the last argument of a ι-redex cannot be a variable. This is a technical solution used in the proof of Lemma 2.2.40. An alternative solution would be to forbid the last argument of a pattern-matching rule being a variable, and when needed let the right-hand sides start with additional abstractions, but then different clauses would have different number of arguments. Such a system would complicate Definition 3.6.2 (Formal call), connected to the size-change analysis in Section 4.1. Yet another solution would be to change the definition of reducibility, (Definition 3.1.8) which would be less natural and involve more complications.

[4] To be used in Lemma 3.6.6 (Key lemma), page 82.

Proposition 2.2.14. $((\lambda x.t)\gamma)u \rightsquigarrow_\beta t[\gamma, u/x]$

Lemma 2.2.15. *If* $f\ \vec{p} = s \in \mathcal{R}$ *then* $f\ \vec{p}\gamma \rightsquigarrow^*\ s\ \gamma$.

Proof. Since $\vec{p}$ is of the form $\vec{q}, \vec{y}$, we have $f\ \vec{q}\gamma, \vec{y}\gamma \rightsquigarrow ((\lambda \vec{y}.s)\ \gamma)(\vec{y}\gamma)$. Then we proceed by iterating the application rule on the substituted expressions. □

Definition 2.2.16 (Equality by convertibility).
The assertion $t \bowtie u$ is defined as $\exists v\ .\ t \rightsquigarrow^*\ v\ \wedge\ u \rightsquigarrow^*\ v$.[5]

Proposition 2.2.17 (Reduction and equality closed under substitution).

$$t \rightsquigarrow^*\ u\ \Rightarrow\ t\ \gamma \rightsquigarrow^*\ u\ \gamma \qquad t \bowtie u\ \Rightarrow\ t\ \gamma \bowtie u\ \gamma$$

Proposition 2.2.18. $t \rightsquigarrow^*\ u\ \Rightarrow\ FV(u) \subseteq FV(t)$.

Raw types closed under reduction

An untyped term which is a non-type might reduce to a type, but a type will always reduce to a type, which can be seen from the grammar of types in Definition 2.1.1. The following properties will be used implicitly throughout the document.

Remark 2.2.19. $T \rightsquigarrow^* \text{Set}\ \Rightarrow\ T = \text{Set}$

Remark 2.2.20. $T \bowtie \text{El}\ t\ \iff\ \exists u\ .\ T = \text{El}\ u\ \wedge\ t \bowtie u$

Remark 2.2.21. $T \bowtie \text{Fun}\ U\ (\lambda x.V)\ \iff$
$\exists U', V'\ .\ T = \text{Fun}\ U'\ (\lambda x.V')\ \wedge\ U \bowtie U'\ \wedge\ V \bowtie V'$

2.2.3 The Church-Rosser property

The property

$$t \rightsquigarrow^*\ u\ \wedge\ t \rightsquigarrow^*\ v\ \Rightarrow\ u \bowtie v$$

called the *Church-Rosser property* or *confluence* is known to hold for a number of combinations of lambda-calculus with rewrite systems, including ours. In our system we are considering a restricted rewrite system, namely functional programs. We have non-overlapping left-linear patterns—also known as *orthogonal rewriting*—combined with β-reduction. Thus we do not need to consider more complicated side conditions, such as so-called *critical pairs*, as necessary in more general term rewriting systems. The confluence of our system should follow from Müller (1992), but their result is stronger than what we actually need here.

Although confluence of our system follows from known results, for being self-contained, we will give a proof using the Martin-Löf–Tait method presented in Martin-Löf (1971, 1972), based on a parallel reflexive reduction relation.

[5] The notation $\bowtie$ is taken from Coquand (1998).

Disjoint and left linear pattern-matching

Definition 2.2.22 (Disjoint patterns).
The patterns $(p_1, \ \ldots \ , p_n)$ and $(q_1, \ \ldots \ , q_n)$ are disjoint if they are not unifiable (as first-order terms).

Definition 2.2.23 (Linear patterns).
The pattern vector $\vec{p}$ is *linear* if each free variable in $\vec{p}$ occurs only once.

The following restriction will be used in the proof of confluent reduction, Lemma 2.2.30, page 35, case 2a for disjointness and 2b for linearity.

Restriction 2.2.24 *(Left-linear disjoint rules).* We assume from now on, that for all constants f, for any rule $f\ \vec{p} = s \in \mathcal{R}$ we have $\vec{p}$ linear. For any two rules $f\ \vec{p} = s_1, f\ \vec{q} = s_2 \in \mathcal{R}$, $\vec{p}$ and $\vec{q}$ are disjoint.

The need of Left-linearity

Allowing non-linear patterns, we would not have a confluent system without introducing further restrictions on our rules. The following example illustrates this:

Example 2.2.25. Consider the constants f and ω:

$$\begin{array}{ll} f\ x\ x = 0 & \\ f\ x\ (\mathrm{s}\ x) = 1 & \qquad \omega = \mathrm{s}\ \omega \end{array}$$

Given the definitions of f and ω above, we can infer both $f\ \omega\ \omega \rightsquigarrow^* 1$ and $f\ \omega\ \omega \rightsquigarrow^* 0$. As we can see, ω has no normal form, and one could argue that this is not a "true" counter example, since the terms we will consider in the end are only normalizing ones. However, when we prove confluence, we do not want to depend on the normalisation property, since that is going to be shown assuming the confluence property.

The need of the parallel reflexive reduction relation

The relation $\rightsquigarrow$ does not have the diamond property, as opposed to its transitive reflexive closure $\rightsquigarrow^*$, that will be proved below. As mentioned in Pollack (1995), the following two examples illustrate the problem.

Example 2.2.26 (Counter example to one-step β diamond property).

1. β is not reflexive:
   We have
   $(\lambda x.y)((\lambda x.x)\ z) \rightsquigarrow (\lambda x.y)\ z \rightsquigarrow y$ and
   $(\lambda x.y)((\lambda x.x)\ z) \rightsquigarrow y$,
   but *not* $y \rightsquigarrow y$.

2. The one-step β-relation does not reduce sub-expressions in parallel:
   We have $(\lambda x.x\ x)((\lambda x.y)\ z) \rightsquigarrow ((\lambda x.y)\ z)\ ((\lambda x.y)\ z)$ and

$(\lambda x.x\ x)((\lambda x.y)\ z) \rightsquigarrow (\lambda x.x\ x)\ y \rightsquigarrow y\ y$,
but *not* $((\lambda x.y)\ z)\ ((\lambda x.y)\ z) \rightsquigarrow y\ y$,
since it would require two steps to obtain.

A similar argument as above can be repeated for the ι-rule, as follows:

Example 2.2.27 (Counter example to one-step ι diamond property).

1. ι is not reflexive:
   Assume given the rule defining f as follows:

$$\mathrm{f}\ x = 0$$

   We have $\mathrm{f}(\mathrm{f}\ x) \rightsquigarrow \mathrm{f}\ 0 \rightsquigarrow 0$,
   and $\mathrm{f}(\mathrm{f}\ x) \rightsquigarrow 0$ but *not* $0 \rightsquigarrow 0$,
   since the one-step ι-relation is not reflexive either.

2. ι is not parallel:
   Assume given the rules defining + and double as follows:

$$x + 0 = x$$
$$x + (\mathrm{s}\ y) = \mathrm{s}\ (x + y)$$

$$\mathrm{double}\ x = x + x$$

   We have $\mathrm{double}(x + 0) \rightsquigarrow \mathrm{double}\ x \rightsquigarrow x + x$,
   and $\mathrm{double}(x + 0) \rightsquigarrow (x + 0) + (x + 0)$,
   but *not* $(x + 0) + (x + 0) \rightsquigarrow x + x$,
   since it would require two steps to obtain. Hence, the one-step ι-relation is not parallel either.

We will follow the Martin-Löf/Tait method and define a parallel reflexive one-step relation.

Definition 2.2.28 (Parallel reflexive reduction). The relation $t \gg u$ is inductively defined as follows:

1. $$\frac{t \gg t' \quad u \gg u'}{(\lambda x.t)\ u \gg t'[u'/x]}$$

2. $$\frac{p_1\ \gamma \gg p_1\ \gamma' \quad \ldots \quad p_n\ \gamma \gg p_n\ \gamma'}{f(p_1\ \gamma)\ \ldots\ (p_n\ \gamma) \gg (\lambda \vec{x}\,.s)\ \gamma'} \begin{cases} f\ p_1\ \ldots\ p_n\ \vec{x} = s\ \in \mathcal{R} \\ p_n \text{ is not a variable} \end{cases}$$

3. $$\frac{}{a \gg a}$$

4. $$\frac{t \gg t'}{\lambda x.t \gg \lambda x.t'}$$

5. $$\frac{t \gg t' \quad u \gg u'}{t\ u \gg t'\ u'}$$

Recall from Definition 2.1.1 (Raw terms), page 26, the category a are atoms of raw terms. Cases 3, 4 and 5 above make the definition reflexive for all terms.

Lemma 2.2.29 (Substitution property of parallel reduction).
$t \gg t' \ \wedge \ \forall x \ . \ \gamma(x) \gg \gamma'(x) \ \Rightarrow \ t \ \gamma \gg t' \ \gamma'$.

Proof. Assume $t \gg t'$ and $\forall x \ . \ \gamma(x) \gg \gamma'(x)$. We prove $t \ \gamma \gg t' \ \gamma'$ by induction on the derivation of $t \gg t'$. We have the following cases:

1. $\dfrac{u \gg u' \quad v \gg v'}{(\lambda x.u) \ v \gg u'[v'/x]}$

   By induction $u \ \gamma \gg u' \ \gamma'$ and $v \ \gamma \gg v' \ \gamma'$.
   By definition $(\lambda x.u \ \gamma) \ (v \ \gamma) \gg (u' \ \gamma')[(v' \ \gamma')/x]$.
   Assume w.l.o.g. (by Remark 2.2.5) that for all y where $y \neq x$,
   $x \notin FV(\gamma(y)), \ x \notin FV(\gamma'(y)), \ \gamma(x) = x$ and $\gamma'(x) = x$.
   By substitution we have $(\lambda x.u \ \gamma) \ (v \ \gamma) = ((\lambda x.u)v) \ \gamma$ and
   $u' \ \gamma'[v' \ \gamma'/x] = u'[v'/x] \ \gamma'$.

2. $\dfrac{p_1 \ \delta \gg p_1 \ \delta' \quad \ldots \quad p_n \ \delta \gg p_n \ \delta'}{f(p_1 \ \delta) \ \ldots \ (p_n \ \delta) \gg (\lambda \vec{x} .s) \ \delta'} \ \begin{cases} f \ p_1 \ \ldots \ p_n \ \vec{x} = s \ \in \mathcal{R} \\ p_n \text{ is not a variable} \end{cases}$

   By induction we have
   $(p_1 \ \delta) \ \gamma \gg (p_1 \ \delta') \ \gamma', \ \ldots \ , \ (p_n \ \delta) \ \gamma \gg (p_n \ \delta') \ \gamma'$.

   We define $\rho(y) = \delta(y) \ \gamma$ and $\rho'(y) = \delta'(y) \ \gamma'$ for all y.

   We have then $(p_i \ \delta) \ \gamma = p_i \ \rho$ and $(p_i \ \delta') \ \gamma' = p_i \ \rho'$.
   Therefore $p_i \ \rho \gg p_i \ \rho'$. By definition then $f \ p_1\rho \ \ldots \ p_n\rho \gg (\lambda \vec{x} .s) \ \rho'$.

   We have $(\lambda \vec{x} .s) \ \rho' = ((\lambda \vec{x} .s) \ \delta') \ \gamma'$.

   Then $(f(p_1 \ \delta) \ \ldots \ (p_n \ \delta))\gamma \gg ((\lambda \vec{x} .s) \ \delta') \ \gamma'$.

3. $\overline{a \gg a}$

   If $a = x$ we have $\gamma(x) \gg \gamma'(x)$ by assumption. Otherwise $a\gamma = a = a\gamma'$, hence $a\gamma \gg a\gamma'$ in both cases.

4. $\dfrac{u \gg u'}{\lambda x.u \gg \lambda x.u'}$

   By induction $u \ \gamma \gg u' \ \gamma'$. By definition $\lambda x.(u \ \gamma) \gg \lambda x.(u' \ \gamma')$.

   Assume w.l.o.g. (by Remark 2.2.5) that for all $y \in FV(u) \cup FV(u')$ where $y \neq x$, we have $x \notin FV(\gamma(y)), \ x \notin FV(\gamma'(y)), \ \gamma(x) = x$ and $\gamma'(x) = x$.
   By substitution $\lambda x.(u \ \gamma) = (\lambda x.u) \ \gamma$ and $\lambda x.(u \ \gamma') = (\lambda x.u) \ \gamma'$,
   hence $(\lambda x.u) \ \gamma \gg (\lambda x.u') \ \gamma'$.

5. $\dfrac{u \gg u' \quad v \gg v'}{u \ v \gg u' \ v'}$

   By induction we have $u \ \gamma \gg u' \ \gamma'$ and $v \ \gamma \gg v' \ \gamma'$.
   By definition we have $(u \ \gamma) \ (v \ \gamma) \gg (u' \ \gamma') \ (v' \ \gamma')$.
   By substitution we have $(u \ v) \ \gamma \gg (u' \ v') \ \gamma'$.

□

Lemma 2.2.30 (Diamond property of parallel reduction).
If $t_1 \gg t_2$ and $t_1 \gg t_3$, then there is t_4 such that $t_2 \gg t_4$ and $t_3 \gg t_4$.

Proof. By induction on the sum of the lengths of the derivations of $t_1 \gg t_2$ and $t_1 \gg t_3$. We enumerate the cases for $t_1 \gg t_2$, and if necessary, for $t_1 \gg t_3$:

1. When at least one of the reductions $t_1 \gg t_2$ or $t_1 \gg t_3$ is an instance of rule 1 of Definition 2.2.28, then t_1 is an expression of the form $(\lambda x.u_1)\ v_1$, and we get the following possible sub-cases:

   (a) $\dfrac{u_1 \gg u_2 \quad v_1 \gg v_2}{(\lambda x.u_1)\ v_1 \gg u_2[v_2/x]}$ $\qquad$ $\dfrac{u_1 \gg u_3 \quad v_1 \gg v_3}{(\lambda x.u_1)\ v_1 \gg u_3[v_3/x]}$

   By induction there is u_4, v_4 such that
   $u_2 \gg u_4$ and $u_3 \gg u_4$ and $v_2 \gg v_4$ and $v_3 \gg v_4$.
   By applying Lemma 2.2.29 twice we get $u_2[v_2/x] \gg u_4[v_4/x]$ and $u_3[v_3/x] \gg u_4[v_4/x]$ respectively.

   (b) $\dfrac{u_1 \gg u_2 \quad v_1 \gg v_2}{(\lambda x.u_1)\ v_1 \gg u_2[v_2/x]}$ $\qquad$ $\dfrac{\lambda x.u_1 \gg \lambda x.u_3 \quad v_1 \gg v_3}{(\lambda x.u_1)\ v_1 \gg (\lambda x.u_3)\ v_3}$

   By induction there is u_4, v_4 such that
   $u_2 \gg u_4$ and $u_3 \gg u_4$ and $v_2 \gg v_4$ and $v_3 \gg v_4$.
   By definition $(\lambda x.u_3)\ v_3 \gg u_4[v_4/x]$.
   By Lemma 2.2.29 $u_2[v_2/x] \gg u_4[v_4/x]$.

   (c) $\dfrac{\lambda x.u_1 \gg \lambda x.u_2 \quad v_1 \gg v_2}{(\lambda x.u_1)\ v_1 \gg (\lambda x.u_2)\ v_2}$ $\qquad$ $\dfrac{u_1 \gg u_3 \quad v_1 \gg v_3}{(\lambda x.u_1)\ v_1 \gg u_3[v_3/x]}$

   Symmetric to the previous case.

2. When at least one of the reductions $t_1 \gg t_2$ or $t_1 \gg t_3$ is an instance of rule 2 of Definition 2.2.28, then t_1 is an expression of the form $f\ (p_1\ \gamma_1)\ \ldots\ (p_n\ \gamma_1)$, and we get the following possible sub-cases:

   (a) $\begin{cases} \dfrac{p_1\ \gamma_1 \gg p_1\ \gamma_2 \quad \ldots \quad p_n\ \gamma_1 \gg p_n\ \gamma_2}{f(p_1\ \gamma_1)\ \ldots\ (p_n\ \gamma_1) \gg (\lambda \vec{x}\,.s)\ \gamma_2} & \begin{cases} f\ p_1\ \ldots\ p_n\ \vec{x} = s\ \in \mathcal{R} \\ p_n \text{ is not a variable} \end{cases} \\ \\ \dfrac{p_1\ \gamma_1 \gg p_1\ \gamma_3 \quad \ldots \quad p_n\ \gamma_1 \gg p_n\ \gamma_3}{f(p_1\ \gamma_1)\ \ldots\ (p_n\ \gamma_1) \gg (\lambda \vec{x}\,.s)\ \gamma_3} & \begin{cases} f\ p_1\ \ldots\ p_n\ \vec{x} = s\ \in \mathcal{R} \\ p_n \text{ is not a variable} \end{cases} \end{cases}$

   By Restriction 2.2.24 all the rules for f in $\mathcal{R}$ are mutually disjoint, and the term $f(p_1\ \gamma_1)\ \ldots\ (p_n\ \gamma_1)$ then uniquely determines the matching rule $f\ p_1\ \ldots\ p_n\ \vec{x} = s\ \in \mathcal{R}$.
   By induction there is γ_4 such that $p_i\ \gamma_2 \gg p_i\ \gamma_4$ and $p_i\ \gamma_3 \gg p_i\ \gamma_4$.
   By definition we have
   $f(p_1\ \gamma_2)\ \ldots\ (p_n\ \gamma_2) \gg (\lambda \vec{x}\,.s)\ \gamma_4$ and $f(p_1\ \gamma_3)\ \ldots\ (p_n\ \gamma_3) \gg (\lambda \vec{x}\,.s)\ \gamma_4$.
   We have $\gamma_2(x) \gg \gamma_4(x)$ and $\gamma_3(x) \gg \gamma_4(x)$ for all $x \in FV(p_i)$.
   By using Lemma 2.2.29 twice, we have
   $(\lambda \vec{x}\,.s)\ \gamma_2 \gg (\lambda \vec{x}\,.s)\ \gamma_4$ and $(\lambda \vec{x}\,.s)\ \gamma_3 \gg (\lambda \vec{x}\,.s)\ \gamma_4$ respectively.

(b)
$$\begin{cases} \dfrac{p_1\ \gamma_1 \gg p_1\ \gamma_2 \quad \dots \quad p_n\ \gamma_1 \gg p_n\ \gamma_2}{f(p_1\ \gamma_1)\ \dots\ (p_n\ \gamma_1) \gg (\lambda \vec{x}.s)\ \gamma_2} & \begin{cases} f\ p_1\ \dots\ p_n\ \vec{x} = s\ \in \mathcal{R} \\ p_n \text{ is not a variable} \end{cases} \\ \dfrac{p_1\ \gamma_1 \gg u_1 \quad \dots \quad p_n\ \gamma_1 \gg u_n}{f(p_1\ \gamma_1)\ \dots\ (p_n\ \gamma_1) \gg f\ u_1\ \dots\ u_n} & \end{cases}$$

By Restriction 2.2.24 all the rules for f in $\mathcal{R}$ are left-linear, and then u_i are expressions of the form $p_i\ \gamma_3$, where $\gamma_1(x) \gg \gamma_3(x)$ for all $x \in FV(p_1, \dots, p_n)$
By induction there is γ_4 such that $p_i\ \gamma_2 \gg p_i\ \gamma_4$ and $p_i\ \gamma_3 \gg p_i\ \gamma_4$.
By definition $f(p_1\ \gamma_3)\ \dots\ (p_n\ \gamma_3) \gg (\lambda \vec{x}.s)\ \gamma_4$.
We have $\gamma_2(x) \gg \gamma_4(x)$ for all $x \in FV(p_i)$.
By Lemma 2.2.29 we have $(\lambda \vec{x}.s)\ \gamma_2 \gg (\lambda \vec{x}.s)\ \gamma_4$.

(c)
$$\begin{cases} \dfrac{p_1\ \gamma_1 \gg u_1 \quad \dots \quad p_n\ \gamma_1 \gg u_n}{f(p_1\ \gamma_1)\ \dots\ (p_n\ \gamma_1) \gg f\ u_1\ \dots\ u_n} & \\ \dfrac{p_1\ \gamma_1 \gg p_1\ \gamma_3 \quad \dots \quad p_n\ \gamma_1 \gg p_n\ \gamma_3}{f(p_1\ \gamma_1)\ \dots\ (p_n\ \gamma_1) \gg (\lambda \vec{x}.s)\ \gamma_3} & \begin{cases} f\ p_1\ \dots\ p_n\ \vec{x} = s\ \in \mathcal{R} \\ p_n \text{ is not a variable} \end{cases} \end{cases}$$

This case is symmetric to the previous one.

3. $\dfrac{}{a \gg a}$

   Then t_4 is a.

4. $\dfrac{u_1 \gg u_2}{\lambda x.u_1 \gg \lambda x.u_2} \qquad \dfrac{u_1 \gg u_3}{\lambda x.u_1 \gg \lambda x.u_3}$

   By induction there is u_4 such that $u_2 \gg u_4$ and $u_3 \gg u_4$.
   By definition $\lambda x.u_2 \gg \lambda x.u_4$ and $\lambda x.u_3 \gg \lambda x.u_4$ respectively.

5. $\dfrac{u_1 \gg u_2 \quad v_1 \gg v_2}{u_1\ v_1 \gg u_2\ v_2} \qquad \dfrac{u_1 \gg u_3 \quad v_1 \gg v_3}{u_1\ v_1 \gg u_3\ v_3}$

   By induction there is u_4, v_4 such that
   $u_2 \gg u_4,\ u_3 \gg u_4$ and $v_2 \gg v_4,\ v_3 \gg v_4$.
   By definition $u_2\ v_2 \gg u_4\ v_4$ and $u_3\ v_3 \gg u_4\ v_4$.

□

Definition 2.2.31 (Multi-step parallel reduction).
The relation $t \gg_n u$ is inductively defined as follows:

$$\frac{}{t \gg_0 t} \qquad \frac{t \gg_m u \quad u \gg v}{t \gg_{m+1} v}$$

Proposition 2.2.32 (Confluence).
If $t \rightsquigarrow^* u$ and $t \rightsquigarrow^* v$ then $t \bowtie u$.

Proof. It is clear that $t \rightsquigarrow^* u$ if and only if $t \gg_n u$ for some n.
Assume $t \rightsquigarrow^* u$ and $t \rightsquigarrow^* v$. Then $t \gg_m u$ for some m, and $t \gg_n v$ for some n. By repeatedly applying Lemma 2.2.30 we can find w such that $u \gg_n w$ and $v \gg_m w$. Hence $u \rightsquigarrow^* w$ and $v \rightsquigarrow^* w$. □

Corollary 2.2.33. *The relation $\bowtie$ is an equivalence relation.*

2.2.4 Notions of normal form and normalization

Note that we consider reduction of open terms, and reduction is performed under abstractions as well as constructors. Therefore every sub-term of a normal term is normal. We will consider *weak normalization*, asserting the existence of a reduction sequence leading to a normal form.

Definition 2.2.34 (Normal form and normalizability).

- $\mathsf{NF}(t)$ means that there is no t' such that $t \rightsquigarrow t'$.
- $\mathsf{WN}(t)$ means that there exists t' such that $t \rightsquigarrow^* t'$, where $\mathsf{NF}(t')$.

Definition 2.2.35 (β-normality). A term t is beta normal iff t contains no sub-term of the form $(\lambda x.u)\ v$.

Remark 2.2.36. The syntactical categories s, S and Ξ are β-normal.[6]

Notation 2.2.37. Let $u \Downarrow v$ be a shorthand for $u \rightsquigarrow^* v \wedge \mathsf{NF}(v)$.

Notation 2.2.38.
Let $(t_1, \ldots, t_n) \Downarrow (u_1, \ldots, u_n)$ be a shorthand for $t_1 \Downarrow u_1, \ldots, t_n \Downarrow u_n$.

Proposition 2.2.39 (Uniqueness of normal form).
If $u \bowtie v$ and $u \Downarrow w$ then $v \Downarrow w$.

Proof. By Proposition 2.2.32 (Confluence). □

The following lemma will be used in the proof of Proposition 3.3.3.

Lemma 2.2.40 (Normalizable variable application).
$\mathsf{WN}(t\ x) \Rightarrow \mathsf{WN}(t)$.

Proof. Assume $\mathsf{WN}(t\ x)$. Then there is a reduction sequence

$$t\ x \rightsquigarrow t_1 \rightsquigarrow t_2 \rightsquigarrow \ \ldots \ \rightsquigarrow t_n$$

such that t_n is normal. We proceed by induction on n. If $n = 0$, then $t\ x$ is normal, hence t is normal. Otherwise $t\ x$ contains a redex. By Definition 2.2.12 (Reduction), $t\ x$ itself cannot be a ι-redex, and the first step $t\ x \rightsquigarrow t_1$ must be derived by one of the following rules:

1. $\dfrac{}{(\lambda y.u)\ x \rightsquigarrow_\beta u[x/y]}$

   Since $u[x/y]$ is normalizable, so is $u[y/y]$, hence $\mathsf{WN}(t)$.

2. $\dfrac{t \rightsquigarrow u}{t\ x \rightsquigarrow u\ x}$

   We have $u\ x = t_1$. Since t_1 reduces to t_n in $n-1$ steps, we have $\mathsf{WN}(u\ x)$. By induction $\mathsf{WN}(u)$. We have $t \rightsquigarrow u$, hence $\mathsf{WN}(t)$.

□

[6] Their syntax are given in Definition 2.1.20, page 28.

2.3 Type system

2.3.1 Rules of inference

Definition 2.3.1 (Formation and inhabitation of types).
The relations $\vdash \Gamma$ (context formation), $\Gamma \vdash T$ (type formation) and $\Gamma \vdash t : T$ (type inhabitation) are mutually inductively defined as follows:

- **Context formation**

$$\frac{}{\vdash ()} \qquad \frac{\Gamma \vdash T}{\vdash \Gamma, x : T}\ x \notin \mathit{supp}(\Gamma)$$

- **Type formation**

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathrm{Set}} \qquad \frac{\Gamma \vdash t : \mathrm{Set}}{\Gamma \vdash \mathrm{El}\ t} \qquad \frac{\Gamma, x : U \vdash V}{\Gamma \vdash (x : U) \to V}$$

- **Type inhabitation**

$$\frac{\vdash \Gamma}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash t : U \quad \Gamma \vdash T}{\Gamma \vdash t : T}\ U \bowtie T$$

$$\frac{\Gamma \vdash t : (x : U) \to V \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : V[u/x]} \qquad \frac{\Gamma, x : U \vdash v : V}{\Gamma \vdash \lambda x.v : (x : U) \to V}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash f : \mathcal{F}(f)} \qquad \frac{\vdash \Gamma}{\Gamma \vdash d : \mathcal{D}(d)} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash u_1 : \mathrm{Set} \quad \ldots \quad \Gamma \vdash u_k : \mathrm{Set}}{\Gamma \vdash c : \mathcal{C}(c)[u_1, \ \ldots\ , u_k]}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \Pi : (x : \mathrm{Set},\ \mathrm{El}\ x \to \mathrm{Set}) \to \mathrm{Set}}$$

$$\frac{\Gamma \vdash t : \mathrm{Set} \quad \Gamma \vdash u : \mathrm{El}\ t \to \mathrm{Set}}{\Gamma \vdash \mathrm{fun}\ : ((x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)) \to \mathrm{El}\ (\Pi\ t\ u)}\ x \notin FV(u)$$

where
$\mathcal{D}(d) = \mathrm{Set}^n \to \mathrm{Set}$
$\mathcal{C}(c) = (\mathrm{El}\ e_1, \ \ldots\ , \mathrm{El}\ e_n) \to \mathrm{El}\ (d\ x_1\ \ldots\ x_k)$
with $FV(e_i) \subseteq \{x_1, \ \ldots\ , x_k\}$.
Note that in the constructor rule, the premise $\vdash \Gamma$ is needed since k may be 0.

Remark 2.3.2. The language, and so the set of defined constants f, can be seen as fixed throughout the whole text. However, the typing specifications $\mathcal{F}$ may be undefined for some constants, in the case where $\mathcal{R}$ have rules for constants not declared in $\mathcal{F}$. In Definition 2.3.17 below, we describe what is a type-correct signature, and only for such signature we are sure that $\mathcal{F}$ will always have a type for f.

Definition 2.3.3 (Inhabitation of vectors).
We define the relation $\Delta \vdash \vec{u} : \Theta$, where $\vdash \Theta$ holds and Θ and $\vec{u}$ have the same lengths n, by induction on n by the clauses:

$$\frac{}{() \vdash () : ()} \qquad \frac{\Delta \vdash t : T \quad \Delta \vdash \vec{t} : \Gamma[t/x]}{\Delta \vdash t, \vec{t} : (x : T, \Gamma)}$$

2.3.2 Basic inversion properties

Lemma 2.3.4 (Type-formation inversion).

1. $\Gamma \vdash \mathrm{Set} \;\Rightarrow \vdash \Gamma$
2. $\Gamma \vdash \mathrm{El}\ t \;\Rightarrow\; \Gamma \vdash t : \mathrm{Set}$
3. $\Gamma \vdash (x : U) \to V \;\Rightarrow\; \Gamma, x : U \vdash V$

Proof. Direct from Definition 2.3.1. □

Lemma 2.3.5 (Iterated function type inversion).
If $\Gamma \vdash (x_1 : T_1, \ \ldots\ , x_n : T_n) \to T$, *then*
$\Gamma \vdash T_1$, $\Gamma, x_1 : T_1 \vdash T_2$ *through* $\Gamma, x_1 : T_1, \ \ldots\ , x_{n-1} : T_{n-1} \vdash T_n$ *and*
$\Gamma, x_1 : T_1, \ \ldots\ , x_n : T_n \vdash T$.

Proof. By iterated inversion of function type formation. □

Lemma 2.3.6 (Context formation from typing). *If* $\mathfrak{D}$ *is a derivation of* $\Gamma \vdash T$ *or* $\Gamma \vdash t : T$ *then* $\mathfrak{D}$ *contains a sub-derivation* $\mathfrak{D}'$ *of* $\vdash \Gamma$.

Proof. By induction on the derivations. □

Corollary 2.3.7. *If* $\mathfrak{D}$ *is a derivation of* $\Gamma, x : U \vdash T$ *or* $\Gamma, x : U \vdash t : T$ *then* $\mathfrak{D}$ *contains a sub-derivation* $\mathfrak{D}'$ *of* $\Gamma \vdash U$.

Lemma 2.3.8.

1. $\Gamma \vdash T \;\Rightarrow\; FV(T) \subseteq \mathit{supp}(\Gamma)$.
2. $\Gamma \vdash t : T \;\Rightarrow\; FV(t) \subseteq \mathit{supp}(\Gamma)$.

Proof. By induction on the derivations and Lemma 2.3.6. □

Corollary 2.3.9. *If* $\vdash t : T$ *then* t *and* T *are closed.*

2.3.3 Typing and patterns

Notation 2.3.10 *(Pattern substitutions).* Substitutions of the form $[y/x]$, $[\text{fun } y/x]$ or $[c\ \vec{y}\,/x]$ will be denoted α, and compositions of substitutions of the form $\alpha_1\ \ldots\ \alpha_n$ will be denoted τ.

For the following definition, please recall Notation 2.2.9, page 30, for substitution in contexts. The name *neighbourhood* in the definition below comes from Martin-Löf's terminology, as referred to from Coquand (1992).

Definition 2.3.11 (Atomic neighbourhood).
Given $\vdash \Delta$ and $\vdash \Gamma$, the relation $\Delta \xrightarrow{\alpha} \Gamma$ is defined by the clauses

1. $\Gamma_1, y : T, \Gamma_2[y/x] \xrightarrow{[y/x]} \Gamma_1, x : T, \Gamma_2$

2. $\Gamma_1, y : (z : \text{El } t) \to \text{El } (u\ z), \Gamma_2[\text{fun } y/x] \xrightarrow{[\text{fun } y/x]} \Gamma_1, x : T, \Gamma_2$
   where $T \bowtie \text{El } (\Pi\ t\ u)$, $\Gamma_1 \vdash \text{El } (\Pi\ t\ u)$ and $z \notin FV(u)$.

3. $\Gamma_1, \Delta[\vec{t}\,/\vec{z}], \Gamma_2[c\ \vec{y}\,/x] \xrightarrow{[c\ \vec{y}\,/x]} \Gamma_1, x : T, \Gamma_2$
   where $\mathcal{C}(c) = \Delta \to \text{El } (d\ \vec{z}\,)$, $\vec{y} = \overrightarrow{supp}(\Delta)$,
   $T \bowtie \text{El } (d\ \vec{t}\,)$ and $\Gamma_1 \vdash \text{El } (d\ \vec{t}\,)$.

In the above definition we have assumed $\vdash \Delta$ for technical reasons. Without this assumption we would need subject reduction, which is proved only later (Lemma 2.4.18, page 55 below). Once we have proved this, we can drop the extra condition.

Remark 2.3.12. When $\Gamma_1, y_1 : U_1,\ \ldots\ , y_m : U_m, \Gamma_2\alpha \xrightarrow{\alpha} \Gamma_1, x : T, \Gamma_2$ holds, then $x \notin FV(T) \cup FV(U_i)$, since $\vdash \Delta$ holds.

Definition 2.3.13 (Compound neighbourhood). Given $\vdash \Delta$ and $\vdash \Gamma$, the relation $\Delta \xrightarrow{\tau} \Gamma$, where τ is a composition of atomic neighbourhoods, is inductively defined by the rules

$$\frac{}{\Gamma \xrightarrow{[]} \Gamma} \qquad \frac{\Theta \xrightarrow{\tau} \Delta \quad \Delta \xrightarrow{\alpha} \Gamma}{\Theta \xrightarrow{\alpha\ \tau} \Gamma}$$

Notation 2.3.14 *(Neighbourhoods obtained from patterns).* We write $\Delta \xrightarrow{[\vec{p}\,/\vec{x}\,]} \Gamma$, where $\vec{x} = \overrightarrow{supp}(\Gamma)$, when we can write $[\vec{p}\,/\vec{x}\,]$ as a neighbourhood τ such that $\vec{x}\,\tau = \vec{p}$ and $\Delta \xrightarrow{\tau} \Gamma$ holds. When there is no confusion about $\vec{x}$, we may omit it.

Note that $\Delta \xrightarrow{\tau} \Gamma$ may have infinitely many possible derivations, and also that Δ is not unique. The properties we are going to use about neighbourhoods will not depend on particular derivations of neighbourhoods, but only on the fact that such derivations exist. Moreover, as we will show in Section 5.1, we can find the unique normal form Δ' of Δ, satisfying $\Delta' \xrightarrow{\tau} \Gamma$.

2.3.4 Typing and the signature

Definition 2.3.15 (Typing conditions of $\mathcal{D}$ and $\mathcal{C}$).
Let $\vdash (\mathcal{D}, \mathcal{C})$ be the property[7]
$\forall d\ .\ \mathcal{D}(d) = \mathrm{Set}^k \to \mathrm{Set}\ \Rightarrow$
$\quad \forall c\ .\ \mathcal{C}(c) = (\mathrm{El}\ e_1,\ \ldots\ , \mathrm{El}\ e_n) \to d\ x_1\ \ldots\ x_k\ \Rightarrow$
$\quad\quad x_1 : \mathrm{Set},\ \ldots\ , x_k : \mathrm{Set} \vdash \mathcal{C}(c)$

Definition 2.3.16 (Typing conditions of $\mathcal{F}$ and $\mathcal{R}$).
Let $\vdash \mathcal{F}$ be the property $\forall f \in \mathcal{F}. \vdash \mathcal{F}(f)$. If $\vdash \mathcal{F}$ holds, then $\vdash \mathcal{R}$ holds iff for all f such that $\mathcal{F}(f) = \Gamma_f \to T_f$, for all rules $f\ \vec{p} = s$ in $\mathcal{R}$ there exists Δ such that $\Delta \xrightarrow{[\vec{p}]} \Gamma_f$ and $\Delta \vdash s : T_f[\vec{p}]$.

Definition 2.3.17 (Typing conditions of Σ).
Let $\vdash \Sigma$ be the property $\vdash (\mathcal{D}, \mathcal{C})\ \wedge\ \vdash \mathcal{F}\ \wedge\ \vdash \mathcal{R}$

Notation 2.3.18. We may write $\vdash_\Sigma$ in a typing judgement to make explicit that it is made with respect to a particular signature Σ. When omitting the subscript, we mean the whole signature, as explained in Restriction 2.1.22. To be more explicit in the above definition, we could have written $\vdash_\Sigma \mathcal{D}, \mathcal{C} \wedge \vdash_\Sigma \mathcal{F} \wedge \vdash_\Sigma \mathcal{R}$.

2.3.5 Examples

We give some examples to show how a definition is validated in our system. These should also give an impression of what kinds of type-theoretic definitions fit in the framework. Before reading further, it may be helpful to recall the notational conventions discussed in Section 2.1.3, page 27.

Notation 2.3.19. In the examples we will allow ourselves to choose identifier names differing from the notation given in Definition 2.1.1. We will use uppercase names for set-valued identifiers and types, and lower case for other objects.

The Cartesian product of a family of sets

We have one built-in *higher-order* but *non-recursive* parameterized data type:

$$\Pi : (x : \mathrm{Set},\ \mathrm{El}\ x \to \mathrm{Set}) \to \mathrm{Set}$$

with constructor

$$\mathrm{fun}\ : ((x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)) \to \mathrm{El}\ (\Pi\ t\ u)$$

[7] Recall Definition 2.1.17 (Data type specification).

We can define the elimination rule formulated in Martin-Löf (1984) (see also Nordström *et al.*, 1990) in our framework as follows:

$$\begin{aligned}\text{funsplit} \ : & (A : \text{Set}, \\ & F : \text{El}\ A \to \text{Set}, \\ & G : \text{El}\ (\Pi\ A\ F) \to \text{Set}, \\ & (f : (a : \text{El}\ A) \to \text{El}\ (F\ a)) \to \text{El}\ (G(\text{fun}\ f)), \\ & b : \text{El}\ (\Pi\ A\ F)) \\ & \to \text{El}\ (G\ b) \\ \text{funsplit}\ A\ F\ G\ g\ (\text{fun}\ f) &= g\ f\end{aligned}$$

To establish $\vdash \mathcal{R}$ in this case we have to find Δ such that[8]

$$\begin{aligned}\Delta \overset{[A,F,G,g,\text{fun}\ f]}{\longrightarrow} & (A : \text{Set},\ F : \text{El}\ A \to \text{Set},\ G : \text{El}\ (\Pi\ A\ F) \to \text{Set}, \\ & g : (f : (a : \text{El}\ A) \to \text{El}\ (F\ a)) \to \text{El}\ (G(\text{fun}\ f)), \\ & b : \text{El}\ (\Pi\ A\ F))\end{aligned}$$

holds, and verify $\Delta \vdash g\ f : \text{El}\ (G\ (\text{fun}\ f))$. We have

$$\begin{aligned}\Delta = (& A : \text{Set}, \\ & F : \text{El}\ A \to \text{Set}, \\ & G : \text{El}\ (\Pi\ A\ F) \to \text{Set}, \\ & g : (f : (a : \text{El}\ A) \to \text{El}\ (F\ a)) \to \text{El}\ (G(\text{fun}\ f)), \\ & f : (x : \text{El}\ A) \to\ \text{El}\ (F\ x))\end{aligned}$$

and we see that $g\ f$ has the correct type, by the application rule.

Set-valued functions

Following Smith (1989) we can define propositional functions as families of types. In a programming point of view, these are rather set-valued functions. The following examples illustrate how such functions can be used for programming. We can define the type of vectors of length n as follows:[9]

$$\begin{aligned}&\text{Vec}\ : (\text{Set}, \text{Nat}) \to \text{Set} \\ &\text{Vec}\ A\ 0 = \top \\ &\text{Vec}\ A\ (\text{s}\ n) = A \times \text{Vec}\ A\ n\end{aligned}$$

We can also define the set of finite sets of size n:

$$\begin{aligned}&\text{Fin}\ : \text{Nat} \to \text{Set} \\ &\text{Fin}\ 0 = \bot \\ &\text{Fin}\ (\text{s}\ n) = \text{Fin}\ n + \top\end{aligned}$$

We can combine these to implement a safe indexing function for vectors. Given a number n and $m \leq n + 1$ and a non-empty vector v of length $n + 1$, return v at position m.

[8] Recall Notation 2.3.14, page 40.

[9] Recall Notation 1.2.1, page 8, for the following examples.

$$\begin{array}{l}\text{index} : (A : \text{Set},\ n : \text{Nat},\ \text{Fin}\ (\text{s}\ n),\ \text{Vec}\ A\ (\text{s}\ n)) \to A\\ \text{index}\ A\ 0\ m\ (a, v') = a\\ \text{index}\ A\ (\text{s}\ n')\ (\text{inl}\ m')\ (a, v') = \text{index}\ A\ n'\ m'\ v'\\ \text{index}\ A\ (\text{s}\ n')\ (\text{inr}\ u)\ (a, v') = a\end{array}$$

Nested data-types

Nested data-types can be defined in our system. Here follows a flattening function for such a type.[10] The 'append' constant is the usual list append function. We are not investigating nested types here, but as they can be defined in our system, we give this example. It shows how a set pattern[11] e can be instantiated in the scheme for constructor arguments.

$$\begin{array}{l}\text{Square} : \text{Set} \to \text{Set}\\ \quad \text{pair} : A \to A \to \text{Square}\ A\end{array}$$

$$\begin{array}{l}\text{Pow} : \text{Set} \to \text{Set}\\ \quad \text{zeroP} : A \to \text{Pow}\ A\\ \quad \text{succP} : \text{Pow}(\text{Square}\ A) \to \text{Pow}\ A\end{array}$$

$$\begin{array}{l}\text{aux} : (A : \text{Set},\ B : \text{Set},\ B \to \text{List}\ A,\ \text{Square}\ B) \to \text{List}\ A\\ \quad \text{aux}\ A\ B\ f\ (\text{pair}\ x_1\ x_2) = \text{append}\ A\ (f\ x_1)\ (f\ x_2)\end{array}$$

$$\begin{array}{l}\text{flatP} : (A : \text{Set},\ B : \text{Set},\ B \to \text{List}\ A,\ \text{Pow}\ B) \to \text{List}\ A\\ \quad \text{flatP}\ A\ B\ f\ (\text{zeroP}\ a) = f\ a\\ \quad \text{flatP}\ A\ B\ f\ (\text{succP}\ psa) = \text{flatP}\ A\ (\text{Square}\ B)\ (\lambda x\ .\ \text{aux}\ A\ B\ f\ x)\ psa\end{array}$$

$$\begin{array}{l}\text{flatPow} : (A : \text{Set},\ \text{Pow}\ A) \to \text{List}\ A\\ \quad \text{flatPow}\ A\ p = \text{flatP}\ A\ A\ (\lambda x\ .\ x :: [])\ p\end{array}$$

Later we will see that the results about normalization and decidable type-correctness in sections 3.3 and 5.1 apply to the examples above as special cases.

2.4 Properties of the type system

2.4.1 Thinning and weakening

Lemma 2.4.1 (Thinning).

1. $\Gamma, \Delta \vdash T\ \wedge\ \vdash \Gamma, y : W, \Delta\ \Rightarrow\ \Gamma, y : W, \Delta \vdash T$

2. $\Gamma, \Delta \vdash t : T\ \wedge\ \vdash \Gamma, y : W, \Delta\ \Rightarrow\ \Gamma, y : W, \Delta \vdash t : T$

[10] Thanks to Ulf Norell.

[11] See Definition 2.1.3 (set patterns), page 26 and Definition 2.1.17 (data type), page 28.

Proof. We proceed by simultaneous induction on the derivations of $\Gamma, \Delta \vdash T$ and $\Gamma, \Delta \vdash t : T$.

1. Assume $\vdash \Gamma, y : W, \Delta$ and $\Gamma, \Delta \vdash T$. We have the following cases:

   (a) $\dfrac{\vdash \Gamma, \Delta}{\Gamma, \Delta \vdash \mathrm{Set}}$

   In this case $\Gamma, y : W, \Delta \vdash \mathrm{Set}$ follows by definition from $\vdash \Gamma, y : W, \Delta$.

   (b) $\dfrac{\Gamma, \Delta \vdash u : \mathrm{Set}}{\Gamma, \Delta \vdash \mathrm{El}\ u}$

   By induction (2) we have $\Gamma, y : W, \Delta \vdash u : \mathrm{Set}$.
   By definition then $\Gamma, y : W, \Delta \vdash \mathrm{El}\ u$ holds.

   (c) $\dfrac{\Gamma, \Delta, x : U \vdash V}{\Gamma, \Delta \vdash (x : U) \to V}$

   By Corollary 2.3.7, we have $\Gamma, \Delta \vdash U$ in a sub-derivation of $\Gamma, \Delta, x : U \vdash V$.
   By assumption $\vdash \Gamma, y : W, \Delta$. Assume w.l.o.g. $x \notin \mathit{supp}(\Gamma, y : W, \Delta)$. Then $\vdash \Gamma, y : W, \Delta, x : U$.
   By induction (1) we have $\Gamma, y : W, \Delta, x : U \vdash V$.
   By definition then $\Gamma, y : W, \Delta \vdash (x : U) \to V$.

2. Assume $\vdash \Gamma, y : W, \Delta$ and $\Gamma, \Delta \vdash t : T$. We have the following cases:

   (a) $\dfrac{\vdash \Gamma, \Delta}{\Gamma, \Delta \vdash x : (\Gamma, \Delta)(x)}$

   By assumption $\vdash \Gamma, y : W, \Delta$. Then since $\Gamma, y : W, \Delta$ is disjoint, we have $y \neq x$, and so $(\Gamma, \Delta)(x) = (\Gamma, y : W, \Delta)(x)$.
   Hence $\Gamma, y : W, \Delta \vdash x : (\Gamma, \Delta)(x)$.

   (b) $\dfrac{\Gamma, \Delta \vdash t : U \quad \Gamma, \Delta \vdash T}{\Gamma, \Delta \vdash t : T}\ U \bowtie T$

   By induction (2) we have $\Gamma, y : W, \Delta \vdash t : U$.
   By induction (1) we have $\Gamma, y : W, \Delta \vdash T$.
   By definition then $\Gamma, y : W, \Delta \vdash t : T$.

   (c) $\dfrac{\Gamma, \Delta \vdash u : (x : U) \to V \quad \Gamma, \Delta \vdash v : U}{\Gamma, \Delta \vdash u\ v : V[v/x]}$

   By induction (2) we have $\Gamma, y : W, \Delta \vdash u : (x : U) \to V$.
   By induction (2) we have $\Gamma, y : W, \Delta \vdash v : U$.
   By definition then $\Gamma, y : W, \Delta \vdash u\ v : V[v/x]$.

(d) $$\frac{\Gamma, \Delta, x : U \vdash v : V}{\Gamma, \Delta \vdash \lambda x.v : (x : U) \to V}$$

By Corollary 2.3.7, we have $\Gamma, \Delta \vdash U$ in a sub-derivation of $\Gamma, \Delta, x : U \vdash v : V$.

By assumption $\vdash \Gamma, y : W, \Delta$. Assume w.l.o.g. $x \notin supp(\Gamma, y : W, \Delta)$. Then $\vdash \Gamma, y : W, \Delta, x : U$.

By induction (2) we have $\Gamma, y : W, \Delta, x : U \vdash v : V$.

By definition then $\Gamma, y : W, \Delta \vdash \lambda x.v : (x : U) \to V$.

(e) $$\frac{\vdash \Gamma, \Delta}{\Gamma, \Delta \vdash f : \mathcal{F}(f)}$$

In this case $\Gamma, y : W, \Delta \vdash t : T$ follows by definition from $\vdash \Gamma, y : W, \Delta$.

(f) $$\frac{\vdash \Gamma, \Delta}{\Gamma, \Delta \vdash d : \mathcal{D}(d)}$$

In this case $\Gamma, y : W, \Delta \vdash t : T$ follows by definition from $\vdash \Gamma, y : W, \Delta$.

(g) $$\frac{\vdash \Gamma, \Delta \quad \Gamma, \Delta \vdash u_1 : \mathrm{Set} \quad \ldots \quad \Gamma, \Delta \vdash u_k : \mathrm{Set}}{\Gamma, \Delta \vdash c : \mathcal{C}(c)[u_1, \ \ldots \ , u_k]} \begin{cases} \mathcal{D}(d) = \mathrm{Set}^k \to \mathrm{Set} \\ \mathcal{C}(c) = (\mathrm{El}\ e_1, \ \ldots \ , \mathrm{El}\ e_n) \\ \qquad\qquad \to \mathrm{El}\ (d\ x_1\ \ldots\ x_k) \\ FV(e_i) \subseteq \{x_1, \ \ldots \ , x_k\} \end{cases}$$

By induction (2) we have $\Gamma, y : W, \Delta \vdash u_i : \mathrm{Set}$. Then by definition $\Gamma, y : W, \Delta \vdash c : \mathcal{C}(c)[u_1, \ \ldots \ , u_k]$.

(h) $$\frac{\vdash \Gamma, \Delta}{\Gamma, \Delta \vdash \Pi : (x : \mathrm{Set}, \ \mathrm{El}\ x \to \mathrm{Set}) \to \mathrm{Set}}$$

In this case $\Gamma, y : W, \Delta \vdash t : T$ follows by definition from $\vdash \Gamma, y : W, \Delta$.

(i) $$\frac{\Gamma, \Delta \vdash u : \mathrm{Set} \quad \Gamma, \Delta \vdash v : \mathrm{El}\ u \to \mathrm{Set}}{\Gamma, \Delta \vdash \mathrm{fun}\ : ((x : \mathrm{El}\ u) \to \mathrm{El}\ (v\ x)) \to \mathrm{El}\ (\Pi\ u\ v)}\ x \notin FV(v)$$

By induction (2) we have $\Gamma, y : W, \Delta \vdash u : \mathrm{Set}$ and $\Gamma, y : W, \Delta \vdash v : \mathrm{El}\ u \to \mathrm{Set}$.
By definition then $\Gamma, y : W, \Delta \vdash \mathrm{fun}\ : ((x : \mathrm{El}\ u) \to \mathrm{El}\ (v\ x)) \to \mathrm{El}\ (\Pi\ u\ v)$.

□

Corollary 2.4.2 (Weakening).

1. $\Gamma \vdash T \ \wedge \ \vdash \Gamma, x : U \ \Rightarrow \ \Gamma, x : U \vdash T$
2. $\Gamma \vdash t : T \ \wedge \ \vdash \Gamma, x : U \ \Rightarrow \ \Gamma, x : U \vdash t : T$

2.4.2 Well-typed substitution

Definition 2.4.3 (Context map).
We will write $\gamma : \Delta \to \Gamma$, to denote $\forall x \in \mathit{supp}(\Gamma)\ .\ \Delta \vdash \gamma(x) : \Gamma(x)\ \gamma$

The notation[12] $\gamma : \Delta \to \Gamma$ may be read as *"γ is a realization of Γ with respect to Δ"* or alternatively *"γ is a* context map *from Δ to Γ"*.

Lemma 2.4.4 (Substitution lemma).
If $\vdash \Delta$ and $\gamma : \Delta \to \Gamma$ then

1. $\Gamma \vdash T \Rightarrow \Delta \vdash T\ \gamma$
2. $\Gamma \vdash t : T \Rightarrow \Delta \vdash t\ \gamma : T\ \gamma$

Proof. We proceed by simultaneous induction on the derivations of $\Gamma \vdash T$ and $\Gamma \vdash t : T$. We have the following cases:

1. Assume $\vdash \Delta$, $\gamma : \Delta \to \Gamma$ and $\Gamma \vdash T$. We have the following cases:

   (a) $\dfrac{\vdash \Gamma}{\Gamma \vdash \mathrm{Set}}$

   We have $\vdash \Delta$ by assumption. Then $\Delta \vdash \mathrm{Set}$ holds by definition.

   (b) $\dfrac{\Gamma \vdash u : \mathrm{Set}}{\Gamma \vdash \mathrm{El}\ u}$

   By induction (2) we have $\Delta \vdash u\ \gamma : \mathrm{Set}$.
   By definition then $\Delta \vdash \mathrm{El}\ u\ \gamma$.

   (c) $\dfrac{\Gamma, x : U \vdash V}{\Gamma \vdash (x : U) \to V}$

   By Corollary 2.3.7 we have $\Gamma \vdash U$ as a sub-derivation of the premise. By induction (1) then $\Delta \vdash U\ \gamma$. Assume w.l.o.g. that $x \notin \mathit{supp}(\Delta)$, $\gamma(x) = x$ and x is fresh for V, x, γ.

   We verify first that $\gamma : (\Delta, x : U\ \gamma) \to (\Gamma, x : U)$ holds as follows: By above we have $\vdash \Delta, x : U\ \gamma$. By Definition 2.3.1 (Type inhabitation, variable rule) we have $\Delta, x : U\ \gamma \vdash x : U\ \gamma$. We have $U\ \gamma = (\Gamma, x : U\ \gamma)(x)$ and $\gamma(x) = x$, and then $\Delta, x : U\ \gamma \vdash \gamma(x) : (\Gamma, x : U)(x)\ \gamma$ holds.

   Then we verify that $\Delta, x : U\ \gamma \vdash \gamma(y) : (\Gamma, x : U)(y)\ \gamma$ holds for y such that $y \neq x$ and $y \in \mathit{supp}(\Gamma)$ as follows: We have by assumption that $\Delta \vdash \gamma(y) : \Gamma(y)\ \gamma$ holds.

   By Corollary 2.4.2 (Weakening) we have $\Delta, x : U\ \gamma \vdash \gamma(y) : \Gamma(y)\ \gamma$. But $(\Gamma, x : U)(y) = \Gamma(y)$, and then $\Delta, x : U\ \gamma \vdash \gamma(y) : (\Gamma, x : U)(y)\ \gamma$ holds.

[12] The notation can be found in Cartmell (1986) and Martin-Löf's substitution calculus (Martin-Löf, 1992).

Thus, $\forall z \in supp(\Gamma, x : U)\ .\ \Delta, x : U\ \gamma \vdash \gamma(z) : (\Gamma, x : U)(z)\ \gamma$ holds, which means $\gamma : (\Delta, x : U\ \gamma) \to (\Gamma, x : U)$.

By induction (1) then $\Delta, x : U\ \gamma \vdash V\ \gamma$.

By definition we get $\Delta \vdash \text{Fun}\ (U\ \gamma)\ (\lambda x.V\ \gamma)$.

By assumption x is fresh for V, x, γ and $\gamma(x) = x$.

By substitution we have $\lambda x.(V\ \gamma) = (\lambda x.V)\ \gamma$.

Then $\Delta \vdash (\text{Fun}\ U\ (\lambda x.V))\ \gamma$.

2. Assume $\vdash \Delta$, $\gamma : \Delta \to \Gamma$ and $\Gamma \vdash t : T$. We have the following cases:

(a) $\dfrac{\vdash \Gamma}{\Gamma \vdash x : \Gamma(x)}$

We have $\gamma : \Delta \to \Gamma$ by assumption. By Definition 2.4.3 (Context map) then $\Delta \vdash \gamma(x) : \Gamma(x)\gamma$.

(b) $\dfrac{\Gamma \vdash t : U \quad \Gamma \vdash T}{\Gamma \vdash t : T}\ U \bowtie T$

By induction (2) we have $\Delta \vdash t\ \gamma : U\ \gamma$ and $\Delta \vdash T\ \gamma$. From $U \bowtie T$ and Proposition 2.2.17 we have $U\ \gamma \bowtie T\ \gamma$. By definition then $\Delta \vdash t\ \gamma : T\ \gamma$.

(c) $\dfrac{\Gamma \vdash u : (x : U) \to V \quad \Gamma \vdash v : U}{\Gamma \vdash u\ v : V[v/x]}$

By induction (2) we have
$\Delta \vdash u\ \gamma : ((x : U) \to V)\ \gamma$ and $\Delta \vdash v\ \gamma : U\ \gamma$.
Assume w.l.o.g. x is fresh for V, x, γ and $\gamma(x) = x$.
Then $(\lambda x.V)\ \gamma = \lambda x.(V\ \gamma)$. Then $\Delta \vdash u\ \gamma : (x : U\ \gamma) \to V\ \gamma$.
By definition $\Delta \vdash (u\ \gamma)(v\ \gamma) : V\gamma[v\ \gamma/x]$.
From the assumption about x we have $V\gamma[v\ \gamma/x] = V[v/x]\ \gamma$.
Then $\Delta \vdash (u\ v)\ \gamma : V[v/x]\ \gamma$.

(d) $\dfrac{\Gamma, x : U \vdash v : V}{\Gamma \vdash \lambda x.v : (x : U) \to V}$

By Corollary 2.3.7 we have $\Gamma \vdash U$ as a sub-derivation of the premise. By induction (1) then $\Delta \vdash U\ \gamma$. Assume w.l.o.g. $x \notin supp(\Delta)$, x is fresh for v, x, γ, x is fresh for V, x, γ and $\gamma(x) = x$.

We verify first that $\gamma : (\Delta, x : U\ \gamma) \to (\Gamma, x : U)$ holds as follows: By above we have $\vdash \Delta, x : U\ \gamma$. By Definition 2.3.1 (Type inhabitation, variable rule) we have $\Delta, x : U\ \gamma \vdash x : U\ \gamma$. We have $U\ \gamma = (\Gamma, x : U\ \gamma)(x)$ and $\gamma(x) = x$, and then $\Delta, x : U\ \gamma \vdash \gamma(x) : (\Gamma, x : U)(x)\ \gamma$ holds.

Then we verify that $\Delta, x : U\ \gamma \vdash \gamma(y) : (\Gamma, x : U)(y)\ \gamma$ holds for y such that $y \neq x$ and $y \in supp(\Gamma)$. We have by assumption that $\Delta \vdash \gamma(y) : \Gamma(y)\ \gamma$ holds.

By Corollary 2.4.2 (Weakening) we have $\Delta, x : U\ \gamma \vdash \gamma(y) : \Gamma(y)\ \gamma$. But $(\Gamma, x : U)(y) = \Gamma(y)$, and then $\Delta, x : U\ \gamma \vdash \gamma(y) : (\Gamma, x : U)(y)\ \gamma$ holds.

Thus, $\forall z \in \mathit{supp}(\Gamma, x : U)\ .\ \Delta, x : U\ \gamma \vdash \gamma(z) : (\Gamma, x : U)(z)\ \gamma$ holds, which means $\gamma : (\Delta, x : U\ \gamma) \to (\Gamma, x : U)$.

By induction (2) then $\Delta, x : U\ \gamma \vdash v\ \gamma : V\ \gamma$.

By definition we get $\Delta \vdash \lambda x.(v\ \gamma) : \mathrm{Fun}\ (U\ \gamma)\ (\lambda x.V\ \gamma)$.

By assumption x is fresh for v, x, γ, x is fresh for V, x, γ and $\gamma(x) = x$. By substitution we have $\lambda x.(v\ \gamma) = (\lambda x.v)\ \gamma$ and $\lambda x.(V\ \gamma) = (\lambda x.V)\ \gamma$. Then $\Delta \vdash (\lambda x.v)\ \gamma : (\mathrm{Fun}\ U\ (\lambda x.V))\ \gamma$.

(e) $$\frac{\vdash \Gamma}{\Gamma \vdash f : \mathcal{F}(f)}$$

From $\vdash \Delta$, both t closed and T closed we have $\Delta \vdash t\ \gamma : T\ \gamma$ by Definition 2.3.1 (Type inhabitation).

(f) $$\frac{\vdash \Gamma}{\Gamma \vdash d : \mathcal{D}(d)}$$

From $\vdash \Delta$, both t closed and T closed we have $\Delta \vdash t\ \gamma : T\ \gamma$ by Definition 2.3.1 (Type inhabitation).

(g) $$\frac{\vdash \Gamma \quad \Gamma \vdash u_1 : \mathrm{Set} \quad \ldots \quad \Gamma \vdash u_k : \mathrm{Set}}{\Gamma \vdash c : \mathcal{C}(c)[u_1,\ \ldots\ , u_k]} \begin{cases} \mathcal{D}(d) = \mathrm{Set}^k \to \mathrm{Set} \\ \mathcal{C}(c) = (\mathrm{El}\ e_1,\ \ldots\ , \mathrm{El}\ e_n) \\ \qquad\qquad \to \mathrm{El}\ (d\ x_1\ \ldots\ x_k) \\ FV(e_i) \subseteq \{x_1,\ \ldots\ , x_k\} \end{cases}$$

By induction (2) then $\Delta \vdash u_i\ \gamma : \mathrm{Set}$. By Definition 2.3.1 (Type inhabitation, constructor rule) we have $\Delta \vdash c\ \gamma : \mathcal{C}(c)[u_1\gamma,\ \ldots\ , u_k\gamma]$, which is equivalent to $\Delta \vdash c\ \gamma : \mathcal{C}(c)[u_1,\ \ldots\ , u_k]\gamma$.

(h) $$\frac{\vdash \Gamma}{\Gamma \vdash \Pi : (x : \mathrm{Set},\ \mathrm{El}\ x \to \mathrm{Set}) \to \mathrm{Set}}$$

From $\vdash \Delta$, both t closed and T closed we have $\Delta \vdash t\ \gamma : T\ \gamma$ by Definition 2.3.1 (Type inhabitation).

(i) $$\frac{\Gamma \vdash u : \mathrm{Set} \quad \Gamma \vdash v : \mathrm{El}\ u \to \mathrm{Set}}{\Gamma \vdash \mathrm{fun}\ : ((x : \mathrm{El}\ u) \to \mathrm{El}\ (v\ x)) \to \mathrm{El}\ (\Pi\ u\ v)}\ x \notin FV(v)$$

By induction (2) then $\Delta \vdash u\ \gamma : \mathrm{Set}$ and $\Delta \vdash v\ \gamma : \mathrm{El}\ (u\ \gamma) \to \mathrm{Set}$.

Assume w.l.o.g. $x \notin FV(v\ \gamma)$, and $\gamma(x) = x$.

By Definition 2.3.1 (Type inhabitation) for 'fun', we have $\Delta \vdash \mathrm{fun}\ \gamma : ((x : \mathrm{El}\ (u\ \gamma)) \to \mathrm{El}\ ((v\ \gamma)\ x)) \to \mathrm{El}\ (\Pi\ (u\ \gamma)\ (v\ \gamma))$ which is equivalent to $\Delta \vdash \mathrm{fun}\ \gamma : (((x : \mathrm{El}\ u) \to \mathrm{El}\ (v\ x)) \to \mathrm{El}\ (\Pi\ u\ v))\ \gamma$

□

2.4.3 From neighbourhoods to context maps

Lemma 2.4.5 (Composition of context maps).
$\vdash \Theta \ \wedge \ \gamma : \Theta \to \Delta \ \wedge \ \delta : \Delta \to \Gamma \ \Rightarrow \ \delta\gamma : \Theta \to \Gamma$

Proof. Assume $\vdash \Theta$, $\gamma : \Theta \to \Delta$ and $\delta : \Delta \to \Gamma$. Assume given $x \in \mathit{supp}(\Gamma)$. From $\delta : \Delta \to \Gamma$ we have $\Delta \vdash \delta(x) : \Gamma(x)\delta$. From $\vdash \Theta$, $\gamma : \Theta \to \Delta$ and Lemma 2.4.4 (Substitution lemma) we have $\Theta \vdash \delta(x)\gamma : \Gamma(x)\delta\gamma$. □

Lemma 2.4.6. $\Delta \xrightarrow{\alpha} \Gamma \ \Rightarrow \ \alpha : \Delta \to \Gamma$.

Proof. Assume $\Delta \xrightarrow{\alpha} \Gamma$. By Definition 2.3.11 we have $\vdash \Delta$ and $\vdash \Gamma$. We have Γ of the form $\Gamma_1, x_k : T_k, \Gamma_2$ and Δ is $\Gamma_1, \Gamma', \Gamma_2\alpha$, $\alpha = [p/x_k]$, where Γ' depends on α and T_k. We verify $\alpha : \Delta \to \Gamma$ by verifying $\alpha : \Delta \to \Gamma_1$, $\alpha : \Delta \to x_k : T_k$ and $\alpha : \Delta \to \Gamma_2$ in turn.

For Γ_1 it is direct.

For $x_k : T_k$, from $\vdash \Delta$, and Definition 2.3.1 (Type inhabitation), when $\alpha(x_k)$ is a variable we use the variable rule, and when in constructor form, we use the conversion rule to get $\Delta \vdash \alpha(x_k) : \Gamma(x_k)\alpha$ for the corresponding forms of α.

For Γ_2, from $\vdash \Delta$, by the variable rule we get $\Delta \vdash \alpha(x) : \Gamma(x)\alpha$ for all $x \in \mathit{supp}(\Gamma_2)$, since these are actually the parts of Δ. □

Lemma 2.4.7. $\Delta \xrightarrow{\tau} \Gamma \;\Rightarrow\; \tau : \Delta \to \Gamma$.

Proof. By induction on the derivation of $\Delta \xrightarrow{\tau} \Gamma$.

- $\Gamma \xrightarrow{[]} \Gamma$.
  Direct.
- $$\frac{\Delta \xrightarrow{\tau'} \Gamma' \quad \Gamma' \xrightarrow{\alpha} \Gamma}{\Delta \xrightarrow{\alpha\,\tau'} \Gamma}$$
  By Lemma 2.4.6 we have $\alpha : \Gamma' \to \Gamma$. By induction from $\Delta \xrightarrow{\tau'} \Gamma'$ we have $\tau' : \Delta \to \Gamma'$. By Lemma 2.4.5 then $\alpha\tau' : \Delta \to \Gamma$.

□

2.4.4 Generation lemma

Lemma 2.4.8. *If* $\vdash \Sigma$ *and* $\Gamma \vdash t : T$ *then* $\Gamma \vdash T$.

Proof. By induction on the derivation and Definition 2.3.17, in the case of constants. It also uses Corollary 2.4.2 (Weakening) and Lemma 2.4.4 (Substitution lemma) for the application case. □

Lemma 2.4.9 (Generation lemma).

1. $\Gamma \vdash x : T \;\Rightarrow\; T \bowtie \Gamma(x)$
2. $\Gamma \vdash t\ u : T \;\Rightarrow$
   $\exists U, V\ .\ \Gamma \vdash t : (x : U) \to V \;\wedge\; \Gamma \vdash u : U \;\wedge\; T \bowtie V[u/x]$
3. $\Gamma \vdash \lambda x.v : T \;\Rightarrow\; \exists U, V\ .\ T \bowtie (x : U) \to V \;\wedge\; \Gamma, x : U \vdash v : V$
4. $\Gamma \vdash d : T \;\Rightarrow\; T = \mathcal{D}(d)$
5. $\Gamma \vdash c : T \;\Rightarrow \vdash \Gamma \;\wedge\; T \bowtie \mathcal{C}(c)[u_1, \ \ldots\ , u_k] \;\wedge\; \Gamma \vdash u_i : \mathrm{Set}$
6. $\Gamma \vdash f : T \;\Rightarrow \vdash \Gamma \;\wedge\; T \bowtie \mathcal{F}(f)$
7. $\Gamma \vdash \Pi : T \;\Rightarrow \vdash \Gamma \;\wedge\; T \bowtie (x : \mathrm{Set},\ \mathrm{El}\ x \to \mathrm{Set}) \to \mathrm{Set}$
8. $\Gamma \vdash \mathrm{fun} : T \;\Rightarrow$
   $T \bowtie ((x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)) \to \mathrm{El}\ (\Pi\ t\ u) \;\wedge$
   $x \notin FV(u) \quad \wedge \quad \Gamma \vdash t : \mathrm{Set} \quad \wedge \quad \Gamma \vdash u : \mathrm{El}\ t \to \mathrm{Set}$

Proof. By induction on the derivations and conversion. Case 4 uses Remark 2.2.19. □

Lemma 2.4.10 (Strong generation lemma). *If* $\Gamma \vdash \lambda x.v : T$ *holds, then* T *is of the form* $(x : U) \to V$ *where* $\Gamma, x : U \vdash v : V$ *holds.*

Proof. By induction on the derivation of $\Gamma \vdash \lambda x.v : T$.

We have two possible cases:

- $\dfrac{\Gamma, x : U \vdash v : V}{\Gamma \vdash \lambda x.v : (x : U) \to V}$

  Here $T = (x : U) \to V$ and the premise give our goal directly.

- $\dfrac{\Gamma \vdash \lambda x.v : T' \quad \Gamma \vdash T}{\Gamma \vdash \lambda x.v : T}\ T' \bowtie T$

  By induction we have $T' = (x : U') \to V'$ and $\Gamma, x : U' \vdash v : V'$.

  By Remark 2.2.21 we have T of the form $(x : U) \to V$, and then $U \bowtie U'$ and $V \bowtie V'$.

  By Lemma 2.3.4 (Type-formation inversion) for the premise $\Gamma \vdash T$ we have $\Gamma, x : U \vdash V$, and from Lemma 2.3.6 we also have $\vdash \Gamma, x : U$. Now suppose that $[] : \Gamma, x : U \to \Gamma, x : U'$ holds. Then by Lemma 2.4.4 we have $\Gamma, x : U \vdash v : V'$. By conversion from $\Gamma, x : U \vdash V$ and $V \bowtie V'$ we get $\Gamma, x : U \vdash v : V$.

  It remains to verify $[] : \Gamma, x : U \to \Gamma, x : U'$.

  For $y \in \mathit{supp}(\Gamma)$, $y \neq x$, we have $\Gamma, x : U \vdash y : (\Gamma, x : U')(y)$, since $\vdash \Gamma, x : U$ and $(\Gamma, x : U')(y) = \Gamma(y) = (\Gamma, x : U)(y)$.

  For x we show $\Gamma, x : U \vdash x : (\Gamma, x : U')(x)$ as follows:

  From previously we have $\Gamma, x : U' \vdash v : V'$, and by Corollary 2.3.7 we have $\Gamma \vdash U'$. By Corollary 2.4.2 (Weakening) and $\vdash \Gamma, x : U$ we have $\Gamma, x : U \vdash U'$. From $\vdash \Gamma, x : U$ we also have $\Gamma, x : U \vdash x : U$, and from $U \bowtie U'$ and the conversion rule we have $\Gamma, x : U \vdash x : U'$.

□

2.4.5 Iterated inversion properties

Lemma 2.4.11 (Typing iterated application).
If $\Gamma \vdash t : (x_1 : T_1, \ \ldots\ , x_n : T_n) \to T$, *and*
$\Gamma \vdash t_1 : T_1, \ \ldots\ , \ \Gamma \vdash t_n : T_n[t_1, \ \ldots\ , t_{n-1}]$
then $\Gamma \vdash t\ t_1\ \ldots\ t_n : T[t_1, \ \ldots\ , t_n]$.

Proof. By Lemma 2.4.4 (Substitution lemma). □

Lemma 2.4.12 (Generation for iterated application).
If $\vdash \Sigma$ *and* $\Gamma \vdash t\ t_1\ \ldots\ t_n : T$ *then there is* $U, U_1, \ \ldots\ , U_n$ *such that* $\Gamma \vdash t : (x_1 : U_1, \ \ldots\ , x_n : U_n) \to U$, $\Gamma \vdash t_i : U_i[t_1, \ \ldots\ , t_{i-1}]$ *and* $T \bowtie U[t_1, \ \ldots\ , t_n]$.

Proof. Assume $\vdash \Sigma$ and $\Gamma \vdash t\ t_1\ \ldots\ t_n : T$. We proceed by induction on n. If $n = 0$ we are done, otherwise the derivation is of the form
$\Gamma \vdash (t\ t_1\ \ldots\ t_{n-1})\ t_n : T$, and by generation there are V_n, V such that

$$\Gamma \vdash t\ t_1\ \ldots\ t_{n-1} : (x_n : V_n) \to V\ ,\quad \Gamma \vdash t_n : V_n \text{ and } T \bowtie V[t_n/x_n]. \tag{2.1}$$

Assume[13] w.l.o.g. that $x_n \notin FV(t_1, \ldots, t_{n-1})$.
By induction there are $V_1, \ldots, V_{n-1}, V'$ such that

$$\Gamma \vdash t : (x_1 : V_1, \ldots, x_{n-1} : V_{n-1}) \to V'\ , \tag{2.2}$$

$$\Gamma \vdash t_1 : V_1, \ldots, \Gamma \vdash t_{n-1} : V_{n-1}[t_1, \ldots, t_{n-2}] \quad \text{and} \tag{2.3}$$

$$(x_n : V_n) \to V \bowtie V'[t_1, \ldots, t_{n-1}]. \tag{2.4}$$

From (2.4) there are V'_n, V'' such that

$$V' = (x_n : V'_n) \to V''. \tag{2.5}$$

Then

$$V_n \bowtie V'_n[t_1, \ldots, t_{n-1}] \text{ and } V \bowtie V''[t_1, \ldots, t_{n-1}]. \tag{2.6}$$

From (2.5) then

$$(x_1 : V_1, \ldots, x_{n-1} : V_{n-1}) \to V' \quad = \\ (x_1 : V_1, \ldots, x_{n-1} : V_{n-1}, x_n : V'_n) \to V''. \tag{2.7}$$

From (2.1) we have $T \bowtie V[t_n/x_n]$, and from (2.6) we have
$V \bowtie V''[t_1, \ldots, t_{n-1}]$, then $V[t_n/x_n] \bowtie V''[t_1, \ldots, t_{n-1}][t_n/x_n]$.
By the assumption made at (2.1) we have $x_n \notin FV(t_1, \ldots, t_{n-1})$,
and we obtain $T \bowtie V''[t_1, \ldots, t_n]$. From (2.7) and (2.2) we have

$$\Gamma \vdash t : (x_1 : V_1, \ldots, x_{n-1} : V_{n-1}, x_n : V'_n) \to V''. \tag{2.8}$$

Recall from (2.3) we have

$$\Gamma \vdash t_1 : V_1, \ldots, \Gamma \vdash t_{n-1} : V_{n-1}[t_1, \ldots, t_{n-2}] \tag{2.9}$$

Then

$$[t_1/x_1, \ldots, t_{n-1}/x_{n-1}]\ :\ \Gamma\ \to\ \Gamma, x_1 : V_1, \ldots, x_{n-1} : V_{n-1}$$

From $\vdash \Sigma$, (2.8) and Lemma 2.4.8 we get

$$\Gamma \vdash (x_1 : V_1, \ldots, x_{n-1} : V_{n-1}, x_n : V'_n) \to V''$$

and then from Lemma 2.3.5 we get

$$\Gamma, x_1 : V_1, \ldots, x_{n-1} : V_{n-1} \vdash V'_n$$

By Lemma 2.4.4 (Substitution lemma) we get $\Gamma \vdash V'_n[t_1, \ldots, t_{n-1}]$. From (2.1) we have $\Gamma \vdash t_n : V_n$ and from (2.6) $V_n \bowtie V'_n[t_1, \ldots, t_{n-1}]$. By Definition 2.3.1 (Type inhabitation, conversion rule), we have $\Gamma \vdash t_n : V'_n[t_1, \ldots, t_{n-1}]$, which completes the proof. □

[13] Recall from Remark 2.2.5 (α-equivalence), page 29 that terms that differ only in the names of their bound variables are identified.

Lemma 2.4.13. *If* $\vdash \Sigma$ *then*

1. $\Gamma \vdash c\ t_1\ \ldots\ t_n : \mathrm{El}\ (d\ \vec{u}\,) \Rightarrow \Gamma \vdash t_i : \mathrm{El}\ (e_i[\vec{u}\,])$, *where* $\mathcal{C}(c) = (\mathrm{El}\ e_1,\ \ldots\ ,\ \mathrm{El}\ e_n) \to \mathrm{El}\ (d\ \vec{z}\,)$.
2. $\Gamma \vdash \mathrm{El}\ (d\ \vec{u}\,) \Rightarrow \Gamma \vdash u_i : \mathrm{Set}$.

Proof. By Lemma 2.4.12 and conversion. □

Lemma 2.4.14.

1. $\Gamma \vdash \mathrm{El}\ (\Pi\ t\ u) \Rightarrow \Gamma \vdash (x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)$, *with* $x \notin FV(u)$.
2. $\Gamma \vdash \mathrm{fun}\ v : \mathrm{El}\ (\Pi\ t\ u) \Rightarrow \Gamma \vdash v : (x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)$, *with* $x \notin FV(u)$.

Proof of 1. Assume $\Gamma \vdash \mathrm{El}\ (\Pi\ t\ u)$. By inversion $\Gamma \vdash \Pi\ t\ u : \mathrm{Set}$. By iterated inversion then $\Gamma \vdash t : \mathrm{Set}$ and $\Gamma \vdash u : \mathrm{El}\ t \to \mathrm{Set}$ is derivable. By inversion $\Gamma \vdash \mathrm{El}\ t$. Assume w.l.o.g. that $x \notin \mathit{supp}(\Gamma)$, then we have $\vdash \Gamma, x : \mathrm{El}\ t$. We have then $\Gamma, x : \mathrm{El}\ t \vdash x : \mathrm{El}\ t$. By Corollary 2.4.2 (Weakening) we have $\Gamma, x : \mathrm{El}\ t \vdash u : \mathrm{El}\ t \to \mathrm{Set}$. We can derive

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \mathrm{El}\ t \vdash u : \mathrm{El}\ t \to \mathrm{Set} \quad \Gamma, x : \mathrm{El}\ t \vdash x : \mathrm{El}\ t}{\Gamma, x : \mathrm{El}\ t \vdash u\ x : \mathrm{Set}}}{\Gamma, x : \mathrm{El}\ t \vdash \mathrm{El}\ (u\ x)}}{\Gamma \vdash (x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)}$$

□

Proof of 2. Similar as above. □

2.4.6 Inversion of neighbourhoods

Lemma 2.4.15 (Inversion of atomic neighbourhood).
$\vdash \Sigma\ \wedge\ \Delta \xrightarrow{\alpha} \Gamma\ \wedge\ (\alpha\gamma) : \Theta \to \Gamma\ \Rightarrow\ \gamma : \Theta \to \Delta$.

Proof. Assume $\vdash \Sigma$, $\Delta \xrightarrow{\alpha} \Gamma$ and $(\alpha\gamma) : \Theta \to \Gamma$.
Let $\Gamma = (x_1 : T_1,\ \ldots\ , x_n : T_n)$. We have

$$\Theta \vdash (\alpha\gamma)(x_1) : T_1(\alpha\gamma),\ \ldots\ ,\ \Theta \vdash (\alpha\gamma)(x_n) : T_n(\alpha\gamma). \tag{2.10}$$

We have to show $\forall y \in \mathit{supp}(\Delta)\ .\ \Theta \vdash \gamma(y) : \Delta(y)\gamma$. From $\Delta \xrightarrow{\alpha} \Gamma$ and Definition 2.3.11 (Atomic neighbourhood), we have Γ of the form $(\Gamma_1, x_k : T_k, \Gamma_2)$ and Δ given by $(\Gamma_1, \Gamma', \Gamma_2\alpha)$, where Γ' depends on α and T_k. Note that since Γ is closed, we have for $l \leq k$ that $x_k \notin FV(T_l)$ holds, and then

$$T_l(\alpha\gamma) = T_l\gamma. \tag{2.11}$$

We verify our goal for Γ_1, Γ' and $\Gamma_2\alpha$ separately:

1. For Γ_1, and $x_i \in \{x_1, \ldots, x_{k-1}\}$:
   From (2.11) we have $T_i(\alpha\gamma) = T_i\gamma$. Since Γ is disjoint, $x_i \neq x_k$, so $\alpha(x_i) = x_i$, and then $(\alpha\gamma)(x_i) = \gamma(x_i)$. Then, from (2.10), we have $\Theta \vdash \gamma(x_1) : T_1\gamma$ through $\Theta \vdash \gamma(x_{k-1}) : T_{k-1}\gamma$. By Definition 2.4.3 then

$$\gamma : \Theta \to \Gamma_1. \tag{2.12}$$

2. For Γ':

   (a) For α of the form $[y/x_k]$:
   We have $[y/x_k]\gamma : \Theta \to \Gamma$. In this case Γ' is $(y : T_k)$, and we have to show $\Theta \vdash \gamma(y) : T_k\gamma$. From (2.10) we have $\Theta \vdash (\alpha\gamma)(x_k) : T_k(\alpha\gamma)$. From (2.11) we have $T_k(\alpha\gamma) = T_k\gamma$. By substitution we have $(\alpha\gamma)(x_k) = \gamma(y)$. Hence $\Theta \vdash \gamma(y) : T_k\gamma$ holds.

   (b) For α of the form $[\text{fun } y/x_k]$:
   We have $T_k \bowtie \text{El } (\Pi\ t\ u)$ and $\Gamma_1 \vdash \text{El } (\Pi\ t\ u)$. In this case Γ' is $(y : (z : \text{El } t) \to \text{El } (u\ z))$ with $z \notin FV(u)$. From (2.10) we have $\Theta \vdash (\alpha\gamma)(x_k) : T_k(\alpha\gamma)$. We have $(\alpha\gamma)(x_k) = \text{fun}(\gamma(y))$ and from (2.11) we have $T_k(\alpha\gamma) = T_k\gamma$, hence $\Theta \vdash \text{fun}(\gamma(y)) : T_k\gamma$. From $\Gamma_1 \vdash \text{El } (\Pi\ t\ u)$, $\vdash \Theta$ and (2.12), by Lemma 2.4.4 (Substitution lemma) we have $\Theta \vdash \text{El } (\Pi\ t\ u)\gamma$. By substitution we have $\Theta \vdash \text{El } (\Pi(t\gamma)(u\gamma))$. From $T_k \bowtie \text{El } (\Pi\ t\ u)$ we have $T_k\gamma \bowtie \text{El } (\Pi\ (t\gamma)(u\gamma))$. By Definition 2.3.1 (Type inhabitation, conversion rule), then $\Theta \vdash \text{fun}(\gamma(y)) : \text{El } (\Pi\ (t\gamma)\ (u\gamma))$. By Lemma 2.4.14 then $\Theta \vdash \gamma(y) : (z : \text{El } (t\ \gamma)) \to \text{El } ((u\ \gamma)\ z)$, with $z \notin FV(u\ \gamma)$. Then we have $(z : \text{El } (t\ \gamma)) \to \text{El } ((u\ \gamma)\ z) = ((z : \text{El } t) \to \text{El } (u\ z))\ \gamma$, and so $\Theta \vdash \gamma(y) : ((z : \text{El } t) \to \text{El } (u\ z))\ \gamma$.

   (c) For α of the form $[c\ y_1\ \ldots\ y_m/x_k]$:
   We have $\mathcal{C}(c) = (\text{El } e_1, \ldots, \text{El } e_m) \to \text{El } (d\ \vec{z})$, $T_k = \text{El } (d\ \vec{t})$, and $\Gamma_1 \vdash \text{El } (d\ \vec{t})$. In this case Γ' is $(y_1 : \text{El } e_1[\vec{t}], \ldots, y_m : \text{El } e_m[\vec{t}])$.
   From (2.10) we have $\Theta \vdash (\alpha\gamma)(x_k) : T_k(\alpha\gamma)$. We have $(\alpha\gamma)(x_k) = c\ y_1\gamma\ \ldots\ y_m\gamma$. From (2.11) we have $T_k(\alpha\gamma) = T_k\gamma$, and then
   $\Theta \vdash c\ y_1\gamma\ \ldots\ y_m\gamma : T_k\gamma$ holds. From $\Gamma_1 \vdash \text{El } (d\ \vec{t})$, $\vdash \Theta$ and (2.12), by Lemma 2.4.4 (Substitution lemma) we have $\Theta \vdash \text{El } (d\ \vec{t})\gamma$. From $T_k \bowtie \text{El } (d\ \vec{t})$ we have $T_k\gamma \bowtie \text{El } (d\ \vec{t})\gamma$.
   By Definition 2.3.1 (Type inhabitation, conversion rule) we have $\Theta \vdash c\ y_1\gamma\ \ldots\ y_n\gamma : \text{El } (d\ \vec{t})\gamma$. By assumption we have $\vdash \Sigma$, and by Lemma 2.4.13 we have
   $\Theta \vdash \gamma(y_1) : \text{El } (e_1[\vec{t}])\gamma$ through $\Theta \vdash \gamma(y_m) : \text{El } (e_m[\vec{t}])\gamma$.

3. For $\Gamma_2\alpha$, and $x_i \in \{x_{k+1}, \ldots, x_n\}$:
   Since Γ is disjoint, we have $x_i \neq x_k$, so $\alpha(x_i) = x_i$, and then $(\alpha\gamma)(x_i) = \gamma(x_i)$. We have $T_i(\alpha\gamma) = (T_i\alpha)\gamma$, and then from (2.10), we have
   $\Theta \vdash \gamma(x_{k+1}) : (T_{k+1}\alpha)\gamma$ through $\Theta \vdash \gamma(x_n) : (T_n\alpha)\gamma$.

□

Lemma 2.4.16 (Inversion of compound neighbourhood).
$\vdash \Sigma \ \wedge \ \Delta \xrightarrow{\tau} \Gamma \ \wedge \ (\tau\ \gamma) : \Theta \to \Gamma \ \Rightarrow \ \gamma : \Theta \to \Delta.$

Proof. Assume $\vdash \Sigma$, $\Delta \xrightarrow{\tau} \Gamma$ and $(\tau\ \gamma) : \Theta \to \Gamma$.
We proceed by induction on the derivation of $\Delta \xrightarrow{\tau} \Gamma$.

1. $\dfrac{}{\Gamma \xrightarrow{[]} \Gamma}$
   In this case we have $\Delta = \Gamma$, and then $\gamma : \Theta \to \Delta$.

2. $\dfrac{\Delta \xrightarrow{\tau'} \Gamma' \quad \Gamma' \xrightarrow{\alpha} \Gamma}{\Delta \xrightarrow{\alpha\ \tau'} \Gamma}$
   We have $(\alpha\ \tau')\gamma : \Theta \to \Gamma$.

   By associativity of substitution we have $(\alpha\ \tau')\ \gamma$ equivalent to $\alpha\ (\tau'\ \gamma)$, and then $\alpha\ (\tau'\ \gamma) : \Theta \to \Gamma$. By assumption we have $\vdash \Sigma$, and from $\Gamma' \xrightarrow{\alpha} \Gamma$ and Lemma 2.4.15 then $(\tau'\ \gamma) : \Theta \to \Gamma'$. Then, from $\Delta \xrightarrow{\tau'} \Gamma'$ and induction we have $\gamma : \Theta \to \Delta$.

□

Lemma 2.4.17. *If* $\Delta \xrightarrow{[\vec{p}\,,\vec{q}\,/\vec{x}\,,\vec{y}\,]} \Gamma_1, \Gamma_2$, $|\vec{p}\,| = |\vec{x}\,|$ *and* $\vec{x} = \overrightarrow{supp}(\Gamma_1)$, *then there are* Δ_1 *and* Δ_2 *such that* $\Delta = \Delta_1, \Delta_2$ *where* $\Delta_1 \xrightarrow{[\vec{p}\,/\vec{x}\,]} \Gamma_1$.

Proof. By induction on the derivation of $\Delta \xrightarrow{[\vec{p}\,,\vec{q}\,/\vec{x}\,,\vec{y}\,]} \Gamma$. □

2.4.7 Subject reduction

Lemma 2.4.18 (Subject reduction). *If* $\vdash \Sigma$ *then*

1. *If* $\Gamma \vdash T$ *and* $T \rightsquigarrow U$ *then* $\Gamma \vdash U$.
2. *If* $\Gamma \vdash t : T$ *and* $t \rightsquigarrow u$ *then* $\Gamma \vdash u : T$.

Proof of 2. Assume $\vdash \Sigma$, $\Gamma \vdash t : T$ and $t \rightsquigarrow t'$. We proceed by induction on the derivation of $t \rightsquigarrow t'$.

1. $\dfrac{}{(\lambda x.v)\ u \rightsquigarrow_\beta v[u/x]}$
   By Lemma 2.4.9 (Generation lemma) there is U and V such that $\Gamma \vdash u : U$, $\Gamma \vdash \lambda x.v : (x : U) \to V$, and $T \bowtie V[u/x]$.

   By Lemma 2.4.10 (Strong generation lemma), from $\Gamma \vdash \lambda x.v : (x : U) \to V$ we have $\Gamma, x : U \vdash v : V$. From Corollary 2.3.7 we have $\Gamma \vdash U$, and from Lemma 2.3.6 we have $\vdash \Gamma, x : U$ and $x \notin supp(\Gamma)$. For $x_i \in supp(\Gamma)$ we have $\Gamma \vdash [u/x](x_i) : (\Gamma, x : U)(x_i)$, because $[u/x](x_i) = x_i$ and $(\Gamma, x : U)(x_i) = (\Gamma)(x_i)$. For x we have $\Gamma \vdash [u/x](x) : (\Gamma, x : U)(x)$, because $[u/x](x) = u$ and $(\Gamma, x : U)(x) = U$. Thus $[u/x] : \Gamma \to \Gamma, x : U$.

   From $\Gamma, x : U \vdash v : V$ and Lemma 2.4.4 (Substitution lemma) we have $\Gamma \vdash v[u/x] : V[u/x]$.

From $\vdash \Sigma$, $\Gamma \vdash t : T$ and Lemma 2.4.8 we have $\Gamma \vdash T$. By Definition 2.3.1 (Type inhabitation, conversion rule) we get $\Gamma \vdash v[u/x] : T$.

2. $\dfrac{}{(f\ p_1\ \ldots\ p_n)\ \gamma \rightsquigarrow_\iota (\lambda \vec{y}\,.s)\ \gamma}$ $\begin{cases} f\ p_1\ \ldots\ p_n\ \vec{y} = s\ \in \mathcal{R} \\ p_n \text{ is not a variable} \end{cases}$

We have $\mathcal{F}(f)$ of the form $\Gamma_1, \Gamma_2 \to U$, where $\Gamma_1 = (x_1 : U_1,\ \ldots\ , x_n : U_n)$, $\Gamma_2 = (x_{n+1} : U_{n+1},\ \ldots\ , x_{n+m} : U_{n+m})$ and $ar(f) = n + m$.

Let $\vec{p} = (p_1,\ \ldots\ , p_n)$. Recall $\vec{y} = (y_1,\ \ldots\ , y_m)$.

From $\vdash \Sigma$, $\Gamma \vdash f\ (p_1\gamma)\ \ldots\ (p_n\gamma) : T$ and Lemma 2.4.12, Lemma 2.4.9 (Generation lemma) and conversion we have

$$\Gamma \vdash f : \mathcal{F}(f), \quad (2.13)$$

$$\Gamma \vdash p_1\gamma : U_1,\ \ldots\ , \Gamma \vdash p_n\gamma : U_n[p_1\gamma,\ \ldots\ , p_{n-1}\gamma], \text{and} \quad (2.14)$$

$$T \bowtie ((x_{n+1} : U_{n+1},\ \ldots\ , x_{n+m} : U_{n+m}) \to U)[p_1\gamma,\ \ldots\ , p_n\gamma]. \quad (2.15)$$

From $\vdash \Sigma$ we have $\vdash \mathcal{F}$ and $\vdash \mathcal{R}$, and then there is Δ such that[14]

$$\Delta \xrightarrow{[\vec{p}\,,\vec{y}\,]} \Gamma_1, \Gamma_2\ \wedge\ \Delta \vdash s : U[\vec{p}\,, \vec{y}\,]. \quad (2.16)$$

By Lemma 2.4.17 we have $\Delta = \Delta_1, \Delta_2$ where

$$\Delta_1 \xrightarrow{[\vec{p}\,]} \Gamma_1 \quad (2.17)$$

From (2.16) and by applying type inhabitation, the abstraction rule $|\vec{y}\,|$ times we get

$$\Delta_1 \vdash \lambda \vec{y}\,.s : \Delta_2 \to U[\vec{p}\,, \vec{y}\,]. \quad (2.18)$$

From (2.14) we have

$$\Gamma \vdash \vec{p}\gamma : \Gamma_1 \quad (2.19)$$

and then from (2.17), (2.19), and the assumption $\vdash \Sigma$, by Lemma 2.4.16 (Inversion of compound neighbourhood) we get

$$\gamma : \Gamma \to \Delta_1 \quad (2.20)$$

and then from (2.18) and Lemma 2.4.4 (Substitution lemma) we get

$$\Gamma \vdash (\lambda \vec{y}\,.s)\ \gamma : (\Delta_2 \to U[\vec{p}\,, \vec{y}\,])\ \gamma$$

But since
$(\Delta_2 \to U[\vec{p}\,, \vec{y}\,])\ \gamma = ((x_{n+1} : U_{n+1},\ \ldots\ , x_{n+m} : U_{n+m}) \to U)[p_1\gamma,\ \ldots\ , p_n\gamma]$,
we have from (2.15) that $T \bowtie (\Delta_2 \to U[\vec{p}\,, \vec{y}\,])\ \gamma$ holds.

By conversion of typing we get

$$\Gamma \vdash (\lambda \vec{y}\,.s)\ \gamma : T$$

[14] Recall Notation 2.3.14, page 40.

3. $\dfrac{v \rightsquigarrow v'}{\lambda x.v \rightsquigarrow \lambda x.v'}$

   By Lemma 2.4.10 (Strong generation lemma), T is of the form $(x : U) \to V$ where $\Gamma, x : U \vdash v : V$ holds. From $v \rightsquigarrow v'$ and induction then $\Gamma, x : U \vdash v' : V$. By Definition 2.3.1 (Type inhabitation, abstraction rule) we have $\Gamma \vdash \lambda x.v' : (x : U) \to V$.

4. $\dfrac{v \rightsquigarrow v'}{v\ u \rightsquigarrow v'\ u}$

   By Lemma 2.4.9 (Generation lemma), there is U and V such that $\Gamma \vdash v : (x : U) \to V$, $\Gamma \vdash u : U$, and $T \bowtie V[u/x]$. From $v \rightsquigarrow v'$ and induction we have $\Gamma \vdash v' : (x : U) \to V$. By Definition 2.3.1 (Type inhabitation, application rule) we have $\Gamma \vdash v'\ u : V[u/x]$. From $\vdash \Sigma$, $\Gamma \vdash v\ u : T$ and Lemma 2.4.8 we have $\Gamma \vdash T$. From $T \bowtie V[u/x]$ and Definition 2.3.1 (Type inhabitation, conversion rule) we get $\Gamma \vdash v'\ u : T$.

5. $\dfrac{u \rightsquigarrow u'}{v\ u \rightsquigarrow v\ u'}$

   By Lemma 2.4.9 (Generation lemma), there is U and V such that $\Gamma \vdash v : (x : U) \to V$, $\Gamma \vdash u : U$, and $T \bowtie V[u/x]$.

   From $u \rightsquigarrow u'$ and induction we have $\Gamma \vdash u' : U$. By Definition 2.3.1 (Type inhabitation, application rule) we have $\Gamma \vdash v\ u' : V[u'/x]$. From $u \rightsquigarrow u'$ and $T \bowtie V[u/x]$ we have $T \bowtie V[u'/x]$.

   From $\vdash \Sigma$, $\Gamma \vdash v\ u : T$ and Lemma 2.4.8 we have $\Gamma \vdash T$.

   From $T \bowtie V[u'/x]$ and Definition 2.3.1 (Type inhabitation, conversion rule) we get $\Gamma \vdash v\ u' : T$.

□

Proof of 1. Assume $\vdash \Sigma$, $\Gamma \vdash T$ and $T \rightsquigarrow T'$. We proceed by induction on the derivation of $T \rightsquigarrow T'$.

1. $\dfrac{t \rightsquigarrow t'}{\mathrm{El}\ t \rightsquigarrow \mathrm{El}\ t'}$

   By Lemma 2.3.4 (Type-formation inversion), from $\Gamma \vdash \mathrm{El}\ t$, we have $\Gamma \vdash t : \mathrm{Set}$. From $t \rightsquigarrow t'$, by Lemma 2.4.18 (2) (Subject reduction, inhabitation) we have $\Gamma \vdash t' : \mathrm{Set}$. By Definition 2.3.1 (Type formation) we have $\Gamma \vdash \mathrm{El}\ t'$.

2. $\dfrac{U \rightsquigarrow U'}{(x : U) \to V \rightsquigarrow (x : U') \to V}$

   By Lemma 2.3.4 (Type-formation inversion), from $\Gamma \vdash (x : U) \to V$, we have $\Gamma, x : U \vdash V$. By Lemma 2.3.6 we have $\vdash \Gamma, x : U$ as a sub-derivation. By Corollary 2.3.7 we have $\Gamma \vdash U$ as a sub-derivation, and $x \notin \mathit{supp}(\Gamma)$. From $U \rightsquigarrow U'$, by induction we have $\Gamma \vdash U'$.

   By Definition 2.3.1 (Context formation) we have $\vdash \Gamma, x : U'$.

We will now verify that $[] : \Gamma, x : U' \to \Gamma, x : U$ holds.

First we show for $x_i \in \mathit{supp}(\Gamma)$ that $\Gamma, x : U' \vdash x_i : (\Gamma, x : U)(x_i)$ holds. We have $\vdash \Gamma, x : U'$ and $(\Gamma, x : U)(x_i) = \Gamma(x_i) = (\Gamma, x : U')(x_i)$.

Secondly, we show for x that $\Gamma, x : U' \vdash x : (\Gamma, x : U)(x)$ holds. We have $\vdash \Gamma, x : U'$. Then $\Gamma, x : U' \vdash x : U'$. From $\Gamma \vdash U$, $\vdash \Gamma, x : U'$ and Corollary 2.4.2 (Weakening) we have $\Gamma, x : U' \vdash U$. By $U' \bowtie U$ and the conversion rule we have $\Gamma, x : U' \vdash x : U$. We have $(\Gamma, x : U)(x) = U$, so we have $\Gamma, x : U' \vdash x : (\Gamma, x : U)(x)$.

From $[] : \Gamma, x : U' \to \Gamma, x : U$, $\Gamma, x : U \vdash V$, by the Substitution Lemma we obtain $\Gamma, x : U' \vdash V$.

By Definition 2.3.1 (Type formation) we have $\Gamma \vdash (x : U') \to V$.

3. $$\frac{V \rightsquigarrow V'}{(x : U) \to V \rightsquigarrow (x : U) \to V'}$$

By Lemma 2.3.4 (Type-formation inversion), from $\Gamma \vdash (x : U) \to V$, we have $\Gamma, x : U \vdash V$. From $V \rightsquigarrow V'$ and induction we get $\Gamma, x : U \vdash V'$. By definition we get $\Gamma \vdash (x : U) \to V'$.

□

2.5 Type checking

2.5.1 A type checking relation

We will define a relation which can be shown equivalent to a fragment of Definition 2.3.1 for β-normal terms. Since we have untyped abstractions, we have decidable type correctness only for this fragment. Notice that normalization is not required for this definition to be sound and complete with respect to Definition 2.3.1 (Typing), see below Lemma 2.5.2, page 59 and Corollary 2.5.6, page 64.

Instead of using this definition, one could have used Definition 2.3.1 (Typing) for β-normal terms, and iterate typing of application and inversion properties.

In Section 5.1 we will show the decidability of this relation, which follows from normalization. Convertibility is tested by normalizing the terms and comparing them syntactically.[15]

Two purposes Our type checking relation serves two purposes. The first is the obvious one, to be a specification of how to type-check terms, being a basis for an implementation of the system. The second purpose is of technical nature. We are going to use it to connect well-typedness of defined constants with well-founded recursion on the call instance relation in order to prove reducibility of recursive constants. See Lemma 3.6.6, page 82.

[15] Recall this means up to α-convertibility.

Definition 2.5.1 (Type checking). The relations $\Gamma \vdash s \uparrow T$ and $\Gamma \vdash S \uparrow$ are inductively defined as follows:[16]

1. Checking type inhabitation

(a) $$\frac{\Gamma \vdash s_i \uparrow T_i[s_1,\ \ldots\ , s_{i-1}]}{\Gamma \vdash x\ s_1\ \ldots\ s_n \uparrow U} \left\{ \begin{array}{l} \Gamma(x) = (x_1 : T_1,\ \ldots\ , x_n : T_n) \to T \\ U \bowtie T[s_1,\ \ldots\ , s_n] \end{array} \right.$$

(b) $$\frac{\Gamma \vdash s_i \uparrow T_i[s_1,\ \ldots\ , s_{i-1}]}{\Gamma \vdash f\ s_1\ \ldots\ s_n \uparrow U} \left\{ \begin{array}{l} \mathcal{F}(f) = (x_1 : T_1,\ \ldots\ , x_n : T_n) \to T \\ U \bowtie T[s_1,\ \ldots\ , s_n] \end{array} \right.$$

(c) $$\frac{\Gamma \vdash s_i \uparrow \mathrm{Set}}{\Gamma \vdash d\ s_1\ \ldots\ s_n \uparrow \mathrm{Set}}\ \mathcal{D}(d) = \mathrm{Set}^n \to \mathrm{Set}$$

(d) $$\frac{\Gamma \vdash s_i \uparrow \mathrm{El}\ e_i[u_1,\ \ldots\ , u_k]}{\Gamma \vdash c\ s_1\ \ldots\ s_n \uparrow \mathrm{El}\ u} \left\{ \begin{array}{l} \mathcal{C}(c) = (\mathrm{El}\ e_1,\ \ldots\ ,\ \mathrm{El}\ e_n) \\ \qquad\qquad \to \mathrm{El}\ (d\ x_1\ \ldots\ x_k) \\ u \rightsquigarrow^* d\ u_1\ \ldots\ u_k \end{array} \right.$$

(e) $$\frac{\Gamma \vdash s_1 \uparrow \mathrm{Set} \quad \Gamma \vdash s_2 \uparrow \mathrm{El}\ s_1 \to \mathrm{Set}}{\Gamma \vdash \Pi\ s_1\ s_2 \uparrow \mathrm{Set}}$$

(f) $$\frac{\Gamma \vdash s \uparrow (x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)}{\Gamma \vdash \mathrm{fun}\ s \uparrow \mathrm{El}\ v} \left\{ \begin{array}{l} v \rightsquigarrow^* \Pi\ t\ u \\ x \notin FV(u) \end{array} \right.$$

(g) $$\frac{\Gamma, x : U \vdash s \uparrow V}{\Gamma \vdash \lambda x.s \uparrow (x : U) \to V}\ x \notin \mathit{supp}(\Gamma)$$

2. Checking type formation

(a) $$\frac{}{\Gamma \vdash \mathrm{Set} \uparrow}$$

(b) $$\frac{\Gamma \vdash s \uparrow \mathrm{Set}}{\Gamma \vdash \mathrm{El}\ s \uparrow}$$

(c) $$\frac{\Gamma \vdash S_1 \uparrow \quad \Gamma, x : S_1 \vdash S_2 \uparrow}{\Gamma \vdash (x : S_1) \to S_2 \uparrow}\ x \notin \mathit{supp}(\Gamma)$$

Note that in the side conditions of the cases 1d and 1f above, any number of computation steps that gives the required form is accepted, so the result is not necessarily the normal form. At this stage we do not need to know that the reduction reaches normal form, and therefore we do not require it. However, when proving decidability of type-correctness, we will need the normalization property of well-typed terms to decide the existence of the desired forms.

2.5.2 Soundness of type checking

Lemma 2.5.2 (Soundness of type checking).
$\vdash \Sigma\ \wedge\ \Gamma \vdash T\ \wedge\ \Gamma \vdash s \uparrow T\ \Rightarrow\ \Gamma \vdash s : T.$

[16] Recall that S and s denotes the β-normal fragment of the language. See Definition 2.1.20, page 28.

Proof. Assume $\vdash \Sigma$, $\Gamma \vdash T$ and $\Gamma \vdash s \uparrow T$. First note that from $\Gamma \vdash T$ and Lemma 2.3.6 we have

$$\vdash \Gamma. \tag{2.21}$$

We proceed by induction on s. The derivation of $\Gamma \vdash s \uparrow T$ gives the following cases:

1. $$\frac{\Gamma \vdash s_i \uparrow U_i[s_1, \ \ldots \ , s_{i-1}]}{\Gamma \vdash x \ s_1 \ \ldots \ s_n \uparrow T} \begin{cases} \Gamma(x) = (y_1 : U_1, \ \ldots \ , y_n : U_n) \to U \\ T \bowtie U[s_1, \ \ldots \ , s_n] \end{cases}$$

   From (2.21) we get $\Gamma \vdash \Gamma(x)$. From Lemma 2.3.5 we get $\Gamma \vdash U_1$, $\Gamma, y_1 : U_1 \vdash U_2$ through $\Gamma, y_1 : U_1, \ \ldots \ , y_{n-1} : U_{n-1} \vdash U_n$, and $\Gamma, y_1 : U_1, \ \ldots \ , y_n : U_n \vdash U$.

   From $\Gamma \vdash U_1$ and induction we get $\Gamma \vdash s_1 : U_1$.

   We have

   $$[s_1/y_1] : \Gamma \to \Gamma, y_1 : U_1$$

   and by Lemma 2.4.4 (Substitution lemma) we have $\Gamma \vdash U_2[s_1/y_1]$. By induction for $\Gamma \vdash s_2 \uparrow U_2[s_1/y_1]$ we get $\Gamma \vdash s_2 : U_2[s_1/y_1]$. We proceed similarly until we get

   $$[s_1/y_1, \ \ldots \ , s_{n-1}/y_{n-1}] \ : \ \Gamma \ \to \ \Gamma, y_1 : U_1, \ \ldots \ , y_{n-1} : U_{n-1}$$

   and by Lemma 2.4.4 (Substitution lemma) get $\Gamma \vdash U_n[s_1, \ \ldots \ , s_{n-1}]$ and by induction from $\Gamma \vdash s_n \uparrow U_n[s_1, \ \ldots \ , s_{n-1}]$ obtain $\Gamma \vdash s_n : U_n[s_1, \ \ldots \ , s_{n-1}]$.

   By Lemma 2.4.11 we get $\Gamma \vdash x \ s_1 \ \ldots \ s_n : U[s_1, \ \ldots \ , s_n]$. By conversion of typing we get $\Gamma \vdash x \ s_1 \ \ldots \ s_n : T$.

2. $$\frac{\Gamma \vdash s_i \uparrow U_i[s_1, \ \ldots \ , s_{i-1}]}{\Gamma \vdash f \ s_1 \ \ldots \ s_n \uparrow U} \begin{cases} \mathcal{F}(f) = (y_1 : U_1, \ \ldots \ , y_n : U_n) \to U \\ T \bowtie U[s_1, \ \ldots \ , s_n] \end{cases}$$

   From $\vdash \Sigma$ we get $\vdash \mathcal{F}(f)$.

   By iteration of Corollary 2.4.2 (Weakening) we get $\Gamma \vdash \mathcal{F}(f)$.

   As in the last case we get by successively applying the substitution lemma and induction

   $$\Gamma \vdash s_1 : U_1 \qquad \ldots \qquad \Gamma \vdash s_n : U_n[s_1, \ \ldots \ , s_{n-1}]$$

   By Lemma 2.4.11 and conversion we get $\Gamma \vdash f \ s_1 \ \ldots \ s_n : T$.

3. $$\frac{\Gamma \vdash s_i \uparrow \mathrm{Set}}{\Gamma \vdash d \ s_1 \ \ldots \ s_n \uparrow \mathrm{Set}} \ \mathcal{D}(d) = \mathrm{Set}^n \to \mathrm{Set}$$

   By assumption $\vdash \Gamma$, hence $\Gamma \vdash \mathrm{Set}$ and $\Gamma \vdash d : \mathrm{Set}^n \to \mathrm{Set}$.

   By induction $\Gamma \vdash s_i : \mathrm{Set}$.

   By Lemma 2.4.11 we get $\Gamma \vdash d \ s_1 \ \ldots \ s_n : T$.

4. $$\frac{\Gamma \vdash s_i \uparrow \text{El } e_i[u_1,\ \ldots\ , u_k]}{\Gamma \vdash c\ s_1\ \ldots\ s_n \uparrow \text{El } u} \begin{cases} \mathcal{C}(c) = (\text{El } e_1,\ \ldots\ ,\ \text{El } e_n) \\ \qquad\quad \to \text{El } (d\ y_1\ \ldots\ y_k) \\ u \rightsquigarrow^* d\ u_1\ \ldots\ u_k \end{cases}$$

   From $\Gamma \vdash \text{El } u$, $u \rightsquigarrow^* d\ u_1\ \ldots\ u_k$ and Lemma 2.4.18 (Subject reduction) we have $\Gamma \vdash \text{El } (d\ u_1\ \ldots\ u_k)$. By Lemma 2.4.12 then $\Gamma \vdash u_i : \text{Set}$.

   We have $\vdash \Gamma$, and by Definition 2.3.1 (Typing) we get $\Gamma \vdash c : \mathcal{C}(c)[u_1,\ \ldots\ , u_k]$. By Lemma 2.3.5 we get $\Gamma \vdash \text{El } e_i[u_1,\ \ldots\ , u_k]$. By induction $\Gamma \vdash s_i : \text{El } e_i[u_1,\ \ldots\ , u_k]$. By Lemma 2.4.11 we get $\Gamma \vdash c\ s_1\ \ldots\ s_n : \text{El } (d\ u_1\ \ldots\ u_k)$, and by type inhabitation, conversion rule we get $\Gamma \vdash c\ s_1\ \ldots\ s_n : \text{El } u$.

5. $$\frac{\Gamma \vdash s_1 \uparrow \text{Set} \quad \Gamma \vdash s_2 \uparrow \text{El } s_1 \to \text{Set}}{\Gamma \vdash \Pi\ s_1\ s_2 \uparrow \text{Set}}$$

   From $\vdash \Gamma$ we have $\Gamma \vdash \text{Set}$. By induction $\Gamma \vdash s_1 : \text{Set}$. We have then $\Gamma \vdash \text{El } s_1 \to \text{Set}$. By induction $\Gamma \vdash s_2 : \text{El } s_1 \to \text{Set}$.

   We have $\Gamma \vdash \Pi : (x : \text{Set},\ \text{El } x \to \text{Set}) \to \text{Set}$.

   By Lemma 2.4.11 we get $\Gamma \vdash \Pi\ s_1\ s_2 : \text{Set}$

6. $$\frac{\Gamma \vdash s_1 \uparrow (x : \text{El } t) \to \text{El } (u\ x)}{\Gamma \vdash \text{fun } s_1 \uparrow \text{El } v} \begin{cases} v \rightsquigarrow^* \Pi\ t\ u \\ x \notin FV(u) \end{cases}$$

   From $\vdash \Gamma$ we have $\Gamma \vdash \Pi : (x : \text{Set},\ \text{El } x \to \text{Set}) \to \text{Set}$.

   From $\Gamma \vdash \text{El } v$, $v \rightsquigarrow^* \Pi\ t\ u$ and Lemma 2.4.18 (Subject reduction) we have $\Gamma \vdash \text{El } (\Pi\ t\ u)$. By Lemma 2.4.14 we have $\Gamma \vdash (x : \text{El } t) \to \text{El } (u\ x)$.

   By induction then $\Gamma \vdash s_1 : (x : \text{El } t) \to \text{El } (u\ x)$. Since $x \notin FV(u)$ we have by definition $\Gamma \vdash \text{fun } s_1 : \text{El } (\Pi\ t\ u)$, and then by conversion we get $\Gamma \vdash \text{fun } s_1 : \text{El } v$.

7. $$\frac{\Gamma, x : U \vdash s_1 \uparrow V}{\Gamma \vdash \lambda x.s_1 \uparrow (x : U) \to V}\ x \notin \mathit{supp}(\Gamma)$$

   From $\Gamma \vdash (x : U) \to V$ and Lemma 2.3.4 (Type-formation inversion) we have $\Gamma, x : U \vdash V$.

   By induction $\Gamma, x : U \vdash s_1 : V$.

   By Definition 2.3.1 (Typing) we get $\Gamma \vdash \lambda x.s_1 : (x : U) \to V$.

□

Lemma 2.5.3 (Soundness of type formation checking).
$\vdash \Sigma \wedge \ \vdash \Gamma \ \wedge \ \Gamma \vdash S \uparrow \Rightarrow \ \Gamma \vdash S$

Proof. Assume $\vdash \Sigma$, $\vdash \Gamma$ and $\Gamma \vdash S \uparrow$. We proceed by induction on S. From $\Gamma \vdash S \uparrow$ we have the cases:

1. $\dfrac{}{\Gamma \vdash \text{Set} \uparrow}$.

   By assumption $\vdash \Gamma$, hence $\Gamma \vdash \text{Set}$.

2. $\dfrac{\Gamma \vdash s \uparrow \text{Set}}{\Gamma \vdash \text{El } s \uparrow}$.

   By assumption $\vdash \Gamma$, hence $\Gamma \vdash \text{Set}$. Then, by Lemma 2.5.2 we have $\Gamma \vdash s : \text{Set}$. By definition then $\Gamma \vdash \text{El } s$.

3. $\dfrac{\Gamma \vdash S_1 \uparrow \quad \Gamma, x : S_1 \vdash S_2 \uparrow}{\Gamma \vdash (x : S_1) \to S_2 \uparrow}\ x \notin \mathit{supp}(\Gamma)$

   By assumption $\vdash \Gamma$, and by induction then $\Gamma \vdash S_1$. Since $x \notin \mathit{supp}(\Gamma)$ we have $\vdash \Gamma, x : S_1$. By induction then $\Gamma, x : S_1 \vdash S_2$. By definition then $\Gamma \vdash (x : S_1) \to S_2$.

□

2.5.3 Completeness of type checking

Notation 2.5.4 *(Equality for contexts).* If $\overrightarrow{\mathit{supp}}(\Gamma) = \overrightarrow{\mathit{supp}}(\Delta)$ then $\Gamma \bowtie \Delta$ whenever $\Gamma(x) \bowtie \Delta(x)$ for all $x \in \mathit{supp}(\Gamma)$.

Lemma 2.5.5 (Completeness of type checking with conversion).
If $\vdash \Sigma$ *then* $\Gamma \vdash s : T \ \wedge \ \Gamma \bowtie \Delta \ \wedge \ T \bowtie U \ \Rightarrow \ \Delta \vdash s \uparrow U$.

Proof. Assume $\vdash \Sigma$, $\Gamma \vdash s : T$, $\Gamma \bowtie \Delta$ and $T \bowtie U$. We proceed by induction on s, which may have the following forms:

- $x\ s_1\ \ldots\ s_n$.
  By Lemma 2.4.12 there are $V_1, \ldots, V_n$ and V such that $\Gamma \vdash x : (x_1 : V_1, \ldots, x_n : V_n) \to V$, $\Gamma \vdash s_i : V[s_1, \ldots, s_{i-1}]$ and $T \bowtie V[s_1, \ldots, s_n]$. By Lemma 2.4.9 (Generation lemma) then $(x_1 : V_1, \ldots, x_n : V_n) \to V \bowtie \Gamma(x)$ holds.

  Then $\Gamma(x)$ is of the form $x_1 : V_1', \ldots, x_n : V_n' \to V'$, with $V_i \bowtie V_i'$ and $V \bowtie V'$. From $\Gamma \bowtie \Delta$ we have $\Gamma(x) \bowtie \Delta(x)$. Then $\Delta(x)$ is of the form $x_1 : V_1'', \ldots, x_n : V_n'' \to V''$ with $V_i' \bowtie V_i''$ and $V' \bowtie V''$.

  By transitivity of equality then $V_i \bowtie V_i''$ and $V \bowtie V''$, and then $V_i[s_1, \ldots, s_{i-1}] \bowtie V_i''[s_1, \ldots, s_{i-1}]$ and $V[s_1, \ldots, s_n] \bowtie V''[s_1, \ldots, s_n]$ holds. From $T \bowtie U$ and $T \bowtie V[s_1, \ldots, s_n]$ we have $U \bowtie V''[s_1, \ldots, s_n]$.

  By induction $\Delta \vdash s_i \uparrow V_i''[s_1, \ldots, s_{i-1}]$. By Definition 2.5.1 (Type checking) then $\Delta \vdash x\ s_1\ \ldots\ s_n \uparrow U$.

- $h\ s_1\ \ldots\ s_n$, where $n = ar(h)$.
  By Lemma 2.4.12 there are $V_1, \ldots, V_n$ and V such that
  $\Gamma \vdash h : (x_1 : V_1, \ldots, x_n : V_n) \to V$, $\Gamma \vdash s_i : V[s_1, \ldots, s_{i-1}]$ and $T \bowtie V[s_1, \ldots, s_n]$. We have the following forms of h:
  - f.
    By Lemma 2.4.9 (Generation lemma) then $(x_1 : V_1, \ldots, x_n : V_n) \to V \bowtie \mathcal{F}(f)$.
    Then $\mathcal{F}(f)$ is of the form $x_1 : V_1', \ldots, x_n : V_n' \to V'$ with $V_i \bowtie V_i'$ and $V \bowtie V'$.
    Then $V_i[s_1, \ldots, s_{i-1}] \bowtie V_i'[s_1, \ldots, s_{i-1}]$ and
    $V[s_1, \ldots, s_n] \bowtie V'[s_1, \ldots, s_n]$.
    From $T \bowtie U$ and
    $T \bowtie V[s_1, \ldots, s_n]$ we have $U \bowtie V'[s_1, \ldots, s_n]$.
    By induction $\Delta \vdash s_i \uparrow V_i'[s_1, \ldots, s_{i-1}]$. By Definition 2.5.1 (Type checking) then $\Delta \vdash f\ s_1\ \ldots\ s_n \uparrow U$.
  - d.
    By Lemma 2.4.9 (Generation lemma) $(x_1 : V_1, \ldots, x_n : V_n) \to V = \mathcal{D}(d)$, with $\mathcal{D}(d) = \mathrm{Set}^n \to \mathrm{Set}$. We have $V_i[s_1, \ldots, s_{i-1}] = \mathrm{Set}$. By induction $\Delta \vdash s_i \uparrow \mathrm{Set}$. Since $T \bowtie U$ and $T = \mathrm{Set}$ we have $U = \mathrm{Set}$. By Definition 2.5.1 (Type checking) then $\Delta \vdash d\ s_1\ \ldots\ s_n \uparrow U$.
  - c.
    From previously we know $\Gamma \vdash c : (x_1 : V_1, \ldots, x_n : V_n) \to V$,
    $\Gamma \vdash s_i : V[s_1, \ldots, s_{i-1}]$ and $T \bowtie V[s_1, \ldots, s_n]$.
    $\mathcal{C}(c)$ is an independent function type of the form
    $(\mathrm{El}\ e_1, \ldots, \mathrm{El}\ e_n) \to \mathrm{El}\ (d\ y_1\ \ldots\ y_k)$.
    By Lemma 2.4.9 (Generation lemma) for c we have
    $(x_1 : V_1, \ldots, x_n : V_n) \to V \bowtie$
    $(x_1 : \mathrm{El}\ (e_1[\vec{u}]), \ldots, x_n : \mathrm{El}\ (e_n[\vec{u}])) \to \mathrm{El}\ (d\ \vec{u})$.
    Then $V_i \bowtie \mathrm{El}\ (e_i[\vec{u}])$ and $V \bowtie \mathrm{El}\ (d\ \vec{u})$.
    By the definition of equality there is $\vec{v}$ such that

    $$V_i \rightsquigarrow^* \mathrm{El}\ (e_i[\vec{v}]) \qquad \text{and} \qquad \mathrm{El}\ (e_i[\vec{u}]) \rightsquigarrow^* \mathrm{El}\ (e_i[\vec{v}]),$$

    $$V \rightsquigarrow^* \mathrm{El}\ (d\ \vec{v}) \qquad \text{and} \qquad \mathrm{El}\ (d\ \vec{u}) \rightsquigarrow^* \mathrm{El}\ (d\ \vec{v}).$$

    Since $\mathcal{C}(c)$ is independent we have $x_i \notin FV(\vec{v})$. We have
    $V_i[s_1, \ldots, s_{i-1}] \rightsquigarrow^* \mathrm{El}\ (e_i[\vec{v}])$ and $V[s_1, \ldots, s_n] \rightsquigarrow^* \mathrm{El}\ (d\ \vec{v})$.
    From $T \bowtie U$ and $T \bowtie V[s_1, \ldots, s_n]$ we have $U \bowtie V[s_1, \ldots, s_n]$.
    From the definition of equality and that patterns are closed under reduction there is $\vec{w}$ such that

    $$U \rightsquigarrow^* \mathrm{El}\ (d\ \vec{w}) \qquad \text{and} \qquad V[s_1, \ldots, s_n] \rightsquigarrow^* \mathrm{El}\ (d\ \vec{w}).$$

    Then $V_i[s_1, \ldots, s_{i-1}] \bowtie \mathrm{El}\ (e_i[\vec{w}])$. We have U of the form El u with $u \rightsquigarrow^* d\ \vec{w}$. By induction $\Delta \vdash s_i \uparrow \mathrm{El}\ (e_i[\vec{w}])$. By Definition 2.5.1 (Type checking) then $\Delta \vdash c\ s_1\ \ldots\ s_n \uparrow U$.

– Π.
By Lemma 2.4.9 (Generation lemma) $T \bowtie (x_1 : \text{Set}, \text{El } x_1 \to \text{Set}) \to \text{Set}$. From $\Gamma \vdash s_1 : V_1$ and $\Gamma \vdash s_2 : V_2[s_1/x_1]$ we know $\Gamma \vdash s_1 : \text{Set}$ and $\Gamma \vdash s_2 : \text{El } s_1 \to \text{Set}$.
By induction $\Delta \vdash s_1 \uparrow \text{Set}$ and $\Delta \vdash s_2 \uparrow \text{El } s_1 \to \text{Set}$. Since $U = \text{Set}$ we have by definition that $\Delta \vdash \Pi\ s_1\ s_2 \uparrow U$.

– fun.
By Lemma 2.4.9 (Generation lemma) from $\Gamma \vdash \text{fun} : (x_1 : V_1) \to V$ we have

$$(x_1 : V_1) \to V \bowtie (x_1 : (y : \text{El } t) \to \text{El } (u\ y)) \to \text{El } (\Pi\ t\ u)$$

with $y \notin FV(u)$, $\Gamma \vdash t : \text{Set}$ and $\Gamma \vdash u : \text{El } t \to \text{Set}$.
We have $V_1 \bowtie (y : \text{El } t) \to \text{El } (u\ y)$ and $V \bowtie \text{El } (\Pi\ t\ u)$.
From $U \bowtie T$, $T \bowtie V[s_1/x_1]$ and $V \bowtie \text{El } (\Pi\ t\ u)$ we have $U = \text{El } v$ with t', u' such that $v \rightsquigarrow^* \Pi\ t'\ u'$ and $\Pi\ t\ u \rightsquigarrow^* \Pi\ t'\ u'$.
We have $\Gamma \vdash s_1 : V_1$ and $V_1 \bowtie (y : \text{El } t') \to \text{El } (u'\ y)$. Since $y \notin FV(u)$ and $u \rightsquigarrow^* u'$ we have $y \notin FV(u')$.
By induction $\Delta \vdash s_1 \uparrow (y : \text{El } t') \to \text{El } (u'\ y)$ and by definition then $\Delta \vdash \text{fun } s_1 \uparrow \text{El } v$.

- $\lambda x.s'$.
By Lemma 2.4.10 (Strong generation lemma) we have $T = (x : T_1) \to T_2$ and $\Gamma, x : T_1 \vdash s' : T_2$. Since $T \bowtie U$ we have U of the form $(x : U_1) \to U_2$ where $T_1 \bowtie U_1$ and $T_2 \bowtie U_2$. Then $\Gamma, x : T_1 \bowtie \Delta, x : U_1$. By induction $\Delta, x : U_1 \vdash s' \uparrow U_2$. Assume w.l.o.g. that $x \notin \mathit{supp}(\Gamma)$. By definition $\Delta \vdash \lambda x.s' \uparrow (x : U_1) \to U_2$.

□

Corollary 2.5.6 (Completeness of type checking).
$\vdash \Sigma \ \wedge\ \Gamma \vdash s : T \ \Rightarrow\ \Gamma \vdash s \uparrow T$

Lemma 2.5.7 (Completeness of type formation checking).
$\vdash \Sigma \ \wedge \ \Gamma \vdash S \ \Rightarrow \ \Gamma \vdash S \uparrow$.

Proof. Assume $\vdash \Sigma$ and $\Gamma \vdash S$. We proceed by induction on S. By inversion of Definition 2.3.1 S is well-formed by the following derivations:

- $\dfrac{\vdash \Gamma}{\Gamma \vdash \mathrm{Set}}$

  By definition $\Gamma \vdash \mathrm{Set} \uparrow$.

- $\dfrac{\Gamma \vdash s : \mathrm{Set}}{\Gamma \vdash \mathrm{El}\ s}$

  By Corollary 2.5.6 we have $\Gamma \vdash s \uparrow \mathrm{Set}$, and by definition $\Gamma \vdash \mathrm{El}\ s \uparrow$.

- $\dfrac{\Gamma \vdash S_1 \quad \Gamma, x : S_1 \vdash S_2}{\Gamma \vdash (x : S_1) \to S_2}$

  By induction we have $\Gamma \vdash S_1 \uparrow$ and $\Gamma, x : S_1 \vdash S_2 \uparrow$. By definition then $\Gamma \vdash (x : S_1) \to S_2 \uparrow$.

□

Chapter 3

Semantics

In this chapter we introduce a semantic notion of reducibility[1]. We prove that well-typed terms are reducible, if all defined constants are reducible. Then we prove the latter condition provided that the call-instance relation (Definition 3.6.3) is well-founded.

3.1 Reducibility

3.1.1 Neutrality

Definition 3.1.1 (Neutral). Neutral terms are inductively defined by the grammar

$$\begin{array}{rcll} b & ::= & x\ t_1\ \ldots\ t_n & \text{where } \mathsf{NF}(t_i) \\ & | & f\ t_1\ \ldots\ t_n & \text{where } \mathsf{NF}(f\ t_1\ \ldots\ t_n),\ n \geq ar(f) \end{array}$$

Notation 3.1.2. We write $\mathsf{NEUTRAL}(t)$ when t is neutral.

Remark 3.1.3. Neutral terms are normal. Note that the notion of neutrality, as a consequence of its dependency of normality, is relative to the rules given in $\mathcal{R}$.

Lemma 3.1.4. *If* $\mathsf{NEUTRAL}(t)$ *and* $\mathsf{NF}(u)$ *then* $\mathsf{NEUTRAL}(t\ u)$.

Proof. We consider Definition 3.1.1. If the head of t is a variable, we are done. If it is a constant, this constant is at least fully applied, so we will not get a redex by applying u. □

3.1.2 Specification of reducibility

We postulate the existence of the semantic predicates having the forms $\mathsf{RED}(T)$ and $\mathsf{RED}_T(t)$. Later we will give an instantiation to show that the specification is not vacuous.

[1] See the discussion in Section 1.5.3, page 20.

Specification 3.1.5 (Reducible sets and elements). Given the data types $\mathcal{D}, \mathcal{C}$ and the rules $\mathcal{R}$, we specify the predicate $\mathsf{RED}_{\mathrm{Set}}$ for sets, and a family of predicates $\mathsf{RED}_{\mathrm{El}\ t}$ indexed by terms t satisfying $\mathsf{RED}_{\mathrm{Set}}(t)$. We give clauses 1 and 2 mutually by induction-recursion.

1. $\mathsf{RED}_{\mathrm{Set}}(t)$ holds iff one the following conditions holds[2]
   (a) $\exists d, t_1, \ \ldots\ , t_m\ .$
   $$\begin{cases} t \Downarrow d\ t_1\ \ldots\ t_m \\ \mathcal{D}(d) = \mathrm{Set}^m \to \mathrm{Set} \\ \mathsf{RED}_{\mathrm{Set}}(t_1),\ \ldots\ ,\ \mathsf{RED}_{\mathrm{Set}}(t_m) \end{cases}$$
   (b) $\exists t_1, t_2\ .$
   $$\begin{cases} t \Downarrow \Pi\ t_1\ t_2 \\ \mathsf{RED}_{\mathrm{Set}}(t_1) \\ \forall v.\mathsf{RED}_{(\mathrm{El}\ t_1)}(v) \Rightarrow \mathsf{RED}_{\mathrm{Set}}(t_2\ v) \end{cases}$$
   (c) $\exists b\ .\ t \Downarrow b$

2. If $\mathsf{RED}_{\mathrm{Set}}(t)$ holds, we specify for each of the possible cases, if:
   (a) $\exists d, t_1, \ \ldots\ , t_m\ .$
   $$\begin{cases} t \Downarrow d\ t_1\ \ldots\ t_m \\ \mathcal{D}(d) = \mathrm{Set}^m \to \mathrm{Set} \\ \mathsf{RED}_{\mathrm{Set}}(t_1),\ \ldots\ ,\ \mathsf{RED}_{\mathrm{Set}}(t_m) \end{cases}$$
   then $\mathsf{RED}_{(\mathrm{El}\ t)}(u)$ holds iff one the following conditions holds[3]
      i. $\exists c, u_1, \ \ldots\ , u_n\ .$
      $$\begin{cases} u \Downarrow c\ u_1\ \ldots\ u_n \\ \mathcal{C}(c) = (\mathrm{El}\ e_1,\ \ldots\ ,\ \mathrm{El}\ e_n) \to \mathrm{El}\ (d\ x_1\ \ldots\ x_m) \\ \mathsf{RED}_{\mathrm{Set}}(e_i[t_1,\ \ldots\ , t_m])\ \wedge\ \mathsf{RED}_{(\mathrm{El}\ e_i[t_1,\ \ldots\ ,t_m])}(u_i) \end{cases}$$
      ii. $\exists b\ .\ u \Downarrow b$
   (b) $\exists t_1, t_2\ .$
   $$\begin{cases} t \Downarrow \Pi\ t_1\ t_2 \\ \mathsf{RED}_{\mathrm{Set}}(t_1) \\ \forall v.\mathsf{RED}_{(\mathrm{El}\ t_1)}(v) \Rightarrow \mathsf{RED}_{\mathrm{Set}}(t_2\ v) \end{cases}$$
   then $\mathsf{RED}_{(\mathrm{El}\ t)}(u)$ holds iff one the following conditions holds
      i. $\exists u'.$
      $$\begin{cases} u \Downarrow \mathrm{fun}\ u' \\ \forall v.\mathsf{RED}_{(\mathrm{El}\ t_1)}(v) \Rightarrow \mathsf{RED}_{(\mathrm{El}\ (t_2\ v))}(u'\ v) \end{cases}$$
      ii. $\exists b\ .\ u \Downarrow b$
   (c) $\exists b\ .\ t \Downarrow b$,
   then $\mathsf{RED}_{(\mathrm{El}\ t)}(u)$ holds whenever $\exists b'\ .\ u \Downarrow b'$.

[2] Recall from Notation 2.2.37 that $t \Downarrow u$ means "t normalizes to u". Also recall that terms of the form b are *neutral*, Definition 3.1.1, page 67.

[3] Recall that terms of the form e are first-order *set patterns*, Definition 2.1.3, page 26. See also Definition 2.1.17 (Data type specification), page 28.

Definition 3.1.6 (Rank of raw types).

$$\begin{aligned} rank(\text{Set}) &= 0 \\ rank(\text{El } t) &= 0 \\ rank(\text{Fun } U\ (\lambda x.V)) &= 1 + max(rank(U), rank(V)) \end{aligned}$$

Remark 3.1.7. It is clear that the rank of a type is unaffected by reduction as well as substitution.

Specification 3.1.8 (Reducibility).
We specify the predicate RED for types T by induction on $rank(T)$, and for each case for which $\mathsf{RED}(T)$ holds, we specify the predicate RED_T.

1. $\mathsf{RED}(\text{Set})$ holds,
   and then $\mathsf{RED}_{\text{Set}}(t)$ is given by Specification 3.1.5.

2. $\mathsf{RED}(\text{El } t)$ holds whenever $\mathsf{RED}_{\text{Set}}(t)$ holds,
   and then $\mathsf{RED}_{(\text{El } t)}(u)$ is given by Specification 3.1.5.

3. $\mathsf{RED}(\text{Fun } U\ (\lambda x.V))$ holds whenever
   $\mathsf{RED}(U)$ and $\forall u.\mathsf{RED}_U(u) \Rightarrow \mathsf{RED}(V[u/x])$ holds,
   and then $\mathsf{RED}_{(\text{Fun } U\ (\lambda x.V))}(t)$ holds whenever
   $\forall u.\mathsf{RED}_U(u) \Rightarrow \mathsf{RED}_{(V[u/x])}(t\ u)$.

3.1.3 Examples

We illustrate by examples what it means to be a reducible set, and in such a set, what it means to be a reducible element. Let us consider the example with vectors of length n, from page 42. The expression Vec Bool $(2+1)$ is reducible in Set, which can be seen from its normalization sequence:

$$\begin{aligned} &\text{Vec Bool } (2+1) \rightsquigarrow \\ &\text{Vec Bool } 3 \rightsquigarrow \\ &\text{Bool} \times \text{Vec Bool } 2 \rightsquigarrow \\ &\text{Bool} \times (\text{Bool} \times \text{Vec Bool } 1) \rightsquigarrow \\ &\text{Bool} \times (\text{Bool} \times (\text{Bool} \times \text{Vec Bool } 0)) \rightsquigarrow \\ &\text{Bool} \times (\text{Bool} \times (\text{Bool} \times \top)) \end{aligned}$$

Informally we can infer

$$\dfrac{\mathsf{RED}_{\text{Set}}(\text{Bool}) \qquad \dfrac{\mathsf{RED}_{\text{Set}}(\text{Bool}) \qquad \dfrac{\mathsf{RED}_{\text{Set}}(\text{Bool}) \quad \mathsf{RED}_{\text{Set}}(\top)}{\mathsf{RED}_{\text{Set}}(\text{Bool} \times \text{Vec Bool } 0)}}{\mathsf{RED}_{\text{Set}}(\text{Bool} \times \text{Vec Bool } 1)}}{\dfrac{\mathsf{RED}_{\text{Set}}(\text{Bool} \times \text{Vec Bool } 2)}{\dfrac{\mathsf{RED}_{\text{Set}}(\text{Vec Bool } 3)}{\mathsf{RED}_{\text{Set}}(\text{Vec Bool } (2+1))}}}$$

with the side conditions $\mathcal{D}(\times) = \text{Set}^2 \to \text{Set}$ and $\mathcal{D}(\text{Bool}) = \text{Set}$.

We have $\mathsf{RED}_{(\text{El (Vec Bool 3)})}(\text{true}, (\text{and true false}, ((\lambda x.x)\ \text{true}, \text{unit})))$ because $(\text{true}, (\text{and true false}, ((\lambda x.x)\ \text{true}, \text{unit}))) \Downarrow (\text{true}, (\text{false}, (\text{true}, \text{unit})))$ and we can infer informally

$$\dfrac{\mathsf{RED}_{(\text{El Bool})}(\text{true}) \qquad \dfrac{\mathsf{RED}_{(\text{El Bool})}(\text{false}) \qquad \dfrac{\mathsf{RED}_{(\text{El Bool})}(\text{true}) \quad \mathsf{RED}_{(\text{El }\top)}(\text{unit})}{\mathsf{RED}_{(\text{El (Vec Bool 1)})}(\text{true}, \text{unit})}}{\mathsf{RED}_{(\text{El (Vec Bool 2)})}(\text{false}, (\text{true}, \text{unit}))}}{\mathsf{RED}_{(\text{El (Vec Bool 3)})}(\text{true}, (\text{false}, (\text{true}, \text{unit})))}$$

with the side conditions $\mathcal{C}(,) = (\text{El } x,\ \text{El } y) \to \text{El } (x \times y)$, $\mathcal{C}(\text{true}) = \text{El Bool}$, $\mathcal{C}(\text{false}) = \text{El Bool}$ and $\mathcal{C}(\text{unit}) = \text{El } \top$.

Neutrality The term 'odd(s (s x))' is reducible in El Bool, as seen from the computation[4]

$$\text{odd(s (s } x)) \rightsquigarrow \text{odd } x$$

where 'odd x' is neutral. Another (yet simple) example involving lists, is the term 'length(5::3::y)', which is reducible in El Nat, since it normalizes by the sequence

$$\text{length(5::3::}y) \rightsquigarrow \text{s(length(3::}y)) \rightsquigarrow \text{s(s(length } y))$$

and the term 's (s (length y))' is reducible in El Nat. The computation of 'length y' is blocked, and the function is fully applied, so it is neutral.

Weak normalization and reducibility Note that a reducible term may have no type, and that it may contain sub-terms that have no normal form. For instance we have $\mathsf{RED}_{(\text{El Bool})}(((\lambda x.\lambda y.y)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)))\ \text{true})$ because $((\lambda x.\lambda y.y)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)))\ \text{true} \Downarrow \text{true}$, even if $(\lambda x.x\ x)\ (\lambda x.x\ x)$ has no normal form.

3.2 The soundness of the reducibility predicates

We give a sketch of how the reducibility predicate specifications of Specification 3.1.5 can be justified using a set-theoretical interpretation. The approach is similar to Scott (1975) and Aczel (1980).

3.2.1 A potential counter-example

Assume that the following data type would be part of our system:

$$\begin{aligned} &\text{D} : \text{Set} \\ &\text{abstr} : (\text{El D} \to \text{El D}) \to \text{El D} \end{aligned}$$

[4]For these examples, recall the computation rules we gave in the introduction, page 8 and on.

and we have the constant

$$\begin{aligned}&\text{app} : \text{El D} \to \text{El D} \to \text{El D}\\&\text{app (abstr } f)\ x = f\ x\end{aligned}$$

One can add to the specification of reducibility the following clauses:

1. $\mathsf{RED}_{\text{Set}}(\text{D})$
2. $\mathsf{RED}_{(\text{El D})}(u)$ iff
   $(\exists u'.u \Downarrow \text{abstr } u' \wedge$
   $\quad \forall v.\mathsf{RED}_{(\text{El D})}(v) \Rightarrow \mathsf{RED}_{(\text{El D})}(u'\ v)) \vee$
   $\exists b.u \Downarrow b$

One can show $\mathsf{RED}_{(\text{El D}\to\text{El D}\to\text{El D})}(\text{app})$. We can construct the closed term

$$\begin{aligned}&\text{omega} : \text{El D}\\&\text{omega} = \text{app (abstr } (\lambda x.\text{app } x\ x))\ (\text{abstr } (\lambda x.\text{app } x\ x))\end{aligned}$$

We would then show $\mathsf{RED}_{(\text{El D})}(\text{omega})$. This term has no normal form, and cannot be reduced to an expression in constructor form. What goes wrong is that there is no predicate RED satisfying the conditions 1 and 2 above. Clearly, there is a need to justify the reducibility predicate.

3.2.2 Reducibility predicates as a hierarchy of sets

We give below a motivation[5] of the existence of the predicates $\mathsf{RED}_{\text{Set}}(t)$ and $\mathsf{RED}_{(\text{El } t)}(u)$ given in Specification 3.1.5. Concerning the constructive validity of this method, cf. Aczel (1980).

We are going to build a hierarchy of sets indexed by ordinal numbers. For each level α, we build a set $\mathcal{S}^{(\alpha)}$ of set expressions, and for each $t \in \mathcal{S}^{(\alpha)}$, we form the set $\mathcal{E}_t^{(\alpha)}$ of elements in t. At each level α, to construct $\mathcal{S}^{(\alpha)}$ we first construct a set $\mathcal{L}^{(\alpha)}$. The set $\mathcal{S}^{(\alpha)}$ consists of first-order set constructor trees with leaves taken from $\mathcal{L}^{(\alpha)}$. The following properties are essential:

Proposition 3.2.1.
If $\alpha < \beta$ *then* $\mathcal{S}^{(\alpha)} \subseteq \mathcal{S}^{(\beta)}$, *and if* $t \in \mathcal{S}^{(\alpha)}$ *then* $\mathcal{E}_t^{(\alpha)} = \mathcal{E}_t^{(\beta)}$.

Definition 3.2.2 (Computational closure). For a given set of terms $\mathcal{S}^{(\alpha)}$, by $\mathcal{S}^{(\alpha)+}$, we mean $\left\{t \;\middle|\; t \rightsquigarrow^* t' \ \wedge\ t' \in \mathcal{S}^{(\alpha)}\right\}$. For all terms t, given $\mathcal{E}_t^{(\alpha)}$, by $\mathcal{E}_t^{(\alpha)+}$, we mean $\left\{u \;\middle|\; u \rightsquigarrow^* u' \ \wedge\ t \rightsquigarrow^* t' \ \wedge\ u' \in \mathcal{E}_{t'}^{(\alpha)}\right\}$.

[5]The main ideas behind this motivation come from Thierry Coquand. It took its present form through discussions with the author.

Definition 3.2.3 (Set leaves, Sets and Elements).

1. At level 0:
   (a) $\mathcal{L}^{(0)} = \{t \mid \mathsf{NEUTRAL}(t)\}$
   (b) $\mathcal{S}^{(0)} = \{e[t_1, \ldots, t_n] \mid t_1 \in \mathcal{L}^{(0)}, \ldots, t_n \in \mathcal{L}^{(0)}\}$
   (c) Given $t \in \mathcal{S}^{(0)}$,
   the relation $u \in \mathcal{E}_t^{(0)}$ is specified by induction on u.
      i. $c\ u_1\ \ldots\ u_n \in \mathcal{E}_t^{(0)}$ iff $\mathsf{NF}(u_i)$,
      $\mathcal{C}(c) = (\mathrm{El}\ e_1, \ldots, \mathrm{El}\ e_n) \to \mathrm{El}\ (d\ \vec{x})$, $t = d\ \vec{t}$ and $u_i \in \mathcal{E}^{(0)}_{e_i[\vec{t}]}$.
      ii. $u \in \mathcal{E}_t^{(0)}$ if $\mathsf{NEUTRAL}(u)$.
2. At level $\alpha + 1$:
   (a) $\mathcal{L}^{(\alpha+1)} = \mathcal{L}^{(\alpha)} \cup \{\Pi\ t_1\ t_2 \mid t_1 \in \mathcal{S}^{(\alpha)} \wedge \forall v.v \in \mathcal{E}_{t_1}^{(\alpha)} \Rightarrow t_2 v \in \mathcal{S}^{(\alpha)+}\}$
   (b) $\mathcal{S}^{(\alpha+1)} = \{e[t_1, \ldots, t_n] \mid t_1 \in \mathcal{L}^{(\alpha+1)}, \ldots, t_n \in \mathcal{L}^{(\alpha+1)}\}$
   (c) Given $t \in \mathcal{S}^{(\alpha+1)}$,
   the relation $u \in \mathcal{E}_t^{(\alpha+1)}$ is specified by induction on u.
      i. $c\ u_1\ \ldots\ u_n \in \mathcal{E}_t^{(\alpha+1)}$ iff $\mathsf{NF}(u_i)$,
      $\mathcal{C}(c) = (\mathrm{El}\ e_1, \ldots, \mathrm{El}\ e_n) \to \mathrm{El}\ (d\ \vec{x})$, $t = d\ \vec{t}$ and $u_i \in \mathcal{E}^{(\alpha+1)}_{e_i[\vec{t}]}$.
      ii. fun $u_1 \in \mathcal{E}_t^{(\alpha+1)}$ iff $\mathsf{NF}(u_1)$, $t \in \mathcal{L}^{(\alpha+1)}$, $t = \Pi\ t_1\ t_2$ and $\forall w.w \in \mathcal{E}_{t_1}^{(\alpha)} \Rightarrow u_1\ w \in \mathcal{E}_{t_2 w}^{(\alpha)+}$.
      iii. $u \in \mathcal{E}_t^{(\alpha+1)}$ if $\mathsf{NEUTRAL}(u)$.
3. At limit λ
   (a) $\mathcal{L}^{(\lambda)} = \bigcup_{\alpha<\lambda} \mathcal{L}^{(\alpha)}$
   (b) $\mathcal{S}^{(\lambda)} = \bigcup_{\alpha<\lambda} \mathcal{S}^{(\alpha)}$
   (c) $\mathcal{E}_t^{(\lambda)} = \bigcup_{\alpha<\lambda} \mathcal{E}_t^{(\alpha)}$

Proposition 3.2.4 (Existence of fixed point). *There exists an ordinal number $\alpha_0 < \omega_1$ such that*

1. $\mathcal{S}^{(\alpha_0)+} = \mathcal{S}^{(\alpha_0+1)+}$
2. $\forall t\ .\ t \in \mathcal{S}^{(\alpha_0)+} \Rightarrow \mathcal{E}_t^{(\alpha_0)+} = \mathcal{E}_t^{(\alpha_0+1)+}$

Motivation. We have an increasing sequence $\mathcal{S}^{(\alpha)+}$ of subsets of a given countable set, hence there exists $\alpha_0 < \omega_1$ such that $\mathcal{S}^{(\alpha_0)+} = \mathcal{S}^{(\alpha_0+1)+}$. The same holds for each $t \in \mathcal{S}^{(\alpha_0)+}$ for $\mathcal{E}_t^{(\alpha_0)+}$. □

We claim without proof the following equivalence between terms satisfying the predicates of Specification 3.1.5 and Definition 3.2.3.

Proposition 3.2.5 (Set interpretation of reducibility).

$$\mathsf{RED}_{\mathrm{Set}}(t) \iff t \in \mathcal{S}^{(\alpha_0)+} \qquad \mathsf{RED}_{(\mathrm{El}\ t)}(u) \iff t \in \mathcal{S}^{(\alpha_0)+} \ \wedge\ u \in \mathcal{E}_t^{(\alpha_0)+}$$

Example 3.2.6 (A lower bound). We do have $\alpha_0 > \omega$. Consider the constant[6]

$$\begin{aligned} &F : (n : \mathrm{El\ Nat}) \to \mathrm{Set} \\ &F\ 0 = \mathrm{Nat} \\ &F\ (\mathrm{s}\ n) = \Pi\ \mathrm{Nat}\ (\lambda\_.F\ n) \end{aligned}$$

There is no natural number m such that $\Pi\ \mathrm{Nat}\ (\lambda n.F\ n) \in \mathcal{S}^{(m)+}$, but one can show it belongs to $\mathcal{S}^{(\omega+1)+}$. We have $\mathcal{S}^{(\omega)+} \neq \mathcal{S}^{(\omega+1)+}$.

3.3 Normalization of reducible terms

Remark 3.3.1. From Specification 3.1.8 and the transitivity of $\bowtie$ (Corollary 2.2.33) we have that $T \bowtie T'$ and $t \bowtie t'$ implies $\mathsf{RED}(T) \Leftrightarrow \mathsf{RED}(T')$ and $\mathsf{RED}_T(t) \Leftrightarrow \mathsf{RED}_{T'}(t')$.

Lemma 3.3.2.

1. $\mathsf{RED}(\mathrm{Fun}\ U\ ((\lambda x.V)\ \gamma)) \iff$
   $\mathsf{RED}(U) \wedge \forall u.\mathsf{RED}_U(u) \Rightarrow \mathsf{RED}(V[\gamma, u/x])$.
2. $\mathsf{RED}(\mathrm{Fun}\ U\ ((\lambda x.V)\ \gamma)) \wedge \mathsf{RED}_{(\mathrm{Fun}\ U\ ((\lambda x.V)\ \gamma))}(t) \iff$
   $\mathsf{RED}(U) \wedge \forall u.\mathsf{RED}_U(u) \Rightarrow \mathsf{RED}(V[\gamma, u/x]) \wedge \mathsf{RED}_{(V[\gamma,u/x])}(t\ u)$.

Proof. By substitution properties 2.2.7 and Proposition 2.2.14. □

Proposition 3.3.3.
For all T, t, if $\mathsf{RED}(T)$ then

1. $\mathsf{RED}_T(t) \Rightarrow \mathsf{WN}(t)$
2. $t \Downarrow b \Rightarrow \mathsf{RED}_T(t)$

Proof. Assume $\mathsf{RED}(T)$. We prove 1 and 2 by induction on $rank(T)$.

1: Assume $\mathsf{RED}_T(t)$.

- When T is Set or El u,
  by Specification 3.1.5 we have $\mathsf{WN}(t)$ directly.
- When T is Fun U $(\lambda x.V)$,
  we have from Specification 3.1.8, case 3, that $\mathsf{RED}(\mathrm{Fun}\ U\ (\lambda x.V))$ holds by $\mathsf{RED}(U)$ and $\forall u.\mathsf{RED}_U(u) \Rightarrow \mathsf{RED}(V[u/x]))$. Then $\mathsf{RED}_{(\mathrm{Fun}\ U\ (\lambda x.V))}(t)$ holds whenever $\forall u.\mathsf{RED}_U(u) \Rightarrow \mathsf{RED}_{(V[u/x])}(t\ u)$. We have $\mathsf{NEUTRAL}(x)$. From $\mathsf{RED}(U)$ and induction (2) we have $\mathsf{RED}_U(x)$. Then $\mathsf{RED}(V[x/x])$ and $\mathsf{RED}_{(V[x/x])}(t\ x)$. From $\mathsf{RED}(V)$ and induction (1) we have $\mathsf{WN}(t\ x)$. By Lemma 2.2.40 we have $\mathsf{WN}(t)$.

[6]The symbol '_' denotes some variable not free in $F\ n$.

2: Assume $t \Downarrow b$.

- When T is Set or El u, from $\mathsf{RED}(T)$ and Specification 3.1.5, in all their sub-cases we have $\mathsf{RED}_T(t)$ directly.

- When T is Fun U $(\lambda x.V)$,
  we have from Specification 3.1.8, case 3, that $\mathsf{RED}(\text{Fun } U\ (\lambda x.V))$ holds by $\mathsf{RED}(U)$ and $\forall u.\mathsf{RED}_U(u) \Rightarrow \mathsf{RED}(V[u/x])$.
  We have to show $\forall u.\mathsf{RED}_U(u) \Rightarrow \mathsf{RED}_{(V[u/x])}(t\ u)$.
  Assume given u such that $\mathsf{RED}_U(u)$.
  Then we have $\mathsf{RED}(V[u/x])$. From $\mathsf{RED}(U)$ and induction (1) there exists u' such that $u \Downarrow u'$. By Lemma 3.1.4 we have $\mathsf{NEUTRAL}(b\ u')$.
  Then by induction (2) we get $\mathsf{RED}_{(V[u/x])}(b\ u')$.
  By Remark 3.3.1 we have $\mathsf{RED}_{(V[u/x])}(t\ u)$.

□

Corollary 3.3.4. $\mathsf{RED}(T) \Rightarrow \mathsf{WN}(T)$.

Proof. Similar as above, by induction on $rank(T)$. □

3.4 Properties of reducibility

3.4.1 Reducibility and vectors

Definition 3.4.1 (Reducibility for Vectors).

$$\frac{}{\mathsf{RED}_{()}()} \qquad \frac{\mathsf{RED}_\Gamma(\vec{t}) \quad \mathsf{RED}(T[\vec{t}]) \quad \mathsf{RED}_{(T[\vec{t}])}(t)}{\mathsf{RED}_{(\Gamma,x:T)}(\vec{t},t)}$$

Definition 3.4.2.
$\mathsf{RED}_{(x_1:T_1,\ \ldots\ ,x_n:T_n)}(\gamma)$ holds whenever $\forall i.\ \mathsf{RED}(T_i\gamma) \wedge \mathsf{RED}_{(T_i\gamma)}(\gamma(x_i))$.

Lemma 3.4.3. *If* Γ *is closed,* $\mathsf{RED}_\Gamma(\gamma)$, $\mathsf{RED}_{(T\ \gamma)}(t)$, $x \notin supp(\Gamma)$ *and* $x \notin FV(T)$, *then* $\mathsf{RED}_{(\Gamma,x:T)}([\gamma, t/x])$.

Proof. Assume given γ, Γ, x, t, T. Assume Γ is closed, $\mathsf{RED}_\Gamma(\gamma)$, $\mathsf{RED}_{(T\ \gamma)}(t)$, $x \notin supp(\Gamma)$ and $x \notin FV(T)$. Assume given $(y : U)$ in $(\Gamma, x : T)$.
We have two cases:

- $x = y$
  Then T is U, and $y[\gamma, t/x] = x[\gamma, t/x] = t$, and $T[\gamma, t/x] = T\ \gamma$ because x is not free in T. By assumption we have $\mathsf{RED}_{(T\ \gamma)}(t)$, so we have $\mathsf{RED}_{(T[\gamma,t/x])}(y[\gamma, t/x])$.

- $x \neq y$
  Then $y[\gamma, t/x] = y\ \gamma$. From $\mathsf{RED}_\Gamma(\gamma)$ we have $\mathsf{RED}_{(U\ \gamma)}(y\ \gamma)$.
  We have $U\ \gamma = U[\gamma, t/x]$, since Γ is closed, x is not free in U,
  and so we have $\mathsf{RED}_{(U[\gamma,t/x])}(y[\gamma, t/x])$.

□

Lemma 3.4.4. *If* Γ *is closed and disjoint, then*

$$\mathsf{RED}_\Gamma(\vec{t}) \iff \mathsf{RED}_\Gamma([\vec{t}])$$

Proof. In both directions by induction on the length of Γ, using that Γ is closed and disjoint. □

Lemma 3.4.5.

1. $\mathsf{RED}(\Gamma \to T) \iff \forall \vec{t}.\mathsf{RED}_\Gamma(\vec{t}) \Rightarrow \mathsf{RED}(T[\vec{t}])$.
2. $\mathsf{RED}(\Gamma \to T) \wedge \mathsf{RED}_{(\Gamma \to T)}(t) \iff$
   $\forall \vec{t}.\mathsf{RED}_\Gamma(\vec{t}) \Rightarrow \mathsf{RED}(T[\vec{t}]) \wedge \mathsf{RED}_{(T[\vec{t}])}(t\ \vec{t})$.

Proof. By iteration of Specification 3.1.8 (Reducibility). □

3.4.2 Reducibility and sets

The Cartesian product of a family of sets

Lemma 3.4.6. *If* $x \notin FV(u)$,
$\mathsf{RED}(\text{Fun (El } t) ((\lambda x.\text{El } (u\ x))\gamma))$ *and* $\mathsf{RED}_{(\text{Fun (El } t) ((\lambda x.\text{El } (u\ x))\gamma))}(v)$ *then* $\mathsf{RED}_{(\text{El } \Pi\ t\ (u\ \gamma))}(\text{fun } v)$.

Proof. Assume $x \notin FV(u)$ and

$$\mathsf{RED}(\text{Fun (El } t) ((\lambda x.\text{El } (u\ x))\gamma)) \tag{3.1}$$
$$\mathsf{RED}_{(\text{Fun (El } t) ((\lambda x.\text{El } (u\ x))\gamma))}(v) \tag{3.2}$$

From (3.1), (3.2) and Lemma 3.3.2 we have

$$\mathsf{RED}(\text{El } t) \quad \text{and} \quad \forall w.\mathsf{RED}_{(\text{El } t)}(w) \Rightarrow \mathsf{RED}(\text{El } (u\ x)[\gamma, w/x]) \tag{3.3}$$

and

$$\forall w.\mathsf{RED}_{(\text{El } t)}(w) \Rightarrow \mathsf{RED}_{(\text{El } (u\ x)[\gamma, w/x])}(v\ w) \tag{3.4}$$

From (3.3) we have

$$\mathsf{RED}_{\text{Set}}(t) \quad \text{and} \quad \forall w.\mathsf{RED}_{(\text{El } t)}(w) \Rightarrow \mathsf{RED}_{\text{Set}}((u\ x)[\gamma, w/x]) \tag{3.5}$$

Since $x \notin FV(u)$, for any w we have

$$(u\ x)[\gamma, w/x] = (u\ \gamma)\ w \tag{3.6}$$

hence from (3.5) we get

$$\mathsf{RED}_{\text{Set}}(t) \quad \text{and} \quad \forall w.\mathsf{RED}_{(\text{El } t)}(w) \Rightarrow \mathsf{RED}_{\text{Set}}((u\ \gamma)\ w)$$

which, by Specification 3.1.8 gives

$$\mathsf{RED}(\text{El } \Pi \ t \ (u \ \gamma)) \tag{3.7}$$

Assume given w such that $\mathsf{RED}_{(\text{El } t)}(w)$.
From (3.4) we have

$$\mathsf{RED}_{(\text{El } (u \ x)[\gamma, w/x])}(v \ w) \tag{3.8}$$

From (3.2) and Proposition 3.3.3 there is v' such that $v \Downarrow v'$. From (3.8), Remark 3.3.1 and (3.6) we get

$$\forall w.\mathsf{RED}_{(\text{El } t)}(w) \Rightarrow \mathsf{RED}_{(\text{El } (u \ \gamma) \ w)}(v' \ w) \tag{3.9}$$

From (3.7) and (3.9), by Specification 3.1.8 gives $\mathsf{RED}_{(\text{El } \Pi \ t \ (u \ \gamma))}(\text{fun } v)$. □

Lemma 3.4.7.
$\mathsf{RED}(\text{El } (\Pi \ t \ u)) \ \wedge \ \mathsf{RED}_{(\text{El } (\Pi \ t \ u))}(\text{fun } v) \ \Rightarrow \ \mathsf{RED}((x : \text{El } t) \rightarrow \text{El } (u \ x)) \ \wedge \ \mathsf{RED}_{((x:\text{El } t) \rightarrow \text{El } (u \ x))}(v)$
where $x \notin FV(u)$.

Proof. By unfolding Specification 3.1.8. □

Parameterized data types

Lemma 3.4.8. *If* $FV(e) \subseteq \{x_1, \ \ldots \ , x_n\}$ *and* $\mathsf{RED}_{\text{Set}}(t_1), \ \ldots \ , \mathsf{RED}_{\text{Set}}(t_n)$, *then* $\mathsf{RED}_{\text{Set}}(e[t_1, \ \ldots \ , t_n])$.

Proof. By induction on e. □

Lemma 3.4.9.
$\mathsf{RED}(\text{El } (d \ \vec{t})) \ \wedge \ \mathsf{RED}_{(\text{El } (d \ \vec{t}))}(c \ \vec{u}) \ \Rightarrow \ \mathsf{RED}(\text{El } (e_i[\vec{t}])) \ \wedge \ \mathsf{RED}_{(\text{El } (e_i[\vec{t}]))}(u_i)$
where $\mathcal{C}(c) = (\text{El } e_1, \ \ldots \ , \text{El } e_n) \rightarrow \text{El } (d \ \vec{x})$.

Proof. By Lemma 3.4.8. □

3.4.3 Reducibility and the signature

When we refer to reducibility, this is done relative to a given signature. When we refer to the reducibility of a certain part of the signature itself, we will assume given a data type specification $\mathcal{D}, \mathcal{C}$, where all constructor types are already known to be reducible. What may change according to what part of the signature we consider, is the typing specifications $\mathcal{F}$ for defined constants, and the set of rules $\mathcal{R}$, that the notion of reducibility depends of.

Definition 3.4.10 (Reducibility conditions for $\mathcal{F}$).
Let $\mathsf{RED}(\mathcal{F})$ be the property $\forall f.\mathsf{RED}(\mathcal{F}(f))$.

Notation 3.4.11. When we want to make explicit what reduction rules we refer to in an assertion of the form $\mathsf{RED}(\mathcal{F})$, $\mathsf{RED}(T)$ or $\mathsf{RED}_T(t)$ we write $\mathsf{RED}^{\mathcal{R}}(\mathcal{F})$, $\mathsf{RED}^{\mathcal{R}}(T)$ and $\mathsf{RED}^{\mathcal{R}}_T(t)$ respectively.

Definition 3.4.12 (Reducible signature). Let $\Sigma = (\mathcal{D}, \mathcal{C}, \mathcal{F}, \mathcal{R})$.
$\mathsf{RED}(\Sigma)$ holds iff $\mathsf{RED}^{\mathcal{R}}(\mathcal{F})$ and for all f such that $f \in \mathcal{F}$, we have $\mathsf{RED}^{\mathcal{R}}_{(\mathcal{F}(f))}(f)$.

Note that the latter definition allows the presence of rules in $\mathcal{R}$, that have no type given in $\mathcal{F}$. We will exploit this freedom in the proofs of Lemma 3.6.6 (Key lemma), page 82, and Theorem 5.2.2 (A procedure for type-checking the signature), page 98.

Lemma 3.4.13. *If* $\mathsf{RED}(\mathcal{F}(f))$ *holds, and* f *has no rule in* $\mathcal{R}$*, then* $\mathsf{RED}_{(\mathcal{F}(f))}(f)$.

Proof. Given $f \in \mathcal{F}$, let $\mathcal{F}(f) = \Gamma_f \to T_f$. Assume f has no rule in $\mathcal{R}$. Assume given $\vec{t}$ such that $\mathsf{RED}_{\Gamma_f}(\vec{t})$. By $\mathsf{RED}(\mathcal{F}(f))$ and Lemma 3.4.5 we have $\mathsf{RED}(T_f[\vec{t}])$. By Proposition 3.3.3 (1) then $\vec{t} \Downarrow \vec{u}$ for some $\vec{u}$. Since f has no rule in $\mathcal{R}$, we have then $\mathsf{NEUTRAL}(f\ \vec{u})$ (Definition 3.1.1). By Proposition 3.3.3 (2) then $\mathsf{RED}_{T_f[\vec{u}]}(f\ \vec{u})$. By Remark 3.3.1 then $\mathsf{RED}_{T_f[\vec{t}]}(f\ \vec{t})$. □

3.5 Reducibility of well-typed terms

Lemma 3.5.1 (Reducibility of well-typed terms).
If $\forall f \in \mathcal{F}.\mathsf{RED}^{\mathcal{R}}(\mathcal{F}(f)) \wedge \mathsf{RED}^{\mathcal{R}}_{(\mathcal{F}(f))}(f)$ *then*

1. $\Gamma \vdash T \Rightarrow \forall \gamma\ .\ \mathsf{RED}_\Gamma(\gamma) \Rightarrow \mathsf{RED}(T\ \gamma)$.
2. $\Gamma \vdash t : T \Rightarrow \forall \gamma\ .\ \mathsf{RED}_\Gamma(\gamma) \Rightarrow \mathsf{RED}(T\ \gamma) \wedge \mathsf{RED}_{(T\ \gamma)}(t\ \gamma)$.

Proof. We prove 1 and 2 simultaneously by induction on the typing derivations.

1. Assume $\Gamma \vdash T$. Assume given γ such that $\mathsf{RED}_\Gamma(\gamma)$.
   We have the following cases:

   (a) $\dfrac{\vdash \Gamma}{\Gamma \vdash \mathrm{Set}}$
   We have $\mathsf{RED}(\mathrm{Set})$ directly.

   (b) $\dfrac{\Gamma \vdash t : \mathrm{Set}}{\Gamma \vdash \mathrm{El}\ t}$
   By induction (2) we have $\mathsf{RED}_{\mathrm{Set}}(t\ \gamma)$. By Specification 3.1.8, case 2 we have $\mathsf{RED}(\mathrm{El}\ t\ \gamma)$.

   (c) $\dfrac{\Gamma, x : U \vdash V}{\Gamma \vdash (x : U) \to V}$
   By Corollary 2.3.7, the derivation of $\Gamma, x : U \vdash V$ contains a subderivation of $\Gamma \vdash U$. By induction (1) we have $\mathsf{RED}(U\ \gamma)$.
   Assume given u such that $\mathsf{RED}_{(U\ \gamma)}(u)$.
   From Lemma 2.3.6 and $\Gamma, x : U \vdash V$ we have $\vdash \Gamma, x : U$ and then $x \notin FV(U)$. By Lemma 3.4.3 we have $\mathsf{RED}_{(\Gamma, x:U)}([\gamma, u/x])$.
   By induction (1) then $\mathsf{RED}(V[\gamma, u/x])$.
   By Lemma 3.3.2 we have $\mathsf{RED}(\mathrm{Fun}\ (U\ \gamma)\ ((\lambda x.V)\gamma))$, which is equivalent to $\mathsf{RED}((\mathrm{Fun}\ U\ (\lambda x.V))\ \gamma)$.

2. Assume $\Gamma \vdash t : T$. Assume given γ such that $\mathsf{RED}_\Gamma(\gamma)$.
   We have the following cases:

   (a) $\dfrac{\vdash \Gamma}{\Gamma \vdash x : \Gamma(x)}$

   $\mathsf{RED}_{(x_1:T_1,\ \ldots\ ,x_n:T_n)}(\gamma)$ implies
   $\mathsf{RED}(T_k[x_1,\ \ldots\ , x_{k-1}]\gamma)$ and $\mathsf{RED}_{(T_k[x_1,\ \ldots\ ,x_{k-1}]\gamma)}(\gamma(x_k))$,
   with $x = x_k$ in $supp(\Gamma)$, which is equivalent to
   $\mathsf{RED}(\Gamma(x)\gamma)$ and $\mathsf{RED}_{(\Gamma(x)\gamma)}(\gamma(x))$.

   (b) $\dfrac{\Gamma \vdash t : U \quad \Gamma \vdash T}{\Gamma \vdash t : T}\ U \bowtie T$

   By induction (2) we have $\mathsf{RED}(U\ \gamma)$ and $\mathsf{RED}_{(U\ \gamma)}(t\ \gamma)$. From $U \bowtie T$ we have $U\gamma \bowtie T\gamma$, by Remark 3.3.1 we have $\mathsf{RED}_{(T\ \gamma)}(t\ \gamma)$.

   (c) $\dfrac{\Gamma \vdash t : (x : U) \to V \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : V[u/x]}$

   By induction (2) we have $\mathsf{RED}(((x : U) \to V)\gamma)$ and $\mathsf{RED}_{(((x:U)\to V)\gamma)}(t\ \gamma)$. By Lemma 3.3.2 we have $\mathsf{RED}(U\ \gamma)$ and $\forall v.\mathsf{RED}_{(U\ \gamma)}(v) \Rightarrow \mathsf{RED}(V[\gamma, v/x]) \wedge \mathsf{RED}_{(V[\gamma,v/x])}((t\ \gamma)\ v)$.
   By induction (2) we have $\mathsf{RED}_{(U\ \gamma)}(u\ \gamma)$. From above then $\mathsf{RED}(V[\gamma, (u\ \gamma)/x])$ and $\mathsf{RED}_{(V[\gamma,(u\ \gamma)/x])}((t\ \gamma)\ (u\ \gamma))$.
   By the substitution laws we have $\mathsf{RED}(V[u/x]\gamma)$ and $\mathsf{RED}_{(V[u/x]\gamma)}((t\ u)\gamma)$.

   (d) $\dfrac{\Gamma, x : U \vdash v : V}{\Gamma \vdash \lambda x.v : (x : U) \to V}$

   By Corollary 2.3.7, $\Gamma, x : U \vdash V$ contains a sub-derivation of $\Gamma \vdash U$.
   By induction (1) we have $\mathsf{RED}(U\ \gamma)$. Assume given u such that $\mathsf{RED}_{(U\ \gamma)}(u)$. From $\Gamma, x : U \vdash v : V$ and Lemma 2.3.6 we have $\vdash \Gamma, x : U$, hence $x \notin FV(U)$. By Lemma 3.4.3 we have $\mathsf{RED}_{(\Gamma,x:U)}([\gamma, u/x])$. By induction (2) then $\mathsf{RED}(V[\gamma, u/x])$ and $\mathsf{RED}_{(V[\gamma,u/x])}(v[\gamma, u/x]))$.
   By Proposition 2.2.14, $((\lambda x.v)\gamma)\ u \rightsquigarrow_\beta v[\gamma, u/x]$ and by Remark 3.3.1 we have $\mathsf{RED}_{(V[\gamma,u/x])}(((\lambda x.v)\gamma)\ u)$.
   By Lemma 3.3.2 we have $\mathsf{RED}_{(\mathrm{Fun}\ (U\ \gamma)\ ((\lambda x.V)\gamma))}((\lambda x.v)\ \gamma)$, which is equivalent to $\mathsf{RED}_{((\mathrm{Fun}\ U\ (\lambda x.V))\ \gamma)}((\lambda x.v)\ \gamma)$.

   (e) $\dfrac{\vdash \Gamma}{\Gamma \vdash f : \mathcal{F}(f)}$

   By assumption we have $\mathsf{RED}(\mathcal{F}(f))$ and $\mathsf{RED}_{(\mathcal{F}(f))}(f)$. Since $\mathcal{F}(f)$ is closed we have $\mathsf{RED}(\mathcal{F}(f)\ \gamma)$ and $\mathsf{RED}_{(\mathcal{F}(f)\ \gamma)}(f\ \gamma)$.

(f) $$\frac{\vdash \Gamma}{\Gamma \vdash d : \mathcal{D}(d)}$$

The type $\mathcal{D}(d)$ is of the form $\mathrm{Set}^n \to \mathrm{Set}$. Assume given $\vec{t}$ such that $\mathsf{RED}_{\mathrm{Set}}(t_i)$. By Specification 3.1.5, case 1, we have $\mathsf{WN}(t_i)$. By Specification 3.1.5, case 1 we have $\mathsf{RED}_{\mathrm{Set}}(d\ t_1\ \ldots\ t_n)$. Iterating Specification 3.1.8, case 3 n times, then $\mathsf{RED}_{(\mathrm{Set}^n \to \mathrm{Set})}(d)$, and so $\mathsf{RED}_{((\mathrm{Set}^n \to \mathrm{Set})\ \gamma)}(d\ \gamma)$.

(g) $$\frac{\vdash \Gamma \quad \Gamma \vdash u_1 : \mathrm{Set} \quad \ldots \quad \Gamma \vdash u_k : \mathrm{Set}}{\Gamma \vdash c : \mathcal{C}(c)[u_1,\ \ldots\ , u_k]}$$

By induction (2) we have $\mathsf{RED}_{\mathrm{Set}}(u_i\ \gamma)$.

$\mathcal{C}(c)[u_1,\ \ldots\ , u_k]\gamma$ is an independent function type of the form $(\mathrm{El}\, e_1[u_1,\ \ldots\ , u_k]\gamma, \ldots, \mathrm{El}\, e_n[u_1,\ \ldots\ , u_k]\gamma) \to \mathrm{El}\, ((d\ u_1\ \ldots\ u_k)\ \gamma)$.

From $\mathsf{RED}_{\mathrm{Set}}(u_i\gamma)$ and Lemma 3.4.8 we have $\mathsf{RED}(\mathrm{El}\, e_j[u_1,\ \ldots\ , u_k]\gamma)$.

From $\mathsf{RED}_{\mathrm{Set}}(u_i\gamma)$ and Specification 3.1.5, case 1a we have $\mathsf{RED}_{\mathrm{Set}}((d\ u_1\ \ldots\ u_k)\gamma)$.

By Specification 3.1.5, case 1a we have $(d\ u_1\ \ldots\ u_k)\gamma \Downarrow d\ v_1\ \ldots\ v_k$ where $\mathsf{RED}_{\mathrm{Set}}(v_j)$ holds.

By Specification 3.1.5, case 2 we have $\mathsf{RED}((\mathrm{El}\ d\ u_1\ \ldots\ u_k)\gamma)$.

Assume given $t_1,\ \ldots\ , t_n$ such that $\mathsf{RED}_{(\mathrm{El}\ e_i[u_1,\ \ldots\ ,u_k]\gamma)}(t_i)$. By Proposition 3.3.3 (1), we have $\mathsf{WN}(t_i)$, hence $c\ t_1\ \ldots\ t_n \Downarrow c\ t'_1\ \ldots\ t'_n$. By Remark 3.3.1 we have $\mathsf{RED}_{(\mathrm{El}\ e_i[u_1,\ \ldots\ ,u_k]\gamma)}(t'_i)$.

We have satisfied case 2(a)i of Specification 3.1.5, and then $\mathsf{RED}_{(\mathrm{El}\ (d\ u_1\ \ldots\ u_k)\gamma)}(c\ t_1\ \ldots\ t_n)$. By applying case 3 of Specification 3.1.8 n times we get $\mathsf{RED}_{(\mathcal{C}(c)[u_1,\ \ldots\ ,u_k]\gamma)}(c\ \gamma)$.

(h) $$\frac{\vdash \Gamma}{\Gamma \vdash \Pi : (x : \mathrm{Set},\ \mathrm{El}\ x \to \mathrm{Set}) \to \mathrm{Set}}$$

Assume given t such that $\mathsf{RED}_{\mathrm{Set}}(t)$.
It is straight-forward to see that $\mathsf{RED}((x : \mathrm{El}\ t) \to \mathrm{Set})$ holds.
Assume given u such that $\mathsf{RED}_{((x:\mathrm{El}\ t)\to\mathrm{Set})}(u)$ holds.

By Proposition 3.3.3 (1), we have $\mathsf{WN}(t)$ and $\mathsf{WN}(u)$.
By Specification 3.1.8, case 1b we get $\mathsf{RED}_{\mathrm{Set}}(\Pi\ t\ u)$.

We conclude $\mathsf{RED}((x : \mathrm{Set},\ \mathrm{El}\ x \to \mathrm{Set}) \to \mathrm{Set})$ and $\mathsf{RED}_{((x:\mathrm{Set},\ \mathrm{El}\ x\to\mathrm{Set})\to\mathrm{Set})}(\Pi)$. Since these terms are closed, we have $\mathsf{RED}(((x : \mathrm{Set},\ \mathrm{El}\ x \to \mathrm{Set}) \to \mathrm{Set})\gamma)$ and $\mathsf{RED}_{(((x:\mathrm{Set},\ \mathrm{El}\ x\to\mathrm{Set})\to\mathrm{Set})\gamma)}(\Pi\ \gamma)$.

(i) $$\frac{\Gamma \vdash t : \mathrm{Set} \quad \Gamma \vdash u : \mathrm{El}\ t \to \mathrm{Set}}{\Gamma \vdash \mathrm{fun}\ : ((x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)) \to \mathrm{El}\ (\Pi\ t\ u)}\ x \notin FV(u)$$

The type in the conclusion is an independent function type with its domain being a dependent type. With the domain written in Fun-

notation our goal becomes

$$\mathsf{RED}(((\text{Fun } (\text{El } t)\ (\lambda x.\text{El } (u\ x))) \to \text{El } (\Pi\ t\ u))\ \gamma)$$
$$\mathsf{RED}_{(((\text{Fun } (\text{El } t)\ (\lambda x.\text{El } (u\ x))) \to \text{El } (\Pi\ t\ u))\ \gamma)}(\text{fun } \gamma).$$

Substituting γ into the sub-expressions we get the goal

$$\mathsf{RED}((\text{Fun } (\text{El } t\ \gamma)\ ((\lambda x.\text{El } (u\ x))\gamma)) \to \text{El } (\Pi\ (t\ \gamma)(u\ \gamma))) \quad (3.10)$$
$$\mathsf{RED}_{((\text{Fun } (\text{El } t\ \gamma)\ ((\lambda x.\text{El } (u\ x))\gamma)) \to \text{El } (\Pi\ (t\ \gamma)(u\ \gamma)))}(\text{fun}). \quad (3.11)$$

By induction (2) for $\Gamma \vdash t : \text{Set}$ we have $\mathsf{RED}_{\text{Set}}(t\ \gamma)$.
By induction (2) for $\Gamma \vdash u : \text{El } t \to \text{Set}$ we have $\mathsf{RED}(\text{El } t\gamma \to \text{Set})$ and $\mathsf{RED}_{(\text{El } t\gamma \to \text{Set})}(u\ \gamma)$. By Specification 3.1.8, and $x \notin FV(u)$ then
$\mathsf{RED}(\text{Fun } (\text{El } t\ \gamma)\ ((\lambda x.\text{El } (u\ x))\gamma))$ holds. Assume given v such that $\mathsf{RED}_{(\text{Fun } (\text{El } t\ \gamma)\ ((\lambda x.\text{El } (u\ x))\gamma))}(v)$ holds. From $x \notin FV(u)$ and Lemma 3.4.6 we have $\mathsf{RED}_{(\text{El } (\Pi\ (t\ \gamma)(u\ \gamma)))}(\text{fun } v)$, then (3.10) and (3.11) follows.

□

3.6 Reducibility of defined constants

3.6.1 Call relation

Notation 3.6.1 *(Sub-term).*
We will write $u \trianglelefteq v$ for u being a sub-term of v or $u = v$.

Definition 3.6.2 (Formal call).
$(f, (p_1, \ \dots\ , p_m)) \succ (g, (u_1, \ \dots\ , u_n))$ holds whenever there is a rule $f\ p_1\ \dots\ p_n = s \in \mathcal{R}$, $ar(f) = m$, $ar(g) = n$, and $g\ u_1\ \dots\ u_n \trianglelefteq s$.

Note that in the definition above, u_i may contain free variables other than those in $\vec{p}$, since s is not necessarily first-order.

Definition 3.6.3 (Call instance).
$(f, (t_1, \ \dots\ , t_m)) \overset{\sim}{\succ} (g, (u_1\gamma, ..., u_n\gamma))$ holds whenever
$t_1 \rightsquigarrow^* p_1\ \gamma, \ \dots\ , t_m \rightsquigarrow^* p_m\ \gamma$, $\mathsf{WN}(t_i)$ and $(f, (p_1, \ \dots\ , p_m)) \succ (g, (u_1, \ \dots\ , u_n))$.

3.6.2 Reducibility and neighbourhoods

Lemma 3.6.4. $\mathsf{RED}_\Gamma(\alpha\gamma)\ \wedge\ \Delta \xrightarrow{\alpha} \Gamma\ \Rightarrow\ \mathsf{RED}_\Delta(\gamma)$

Proof. Assume $\Delta \xrightarrow{\alpha} \Gamma$ and $\mathsf{RED}_\Gamma(\alpha\gamma)$. Let $\Gamma = (x_1 : T_1, \ \dots\ , x_n : T_n)$. For $i \in \{1, \ \dots\ , n\}$ we have $\mathsf{RED}(T_i\alpha\gamma)$ and $\mathsf{RED}_{(T_i\alpha\gamma)}(\alpha\gamma(x_i))$. We have α of the form $[y/x]$, $[\text{fun}\ \ y/x_k]$ or $[c\ y_1\ \dots\ y_m/x_k]$ for some $k \in \{1, \ \dots\ , n\}$. We have Γ of the form $(\Gamma_1, x_k : T_k, \ \Gamma_2)$, and $\Delta = (\Gamma_1, \Theta, \Gamma_2\alpha)$, where Θ depends on α and T_k. We verify $\mathsf{RED}_\Delta(\gamma)$ for each of the three parts of which Δ is constructed.

1. For $x_1, \ldots, x_{k-1}$, to show $\mathsf{RED}(T_i\gamma)$ and $\mathsf{RED}_{(T_i\gamma)}(\gamma(x_i))$:

   Since Γ is closed we have $FV(T_i) \subseteq \{x_1, \ldots, x_{i-1}\}$, and then $T_i\alpha\gamma = T_i\gamma$. We have $(\alpha\gamma)(x_i) = \gamma(x_i)$, and the goal then follows from $\mathsf{RED}(T_i\alpha\gamma)$ and $\mathsf{RED}_{(T_i\alpha\gamma)}((\alpha\gamma)(x_i))$, known by assumption.

2. For Θ, dependent on α and T_k, α being of the form

   (a) $[y/x_k]$.
   Direct from Specification 3.1.8 (Reducibility).

   (b) $[\text{fun } y/x_k]$, and $T_k \bowtie \text{El } (\Pi\ t\ u)$,
   with $\Theta = (y : (z : \text{El } t) \to \text{El } (u\ z))$, $z \notin FV(u)$, to show
   $\mathsf{RED}(((z : \text{El } t) \to \text{El } (u\ z))\gamma)$ and $\mathsf{RED}_{(((z:\text{El } t)\to\text{El } (u\ z))\gamma)}(\gamma(y))$:
   We have $(\alpha\gamma)(x_k) = \text{fun}(\gamma(y))$.
   By assumption we have $\mathsf{RED}(T_k\alpha\gamma)$ and $\mathsf{RED}_{(T_k\alpha\gamma)}((\alpha\gamma)(x_k))$.
   By Remark 2.3.12 we have $x_k \notin FV(T_k) \cup FV(\Pi\ t\ u)$, and we have $T_k\alpha\gamma = T_k\gamma$ and then $(\Pi\ t\ u)\alpha\gamma = (\Pi\ t\ u)\gamma$. From $T_k \bowtie \text{El } (\Pi\ t\ u)$ then $T_k\alpha\gamma \bowtie \text{El } (\Pi\ t\ u)\gamma$. By Remark 3.3.1 we get $\mathsf{RED}(\text{El } (\Pi\ t\ u)\gamma)$ and $\mathsf{RED}_{(\text{El } (\Pi\ t\ u)\gamma)}(\text{fun}(\gamma(y)))$.
   We have $z \notin FV(u)$ and then by Lemma 3.4.7 we have
   $\mathsf{RED}(((z : \text{El } t) \to \text{El } (u\ z))\gamma)$ and $\mathsf{RED}_{(((z:\text{El } t)\to\text{El } (u\ z))\gamma)}(\gamma(y))$.

   (c) $[c\ y_1\ \ldots\ y_m/x_k]$, and $T_k \bowtie \text{El } (d\ \vec{u}\,)$,
   with $\mathcal{C}(c) = (\text{El } e_1, \ldots, \text{El } e_m) \to d\ \vec{z}$ and
   $\Theta = (y_1 : \text{El } e_1[\vec{u}], \ldots, y_m : \text{El } e_m[\vec{u}])$,
   for $y_1, \ldots, y_m$, to show $\mathsf{RED}(\text{El } (e_j[\vec{u}]\gamma))$ and $\mathsf{RED}_{(\text{El } (e_j[\vec{u}]\gamma))}(\gamma(y_j))$:
   By assumption we have $\mathsf{RED}(T_k\alpha\gamma)$ and $\mathsf{RED}_{(T_k\alpha\gamma)}((\alpha\gamma)(x_k))$.
   We have $(\alpha\gamma)(x_k) = [c\ y_1\ \ldots\ y_m/x_k]\gamma(x_k) = c\ \gamma(y_1), \ldots, \gamma(y_m)$.
   From $T_k \bowtie \text{El } (d\ \vec{u}\,)$ and from $x_k \notin FV(T_k) \cup FV(\text{El } (d\ \vec{u}\,))$ we have $\text{El } (d\ \vec{u}\,\alpha\gamma) = \text{El } (d\ \vec{u}\,\gamma)$, and then $T_k\alpha\gamma \bowtie \text{El } (d\ \vec{u}\,\gamma)$. By Remark 3.3.1 then $\mathsf{RED}(\text{El } (d\ \vec{u}\,\gamma))$ and $\mathsf{RED}_{(\text{El } (d\ \vec{u}\,\gamma))}(c\ \gamma(y_1), \ldots, \gamma(y_m))$ holds. By Lemma 3.4.9 then $\mathsf{RED}(\text{El } (e_j[\vec{u}]\gamma))$ and $\mathsf{RED}_{(\text{El } (e_j[\vec{u}]\gamma))}(\gamma(y_j))$.

3. For $x_{k+1}, \ldots, x_n$, to show $\mathsf{RED}((T_i\alpha)\gamma)$ and $\mathsf{RED}_{((T_i\alpha)\gamma)}(\gamma(x_i))$:

   We have $(\alpha\gamma)(x_i) = \gamma(x_i)$, and the goal then follows from $\mathsf{RED}(T_i\alpha\gamma)$ and $\mathsf{RED}_{(T_i\alpha\gamma)}((\alpha\gamma)(x_i))$, known by assumption.

□

Lemma 3.6.5. $\mathsf{RED}_\Gamma(\tau\gamma)\ \wedge\ \Delta \xrightarrow{\tau} \Gamma\ \Rightarrow\ \mathsf{RED}_\Delta(\gamma)$

Proof. By induction on the derivation of $\Delta \xrightarrow{\tau} \Gamma$ and Lemma 3.6.4. □

3.6.3 Proof of reducibility for defined constants

Lemma 3.6.6 (Key lemma). *If* $\overset{\sim}{\succ}$ *is well-founded,* $\mathsf{RED}(\mathcal{F})$ *and* $\vdash \Sigma$, *then* $\forall f.\mathsf{RED}_{(\mathcal{F}(f))}(f)$.

Proof. Assume $\overset{\sim}{\succ}$ is well-founded, $\mathsf{RED}(\mathcal{F})$ and $\vdash \Sigma$.
Note that since $\mathsf{RED}(\mathcal{F})$ holds, for any $\mathcal{F}(f) = \Gamma_f \to T_f$ we have $\mathsf{RED}(\Gamma_f \to T_f)$. Then by Lemma 3.4.5 we have $\mathsf{RED}(T_f[\vec{t}])$ for any $\vec{t}$ such that $\mathsf{RED}_{\Gamma_f}(\vec{t})$. Let $\Phi(f, \vec{t})$ be the property $\mathsf{RED}_{\Gamma_f}(\vec{t}) \Rightarrow \mathsf{RED}_{(T_f[\vec{t}])}(f\ \vec{t})$.

We will show

$$\forall f.\forall \vec{t}\,.(\forall g.\forall \vec{u}\,.(f, \vec{t}) \overset{\sim}{\succ} (g, \vec{u}) \Rightarrow \Phi(g, \vec{u})) \ \Rightarrow\ \Phi(f, \vec{t}) \tag{3.12}$$

which, by the principle of well-founded induction implies $\forall f.\forall \vec{t}.\ \Phi(f, \vec{t})$.
By Lemma 3.4.5, then $\forall f.\mathsf{RED}_{(\mathcal{F}(f))}(f)$ follows.

Proof of (3.12):
Assume given f, $\vec{t}$ with $|\vec{t}\,| = ar(f)$ and $\mathcal{F}(f) = \Gamma_f \to T_f$.
Assume

$$\forall g.\forall \vec{u}\,.(f, \vec{t}) \overset{\sim}{\succ} (g, \vec{u}) \Rightarrow \Phi(g, \vec{u}) \tag{3.13}$$

From $\mathsf{RED}(\mathcal{F})$ we have $\mathsf{RED}(\Gamma_f \to T_f)$. Assume $\mathsf{RED}_{\Gamma_f}(\vec{t})$.
By Lemma 3.4.5 we have $\mathsf{RED}(T_f[\vec{t}])$.
From $\mathsf{RED}_{\Gamma_f}(\vec{t})$ and Proposition 3.3.3 (1) there exists $\vec{u}$ such that $\vec{t} \Downarrow \vec{u}$.
If $f\ \vec{u}$ is normal, then since $ar(f) = |\vec{u}\,|$, by Definition 3.1.1, $f\ \vec{u}$ is neutral, and from Proposition 3.3.3 (2) we have $\mathsf{RED}_{(T_f[\vec{t}])}(f\ \vec{u})$, which, by Remark 3.3.1 implies $\mathsf{RED}_{(T_f[\vec{t}])}(f\ \vec{t})$.
Otherwise, since $\vec{u}$ is normal, by Definition 2.2.12, $f\ \vec{u}$ is a ι-redex of the form $f\ \vec{p}\gamma$ where, by Lemma 2.2.15 we get

$$\vec{u} = \vec{p}\gamma, \qquad f\ \vec{p} = s_0 \in \mathcal{R} \qquad \text{and} \qquad f\ \vec{p}\gamma \rightsquigarrow^* s_0\ \gamma \tag{3.14}$$

In this case, by Remark 3.3.1, $\mathsf{RED}_{(T_f[\vec{t}])}(f\ \vec{t})$ follows from $\mathsf{RED}_{(T_f[\vec{t}])}(s_0\ \gamma)$, that we will prove below. From the two cases above we can conclude $\Phi(f, \vec{t})$.

Proof of $\mathsf{RED}_{(T_f[\vec{t}])}(s_0\ \gamma)$:

From $\vdash \Sigma$ we have

$$\Delta_0 \xrightarrow{[\vec{p}]} \Gamma_f \tag{3.15}$$

and

$$\Delta_0 \vdash s_0 : T_f[\vec{p}]. \tag{3.16}$$

From (3.16) and Corollary 2.5.6 (Completeness) we have

$$\Delta_0 \vdash s_0 \uparrow T_f[\vec{p}]. \tag{3.17}$$

We will show the following property[7] about the sub-terms s of s_0, by induction on s.

$$\begin{array}{l}(\forall\Theta)(\forall U)(\forall\sigma)\ .\\ \quad\big(\ s \trianglelefteq s_0\ \wedge\ \Theta \text{ extends } \Delta_0\ \wedge\ \Theta \vdash s \uparrow U\ \wedge\\ \quad\ \ \sigma_{|\Delta_0} = \gamma_{|\Delta_0}\ \wedge\ \mathsf{RED}_\Theta(\sigma)\ \wedge\ \mathsf{RED}(U\sigma)\ \big) \Rightarrow\\ \quad \mathsf{RED}_{(U\sigma)}(s\ \sigma)\end{array} \tag{3.18}$$

Once (3.18) is proved, we can finish our argument as follows:
Recall from previously that we have $\mathsf{RED}_{\Gamma_f}(\vec{t})$ and $\mathsf{RED}(T_f[\vec{t}])$.
From $\vec{t} \Downarrow \vec{p}\,\gamma$ and Remark 3.3.1 then $\mathsf{RED}_{\Gamma_f}([\vec{p}\,\gamma])$ and $\mathsf{RED}(T_f[\vec{p}\,\gamma])$.
Since $\mathcal{F}(f)$ is closed we have for $\Gamma_f = (x_1 : T_1,\ \ldots\ , x_n : T_n)$ that
$T_i[p_1\gamma,\ \ldots\ , p_{i-1}\gamma] = T_i[p_1,\ \ldots\ , p_{i-1}]\gamma$ and
$T_f[p_1\gamma,\ \ldots\ , p_n\gamma] = T_f[p_1,\ \ldots\ , p_n]\gamma$.
We have then $\mathsf{RED}_{\Gamma_f}([\vec{p}]\gamma)$ and $\mathsf{RED}(T_f[\vec{p}]\gamma)$.
From (3.15) and Lemma 3.6.5, we get $\mathsf{RED}_{\Delta_0}(\gamma)$.
We have $s_0 \trianglelefteq s_0$, Δ_0 extends Δ_0, and from (3.17) we have $\Delta_0 \vdash s_0 \uparrow T[\vec{p}]$.
We have also $\gamma_{|\Delta_0} = \gamma_{|\Delta_0}$. The preconditions for (3.18) are then fulfilled, and we get $\mathsf{RED}_{(T_f[\vec{p}]\gamma)}(s_0\ \gamma)$. From $T_f[\vec{p}\,\gamma] = T_f[\vec{p}]\gamma$, $\vec{t} \Downarrow \vec{p}\,\gamma$ and Remark 3.3.1 we have $\mathsf{RED}_{(T_f[\vec{t}])}(s_0\ \gamma)$.

Proof of (3.18):
Assume given s, Θ, U, σ such that $s \trianglelefteq s_0$, Θ extends Δ_0, $\Theta \vdash s \uparrow U$, $\sigma_{|\Delta_0} = \gamma_{|\Delta_0}$, and $\mathsf{RED}_\Theta(\sigma)$. Assume $\mathsf{RED}(U\sigma)$.
We proceed by induction on s, and we analyze the cases of last step of the derivation of $\Theta \vdash s \uparrow U$.

1. $\dfrac{\Theta \vdash s_i \uparrow T_i[s_1,\ \ldots\ , s_{i-1}]}{\Theta \vdash x\ s_1\ \ldots\ s_n \uparrow U}\ \left\{\begin{array}{l}\Theta(x) = (x_1 : T_1,\ \ldots\ , x_n : T_n) \to T\\ U \bowtie T[s_1,\ \ldots\ , s_n]\end{array}\right.$

   Let $V_0 = (x_1 : T_1,\ \ldots\ , x_n : T_n) \to T$.
   We can write V_0 in the form Fun T_1 $(\lambda x_1.V_1)$, with $V_1 =$ Fun T_2 $(\lambda x_2.V_2)$, $V_2 =$ Fun T_3 $(\lambda x_3.V_3)$, $\ldots$, $V_{n-1} =$ Fun T_n $(\lambda x_n.T)$, and $V_n = T$.

   We have $\mathsf{RED}((\text{Fun } T_1\ (\lambda x_1.V_1))\ \sigma)$. From Specification 3.1.8 and Lemma 3.3.2 we obtain $\mathsf{RED}(T_1\sigma)$ and $\forall u.\mathsf{RED}_{(T_1\sigma)}(u) \Rightarrow \mathsf{RED}(V_1[\sigma, u/x_1])$.

   We have $\Theta \vdash s_1 \uparrow T_1$. By induction we get $\mathsf{RED}_{(T_1\sigma)}(s_1\sigma)$.

   Choose $k \in \{1,\ \ldots\ , n-1\}$. Assume $\mathsf{RED}(V_k[s_1,\ \ldots\ , s_k]\sigma)$.

   From Specification 3.1.8 and Lemma 3.3.2 we obtain

   $$\mathsf{RED}(T_{k+1}[s_1,\ \ldots\ , s_k]\sigma) \tag{3.19}$$
   $$\forall u.\mathsf{RED}_{(T_{k+1}[s_1,\ \ldots\ ,s_k]\sigma)}(u) \Rightarrow \mathsf{RED}(V_{k+1}[s_1,\ \ldots\ , s_k][\sigma, u/x_{k+1}]) \tag{3.20}$$

   From $\Theta \vdash s_{k+1} \uparrow T_{k+1}[s_1,\ \ldots\ , s_k]$ and (3.19), by induction we get

   $$\mathsf{RED}_{(T_{k+1}[s_1,\ \ldots\ ,s_k]\sigma)}(s_{k+1}\sigma) \tag{3.21}$$

[7] Recall the notations 2.1.10, page 27 (Extended context) and 2.2.10, page 30 (Substitution restricted by context).

By (3.20) and (3.21) we get $\mathsf{RED}(V_{k+1}[s_1, \ldots, s_k][\sigma, s_{k+1}\sigma/x_{k+1}])$.

By the substitution laws we have $\mathsf{RED}(V_{k+1}[s_1, \ldots, s_{k+1}]\sigma)$.

For k assuming values $1, \ldots, n-1$ we finally obtain $\mathsf{RED}(T[s_1, \ldots, s_n]\sigma)$ and $\mathsf{RED}_{(x_1:T_1\sigma, \ldots, x_n:T_n\sigma)}(s_1\sigma, \ldots, s_n\sigma)$.

By Lemma 3.4.5 we get $\mathsf{RED}_{(T[s_1, \ldots, s_n]\sigma)}((x\ \sigma)(s_1\sigma)\ldots(s_n\sigma))$, which is equivalent to $\mathsf{RED}_{(T[s_1, \ldots, s_n]\sigma)}((x\ s_1\ \ldots\ s_n)\sigma)$.

Since $T[s_1, \ldots, s_n] \bowtie U$, we have $\mathsf{RED}_{(U\ \sigma)}((x\ s_1\ \ldots\ s_n)\sigma)$.

2. $$\frac{\Theta \vdash s_i \uparrow T_i[s_1, \ldots, s_{i-1}]}{\Theta \vdash g\ s_1\ \ldots\ s_n \uparrow U} \left\{ \begin{array}{l} \mathcal{F}(g) = (x_1 : T_1, \ldots, x_n : T_n) \to T \\ U \bowtie T[s_1, \ldots, s_n] \end{array} \right.$$

From $\mathsf{RED}(\mathcal{F})$ we have $\mathsf{RED}(\mathcal{F}(g))$. Since $\mathcal{F}(g)$ is closed we have $\mathcal{F}(g) = \mathcal{F}(g)\ \sigma$, hence $\mathsf{RED}(\mathcal{F}(g)\ \sigma)$. As in the previous case, by successive induction and substitution, we obtain

$$\mathsf{RED}_{(T_1\sigma)}(s_1\sigma), \qquad \ldots, \qquad \mathsf{RED}_{(T_n[s_1, \ldots, s_{n-1}]\sigma)}(s_n\sigma).$$

Since $g\ s_1\ \ldots\ s_n \trianglelefteq s_0$ and $ar(g) = n$ we have $(f, \vec{p}) \succ (g, (s_1, \ldots, s_n))$.

By (3.14) we have $\vec{t} \rightsquigarrow^* \vec{p}\gamma$. From $\mathsf{RED}_\Theta(\sigma)$ and $\sigma_{|\Delta_0} = \gamma_{|\Delta_0}$ we have $\vec{p}\gamma = \vec{p}\sigma$, then we have $\vec{t} \rightsquigarrow^* \vec{p}\sigma$. By Proposition 3.3.3 (1), we have $\mathsf{WN}(\vec{t})$. We have fulfilled the requirements of Definition 3.6.3, (Call instance) and we get $(f, \vec{t}) \stackrel{\sim}{\succ} (g, (s_1\sigma, \ldots, s_n\sigma))$.

From $\mathsf{RED}_{(x_1:T_1, \ldots, x_n:T_n)}(s_1\sigma, \ldots, s_n\sigma)$ and Assumption (3.13) we have $\Phi(g, (s_1\sigma, \ldots, s_n\sigma))$, hence $\mathsf{RED}_{T[s_1\sigma, \ldots, s_n\sigma]}((g\ s_1\ \ldots\ s_n)\ \sigma)$.
Since $\mathcal{F}(g)$ is closed we have $T[s_1\sigma, \ldots, s_n\sigma] = T[s_1, \ldots, s_n]\sigma$, hence $\mathsf{RED}_{T[s_1, \ldots, s_n]\sigma}((g\ s_1\ \ldots\ s_n)\ \sigma)$. Since $T[s_1, \ldots, s_n] \bowtie U$, we have $\mathsf{RED}_{(U\ \sigma)}((g\ s_1\ \ldots\ s_n)\sigma)$.

3. $$\frac{\Theta \vdash s_i \uparrow \mathrm{Set}}{\Theta \vdash d\ s_1\ \ldots\ s_n \uparrow \mathrm{Set}}\ \mathcal{D}(d) = \mathrm{Set}^n \to \mathrm{Set}$$

By induction we have $\mathsf{RED}_{\mathrm{Set}}(s_i\ \sigma)$. By Proposition 3.3.3 (1), there are t_i such that $s_i\ \sigma \Downarrow t_i$, and by Remark 3.3.1 we have $\mathsf{RED}_{\mathrm{Set}}(t_i)$.

By Specification 3.1.8 we have $\mathsf{RED}_{\mathrm{Set}}(d\ s_1\sigma\ \ldots\ s_n\sigma)$.

4. $$\frac{\Theta \vdash s_i \uparrow \mathrm{El}\ e_i[u_1, \ldots, u_k]}{\Theta \vdash c\ s_1\ \ldots\ s_n \uparrow \mathrm{El}\ u} \left\{ \begin{array}{l} \mathcal{C}(c) = (\mathrm{El}\ e_1, \ldots, \mathrm{El}\ e_n) \\ \qquad\qquad \to \mathrm{El}\ (d\ x_1\ \ldots\ x_k) \\ u \rightsquigarrow^* d\ u_1\ \ldots\ u_k \end{array} \right.$$

From $\mathsf{RED}(\mathrm{El}\ u\ \sigma)$ and Remark 3.3.1 we have $\mathsf{RED}(\mathrm{El}\ (d\ u_1\sigma\ \ldots\ u_k\sigma))$. By Specification 3.1.8 we have $\mathsf{RED}_{\mathrm{Set}}(d\ u_1\sigma\ \ldots\ u_k\sigma)$, and furthermore we have $\mathsf{RED}_{\mathrm{Set}}(u_1\ \sigma), \ldots, \mathsf{RED}_{\mathrm{Set}}(u_k\ \sigma)$.

From Lemma 3.4.8 it follows that $\mathsf{RED}_{\mathrm{Set}}(e_i[u_1\sigma, \ldots, u_k\sigma])$ holds.

Since $FV(e_i) \subseteq \{x_1, \ldots, x_k\}$, we have $e_i[u_1\sigma, \ldots, u_k\sigma] = e_i[u_1, \ldots, u_k]\sigma$. It follows that $\mathsf{RED}_{\mathrm{Set}}(e_i[u_1, \ldots, u_k]\sigma)$ holds, hence $\mathsf{RED}(\mathrm{El}\ e_i[u_1, \ldots, u_k]\sigma)$.
From $\Theta \vdash s_i \uparrow \mathrm{El}\ e_i[u_1, \ldots, u_k]$ and induction we get $\mathsf{RED}_{(\mathrm{El}\ e_i[u_1, \ldots, u_k]\sigma)}(s_i\ \sigma)$.

By above equality we also have $\mathsf{RED}_{(\mathrm{El}\ e_i[u_1\sigma, \ldots, u_k\sigma])}(s_i\ \sigma)$.

By Specification 3.1.8 we can obtain $\mathsf{RED}_{(\mathrm{El}\ (d\ u_1\sigma\ \ldots\ u_k\sigma))}(c\ s_1\sigma\ \ldots\ s_n\sigma)$, and from what we know above we can conclude $\mathsf{RED}_{(\mathrm{El}\ u\ \sigma)}((c\ s_1\ \ldots\ s_n)\ \sigma)$.

5. $$\frac{\Theta \vdash s_1 \uparrow \mathrm{Set} \quad \Theta \vdash s_2 \uparrow \mathrm{El}\ s_1 \to \mathrm{Set}}{\Theta \vdash \Pi\ s_1\ s_2 \uparrow \mathrm{Set}}$$

We have $\mathsf{RED}(\mathrm{Set}\ \sigma)$, and by induction for $\Theta \vdash s_1 \uparrow \mathrm{Set}$ we get $\mathsf{RED}_{\mathrm{Set}}(s_1\sigma)$.

By Specification 3.1.8 the latter is equivalent to $\mathsf{RED}(\mathrm{El}\ s_1\ \sigma)$. We have $\mathsf{RED}(\mathrm{Set}\ \rho)$ for all ρ, then in particular $\mathsf{RED}(\mathrm{Fun}\ (\mathrm{El}\ s_1\sigma)\ ((\lambda x.\mathrm{Set})\ \sigma))$.

By induction for $\Theta \vdash s_2 \uparrow \mathrm{El}\ s_1 \to \mathrm{Set}$, we get

$\mathsf{RED}_{(\mathrm{Fun}\ (\mathrm{El}\ s_1\sigma)\ (\lambda x.\mathrm{Set}\ \sigma))}(s_2\sigma)$. By Specification 3.1.8 then

$\forall t.\mathsf{RED}_{(\mathrm{El}\ s_1\sigma)}(t) \Rightarrow \mathsf{RED}_{\mathrm{Set}\ \sigma}((s_2\sigma)\ t)$, and we have satisfied $\mathsf{RED}_{\mathrm{Set}}((\Pi\ s_1\ s_2)\ \sigma)$.

6. $$\frac{\Theta \vdash s_1 \uparrow (x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)}{\Theta \vdash \mathrm{fun}\ s_1 \uparrow \mathrm{El}\ v} \left\{ \begin{array}{l} v \rightsquigarrow^* \Pi\ t\ u \\ x \notin FV(u) \end{array} \right.$$

From $\mathsf{RED}(\mathrm{El}\ v\ \sigma)$ and $v \rightsquigarrow^* \Pi\ t\ u$ and Specification 3.1.8 we get $\mathsf{RED}_{\mathrm{Set}}(\Pi\ (t\ \sigma)\ (u\ \sigma))$. Unfolding Specification 3.1.8 we get

$$\mathsf{RED}_{\mathrm{Set}}(t\ \sigma) \qquad \text{and} \qquad \forall w.\mathsf{RED}_{(\mathrm{El}\ t\sigma)}(w) \Rightarrow \mathsf{RED}_{\mathrm{Set}}((u\ \sigma)\ w) \quad (3.22)$$

To use the induction hypothesis on s_1 we must first show $\mathsf{RED}(\ ((x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x))\ \sigma\ )$.

That is to prove $\mathsf{RED}(\mathrm{Fun}\ (\mathrm{El}\ t\ \sigma)\ ((\lambda x.\mathrm{El}\ (u\ x))\ \sigma))$, which is, knowing $\mathsf{RED}(\mathrm{El}\ t\ \sigma)$, by Lemma 3.3.2, and (3.22), to prove $\forall w.\mathsf{RED}_{(\mathrm{El}\ t\sigma)}(w) \Rightarrow \mathsf{RED}(\mathrm{El}\ (u\ x)[\sigma, w/x])$.
By the side condition $x \notin FV(u)$. Then for any w, $(u\ x)[\sigma, w/x] = (u\ \sigma)\ w$, and by unfolding Specification 3.1.8 we can state the goal by $\forall w.\mathsf{RED}_{(\mathrm{El}\ t\sigma)}(w) \Rightarrow \mathsf{RED}_{\mathrm{Set}}((u\ \sigma)\ w)$, which is known from (3.22) above.

We can use the induction hypothesis for $\Theta \vdash s_1 \uparrow (x : \mathrm{El}\ t) \to \mathrm{El}\ (u\ x)$ and obtain $\mathsf{RED}_{(\mathrm{Fun}\ (\mathrm{El}\ t\ \sigma)\ ((\lambda x.\mathrm{El}\ (u\ x))\ \sigma))}(s_1\sigma)$. By Lemma 3.4.6, we get $\mathsf{RED}_{(\mathrm{El}\ (\Pi\ (t\ \sigma)\ (u\ \sigma)))}(\mathrm{fun}\ s_1\ \sigma)$, or equivalently $\mathsf{RED}_{(\mathrm{El}\ (\Pi\ t\ u)\ \sigma)}(\mathrm{fun}\ s_1\ \sigma)$.
From $v \rightsquigarrow^* \Pi\ t\ u$ we get $v\ \sigma \rightsquigarrow^* (\Pi\ t\ u)\ \sigma$, and by Remark 3.3.1 we get $\mathsf{RED}_{(\mathrm{El}\ v\ \sigma)}(\mathrm{fun}\ s_1\ \sigma)$.

7. $\dfrac{\Theta, x : T \vdash s_1 \uparrow V}{\Theta \vdash \lambda x.s_1 \uparrow (x : T) \to V}\ x \notin supp(\Theta)$

   Assume w.l.o.g. $x \notin FV(T)$.[8] From $\mathsf{RED}(U\sigma)$ we have $\mathsf{RED}(T\sigma)$. Assume given t such that $\mathsf{RED}_{(T\sigma)}(t)$. Since $\mathsf{RED}_\Theta(\sigma)$ holds and $x \notin supp(\Theta)$, and by Lemma 3.4.3 then $\mathsf{RED}_{(\Theta, x:T)}[\sigma, t/x]$ holds.

   Since Θ extends Δ_0, $x \notin FV(T)$ and $x \notin supp(\Theta)$, then $\Theta, x : T$ extends Δ_0. We also have $[\sigma, t/x]_{|\Delta_0} = \gamma_{|\Delta_0}$.

   We know $\mathsf{RED}((\text{Fun } T\ (\lambda x.V))\sigma)$. By Lemma 3.3.2 we have $\mathsf{RED}(T\sigma)$ and $\forall u.\mathsf{RED}_{(T\sigma)}(u) \Rightarrow \mathsf{RED}(V[\sigma, u/x])$.

   We have $\mathsf{RED}(V[\sigma, t/x])$, and by induction then $\mathsf{RED}_{(V[\sigma,t/x])}(s_1[\sigma, t/x])$.

   By Proposition 2.2.14, we have $((\lambda x.s_1)\sigma)\ t \rightsquigarrow_\beta s_1[\sigma, t/x]$.
   Since t was arbitrary we have $\forall t.\mathsf{RED}_{(T\sigma)}(t) \Rightarrow \mathsf{RED}_{(V[\sigma,t/x])}(((\lambda x.s_1)\sigma)\ t)$.

   By Lemma 3.3.2 then $\mathsf{RED}_{((\text{Fun } T\ (\lambda x.V))\sigma)}((\lambda x.s_1)\sigma)$.

□

3.6.4 Normalization of well-typed terms

We can summarize what we have shown so far with the following corollary. As we will see in Section 5.2, its preconditions will be fulfilled after having type-checked the signature sucessfully.

Corollary 3.6.7. *If* $\overset{\sim}{\succ}$ *is well-founded,* $\mathsf{RED}(\mathcal{F})$ *and* $\vdash \Sigma$*, then*

1. $\vdash T \Rightarrow \mathsf{WN}(T)$
2. $\vdash t : T \Rightarrow \mathsf{WN}(t)$

Proof. Assume $\overset{\sim}{\succ}$ is well-founded, $\mathsf{RED}(\mathcal{F})$ and $\vdash \Sigma$. By Lemma 3.6.6 we have $\forall f.\mathsf{RED}_{(\mathcal{F}(f))}(f)$. Assume $\vdash T$ and $\vdash t : T$ respectively. Since the empty substitution is reducible, $\mathsf{WN}(T)$ and $\mathsf{WN}(t)$ follow from Lemma 3.5.1, Corollary 3.3.4 and Proposition 3.3.3 respectively. □

[8] We are free to choose names of bound variables, but in this case we could also use soundness of the premise, and from there we have $\Theta, x : T$ well-formed, and so $x \notin FV(T)$.

Chapter 4

Well-founded recursion

In this chapter we give a syntactic criterion for well-founded recursion, called the *Size-change principle for program termination*, of Lee, Jones and Ben-Amram (2001), and prove that is is sufficient.

4.1 The size-change principle

4.1.1 Size-change graphs and call graph

Definition 4.1.1 (Component relation for constructors). The relation $t > u$ is inductively defined as follows:

$$\frac{}{t > t_k}\ t \rightsquigarrow^* c\ t_1\ \ldots\ t_n \qquad \frac{t > u \quad u > v}{t > v}$$

Lemma 4.1.2. *If $t > u$ and there is t' such that $t \Downarrow t'$, then there is u' such that $u \Downarrow u'$ and u' is a proper sub-term of t'.*

Proof. by Definition 4.1.1 and Proposition 2.2.32 (Confluence). □

Definition 4.1.3 (Size-Change Graph).
A size-change graph $G = (\{1,\ \ldots\ , n\}, \{1,\ \ldots\ , m\}, E)$ is a directed labeled bipartite graph. The arcs in $E(G)$ are of the form $i \stackrel{=}{\rightarrow} j$ or $i \stackrel{>}{\rightarrow} j$ where $i \in \{1,\ \ldots\ , n\}$ and $j \in \{1,\ \ldots\ , m\}$.

Definition 4.1.4 (Call Graph). A call graph $\mathbb{G} = (V, E)$ is a directed labeled graph whose vertexes V are the function constants in $\mathcal{R}$. The arcs in $E(\mathbb{G})$ are of the form (f, g, G_c). Let n, m be the number of parameters for f, g. For every formal call $c = (f, (p_1,\ \ldots\ , p_n)) \succ (g, (t_1,\ \ldots\ , t_m))$ there is an arc (f, g, G_c) in $E(\mathbb{G})$. G_c is the size-change graph determined as follows:

- $k \stackrel{=}{\rightarrow} l$ is an arc in G_c if and only if $p_k = t_l$.
- $k \stackrel{>}{\rightarrow} l$ is an arc in G_c if and only if $p_k > t_l$.

Definition 4.1.5 (Path). A path $\mathbb{P}$ in $\mathbb{G}$ is a sequence of adjacent arcs of $E(\mathbb{G})$ of the form $(f_1, f_2, G_1), (f_2, f_3, G_2), \ldots$

Definition 4.1.6 (Thread). If $\mathbb{P}$ is a path $(f_1, f_2, G_1), (f_2, f_3, G_2) \ldots$ of $\mathbb{G}$, a *thread* ν of $\mathbb{P}$ is a sequence

$$i_k \overset{R_k}{\rightarrow} i_{k+1} \overset{R_{k+1}}{\rightarrow} i_{k+2} \ \ldots$$

such that $i_l \overset{R_l}{\rightarrow} i_{l+1}$ is an arc of G_l, for $l \geq k$.

Definition 4.1.7 (Size-Change Termination - SCT). We have $\mathsf{SCT}(\mathbb{G})$ if: for all infinite paths $\mathbb{P}$ in $\mathbb{G}$, $\mathbb{P}$ has an infinite thread with infinitely many $\overset{>}{\rightarrow}$-transitions.

Theorem 4.1.8. SCT *is a decidable property, if* $\mathbb{G}$ *is given.*

The proof is shown in the paper of Lee *et al.* (2001).

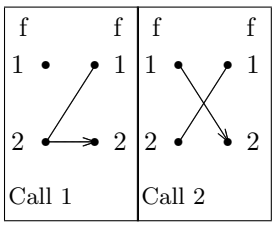
4.1.2 Examples

Maybe the most notable feature of the size-change criterion is that it may accept functions defined with permuted arguments in the recursive calls.

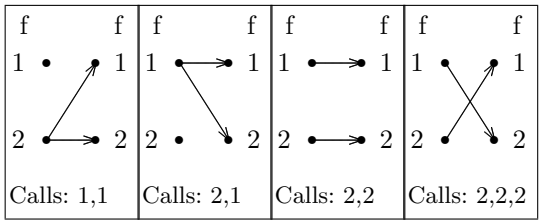
Example 4.1.9 (Permuted and possibly discarded arguments). Here follows a translation into our system from an example (Example 5) presented in Lee, Jones, Ben-Amram (2001). It shows a definition that involves permuted and possibly discarded parameters. We label the calls with prefix superscript.

$$\begin{aligned}
&\mathrm{f} : \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat} \\
&\mathrm{f}\ x\ 0 = x \\
&\mathrm{f}\ 0\ (\mathrm{s}\ y) = {}^{1}\mathrm{f}\ (\mathrm{s}\ y)\ y \\
&\mathrm{f}\ (\mathrm{s}\ x)\ (\mathrm{s}\ y) = {}^{2}\mathrm{f}\ (\mathrm{s}\ y)\ x
\end{aligned}$$

This definition is accepted by the size-change criterion, but there is no lexicographical ordering. We illustrate the corresponding size-change graphs of calls 1 and 2, labelling the $\overset{>}{\rightarrow}$-transitions with headed arrows, and the $\overset{=}{\rightarrow}$-transitions with edges, as follows:



In addition to these, the following graphs can be obtained by iterating composition of graphs 1 and 2, sequences of indexes indicated below:

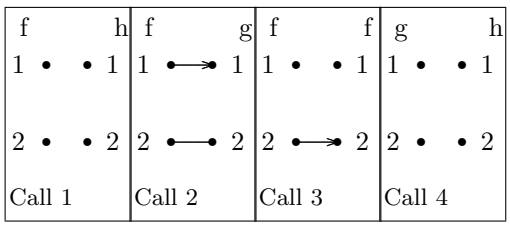


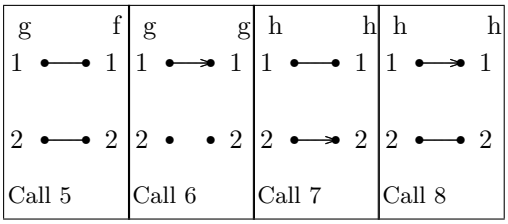
Now, any sequence of calls has an associated sequence of size-change graphs. Consider an arbitrary infinite sequence in the call graph. It can be sectioned into an infinite number of finite sections. For each section the size-change graph obtained by composing the size-change graphs of that section, will be one of the graphs given above. Of these, the first three 1–1, 2–1 and 2–2 are idempotent. They all contain a transition $i \stackrel{>}{\rightarrow} i$, for $i = 1$ or $i = 2$. This implies that an infinite path in the call graph must contain an infinitely decreasing thread, and so the criterion is fulfilled.

Example 4.1.10 (Non well-founded loops). The following mutually recursive program is not size-change terminating:

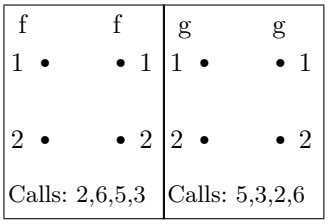
$$
\begin{array}{l}
\mathrm{f} : (\mathrm{Nat},\ \mathrm{Nat}) \rightarrow \mathrm{Nat} \\
\mathrm{f}\ 0\ y = 0 \\
\mathrm{f}\ (\mathrm{s}\ x)\ 0 = 0 \\
\mathrm{f}\ (\mathrm{s}\ x)\ (\mathrm{s}\ y) = {}^{1}\mathrm{h}\ ({}^{2}\mathrm{g}\ x\ (\mathrm{s}\ y))\ ({}^{3}\mathrm{f}\ (\mathrm{s}\ (\mathrm{s}\ (\mathrm{s}\ x)))\ y) \\
\\
\mathrm{g} : (\mathrm{Nat},\ \mathrm{Nat}) \rightarrow \mathrm{Nat} \\
\mathrm{g}\ 0\ y = 0 \\
\mathrm{g}\ (\mathrm{s}\ x)\ 0 = 0 \\
\mathrm{g}\ (\mathrm{s}\ x)\ (\mathrm{s}\ y) = {}^{4}\mathrm{h}\ ({}^{5}\mathrm{f}\ (\mathrm{s}\ x)\ (\mathrm{s}\ y))\ ({}^{6}\mathrm{g}\ x\ (\mathrm{s}\ (\mathrm{s}\ y))) \\
\\
\mathrm{h} : (\mathrm{Nat},\ \mathrm{Nat}) \rightarrow \mathrm{Nat} \\
\mathrm{h}\ 0\ 0 = 0 \\
\mathrm{h}\ 0\ (\mathrm{s}\ y) = {}^{7}\mathrm{h}\ 0\ y \\
\mathrm{h}\ (\mathrm{s}\ x)\ y = {}^{8}\mathrm{h}\ x\ y
\end{array}
$$

There are eight recursive calls, with the corresponding size-change graphs given below:





We can find compositions of these that have no decreasing arc:



This shows that the program is not structurally recursive, and we have found two cycles for which an infinite repetition corresponds to an infinite path in the call graph *without* any infinitely decreasing thread.

Contra-variance and permuted arguments

One motivation for permuted arguments is that it allows us to represent definitions involving contra-variance in a direct way. This happens for instance with sub-typing of function types. Assume we have defined a set to encode two base types and a function type former:

$$\begin{aligned} &\mathrm{Typ} : \mathrm{Set} \\ &\quad \mathrm{Big} : \mathrm{Typ} \\ &\quad \mathrm{Small} : \mathrm{Typ} \\ &\quad \Longrightarrow \ : (\mathrm{Typ}, \mathrm{Typ}) \to \mathrm{Typ} \end{aligned}$$

We define a sub-typing predicate

$$\begin{aligned} \leq \ &: (\mathrm{Typ}, \mathrm{Typ}) \to \mathrm{Set} \\ &\left\{ \begin{array}{rcll} \mathrm{Small} & \leq & \mathrm{Small} & = \top \\ \mathrm{Small} & \leq & \mathrm{Big} & = \top \\ \mathrm{Small} & \leq & (s2 \Longrightarrow t2) & = \bot \\ \mathrm{Big} & \leq & \mathrm{Small} & = \bot \\ \mathrm{Big} & \leq & \mathrm{Big} & = \top \\ \mathrm{Big} & \leq & (s2 \Longrightarrow t2) & = \bot \\ (s1 \Longrightarrow t1) & \leq & \mathrm{Small} & = \bot \\ (s1 \Longrightarrow t1) & \leq & \mathrm{Big} & = \bot \\ (s1 \Longrightarrow t1) & \leq & (s2 \Longrightarrow t2) & = (s2 \leq s1) \times (t1 \leq t2) \end{array} \right. \end{aligned}$$

It is an intuitive definition, and in this case it is easier to see that it terminates than in the previous example, since all the parameters are decreasing in the

recursive call. We can define another predicate having a type that refers to the previous predicate. Let us assume we want to prove that the sub-typing predicate is transitive. We define the constant

$$\begin{array}{l}\text{subTrans} : (x : \text{Typ},\ y : \text{Typ},\ z : \text{Typ},\\ \qquad\qquad h_1 : (x \leq y),\\ \qquad\qquad h_2 : (y \leq z),\\ \qquad\qquad) \to (x \leq z)\end{array}$$

and the recursive case is defined as follows:

$$\begin{array}{l}\text{subTrans}\ (x_1 \Longrightarrow x_2)\ (y_1 \Longrightarrow y_2)\ (z_1 \Longrightarrow z_2)\ (h_{1L},\ h_{1R})\ (h_{2L},\ h_{2R}) =\\ \qquad (\text{subTrans}\ z_1\ y_1\ x_1\ h_{2L}\ h_{1L},\ \text{subTrans}\ x_2\ y_2\ z_2\ h_{1R}\ h_{2R})\end{array}$$

4.2 Well-founded call relation

Theorem 4.2.1. *If* $\mathsf{SCT}(\mathbb{G})$ *then* $\overset{\sim}{\succ}$ *is well-founded.*

Proof. Assume $\mathsf{SCT}(\mathbb{G})$. Then $\overset{\sim}{\succ}$ is well-founded if the existence of an infinite chain in $\overset{\sim}{\succ}$ implies a contradiction. Assume there is an infinite chain

$$\chi = (f_1, \vec{t}_1) \overset{\sim}{\succ} (f_2, \vec{t}_2) \overset{\sim}{\succ} \ \ldots$$

For arbitrary $(f_i, \vec{t}_i) \overset{\sim}{\succ} (f_{i+1}, \vec{t}_{i+1})$ in χ, by Definition 3.6.3 (Call instance) we have

$$\mathsf{WN}(\vec{t}_i) \qquad \vec{t}_i \rightsquigarrow^* \vec{p}_i\gamma_i \qquad (f_i, \vec{p}_i) \succ (f_{i+1}, \vec{u}_i) \qquad \vec{t}_{i+1} = \vec{u}_i\gamma_i \tag{4.1}$$

and by Definition 4.1.4 (Call graph) there is an infinite path $\mathbb{P}$ in $\mathbb{G}$ with adjacent arcs (f_i, f_{i+1}, G_i) such that

$$\begin{array}{lll} k \overset{=}{\to} l \in G_i & \text{whenever} & \vec{p}_i(k) = \vec{u}_i(l)\\ k \overset{>}{\to} l \in G_i & \text{whenever} & \vec{p}_i(k) > \vec{u}_i(l)\end{array} \tag{4.2}$$

By assumption $\mathsf{SCT}(\mathbb{G})$ holds, and then $\mathbb{P}$ has an infinitely decreasing thread ν, starting at some (f_m, f_{m+1}, G_m) in $\mathbb{G}$.
Let $\vec{v}_j = \vec{t}_{m+j},\ \vec{q}_j = \vec{p}_{m+j},\ \delta_j = \gamma_{m+j}$ and $H_j = G_{m+j}$.
We have

$$\nu = k_1 \overset{R_1}{\to} k_2 \in H_1, k_2 \overset{R_2}{\to} k_3 \in H_2, \ldots$$

with infinitely many $\overset{>}{\to}$ transitions. From (4.1) and (4.2) then

$$\begin{array}{lll}\vec{v}_1(k_1) \rightsquigarrow^* \vec{q}_1\delta_1(k_1) & \text{and} & \vec{q}_1\delta_1(k_1)\ R_1\ \vec{v}_2(k_2),\\ \vec{v}_2(k_2) \rightsquigarrow^* \vec{q}_2\delta_2(k_2) & \text{and} & \vec{q}_2\delta_2(k_2)\ R_2\ \vec{v}_3(k_3),\\ \ldots & & \end{array}$$

But from (4.1) we have $\mathsf{WN}(\vec{v}_1(k_1))$. If R_1 is $>$, by Lemma 4.1.2, the normal form of $\vec{v}_2(k_2)$ is a proper sub-term of the normal form of $\vec{q}_1\delta_1(k_1)$. Otherwise, By Proposition 2.2.39, the normal forms are the same. The same holds for the whole sequence $\vec{v}_i(k_i)$. Thus the infinite decrease in $>$, starting with $\vec{v}_1(k_1)$ leads to a contradiction. □

Chapter 5

Main results

In this chapter we apply the results from previous chapters. We prove decidability of type-correctness and show how to extend a theory in a sequence of steps that can be mechanically verified. We also prove logical consistency.

5.1 Decidable type correctness

5.1.1 Checking type formation and inhabitation

Theorem 5.1.1 (Decidable type correctness).
If $\mathsf{RED}(\Sigma)$ *and* $\vdash \Sigma$ *then*[1]

1. *given* $\vdash \Gamma$, *the problem* $\Gamma \vdash S \uparrow$ *is decidable.*
2. *given* $\Gamma \vdash T$, *the problem* $\Gamma \vdash s \uparrow T$ *is decidable.*

Proof of 2. Assume $\vdash \Sigma$, and $\Gamma \vdash T$. First note that from $\Gamma \vdash T$ and Lemma 2.3.6 we have

$$\vdash \Gamma. \tag{5.1}$$

We prove that the problem $\Gamma \vdash s \uparrow T$ is decidable by induction on s. In each production we have to decide the premises and the side conditions. If any of the latter tests fail, the conclusion in the corresponding rule cannot be type correct, by completeness. We have the following cases:

1. $$\frac{\Gamma \vdash s_i \uparrow U_i[s_1, \ \ldots \ , s_{i-1}]}{\Gamma \vdash x \ s_1 \ \ldots \ s_n \uparrow T} \left\{ \begin{array}{l} \Gamma(x) = (y_1 : U_1, \ \ldots \ , y_n : U_n) \to U \\ T \bowtie U[s_1, \ \ldots \ , s_n] \end{array} \right.$$

   First check that $x \in \mathit{supp}(\Gamma)$. From (5.1) we get $\Gamma \vdash \Gamma(x)$. From Lemma 2.3.5 we get
   $\Gamma \vdash U_1$, $\Gamma, y_1 : U_1 \vdash U_2$ through $\Gamma, y_1 : U_1, \ \ldots \ , y_{n-1} : U_{n-1} \vdash U_n$,
   and $\Gamma, y_1 : U_1, \ \ldots \ , y_n : U_n \vdash U$.

[1] Recall that S and s denotes the β-normal fragment of the language. See Definition 2.1.20, page 28.

From $\Gamma \vdash U_1$ and induction we get $\Gamma \vdash s_1 \uparrow U_1$ decidable. By soundness we get $\Gamma \vdash s_1 : U_1$. Then we have

$$[s_1/y_1] : \Gamma \to \Gamma, y_1 : U_1$$

and by Lemma 2.4.4 (Substitution lemma) we have $\Gamma \vdash U_2[s_1/y_1]$. By induction for s_2 we get $\Gamma \vdash s_2 \uparrow U_2[s_1/y_1]$ decidable, and on success, by soundness we have $\Gamma \vdash s_2 : U_2[s_1/y_1]$. We proceed similarly until we get

$$[s_1/y_1,\ \ldots\ , s_{n-1}/y_{n-1}]\ :\ \Gamma\ \to\ \Gamma, y_1 : U_1,\ \ldots\ , y_{n-1} : U_{n-1}$$

and by Lemma 2.4.4 (Substitution lemma) get $\Gamma \vdash U_n[s_1,\ \ldots\ , s_{n-1}]$ and by induction we get $\Gamma \vdash s_n \uparrow U_n[s_1,\ \ldots\ , s_{n-1}]$ decidable and on success obtain $\Gamma \vdash s_n : U_n[s_1,\ \ldots\ , s_{n-1}]$ from soundness. By the substitution lemma and

$$[s_1/y_1,\ \ldots\ , s_n/y_n]\ :\ \Gamma\ \to\ \Gamma, y_1 : U_1,\ \ldots\ , y_n : U_n$$

we get $\Gamma \vdash U[s_1,\ \ldots\ , s_n]$.

By Lemma 2.4.11 we get $\Gamma \vdash x\ s_1\ \ldots\ s_n : U[s_1,\ \ldots\ , s_n]$.

It remains to check $T \bowtie U[s_1,\ \ldots\ , s_n]$. From previously we know $\Gamma \vdash T$ and $\Gamma \vdash U[s_1,\ \ldots\ , s_n]$, and by Lemma 3.5.1 we have $\mathsf{RED}(T)$ and $\mathsf{RED}(U[s_1,\ \ldots\ , s_n])$. By Corollary 3.3.4 there are T' and U' such that $T \Downarrow T'$ and $U[s_1,\ \ldots\ , s_n] \Downarrow U'$. By the uniqueness of normal form $T \bowtie U[s_1,\ \ldots\ , s_n]$ if and only if $T' = U'$.

2. $$\frac{\Gamma \vdash s_i \uparrow U_i[s_1,\ \ldots\ , s_{i-1}]}{\Gamma \vdash f\ s_1\ \ldots\ s_n \uparrow U} \left\{ \begin{array}{l} \mathcal{F}(f) = (y_1 : U_1,\ \ldots\ , y_n : U_n) \to U \\ T \bowtie U[s_1,\ \ldots\ , s_n] \end{array} \right.$$

First check that $f \in \mathcal{F}$. From $\vdash \Sigma$ we get $\vdash \mathcal{F}(f)$.

By iteration of Corollary 2.4.2 (Weakening) we get $\Gamma \vdash \mathcal{F}(f)$.

As in the last case we get by successively applying the substitution lemma and induction that $\Gamma \vdash s_i \uparrow U_i[s_1,\ \ldots\ , s_{i-1}]$ are decidable, together with $\Gamma \vdash s_i : U_i[s_1,\ \ldots\ , s_{i-1}]$ upon success. By the same argument as in the previous case we can check $T \bowtie U[s_1,\ \ldots\ , s_n]$.

3. $$\frac{\Gamma \vdash s_i \uparrow \mathrm{Set}}{\Gamma \vdash d\ s_1\ \ldots\ s_n \uparrow \mathrm{Set}}\ \mathcal{D}(d) = \mathrm{Set}^n \to \mathrm{Set}$$

First check that $\mathcal{D}(d) = \mathrm{Set}^n \to \mathrm{Set}$. By assumption $\vdash \Gamma$, hence $\Gamma \vdash \mathrm{Set}$.

By induction $\Gamma \vdash s_i \uparrow \mathrm{Set}$ are decidable.

4. $$\frac{\Gamma \vdash s_i \uparrow \mathrm{El}\ e_i[u_1,\ \ldots\ , u_k]}{\Gamma \vdash c\ s_1\ \ldots\ s_n \uparrow \mathrm{El}\ u} \left\{ \begin{array}{l} \mathcal{C}(c) = (\mathrm{El}\ e_1,\ \ldots\ ,\ \mathrm{El}\ e_n) \\ \qquad\qquad \to \mathrm{El}\ (d\ y_1\ \ldots\ y_k) \\ u \rightsquigarrow^* d\ u_1\ \ldots\ u_k \end{array} \right.$$

First look up $\mathcal{C}(c) = (\text{El } e_1,\ \ldots\ ,\ \text{El } e_n) \to \text{El } (d\ y_1\ \ldots\ y_k)$.
From $\Gamma \vdash \text{El } u$, Lemma 3.5.1 and Specification 3.1.8, there are $d', v_1,\ \ldots\ , v_m$ such that $u \Downarrow d'\ v_1\ \ldots\ v_m$ where $\mathcal{D}(d') = \text{Set}^m \to \text{Set}$.

Check that $d = d'$ and $k = m$.

Whenever $u \rightsquigarrow^* d\ u_1\ \ldots\ u_k$ we have $u_i \bowtie v_i$, and by Lemma 2.5.5 we have $\Gamma \vdash s_i \uparrow \text{El } e_i[v_1,\ \ldots\ , v_k]$ iff $\Gamma \vdash s_i \uparrow \text{El } e_i[u_1,\ \ldots\ , u_k]$.

By Lemma 2.4.18 (Subject reduction) we have $\Gamma \vdash \text{El } (d\ u_1\ \ldots\ u_k)$. By Lemma 2.4.9 (Generation lemma) then $\Gamma \vdash u_i : \text{Set}$.

From $\vdash \Gamma$ and Definition 2.3.1 (Typing) we get $\Gamma \vdash c : \mathcal{C}(c)[u_1,\ \ldots\ , u_k]$. By Lemma 2.4.8 we have $\Gamma \vdash \mathcal{C}(c)[u_1,\ \ldots\ , u_k]$. By Lemma 2.3.5 (Iterated function type inversion) we get $\Gamma \vdash \text{El } e_i[u_1,\ \ldots\ , u_k]$. By induction $\Gamma \vdash s_i \uparrow \text{El } e_i[u_1,\ \ldots\ , u_k]$ are decidable.

5. $$\frac{\Gamma \vdash s_1 \uparrow \text{Set} \quad \Gamma \vdash s_2 \uparrow \text{El } s_1 \to \text{Set}}{\Gamma \vdash \Pi\ s_1\ s_2 \uparrow \text{Set}}$$

From $\vdash \Gamma$ we have $\Gamma \vdash \text{Set}$. By induction $\Gamma \vdash s_1 \uparrow \text{Set}$ is decidable. Upon success, we have then $\Gamma \vdash \text{El } s_1 \to \text{Set}$. By induction $\Gamma \vdash s_2 \uparrow \text{El } s_1 \to \text{Set}$ is decidable.

6. $$\frac{\Gamma \vdash s_1 \uparrow (x : \text{El } t) \to \text{El } (u\ x)}{\Gamma \vdash \text{fun } s_1 \uparrow \text{El } v} \begin{cases} v \rightsquigarrow^* \Pi\ t\ u \\ x \notin FV(u) \end{cases}$$

From $\Gamma \vdash \text{El } v$ and the fact that the identity substitution [] is reducible, by Lemma 3.5.1 we have that $\mathsf{RED}_{\text{Set}}(v)$ holds. Then we can check that v normalizes to an expression of the form $\Pi\ t'\ u'$.

By the Church-Rosser property, whenever $v \rightsquigarrow^* \Pi\ t\ u$ we have $t \bowtie t'$ and $u \bowtie u'$, and by Lemma 2.5.5 (Completeness with conversion) we have $\Gamma \vdash s_1 \uparrow (x : \text{El } t') \to \text{El } (u'\ x)$ iff $\Gamma \vdash s_1 \uparrow (x : \text{El } t) \to \text{El } (u\ x)$.

From $\Gamma \vdash \text{El } v$, $v \rightsquigarrow^* \Pi\ t\ u$ and Lemma 2.4.18 (Subject reduction) we have $\Gamma \vdash \text{El } (\Pi\ t\ u)$.

By Lemma 2.4.14 then $\Gamma \vdash (x : \text{El } t) \to \text{El } (u\ x)$ with $x \notin FV(u)$. By induction $\Gamma \vdash s_1 \uparrow (x : \text{El } t) \to \text{El } (u\ x)$ is decidable.

7. $$\frac{\Gamma, x : U \vdash s_1 \uparrow V}{\Gamma \vdash \lambda x.s_1 \uparrow (x : U) \to V}\ x \notin supp(\Gamma)$$

First check $x \notin supp(\Gamma)$. From $\Gamma \vdash (x : U) \to V$ and Lemma 2.3.4 (Type-formation inversion) we have $\Gamma, x : U \vdash V$. By induction $\Gamma, x : U \vdash s_1 \uparrow V$ is decidable.

□

Proof of 1. Assume $\vdash \Gamma$. We have to decide $\Gamma \vdash S \uparrow$. We proceed by induction on S. In each production we have to decide the premises and the side conditions.

If any of the latter tests fail, the conclusion in the corresponding rule cannot be type correct, by completeness. We have the following cases:

1. $\dfrac{}{\Gamma \vdash \text{Set} \uparrow}$.

   In this case the relation holds.

2. $\dfrac{\Gamma \vdash s \uparrow \text{Set}}{\Gamma \vdash \text{El } s \uparrow}$.

   By assumption $\vdash \Gamma$, hence $\Gamma \vdash \text{Set}$. Then, by 2 we have $\Gamma \vdash s \uparrow \text{Set}$ decidable.

3. $\dfrac{\Gamma \vdash S_1 \uparrow \quad \Gamma, x : S_1 \vdash S_2 \uparrow}{\Gamma \vdash (x : S_1) \to S_2 \uparrow}\ x \notin \mathit{supp}(\Gamma)$

   By assumption $\vdash \Gamma$, and by induction then $\Gamma \vdash S_1 \uparrow$ is decidable. By soundness then $\Gamma \vdash S_1$. Check $x \notin \mathit{supp}(\Gamma)$. We have $\vdash \Gamma, x : S_1$. By induction then $\Gamma, x : S_1 \vdash S_2 \uparrow$ is decidable.

□

5.1.2 Type-checking of patterns

Definition 5.1.2 (Size of pattern vector).
$\mathit{size}() = 0$
$\mathit{size}(x, \vec{p}\,) = 1 + \mathit{size}(\vec{p}\,)$
$\mathit{size}(\text{fun } x, \vec{p}\,) = 1 + \mathit{size}(\vec{p}\,)$
$\mathit{size}(c\ \vec{q}\,, \vec{p}\,) = 1 + \mathit{size}(\vec{q}\,) + \mathit{size}(\vec{p}\,)$

Lemma 5.1.3. *If* $\mathsf{RED}(\Sigma)$ *and* $\vdash \Sigma$ *then given* $\vec{p}$ *and* Γ *such that* $\vdash \Gamma$ *holds, we can find the unique normal* Δ*, if it exists, such that* $\Delta \xrightarrow{[\vec{p}\,]} \Gamma$ *holds.*

Proof. Assume $\mathsf{RED}(\Sigma)$, $\vdash \Sigma$ and $\vdash \Gamma$.
Let $\vec{p} = (p_1, \ldots, p_n)$ and $\Gamma = (x_1 : T_1, \ldots, x_n : T_n)$. To abbreviate, let $\vec{p}\,' = (p_2, \ldots, p_n)$ and $\Gamma' = (x_2 : T_2, \ldots, x_n : T_n)$. We proceed by induction on $\mathit{size}(\vec{p}\,)$.
First we observe, that from $\vdash \Gamma$ and Corollary 2.3.7 we have $\vdash T_1$. By Lemma 3.5.1 (for the empty substitution), Corollary 3.3.4, and Proposition 2.2.39 (uniqueness of normal form), there is a unique U, such that $T_1 \Downarrow U$. From $\vdash \Sigma$, $\vdash T_1$, $T_1 \Downarrow U$ and Lemma 2.4.18 (Subject reduction) we have $\vdash U$. We may have the following forms of p_1:

1. y.
   As a sub-goal we need $\vdash y : U, \Gamma'[y/x_1]$. We have $[y/x_1] : y : U \to x_1 : T_1$. From $\vdash \Gamma$ we have $x_1 : T_1 \vdash T_2$. By Lemma 2.4.4 (Substitution lemma) we have $y : U \vdash T_2[y/x_1]$. We get
   $[y/x_1] : (y : U,\ x_2 : T_2[y/x_1]) \to (x_1 : T_1, x_2 : T_2)$.

We have $x_1 : T_1, x_2 : T_2 \vdash T_3$. By Lemma 2.4.4 (Substitution lemma) we have $y : U, x_2 : T_2[y/x_1] \vdash T_3[y/x_1]$. Iterating this argument, we finally arrive at $\vdash y : U, \Gamma'[y/x_1]$. By Definition 2.3.11 (Atomic neighbourhood), we have $y : U, \Gamma'[y/x_1] \xrightarrow{[y/x_1]} \Gamma$. By induction we can find a unique normal Δ' such that $\Delta' \xrightarrow{[\vec{p}\,']} y : T_1, \Gamma'[y/x_1]$, if it exists, thus satisfying Definition 2.3.13 (Compound neighbourhood).

2. fun y.
For the goal to be fulfilled, by Definition 2.3.11 (Atomic neighbourhood), we require that U is of the form El $(\Pi\ t\ u)$.

   Since we can choose names of the bound variables, we can make sure $y : (z : \text{El}\ t) \to \text{El}\ (u\ z), \Gamma'[\text{fun}\ y/x_1]$ is closed and disjoint.
   As a sub-goal we need $\vdash y : (z : \text{El}\ t) \to \text{El}\ (u\ z), \Gamma'[\text{fun}\ y/x_1]$. From $\vdash \Sigma$, $\vdash \text{El}\ (\Pi\ t\ u)$ and Lemma 2.4.14 we have $\vdash (z : \text{El}\ t) \to \text{El}\ (u\ z)$.
   We have $[\text{fun}\ y/x_1] : (y : (z : \text{El}\ t) \to \text{El}\ (u\ z)) \to x_1 : T_1$. From $\vdash \Gamma$ we have $x_1 : T_1 \vdash T_2$. By Lemma 2.4.4 (Substitution lemma) we have $y : (z : \text{El}\ t) \to \text{El}\ (u\ z) \vdash T_2[\text{fun}\ y/x_1]$. We get
   $[\text{fun}\ y/x_1] :$
   $$(y : (z : \text{El}\ t) \to \text{El}\ (u\ z),\ x_2 : T_2[\text{fun}\ y/x_1]) \to (x_1 : T_1, x_2 : T_2).$$
   We have $x_1 : T_1, x_2 : T_2 \vdash T_3$. By Lemma 2.4.4 (Substitution lemma) we have $y : (z : \text{El}\ t) \to \text{El}\ (u\ z), x_2 : T_2[\text{fun}\ y/x_1] \vdash T_3[\text{fun}\ y/x_1]$. Iterating this argument, we finally arrive at $\vdash y : (z : \text{El}\ t) \to \text{El}\ (u\ z), \Gamma'[\text{fun}\ y/x_1]$. By induction we can find a unique normal Δ' satisfying
   $\Delta' \xrightarrow{[\vec{p}']} (y : (z : \text{El}\ t) \to \text{El}\ (u\ z), \Gamma'[\text{fun}\ y/x_1])$, if it exists. This satisfies Definition 2.3.13 (Compound neighbourhood).

3. $c\ \vec{q}$.
   For the goal to be fulfilled, by Definition 2.3.11 (Atomic neighbourhood), we require that U is of the form El $(d\ \vec{t}\,)$. Next, check that $\mathcal{C}(c)$ is of the form $\Delta_c \to \text{El}\ (d\ \vec{z}\,)$, and $|\vec{t}\,| = |\vec{z}\,|$. Since we can choose names of the bound variables in $\mathcal{C}(c)$, let $\vec{y} = \overrightarrow{supp}(\Delta_c)$, we can make sure $\Delta_c, \Gamma'[c\ \vec{y}\,/x_1]$ is closed and disjoint. We need $\vdash \Delta_c[\vec{t}\,/\vec{z}\,], \Gamma'[c\ \vec{y}\,/x_1]$ as a sub-goal. From $\vdash \Sigma$, $\vdash \text{El}\ (d\ \vec{t}\,)$ and Lemma 2.4.13 we have $\vdash t_i : \text{Set}$. By Lemma 2.4.4 (Substitution lemma) we have $\vdash \mathcal{C}(c)[\vec{t}\,/\vec{z}\,]$, and so $\vdash \Delta_c[\vec{t}\,/\vec{z}\,]$. We have $[c\ \vec{y}\,/x_1] : \Delta_c[\vec{t}\,/\vec{z}\,] \to x_1 : T_1$. From $\vdash \Gamma$ we have $x_1 : T_1 \vdash T_2$. By Lemma 2.4.4 (Substitution lemma) we have $\Delta_c[\vec{t}\,/\vec{z}\,] \vdash T_2[c\ \vec{y}\,/x_1]$. We get $[c\ \vec{y}\,/x_1] : \Delta_c[\vec{t}\,/\vec{z}\,], x_2 : T_2[c\ \vec{y}\,/x_1] \to x_1 : T_1, x_2 : T_2$.

   We have $x_1 : T_1, x_2 : T_2 \vdash T_3$. By Lemma 2.4.4 (Substitution lemma) we have $\Delta_c[\vec{t}\,/\vec{z}\,], x_2 : T_2[c\ \vec{y}\,/x_1] \vdash T_3[c\ \vec{y}\,/x_1]$. Iterating this argument, we finally arrive at $\vdash \Delta_c[\vec{t}\,/\vec{z}\,], \Gamma'[c\ \vec{y}\,/x_1]$. By induction we can find a unique normal Δ' satisfying $\Delta' \xrightarrow{[\vec{q}\,,\vec{p}\,']} \Delta_c[\vec{t}\,/\vec{z}\,], \Gamma'[c\ \vec{y}\,/x_1]$, it it exists. Thus we satisfy Definition 2.3.13 (Compound neighbourhood).

□

5.2 Type-checking the signature

In this section we show how to type-check a signature built up as a sequence of extensions of the empty signature. This is an important step, where we tie together the previously established results in this dissertation. This procedure corresponds to what is usually called stratification.

Until now, the given signature Σ has been fixed, but below, by technical reasons due to the reducibility method, we will have to make our statements relative to given parts of the signature known so far. Recall notation 2.3.18, page 41, for judgements made relative to an explicit signature.

5.2.1 Type-checking a sequence of extensions

Lemma 5.2.1 (Derivations independent of signature extensions).

1. *(a)* $\Gamma \vdash_{\mathcal{D},\mathcal{C},\mathcal{F},\mathcal{R}} T \;\Rightarrow\; \Gamma \vdash_{\mathcal{D},\mathcal{C},\mathcal{F},\mathcal{R}\mathcal{R}'} T$
   (b) $\Gamma \vdash_{\mathcal{D},\mathcal{C},\mathcal{F},\mathcal{R}} t : T \;\Rightarrow\; \Gamma \vdash_{\mathcal{D},\mathcal{C},\mathcal{F},\mathcal{R}\mathcal{R}'} t : T$
2. *(a)* $\Gamma \vdash_{\mathcal{D},\mathcal{C},\mathcal{F},\mathcal{R}} T \;\Rightarrow\; \Gamma \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}\mathcal{F}',\mathcal{R}} T$
   (b) $\Gamma \vdash_{\mathcal{D},\mathcal{C},\mathcal{F},\mathcal{R}} t : T \;\Rightarrow\; \Gamma \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}\mathcal{F}',\mathcal{R}} t : T$

Proof. This is direct, since the addition of new rules never prevents computations that were possible previously. For extensions of $\mathcal{F}$, the presence of typing specifications for other constants than those in $\mathcal{F}$ will not be used in the derivation that was already established. □

Theorem 5.2.2 (A procedure for type-checking the signature). *Given the signature $\Sigma = (\mathcal{D}, \mathcal{C}, \mathcal{F}_1\mathcal{F}_2, \mathcal{R}_1\mathcal{R}_2)$, where no rules for $\mathcal{F}_2$ are defined in $\mathcal{R}_1$, no rules for $\mathcal{F}_1$ are defined in $\mathcal{R}_2$, no constant declared in $\mathcal{F}_2$ occurs in $\mathcal{F}_1$ nor $\mathcal{R}_1$,[2] and $\overset{\sim}{\succ}$ is well-founded with respect to $\mathcal{R}_1\mathcal{R}_2$, one can decide*
$\vdash \mathcal{D}, \mathcal{C}, \emptyset, \emptyset$,
$\vdash \mathcal{D}, \mathcal{C}, \mathcal{F}_1, \emptyset$, $\vdash \mathcal{D}, \mathcal{C}, \mathcal{F}_1, \mathcal{R}_1$,
$\vdash \mathcal{D}, \mathcal{C}, \mathcal{F}_1\mathcal{F}_2, \mathcal{R}_1$, $\vdash \mathcal{D}, \mathcal{C}, \mathcal{F}_1\mathcal{F}_2, \mathcal{R}_1\mathcal{R}_2$,
in sequence, and on success of the previous steps obtain $\vdash \Sigma$ *and* $\mathsf{RED}(\Sigma)$.

Proof. Similarly as in Theorem 5.1.1, we will perform a sequence of tests. If any on the tests fail, the main goal certainly cannot be fulfilled, by completeness of the type-checking steps. Note that the results from computations below, due to Lemma 5.1.3, are normal and unique. Therefore, in the cases below that depend on previously established computations, we will not lose completeness.
Consider $\Sigma = (\mathcal{D}, \mathcal{C}, \mathcal{F}_1\mathcal{F}_2, \mathcal{R}_1\mathcal{R}_2)$ as given above.
Assume that $\overset{\sim}{\succ}$ is well-founded with respect to $\mathcal{R}_1\mathcal{R}_2$. We perform the procedure as follows:

[2] This condition is necessary for the success of the stratification, but not for the correct behaviour of the procedure, since it will be a consequence of that we first check $\mathcal{F}_1$ and $\mathcal{R}_1$ without the presence of $\mathcal{F}_2$.

1. Verification of $\mathcal{D}, \mathcal{C}$.
   (a) $\vdash \mathcal{D}, \mathcal{C}, \emptyset, \emptyset$.
   Straight-forward from Definition 2.3.15.
   (b) $\mathsf{RED}^{\emptyset}(\emptyset)$.
   Direct from Definition 3.4.10.
   (c) $\mathsf{RED}^{\mathcal{R}_1}(\emptyset)$.
   Direct from Definition 3.4.10.
   (d) $\mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}(\emptyset)$.
   Direct from Definition 3.4.10.
2. Verification of $\mathcal{F}_1$.
   (a) Assume given $f_1 \in \mathcal{F}_1$. Let $S_1 = \mathcal{F}_1(f_1)$.
   (b) Check $\vdash_{\mathcal{D},\mathcal{C},\emptyset,\emptyset} S_1 \uparrow$, and assume it succeeds.
   This is justified by Theorem 5.1.1, using 1a and 1b.
   (c) $\vdash_{\mathcal{D},\mathcal{C},\emptyset,\emptyset} S_1$.
   By Lemma 2.5.3 (Soundness of type formation checking), using 2b and 1a.
   (d) $\vdash_{\mathcal{D},\mathcal{C},\emptyset,\mathcal{R}_1} S_1$.
   By Lemma 5.2.1 (Derivations independent of signature extensions), using 2c.
   (e) $\vdash_{\mathcal{D},\mathcal{C},\emptyset,\mathcal{R}_1\mathcal{R}_2} S_1$.
   By Lemma 5.2.1 (Derivations independent of signature extensions), using 2c.
   (f) $\mathsf{RED}^{\emptyset}(S_1)$.
   By Lemma 3.5.1 (Reducibility of well-typed terms), using 2c and 1b.
   (g) $\mathsf{RED}^{\mathcal{R}_1}(S_1)$.
   By Lemma 3.5.1 (Reducibility of well-typed terms), using 2d and 1c.
   (h) $\mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}(S_1)$.
   By Lemma 3.5.1 (Reducibility of well-typed terms), using 2e and 1d.
   (i) Since f_1 was arbitrary from 2a, we have:
      i. $\vdash_{(\mathcal{D},\mathcal{C},\emptyset,\emptyset)} \mathcal{F}_1$. By 2c.
      ii. $\mathsf{RED}^{\emptyset}(\mathcal{F}_1)$. By 2f.
      iii. $\mathsf{RED}^{\mathcal{R}_1}(\mathcal{F}_1)$. By 2g.
      iv. $\mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}(\mathcal{F}_1)$. By 2h.
   (j) $\vdash \mathcal{D}, \mathcal{C}, \mathcal{F}_1, \emptyset$.
   By Definition 2.3.17 using 2(i)i.

3. Verification of $\mathcal{R}_1$.
   (a) Assume given $f_1' \in \mathcal{F}_1$. We have $\mathcal{F}_1(f_1')$ of the form $\Xi_1 \to S_1'$.
   (b) For f_1', assume given a rule $f_1'\ \vec{p}_1 = s_1 \in \mathcal{R}_1$.
   (c) $\vdash_{\mathcal{D},\mathcal{C},\emptyset,\emptyset} \Xi_1 \to S_1'$.
   By 2(i)i.
   (d) $\Xi_1 \vdash_{\mathcal{D},\mathcal{C},\emptyset,\emptyset} S_1'$.
   By iterating Lemma 2.3.4 from 3c.
   (e) $\vdash_{(\mathcal{D},\mathcal{C},\emptyset,\emptyset)} \Xi_1$.
   By Lemma 2.3.6, using 3d.
   (f) Find the unique and normal Δ_1, if it exists, such that $\Delta_1 \xrightarrow{[\vec{p}\,_1]} \Xi_1$ in the signature $(\mathcal{D},\mathcal{C},\emptyset,\emptyset)$.
   By Lemma 5.1.3, using 3e, 1a and 1b.
   (g) $\vdash_{(\mathcal{D},\mathcal{C},\emptyset,\emptyset)} \Delta_1$.
   By Definition 2.3.13 (Neighbourhood), using 3f.
   (h) $[\vec{p}\,_1] : \Delta_1 \to \Xi_1$ in the signature $(\mathcal{D},\mathcal{C},\emptyset,\emptyset)$.
   By Lemma 2.4.7, using 3f.
   (i) $\Delta_1 \vdash_{\mathcal{D},\mathcal{C},\emptyset,\emptyset} S_1'[\vec{p}\,_1]$.
   By Lemma 2.4.4 (Substitution lemma), using 3d, 3h and 3g.
   (j) $\Delta_1 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\emptyset} S_1'[\vec{p}\,_1]$.
   By Lemma 5.2.1 using 3i.
   (k) Check $\Delta_1 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\emptyset} s_1 \uparrow S_1'[\vec{p}\,_1]$, and assume success.
   By Theorem 5.1.1, using 3j, 2j and 2(i)ii.
   (l) $\Delta_1 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\emptyset} s_1 : S_1'[\vec{p}\,_1]$.
   By Lemma 2.5.2 (Soundness of type checking), using 3k and 2j.
   (m) $\Delta_1 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1} s_1 : S_1'[\vec{p}\,_1]$.
   By Lemma 5.2.1, using 3l.
   (n) Since f_1' and $f_1'\ \vec{p}_1 = s_1$ was arbitrary, by 3a, 3b and Definition 2.3.16, using 3m we have:
      i. $\vdash \mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1$.
      ii. $\vdash \mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1\mathcal{R}_2$.
      Recall from the definition that $\mathcal{R}_1\mathcal{R}_2$ only have to be type correct w.r.t. $\mathcal{F}_1$ in this case, since we quantify over the constants in $\mathcal{F}_1$, and no rules for $\mathcal{F}_1$ are defined in $\mathcal{R}_2$.
   (o) $\overset{\sim}{\succ}$ is well-founded w.r.t. $\mathcal{R}_1$.
   Since $\overset{\sim}{\succ}$ is well-founded w.r.t. $\mathcal{R}_1\mathcal{R}_2$.
   (p) $\forall f \in \mathcal{F}_1.\mathsf{RED}^{\mathcal{R}_1}(\mathcal{F}_1(f)) \wedge \mathsf{RED}^{\mathcal{R}_1}_{(\mathcal{F}_1(f))}(f)$.
   By Lemma 3.6.6 (Key lemma), using 3o, 3(n)i and 2(i)iii.
   (q) $\forall f \in \mathcal{F}_1.\mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}(\mathcal{F}_1(f)) \wedge \mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}_{(\mathcal{F}_1(f))}(f)$.
   By Lemma 3.6.6 (Key lemma), using $\overset{\sim}{\succ}$ well-founded, 3(n)ii and 2(i)iv.

4. Verification of $\mathcal{F}_2$.
   (a) Assume given $f_2 \in \mathcal{F}_1\mathcal{F}_2$. Let $S_2 = \mathcal{F}_1\mathcal{F}_2(f_2)$.
   (b) Check $\vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1} S_2 \uparrow$, and assume it succeeds.
   This is justified by Theorem 5.1.1, using 3p and 3(n)i.
   (c) $\vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1} S_2$.
   By Lemma 2.5.3 (Soundness of type formation checking), using 4b and 3(n)i.
   (d) $\vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1\mathcal{R}_2} S_2$.
   By Lemma 5.2.1 (Derivations independent of signature extensions), using 4c.
   (e) $\mathsf{RED}^{\mathcal{R}_1}(S_2)$.
   By Lemma 3.5.1 (Reducibility of well-typed terms), using 4c and 3p.
   (f) $\mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}(S_2)$.
   By Lemma 3.5.1 (Reducibility of well-typed terms), using 4d and 3q.
   (g) Since f_2 was arbitrary from 4a, we have:
      i. $\vdash_{(\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1)} \mathcal{F}_1\mathcal{F}_2$. By 4c.
      ii. $\mathsf{RED}^{\mathcal{R}_1}(\mathcal{F}_1\mathcal{F}_2)$. By 4e.
      iii. $\mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}(\mathcal{F}_1\mathcal{F}_2)$. By 4f.
   (h) $\vdash \mathcal{D}, \mathcal{C}, \mathcal{F}_1\mathcal{F}_2, \mathcal{R}_1$.
   By Definition 2.3.17 using 4(g)i, 3(n)i and Lemma 5.2.1.

5. Verification of $\mathcal{R}_2$.

   (a) Assume given $f_2' \in \mathcal{F}_1\mathcal{F}_2$. We have $\mathcal{F}_1\mathcal{F}_2(f_2')$ of the form $\Xi_2 \to S_2'$.

   (b) For f_2', assume given a rule $f_2'\ \vec{p}_2 = s_2 \in \mathcal{R}_1\mathcal{R}_2$.

   (c) $\vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1} \Xi_2 \to S_2'$.
   By 4(g)i.

   (d) $\Xi_2 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1} S_2'$.
   By iterating Lemma 2.3.4 from 5c.

   (e) $\vdash_{(\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1)} \Xi_2$.
   By Lemma 2.3.6, using 5d.

   (f) Find Δ_2, if it exists, such that $\Delta_2 \xrightarrow{[\vec{p}_2]} \Xi_2$ in the signature $(\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1)$.
   By Lemma 5.1.3, using 5e, 3(n)i and 3p.

   (g) $\vdash_{(\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1)} \Delta_2$.
   By Definition 2.3.13 (Neighbourhood), using 5f.

   (h) $[\vec{p}_2] : \Delta_2 \to \Xi_2$ in the signature $(\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1)$.
   By Lemma 2.4.7, using 5f.

   (i) $\Delta_2 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1,\mathcal{R}_1} S_2'[\vec{p}_2]$.
   By Lemma 2.4.4 (Substitution lemma), using 5h, 5d and 5g.

   (j) $\Delta_2 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1\mathcal{F}_2,\mathcal{R}_1} S_2'[\vec{p}_2]$.
   By Lemma 5.2.1 using 5i.

   (k) $\forall f \in \mathcal{F}_1\mathcal{F}_2.\mathsf{RED}^{\mathcal{R}_1}(\mathcal{F}_1\mathcal{F}_2(f)) \wedge \mathsf{RED}^{\mathcal{R}_1}_{(\mathcal{F}_1\mathcal{F}_2(f))}(f)$.
   Assume given $f \in \mathcal{F}_1\mathcal{F}_2$. Let $\mathcal{F}_2(f) = \Gamma_f \to T_f$. If $f \in \mathcal{F}_1$, we use 3p. Otherwise $f \in \mathcal{F}_2$. Then by assumption there are no rules for f in $\mathcal{R}_1$. In this case we use Lemma 3.4.13 and 4(g)ii.

   (l) Check $\Delta_2 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1\mathcal{F}_2,\mathcal{R}_1} s_2 \uparrow S_2'[\vec{p}_2]$, and assume success.
   By Theorem 5.1.1, using 5j, 4h and 5k.

   (m) $\Delta_2 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1\mathcal{F}_2,\mathcal{R}_1} s_2 : S_2'[\vec{p}_2]$.
   By Lemma 2.5.2 (Soundness of type inhabitation checking), using 5l and 4h.

   (n) $\Delta_2 \vdash_{\mathcal{D},\mathcal{C},\mathcal{F}_1\mathcal{F}_2,\mathcal{R}_1\mathcal{R}_2} s_2 : S_2'[\vec{p}_2]$.
   By Lemma 5.2.1, using 5m.

   (o) $\vdash \mathcal{D},\mathcal{C},\mathcal{F}_1\mathcal{F}_2,\mathcal{R}_1\mathcal{R}_2$.
   Since f_2' and $f_2'\ \vec{p}_2 = s_2$ was arbitrary from 5a and 5b, by Definition 2.3.16, using 5n.

   (p) $\forall f \in \mathcal{F}_1\mathcal{F}_2.\mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}(\mathcal{F}_1\mathcal{F}_2(f)) \wedge \mathsf{RED}^{\mathcal{R}_1\mathcal{R}_2}_{(\mathcal{F}_1\mathcal{F}_2(f))}(f)$.
   By Lemma 3.6.6 (Key lemma), using $\overset{\sim}{\succ}$ well-founded w.r.t. $\mathcal{R}_1\mathcal{R}_2$, 5o and 4(g)iii.

□

The procedure above can generalized to an arbitrary number of extensions. Then one has to add for each conclusion of a verification step, a list of statements about the reducibility with respect to all the rule sets $\mathcal{R}_1, \ldots, \mathcal{R}_n$ that will be verified throughout the proof, and so the size of the proof will grow rapidly, using the technique presented here. An alternative solution could have been to provide a lemma that $\mathsf{RED}_T^{\mathcal{R}}(t)$ implies $\mathsf{RED}_T^{\mathcal{R}\mathcal{R}'}(t)$, provided that $\mathcal{R}'$ does not define any of the constants defined in $\mathcal{R}$ (omitting the details). However, this seems very hard to prove in our setting.

Even if we have a decision procedure for a given sequence of extensions, we have no decision procedure for $\vdash \Sigma$ given at once, and it doesn't seem possible to prove $\mathsf{RED}(\Sigma)$ from $\overset{\sim}{\succ}$ well-founded and $\vdash \Sigma$ alone. But for each instance of $\vdash \Sigma$ that has been obtained from the stratification procedure above, we have $\mathsf{RED}(\Sigma)$.

Towards a more liberal stratification Note that each $\mathcal{F}_k$ corresponds to a block of mutual definitions. We believe that this procedure allows us to add parts of $\mathcal{R}_k$ in several iterations of the above steps, without extending $\mathcal{F}_k$. Then it should be possible to type-check one branch $f\ \vec{p} = s_1$ of a definition, and then use this computation rule when checking another branch $g\ \vec{q} = s_2$ of the same block of definitions. Then it holds also when f and g are the same constant. Our present argument does not apply for this improvement, but we conjecture that it is a correct extension.

5.3 Consistency

We show that there exists a proposition expressible in our system, that cannot be proved in the system. We introduce the empty data type $\bot$ in the signature with $\mathcal{D}(\bot) = \text{Set}$. Accordingly there are no constructors c such that $\mathcal{C}(c) = \Gamma \to \text{El}\ \bot$.

So far we have overlooked the issue of defining and checking exhaustive pattern matching. To show consistency in the sense that the empty ground type cannot be inhabited, we need to know that a well-typed, closed full application of a defined constant is always a redex. The following property is enforced by exhaustive pattern-matching:

Definition 5.3.1 (Exhaustiveness). The property $\mathsf{EXHAUSTIVE}(\mathcal{F}, \mathcal{R})$ holds iff for all f such that $\mathcal{F}(f) = (x_1 : T_1,\ \ldots\ , x_n : T_n) \to T$, where $n = ar(f)$, if $\vdash t_1 : T_1$ through $\vdash t_n : T_n[t_1,\ \ldots\ , t_{n-1}]$ holds, then there is a rule $f\ p_1\ \ldots\ p_n = s \in \mathcal{R}$, such that $(t_1,\ \ldots\ , t_n) = (p_1\gamma,\ \ldots\ , p_n\gamma)$, for some γ.

Stated as above, the exhaustiveness property is undecidable. Given some constant f of type $\Gamma_f \to T_f$, we cannot know in general if the telescope[3] Γ_f is inhabitable or not, so we cannot know if an empty set of rules for f should be considered exhaustive or not. For instance, consider the denial of Goldbach's conjecture: if we instantiate T_1 to be the natural numbers, and for all natural numbers n, let $T_2[n]$ be the property that if n is an even number greater than two, it cannot be written as the sum of two primes. Most likely, the empty set of computation rules would be exhaustive for f !

However, we can restrict ourselves to *observable* emptiness of the telescope, and allow an empty set of rules only in the case where the empty type appears as one of the types in the given telescope. Then we can decide exhaustiveness (and in the same time disjointness) by adapting for dependent types a more general method that was presented in Huet & Lévy (1991). The algorithm can be sketched as follows: see the set of left-hand sides of the equations as a matrix of patterns. Require that there is a column (choose the leftmost one) starting with only constructors. These constructors are required to exhaust the ground type corresponding to the appropriate column position. Now, partition the sub-matrix to the right of this column by each of these constructors. For each cell the corresponding sub-telescope is now instantiated by the particular constructor and we repeat the same procedure recursively for each such block-matrix.

In McBride (2000) the issue of exhaustiveness is discussed in more detail. McBride also suggests techniques how to give approximations of automatic emptiness-detection.

[3] See Notation 2.1.12, page 27.

Lemma 5.3.2.
If $\mathsf{EXHAUSTIVE}(\mathcal{F}, \mathcal{R})$ *holds, there is no neutral term which is well-typed in the empty context.*

Proof. Assume $\mathsf{EXHAUSTIVE}(\mathcal{F}, \mathcal{R})$. Assume $\vdash b : T$ for some neutral[4] term b and some type T. By Corollary 2.3.9 we have b closed. We have the following two forms for how b may be constructed:

- $b = x\ \vec{t}$, where $\vec{t}$ are normal.
  We have x free in b, which contradicts that b is closed.
- $b = f\ t_1\ \ldots\ t_n$, where $f\ t_1\ \ldots\ t_n$ is normal and $n \geq ar(f)$.
  From $\vdash f\ t_1\ \ldots\ t_n : T$, since t_i are normal, and by Corollary 2.5.6 (Completeness of type inhabitation checking) we have $\vdash f\ t_1\ \ldots\ t_n \uparrow T$.
  By definition then $\mathcal{F}(f) = (x_1 : U_1,\ \ldots\ , x_n : U_n) \to U$, with $\vdash t_i \uparrow U_i[t_1,\ \ldots\ , t_{i-1}]$ and $T \bowtie U[t_1,\ \ldots\ , t_n]$. By Lemma 2.5.2 (Soundness of type inhabitation checking), we have $\vdash t_i : U_i[t_1,\ \ldots\ , t_{i-1}]$. By exhaustiveness $f\ t_1\ \ldots\ t_n$ is a redex, which is a contradiction.

□

Theorem 5.3.3 (Consistency).
Let $\bot : \mathrm{Set}$ *be a data type with no constructors.*
If $\mathsf{RED}(\Sigma)$ *and* $\mathsf{EXHAUSTIVE}(\mathcal{F}, \mathcal{R})$*, then there is no derivation of* $\vdash t : \mathrm{El}\ \bot$.

Proof. Assume $\mathsf{EXHAUSTIVE}(\mathcal{F}, \mathcal{R})$. Assume there is a derivation of $\vdash t : \mathrm{El}\ \bot$. By Lemma 3.5.1 we have $\mathsf{RED}_{(\mathrm{El}\ \bot)}(t)$. By Specification 3.1.8, either

1. there is $c, \vec{t}$ such that $t \Downarrow c\ \vec{t}$ and $\mathcal{C}(c) = (\mathrm{El}\ e_1,\ \ldots\ ,\ \mathrm{El}\ e_n) \to \mathrm{El}\ \bot$.
   This is a contradiction, since there is no such constructor.
2. there is a neutral term b such that $t \Downarrow b$.
   By Lemma 2.4.18 (Subject reduction) we have $\vdash b : \mathrm{El}\ \bot$, but we have from Lemma 5.3.2, that this is a contradiction.

□

[4] See Definition 3.1.1, page 67.

Chapter 6

Discussion

6.1 Conclusions

We have presented a constructive predicative intensional type-theoretic formalism based on a variation of Martin-Löf's logical framework[1], with non-judgemental Church-Rosser convertibility, first-order parameterized algebraic data-types and recursive definitions defined by pattern-matching. The syntactic core language is the untyped lambda-calculus.

We have proved normalization for the proposed system, based on the reducibility method[2] formulated for weak normalization. We have also proved subject reduction and logical consistency.

We proved well-typed terms reducible under the assumption that all constants are reducible. We have defined a relation of *call-instance*, about recursive calls,[3] and proved that the latter condition is satisfied if this relation is well-founded.

To establish well-foundedness, we showed that the *size-change principle for program termination* with a structural term ordering is a sufficient syntactical condition. This shows that type-correctness and the size-change criterion together are sufficient for decidable type-correctness. In Section 5.1 we showed how a group of new constants can be type-checked and added to the signature.

Our approach gives us modularity: once we can prove that some particular syntactic criterion, like the size-change principle, implies well-foundedness, our normalization result follows from it. For instance one could use the recent work of Krauss (2007), who gave a formalization of the size-change principle in Isabelle (Paulson, 1994). Krauss uses the ideas of Manolios & Vroon (2006), which gives an enhancement of the size-change principle, making the analysis more sensitive to branching in the program.

[1] Cf. Nordström *et al.* (1990).
[2] Recall the discussion in Section 1.5.3, page 20.
[3] Cf. Definition 3.6.3, page 80.

6.1.1 Comparison with our Licentiate Thesis

Here follows a list of differences from our past work in Wahlstedt (2004).

- The dependent Cartesian product of a family of sets.

  Without this construction, as we pointed out in Wahlstedt (2004), one could have used a simpler method for proving normalization.

  One can treat all the ground types as one big type, and hence essentially adapt the proof to a simply typed system similar to Gödel's system T. But when we add $\Pi\ A\ F$ in Set, this simply typed method does not apply any more. For instance we can define the following recursive set-valued function, which appears in Hancock (2000), page 52.[4]

  $$\begin{aligned} &\mathrm{F} : \mathrm{Nat} \to \mathrm{Set} \\ &\mathrm{F}\ 0 = \mathrm{Nat} \\ &\mathrm{F}\ (\mathrm{s}\ n) = \Pi\ (\mathrm{F}\ n)\ (\lambda\_.\mathrm{F}\ n) \end{aligned}$$

  Then the type $(n : \mathrm{Nat}) \to \mathrm{F}\ n$ can be defined, and it seems difficult to reduce this type to a simple type.

- Parameterized data-types.

  The present system has first-order parameterized data-types, whereas in the past work we had only first-order *non*-parameterized data types.

- Defined constants with arbitrary types and curried functions.

  In the past work, defined constants were purely first-order, and a type of such a constant was constrained to be a list of first-order data types. Here we allow arbitrary types.

- In our past work, in order to prove reducibility of defined constants, we extended the size-change analysis to deal with both term- and type-level dependencies. We introduced a "type-level" call relation, keeping track of the function symbols appearing in the type declarations of other function symbols. We used then the union of the latter relation and the call-instance relation (here in Definition 3.6.3), and its well-foundedness was shown to imply reducibility of the constants. In this work we instead stratify the signature considering a sequence of extensions. See Section 5.2, where this is described.

6.1.2 Technical difficulties

The main difficulty was in the proof of Lemma 3.6.6, that a well-founded recursion relation implies reducible constants. Instead of proving the goal directly by induction on the typing derivations we prove it by induction on the β-normal fragment of the language, for which type-checking is decidable. We then get the

[4] See also example 3.2.6, page 73.

analytic property, that sub-derivations only concern sub-terms of the conclusion, and this way we get a connection between calls and derivations, enabling us to proceed by well-founded induction on the *call-instance* relation.

6.1.3 Comparison with CAC

The Calculus of Algebraic Constructions (Blanqui, 2005) is the system closest to ours. It is stronger than our system: besides that it is impredicative, it accepts more general kinds of rewriting systems, for instance non-linear and overlapping patterns. It also allows the definition of recursive higher-order data types. Blanqui proved strong normalization, and therefore his reducibility predicates are different from ours. What is more significant is that our only assumption about recursive definitions is that the call relation is *well-founded*, which makes our result independent of particular syntactic constraints. For instance, our result applies to size-change termination, whereas CAC does not (at least not without further investigations). Blanqui's system is based on an extension of the *General Schema* (Jouannaud & Okada, 1991), and it allows a large class of rewriting systems to be accepted. However, it involves also a considerable system of constraints to be satisfied for various kinds of rewriting systems, which seems hard to give a simple presentation. In this sense his approach is less modular.[5]

6.2 Future work

An obvious further direction is to develop more examples in our system, and to implement a type-checker. An interesting case study would be to translate the proof of Hancock (2000) that ϵ_0 is well-founded. This was formalized in Agda (Coquand, 2006), without any use of transfinite recursion. Another task to consider in connection to our proof is to formalize it. To do this would be an extensive effort, requiring more time resources than is available in the scope of this dissertation, but it would certainly be interesting. Yet another natural follow-up work would be to increase the strength of our system. Instead of having a "hard-wired" data-type Π as in the present work, we should permit strictly positive higher-order recursive parameterized data-types with dependently typed constructors. It would also be interesting to incorporate inductive-recursive definitions. Among other further directions we give a list below:

- Method of proving reducibility of constants.
  We believe that our method of proving reducibility of constants could be applied to other problems were recursive constants are involved. For functional programming, it could be useful to extend the work of Danielsson *et al.* (2006) with recursive definitions, there considering PER semantics instead of reducibility.

[5] However, a similar but more general approach is taken in Blanqui (2003).

- Decrease strength.
  Instead of increasing strength, an investigation in the opposite direction could be to remove Π from Set and see what can be done in this system.

- Type-checking of the signature.
  We would like to be able to stratify the signature following a more liberal discipline than in the present work. See the discussion after Theorem 5.2.2, page 98. As a further relaxation, it should be possible to consider a group of mutually defined constants to have types that depend on constants declared earlier in the same group. It would also be interesting to investigate further under what conditions one can allow the execution of certain rules for a given definition when checking the type of some other rules of the same definition. In connection to this, it would be a challenge to see if there are examples that motivate such a type-checking discipline.

- Justification of reducibility predicates.
  The justification presented in Section 3.2, page 70 is yet just a rough sketch. To our knowledge only little work has been done in this direction, besides the analysis of Scott (1975) and Aczel (1980), that were done with simple types. It would certainly be worthwhile to develop more rigorous justifications of such definitions in the presence of dependent types. Also, as pointed out by Aczel, one should try to do this constructively.

- Develop a decision procedure for exhaustive pattern-matching.
  In its general form this property is undecidable with dependent types, but it would certainly be worth investigating appropriate restrictions under which this property is decidable. See the discussion in Section 5.3, page 104.

- Prove strong normalization.
  It should be possible to adapt our proof for strong normalization using a domain model as of Coquand & Spiwack (2006). We conjecture that if we adapt our notion of reducibility for proving strong normalization, the well-foundedness of the call-instance relation (Definition 3.6.3) will coincide with reducibility.

- A constructive understanding of the size-change principle.
  The proof by Lee *et al.* (2001) of the decidability of SCT uses Ramsey's theorem, the infinite binary version, which is also used in the theory of Büchi automata. Looking at what is actually used from the proof of Ramsey's theorem, we believe that a weaker, constructively valid method can be extracted. A constructive version of Ramsey's theorem has been given by Fridlender (1997).

Bibliography

Abel, Andreas. 1999. *A Semantic Analysis of Structural Recursion.* M.Phil. thesis, Ludwig-Maximilians-University Munich.

Abel, Andreas. 2004. Termination Checking with Types. *RAIRO – Theoretical Informatics and Applications*, **38**(4), 277–319. Special Issue: Fixed Points in Computer Science (FICS'03).

Abel, Andreas. 2006a. Implementing a Normalizer Using Sized Heterogeneous Types. *In:* McBride, Connor, & Uustalu, Tarmo (eds), *Workshop on Mathematically Structured Functional Programming, MSFP 2006, Kuressaare, Estonia, July 2, 2006.* electronic Workshop in Computing (eWiC). The British Computer Society (BCS).

Abel, Andreas. 2006b. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* Ph.D. thesis, Ludwig-Maximilians-Universität München.

Abel, Andreas. 2006c. Semi-continuous Sized Types and Termination. *Pages 72–88 of:* Ésik, Zoltán (ed), *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 21-24, 2006, Proceedings.* Lecture Notes in Computer Science, vol. 4207. Springer-Verlag.

Abel, Andreas. 2006d. Towards Generic Programming with Sized Types. *Pages 10–28 of:* Uustalu, Tarmo (ed), *Mathematics of Program Construction: 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006. Proceedings.* Lecture Notes in Computer Science, vol. 4014. Springer-Verlag.

Abel, Andreas, & Altenkirch, Thorsten. 2002. A Predicative Analysis of Structural Recursion. *Journal of Functional Programming*, **12**(1), 1–41.

Ackermann, Wilhelm. 1928. On Hilbert's construction of the real numbers. English translation in van Heijenoort (1977).

Aczel, Peter. 1980. Frege structures and the notions of proposition, truth and set. *Pages 31–59 of: The Kleene Symposium (Proc. Sympos., Univ. Wisconsin, Madison, Wis., 1978).* Stud. Logic Foundations Math., vol. 101. Amsterdam: North-Holland.

Aczel, Peter, & Rathjen, Michael. 1997. *Notes on Constructive Set Theory.* Tech. rept. Institut Mittag-Leffler, The Royal Swedish Academy of Sciences. ISSN 1103-467X, Preprint series: Mathematical Logic - 2000/2001, No. 40.

Andrews, Peter. Fall 2006. Church's Type Theory. *In:* Zalta, Edward N. (ed), *The Stanford Encyclopedia of Philosophy.* `http://plato.stanford.edu/archives/fall2006/entries/type-theory-church/%`.

Appel, K., & Haken, W. 1976. Every planar graph is four colourable. *Bulletin of the American Mathematical Society*, **82**(5).

Augustsson, Lennart. 1985. Compiling Pattern Matching. *Pages 368–381 of: Functional Programming Languages and Computer Architecture.* Lecture Notes in Computer Science, vol. 201. Berlin: Springer-Verlag.

Backhouse, R. C. 1986. *On the Meaning and Construction of the Rules in Martin-Löf's Theory of Types.* Computing Science Notes CS 8606. Department of Mathematics and Computing Science, University of Groningen.

Backhouse, Roland, & Chisholm, Paul. 1989. Do-It-Yourself Type Theory. *Formal Aspects of Computing*, **1**(1), 19–84.

Barthe, G., Grégoire, B., & Pastawski, F. 2006. Type-based termination of recursive definitions in the Calculus of Inductive Constructions. *In: Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06).* Lecture Notes in Artificial Intelligence. Springer-Verlag. Available at `http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Publi/CICso%mbrero.pdf.gz`.

Barthe, Gilles, Frade, Maria João, Giménez, E., Pinto, Luis, & Uustalu, Tarmo. 2004. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, **14**(1), 97–141. Available at `http://dx.doi.org/10.1017/S0960129503004122`.

Ben-Amram, Amir M. 2002. General Size-Change Termination and Lexicographic Descent. *Pages 3–17 of:* Mogensen, Torben, Schmidt, David, & Sudborough, I. Hal (eds), *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones.* Lecture Notes in Computer Science, vol. 2566. Springer-Verlag.

Berger, U. 2005. Continuous Semantics for Strong Normalization. *Pages 23–34 of:* Cooper, S.B., Löwe, B., & Torenvliet, L. (eds), *CiE 2005: New Computational Paradigms.* Lecture Notes in Computer Science, vol. 3526.

Berry, Gérard. 1978. Stable Models of Typed lambda-Calculi. *Pages 72–89 of:* Ausiello, Giorgio, & Böhm, Corrado (eds), *Automata, Languages and Programming, Fifth Colloquium, Udine, Italy, July 17-21, ICALP 1978, Proceedings.* Lecture Notes in Computer Science, vol. 62. Springer.

Berry, Gérard. 1979. *Modeles completement adequats et stable de lambda-calculs.* Ph.D. thesis, Universite Paris VII.

Björk, Magnus. 2000. *Compiling Embedded ML.* M.Phil. thesis, Chalmers University of Technology.

Blanqui, Frédéric. 2003. Inductive Types in the Calculus of Algebraic Constructions. *Pages 46–59 of:* Hofmann, Martin (ed), *TLCA*. Lecture Notes in Computer Science, vol. 2701. Springer.

Blanqui, Frédéric. 2004. A type-based termination criterion for dependently-typed higher-order rewrite systems. *Pages 24–39 of: Rewriting techniques and applications*. Lecture Notes in Comput. Sci., vol. 3091. Berlin: Springer.

Blanqui, Frédéric. 2005. Definitions by Rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, **15**(1), 37–92.

Blazy, Sandrine, Dargaye, Zaynah, & Leroy, Xavier. 2006. Formal Verification of a C Compiler Front-End. *Pages 460–475 of: FM 2006: Int. Symp. on Formal Methods*. Lecture Notes in Computer Science, vol. 4085. Springer-Verlag.

Bove, A., & Capretta, V. 2005. Modelling General Recursion in Type Theory. *Mathematical Structures in Computer Science*, **15**(February), 671–708. Cambridge University Press.

Bove, Ana. 2002. *General Recursion in Type Theory.* Ph.D. thesis, Chalmers University of Technology.

Breazu-Tannen, V. 1988. Combining Algebra and Higher-Order Types. *In: Proc. LICS'88.*

Burstall, R. M. 1969. Proving Properties of Programs by Structural Induction. *The Computer Journal*, **12**(1), 41–48.

Cartmell, John. 1986. Generalised Algebraic Theories and Contextual Categories. *Annals of Pure and Applied Logic*, **32**(3), 209–243.

Church, Alonzo. 1941. *The Calculi of Lambda-Conversion.* Annals of Mathematics Studies, no. 6. Princeton, N. J.: Princeton University Press.

Colson, Loïc. 1989. About Primitive Recursive Algorithms. *Pages 194–206 of:* Ausiello, Giorgio, Dezani-Ciancaglini, Mariangiola, & Rocca, Simona Ronchi Della (eds), *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings.* Lecture Notes in Computer Science, vol. 372. Springer.

Constable, R. L., & Mendler, N. P. 1985. Recursive definitions in type theory. *Pages 61–78 of: Logics of programs (Brooklyn, N.Y., 1985)*. Lecture Notes in Comput. Sci., vol. 193. Berlin: Springer.

Coquand, Catarina. 1998. A realizability interpretation of Martin-Löf's type theory. *In:* Sambin, G., & Smith, J. (eds), *Twenty-Five Years of Constructive Type Theory.* Oxford University Press.

Coquand, Catarina. 2006. *The Agda Home Page.* Department of Computer Science, Chalmers University of Technology and Göteborgs Universitet. `http://www.cs.chalmers.se/~catarina/agda/`.

Coquand, Thierry. 1985. *Une Théorie des Constructions.* Ph.D. thesis, Université Paris VII.

Coquand, Thierry. 1992. Pattern Matching with Dependent Types. *Pages 71–84 of:* Nordström, B., Pettersson, K., & Plotkin, G. (eds), *Informal Proc. of Wksh. on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992.* Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ. `http://www.cs.chalmers.se/~coquand/pattern.ps`.

Coquand, Thierry, & Paulin-Mohring, Christine. 1990. Inductively defined types. *In:* Martin-Löf, P., & Mints, G. (eds), *Proceedings of Colog'88.* Lecture Notes in Computer Science, vol. 417. Springer-Verlag.

Coquand, Thierry, & Spiwack, Arnaud. 2006. A Proof of Strong Normalisation using Domain Theory. *Pages 307–316 of: LICS.* IEEE Computer Society.

Cornes, Cristina. 1997 (Nov.). *Conception d'un langage de haut niveau de representation de preuves: Récurrence par filtrage de motifs; Unification en présence de types inductifs primitifs; Synthèse de lemmes d'inversion.* Thèse de Doctorat, Université Paris 7. Available at `http://pauillac.inria.fr/~cornes/Papers/thesis.ps.gz`.

Curry, Haskell B., & Feys, R. 1958. *Combinatory Logic.* Vol. 1. North Holland.

Danielsson, Hughes, Jansson, & Gibbons. 2006. Fast and Loose Reasoning is Morally Correct. *SPNOTICES: ACM SIGPLAN Notices*, **41**.

de Bruijn, N. G. 1968. The Mathematical Language AUTOMATH, Its Usage, and Some of Its Extensions. *Pages 29–61 of:* Laudet, M. (ed), *Proceedings of the Symposium on Automatic Demonstration.* Versailles, France: Springer-Verlag LNM 125.

Dedekind, Richard. 1888. *Was sind und was sollen die zahlen ?* Braunschweig: F. Vieweg. Translated by W.W. Beman and W. Ewald in Ewald (1996) 787–832.

Dybjer, Peter. 1994. Inductive Families. *Formal Aspects of Computing*, **6**(4), 440–465.

Dybjer, Peter. 2000. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symbolic Logic*, **65**(2), 525–549.

Ewald, William Bragg (ed). 1996. *From Kant to Hilbert: a source book in the foundations of mathematics. Vol. I, II.* Oxford Science Publications. New York: The Clarendon Press Oxford University Press. Compiled, edited and with introductions by William Ewald.

Feferman, S. (ed). 1986. *Kurt Gödel Collected Works.* Oxford, UK: Oxford University Press.

Fridlender, Daniel. 1997. *Higman's Lemma in Type Theory.* PhD thesis, Dept. of Computing Science, Chalmers Univ. of Techn. and Univ. of Göteborg.

Gentzen, Gerhard. 1969. *The collected papers of Gerhard Gentzen.* Edited by M. E. Szabo. Studies in Logic and the Foundations of Mathematics. Amsterdam: North-Holland Publishing Co.

Giesl, Jürgen, Thiemann, René, Schneider-Kamp, Peter, & Falke, Stephan. 2004. Automated Termination Proofs with AProVE. *Pages 210–220 of:* van Oostrom, Vincent (ed), *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings.* Lecture Notes in Computer Science, vol. 3091. Springer-Verlag.

Giménez, Eduardo. 1995. Codifying guarded definitions with recursive schemes. *Pages 39–59 of: Types for proofs and programs (Båstad, 1994).* Lecture Notes in Comput. Sci., vol. 996. Berlin: Springer.

Giménez, Eduardo. 1998. Structural recursive definitions in type theory. *Pages 397–408 of: Automata, languages and programming (Aalborg, 1998).* Lecture Notes in Comput. Sci., vol. 1443. Berlin: Springer.

Girard, J-Y., Lafont, Y., & Taylor, P. 1989. Proofs and Types. Cambridge University Press.

Girard, Jean-Yves. 1971. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. *Pages 63–92 of: Proceedings 2nd Scandinavian Logic Symposium.* Amsterdam: North-Holland.

Gonthier, Georges. 2004. *A computer-checked proof of the Four Colour Theorem.* Available at `http://research.microsoft.com/~gonthier/`.

Hancock, Peter. 2000. *Ordinals and Interactive Programs.* Ph.D. thesis, University of Edinburgh.

Herbrand, Jacques. 1931. On the Consistency of Arithmetic. English translation in van Heijenoort (1977).

Hilbert, David. 1925. Über das Unendliche. *Mathematische Annalen*, **95**, 161–90. Translated by Stefan Bauer-Mengelberg and Dagfinn Føllesdal in van Heijenoort (1977).

Howard, W. 1980. The formulae-as-types notion of construction. *Pages 479–490 of:* J. P. Seldin and J. R. Hindley (ed), *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism.* Academic Press. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard, 1979.

Huet, Gérard P., & Lévy, Jean-Jacques. 1991. Computations in Orthogonal Rewriting Systems, II. *Pages 415–443 of: Computational Logic - Essays in Honor of Alan Robinson.*

Hughes, John, Pareto, Lars, & Sabry, Amr. 1996. Proving the Correctness of Reactive Systems using Sized Types. *In:* Jr, Guy L. Steele (ed), *Principles of Programming Languages*, vol. 23. St Petersburg, Florida: ACM.

Jouannaud, & Okada. 1997. Abstract Data Type Systems. *TCS: Theoretical Computer Science*, **173**.

Jouannaud, Jean-Pierre, & Okada, Mitsuhiro. 1991. Executable Higher-Order Algebraic Specification Languages. *Pages 350–361 of: Proceedings, 6th Symposium on Logic in Computer Science.* IEEE.

Kleene, S. C. 1938. On a notation for ordinal numbers. *Journal of Symbolic Logic*, **3**, 150–155.

Kleene, Stephen Cole. 1945. On the Interpretation of Intuitionistic Number Theory. *The Journal of Symbolic Logic*, **10**(4), 109–124.

Kolmogorov, Andrei Nikolaevich. 1932. Zur Deutung der intuitionistischen Logik. *Mathematischen Zeitschrift*, **35**, 58–65. English translation in P. Mancosu, Ed., From Brouwer to Hilbert : the debate on the foundations of mathematics in the 1920s, Oxford University Press, 1998.

Krauss, Alexander. 2007. Certified Size-Change Termination. *Pages 460–476 of:* Pfenning, Frank (ed), *Automated Deduction — CADE-21 International Conference.* Lecture Notes in Computer Science, vol. 4603. Springer. To appear.

Landin, P. J. 1964. The Mechanical Evaluation of Expressions. *The Computer Journal*, **6**(4), 308–320.

Landin, P. J. 1966. The Next 700 Programming Languages. *Communications of the ACM*, **9**(3), 157–164. Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, August 8–12, 1965.

Lee, C. S., Jones, N. D., & Ben-Amram, A. M. 2001. The Size-Change Principle for Program Termination. *Pages 81–92 of: Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages.* New York: ACM.

Leroy, Xavier. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *Pages 42–54 of: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York, NY, USA: ACM Press.

Leroy, Xavier, Doligez, Damien, Garrigue, Jacques, Rémy, Didier, & Vouillon, Jérôme. 2004. *The Objective Caml system, Documentation and user's manual.* release 3.09 edn. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

Manolios, Panagiotis, & Vroon, Daron. 2006. Termination Analysis with Calling Context Graphs. *Pages 401–414 of:* Ball, Thomas, & Jones, Robert B. (eds), *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings.* Lecture Notes in Computer Science, vol. 4144. Springer.

Martin-Löf, Per. 1971. Hauptsatz for the intuitionistic theory of iterated inductive definitions. *Pages 179–216. Studies in Logic and the Foundations of Mathematics, Vol. 63 of: Proceedings of the Second Scandinavian Logic Symposium (Univ. Oslo, Oslo, 1970).* Amsterdam: North-Holland.

Martin-Löf, Per. 1971. *A Theory of Types.* Tech. rept. 71-3. University of Stockholm.

Martin-Löf, Per. 1972. An Intuitionistic Theory of Types. *In:* Sambin, G., & Smith, J. (eds), *Twenty-Five Years of Constructive Type Theory.* Oxford University Press. Edited in 1998.

Martin-Löf, Per. 1984. *Intuitionistic Type Theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.* Bibliopolis, Napoli.

Martin-Löf, Per. 1992. Substitution Calculus. September. Lecture notes from the logic seminar, University of Stockholm.

Martin-Löf, Per. 1996. On the Meaning of the Logical Constants and the Justifications of the Logical Laws. *Nordic Journal of Philosophical Logic*, **1**(1), 3–10.

Matthes, Ralph. 1998 (May). *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types.* Ph.D. thesis, Ludwig-Maximilians-University.

McBride, Conor. 2000. *Dependently Typed Functional Programs and their Proofs.* Ph.D. thesis, LFCS, University of Edinburgh, Edinburgh, Scotland. Available at `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

McBride, Conor, & McKinna, James. 2004. The view from the left. *Journal of Functional Programming*, **14**(1), 69–111.

McCarthy, John. 1962. Checking mathematical proofs by computer. *In: Proceedings Symposium on Recursive Function Theory.* American Mathematical Society.

McCarthy, John. 1963a. A basis for a mathematical theory of computations. *Pages 33–70 of:* Braffort, & Hershberg (eds), *Computer Programming and Formal Systems.*

McCarthy, John. 1963b. Towards a mathematical science of computation. *Pages 21–28 of: Information Processing: The 1962 IFIP Congress.*

Mendler, Nax P. 1987. Recursive Types and Type Constraints in Second-Order Lambda Calculus. *Pages 30–36 of: Symposium on Logic in Computer Science (LICS '87).* IEEE Computer Society.

Milner, Robin, Tofte, Mads, & Harper, Robert. 1990. *The Definition of Standard ML.* MIT Press.

Müller, Fritz. 1992. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, **41**(6), 293–299.

Newman, James R. 1956. *The world of mathematics.* Simon and Shuster, Inc, New York.

Nordström, Bengt, Petersson, Kent, & Smith, Jan M. 1990. *Programming in Martin-Löf's Type Theory* . Oxford University Press.

Odifreddi, Piergiorgio. Fall 2006. Recursive Functions. *In:* Zalta, Edward N. (ed), *The Stanford Encyclopedia of Philosophy.* `http://plato.stanford.edu/archives/fall2006/entries/recursive-functions%/`.

Pareto, Lars. 2000. *Types for crash prevention.* Ph.D. thesis, Chalmers University of Technology.

Paulson, Lawrence C. 1994. *Isabelle: A Generic Theorem Prover.* Lecture Notes in Computer Science, vol. 828. Springer Verlag.

Péter, Rózsa. 1967. *Recursive functions.* Third revised edition. Translated from the German by István Földes. New York: Academic Press.

Peyton Jones, Simon. 2003. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press. ISBN 0521826144.

Pfenning, F., & Paulin-Mohring, C. 1990. Inductively defined types in the Calculus of Constructions. *In: Proceedings of Mathematical Foundations of Programming Semantics.* LNCS 442. Springer-Verlag.

Pollack, Robert. 1995. Polishing Up the Tait–Martin-Löf Proof of the Church-Rosser Theorem. *In: Proceedings of De Wintermöte '95.* Department of Computing Science, Chalmers Univ. of Technology, Göteborg, Sweden. `http://www.dcs.ed.ac.uk/home/rap/export/churchrosser.ps.gz`.

Prawitz, Dag. 1965. *Natural deduction. A proof-theoretical study.* Acta Universitatis Stockholmiensis. Stockholm Studies in Philosophy, No. 3. Stockholm: Almqvist & Wiksell.

Schönfinkel, Moses. 1924. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, **92**, 305–316. Translated into English and republished as "On the building blocks of mathematical logic" in (van Heijenoort, 1977, pp. 355–366).

Scott, Dana S. 1975. Combinators and classes. *Pages 1–26 of:* Böhm, Corrado (ed), *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 25-27, 1975.* Lecture Notes in Computer Science, vol. 37. Springer-Verlag.

Seldin, Jonathan. 2002 (May). Curry's anticipation of the types used in programming languages. *Pages 148–163 of: Proceedings of the Annual Meeting of the Canadian Society for History and Philosophy of Mathematics, Toronto, Ontario.*

Smith, Jan. 1983. The Identification of Propositions and types in Martin-Löf's Type Theory: A Programming Example. *Pages 445–456 of: Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory.* London, UK: Springer-Verlag.

Smith, Jan M. 1989. Propositional Functions and Families of Types. *Notre Dame Journal of Formal Logic*, **30**(3), 442–458.

Tait, William W. 1967. Intensional Interpretation of Functionals of Finite Type. *Pages 198–212 of: Journal of Symbolic Logic*, vol. 32:2.

Tait, William W. 1975. A realizability interpretation of the theory of species. *Pages 240–251. Lecture Notes in Math., Vol. 453 of: Logic Colloquium (Boston, Mass., 1972-1973).* Berlin: Springer.

The Coq Development Team. 2006. *The Coq Proof Assistant : Reference Manual : Version 8.0.* Tech. rept. INRIA, Roquencourt, France. Available at `http://coq.inria.fr/doc/main.html`.

Uustalu, Tarmo, & Vene, Varmo. 2002. Least and greatest fixed points in intuitionistic natural deduction. *Theoret. Comput. Sci.*, **272**(1-2), 315–339. Theories of types and proofs (Tokyo, 1997).

van Heijenoort, Jean (ed). 1977. *From Frege to Gödel, a Source Book in Mathematical Logic, 1879-1931.* 3 edn. Cambridge: Harvard University Press.

Vogel, Helmut. 1976. Ein starker Normalisationssatz für die bar-rekursiven Funktionale. *Arch. Math. Logik Grundlagenforsch.*, **18**(1–2), 81–84.

Wahlstedt, David. 2000. *Detecting Termination Using Size-Change in Parameter Values.* M.Phil. thesis, Göteborgs Universitet. `http://www.cs.chalmers.se/~davidw/`.

Wahlstedt, David. 2004. *Type Theory with First-Order Data Types and Size-Change Termination.* Tech. rept. Chalmers University of Technology. Licentiate thesis, No. 36L.