Programming in Martin-Löf's Type Theory

An Introduction

Bengt Nordström
Kent Petersson
Jan M. Smith

Department of Computing Sciences
University of Göteborg / Chalmers
S-412 96 Göteborg
Sweden

This book was published by Oxford University Press in 1990. It is now out of print. This version is available from `www.cs.chalmers.se/Cs/Research/Logic`.

Preface

It is now 10 years ago that two of us took the train to Stockholm to meet Per Martin-Löf and discuss his ideas on the connection between type theory and computing science. This book describes different type theories (theories of types, polymorphic and monomorphic sets, and subsets) from a computing science perspective. It is intended for researchers and graduate students with an interest in the foundations of computing science, and it is mathematically self-contained.

We started writing this book about six years ago. One reason for this long time is that our increasing experience in using type theory has made several changes of the theory necessary. We are still in this process, but have nevertheless decided to publish the book now.

We are, of course, greatly indebted to Per Martin-Löf; not only for creating the subject of this book, but also for all the discussions we have had with him. Beside Martin-Löf, we have discussed type theory with many people and we in particular want to thank Samson Abramsky, Peter Aczel, Stuart Anderson, Roland Backhouse, Bror Bjerner, Robert Constable, Thierry Coquand, Peter Dybjer, Roy Dyckhoff, Gerard Huet, Larry Paulson, Christine Paulin-Mohring, Anne Salvesen, Björn von Sydow, and Dan Synek. Thanks to Dan Synek also for his co-authorship of the report which the chapter on trees is based on.

Finally, we would like to thank STU, the National Swedish Board For Technical Development, for financial support.

Bengt Nordström, Kent Petersson and Jan Smith

Göteborg, Midsummer Day 1989.

Contents

Chapter 1

Introduction

In recent years several formalisms for program construction have been introduced. One such formalism is the type theory developed by Per Martin-Löf. It is well suited as a theory for program construction since it is possible to express both specifications and programs within the same formalism. Furthermore, the proof rules can be used to derive a correct program from a specification as well as to verify that a given program has a certain property. This book contains an introduction to type theory as a theory for program construction.

As a programming language, type theory is similar to typed functional languages such as Hope [18] and ML [44], but a major difference is that the evaluation of a well-typed program always terminates. In type theory it is also possible to write specifications of programming tasks as well as to develop provably correct programs. Type theory is therefore more than a programming language and it should not be compared with programming languages, but with formalized programming logics such as LCF [44] and PL/CV [24].

Type theory was originally developed with the aim of being a clarification of constructive mathematics, but unlike most other formalizations of mathematics type theory is not based on first order predicate logic. Instead, predicate logic is interpreted within type theory through the correspondence between propositions and sets [28, 52]. A proposition is interpreted as a set whose elements represent the proofs of the proposition. Hence, a false proposition is interpreted as the empty set and a true proposition as a non-empty set. Chapter 2 contains a detailed explanation of how the logical constants correspond to sets, thus explaining how a proposition could be interpreted as a set. A set cannot only be viewed as a proposition; it is also possible to see a set as a problem description. This possibility is important for programming, because if a set can be seen as a description of a problem, it can, in particular, be used as a specification of a programming problem. When a set is seen as a problem, the elements of the set are the possible solutions to the problem; or similarly if we see the set as a specification, the elements are the programs that satisfy the specification. Hence, set membership and program correctness are the same problem in type theory, and because all programs terminate, correctness means total correctness.

One of the main differences between the type theory presentation in this book and the one in [69] is that we use a uniform notation for expressions. Per Martin-Löf has formulated a theory of mathematical expressions in general, which is presented in chapter 3. We describe how arbitrary mathematical expressions are formed and introduce an equality between expressions. We also show how defined constants can be introduced as abbreviations of more complicated expressions.

In Part I we introduce a polymorphic version of type theory. This version is the same as the one presented by Martin-Löf in Hannover 1979 [69] and in his book *Intuitionistic Type Theory* [70] except that we use an intensional version of the equality.

Type theory contains rules for making judgements of the following four forms:

A is a set
A_1 and A_2 are equal sets
a is an element in the set A
a_1 and a_2 are equal elements in the set A

The semantics of type theory explains what judgements of these forms mean. Since the meaning is explained in a manner quite different from that which is customary in computer science, let us first describe the context in which the meaning is explained. When defining a programming language, one often explains its notions in terms of mathematical objects like sets and functions. Such a definition takes for granted the existence and understanding of these objects. Since type theory is intended to be a fundamental conceptual framework for the basic notions of constructive mathematics, it is infeasible to explain the meaning of type theory in terms of some other mathematical theory. The meaning of type theory is explained in terms of computations. The first step in this process is to define the syntax of programs and how they are computed. We first introduce the canonical expressions which are the expressions that can be the result of programs. When they are defined, it is possible to explain the judgements, first the assumption-free and then the hypothetical. A set is explained in terms of canonical objects and their equality relation, and when the notion of set is understood, the remaining judgement forms are explained. Chapter 4 contains a complete description of the semantics in this manner.

The semantics of the judgement forms justifies a collection of general rules about assumptions, equality and substitution which is presented in chapter 5.

In the following chapters (7 – 17), we introduce a collection of sets and set forming operations suitable both for mathematics and computer science. Together with the sets, the primitive constants and their computation rules are introduced. We also give the rules of a formal system for type theory. The rules are formulated in the style of Gentzen's natural deduction system for predicate logic and are justified from

- the semantic explanations of the judgement forms,
- the definitions of the sets, and
- the computation rules of the constants.

We do not, however, present justifications of all rules, since many of the justifications follow the same pattern.

There is a major disadvantage with the set forming operations presented in part I because programs sometimes will contain computationally irrelevant parts. In order to remedy this problem we will in part II introduce rules which makes it possible to form subsets. However, if we introduce subsets in the same way as we introduced the other set forming operations, we cannot justify a satisfactory elimination rule. Therefore, we define a new theory, the subset theory, and explain the judgements in this new theory by translating them into judgements in the basic theory, which we already have given meaning to in part I.

In part III, we briefly describe a theory of types and show how it can be used as an alternative way of providing meaning to the judgement forms in type theory. The origin of the ideas in this chapter is Martin-Löf's analysis of the notions of proposition, judgement and proof in [71]. The extension of type theory presented is important since it makes it possible to introduce more general assumptions within the given formalism. We also show how the theory of types could be used as a framework for defining some of the sets which were introduced in part I.

In part IV we present some examples from logic and programming. We show how type theory can be used to prove properties of programs and also how to formally derive programs for given specifications. Finally we describe how abstract data types can be specified and implemented in type theory.

1.1 Using type theory for programming

Type theory, as it is used in this book, is intended as a theory for program construction. The programming development process starts with the task of the program. Often, this is just existing in the head of the programmer, but it can also exist explicitly as a specification that expresses what the program is supposed to do. The programmer, then, either directly writes down a program and proves that it satisfies the given specification, or successively derives a program from the specification. The first method is called *program verification* and the second *program derivation* . Type theory supports both methods and it is assumed that it is the programmer who bridges the gap between the specification and the program.

There are many examples of correctness proofs in the literature and proofs done in Martin-Löf's type theory can be found in [20, 75, 82]. A theory which is similar to type theory is Huet and Coquand's Calculus of Constructions [27] and examples of correctness proofs in this theory can be found in [74].

There are fewer examples of formal program derivations in the literature. Manna and Waldinger have shown how to derive a unification algorithm using their tableau method [63] and there are examples developed in Martin-Löf's type theory in Backhouse et al [6] and in the Theory of Constructions in Paulin-Mohring [80]. A formal derivation of the partitioning problem using type theory is presented in [87]; a slightly changed version of this derivation is also presented in chapter 22.

In the process of formal program development, there are two different stages and usually two different languages involved. First, we have the specification process and the specification language, and then the programming process and the programming language. The specification process is the activity of finding and formulating the problem which the program is to solve. This process is not dealt with in this book. We assume that the programmer knows what problem to solve and is able to express it as a specification. A specification is in type theory expressed as a set, the set of all correct programs satisfying the specification. The programming process is the activity of finding and formulating a program which satisfies the specification. In type theory, this means that the programmer constructs an element in the set which is expressed by the specification. The programs are expressed in a language which is a functional programming language. So it is a programming language without assignments and other side effects. The process of finding a program satisfying a specification can be formalized in a programming logic, which has rules for deducing the correctness of programs. So the formal language of type theory is used as a programming language, a specification language and a programming logic.

The language for sets in type theory is similar to the type system in programming languages except that the language is much more expressive. Besides the usual set forming operations which are found in type systems of programming languages (Bool, $A+B$, $A \to B$, $A \times B$, $\mathsf{List}(A)$, etc.) there are operations which make it possible to express properties of programs using the usual connectives in predicate logic. It is possible to write down a specification without knowing if there is a program satisfying it. Consider for example

$$(\exists a \in \mathsf{N}^+)(\exists b \in \mathsf{N}^+)(\exists c \in \mathsf{N}^+)(\exists n \in \mathsf{N}^+)(n > 2 \,\&\, a^n + b^n = c^n)$$

which is a specification of a program which computes four natural numbers such that Fermat's last theorem is false. It is also possible that a specification is satisfied by several different programs. Trivial examples of this are "specifications" like N, $\mathsf{List}(\mathsf{N}) \to \mathsf{List}(\mathsf{N})$ etc. More important examples are the sorting problem (the order of the elements of the output of a sorting program should not be uniquely determined by the input), compilers (two compilers producing different code for the same program satisfies the same specification as long as the code produced computes the correct input-output relation), finding an index of a maximal element in an array, finding a shortest path in a graph etc.

The language to express the elements in sets in type theory constitutes a typed functional programming language with lazy evaluation order. The program forming operations are divided into constructors and selectors. Constructors are used to construct objects in a set from other objects, examples are 0, succ, pair, inl, inr and λ . Selectors are used as a generalized pattern matching: What in ML is written as

```
case p of (x,y) => d
```

is in type theory written as

$$\mathsf{split}(p, (x, y)d)$$

and if we in ML define the disjoint union by

```
datatype ('A,'B)Dunion = inl of 'A | inr of 'B
```

then the ML-expression

```
case p of inl(x) => d
        | inr(y) => e
```

is in type theory written as

$$\mathsf{when}(p, (x)d, (y)e)$$

General recursion is not available. Iteration is expressed by using the selectors associated with the inductively defined sets like N and $\mathsf{List}(A)$. For these sets, the selectors work as operators for primitive recursion over the set. For instance, to find a program $f(n)$ on the natural numbers which solves the equations

$$\begin{cases} f(0) & = & d \\ f(n+1) & = & h(n, f(n)) \end{cases}$$

one uses the selector natrec associated with the natural numbers. The equations are solved by making the definition:

$$f(n) \equiv \mathsf{natrec}(n, d, (x,y)h(x,y))$$

In order to solve recursive equations which are not primitive recursive, one must use the selectors of inductive types together with high order functions. Examples of how to obtain recursion schemas other than the primitive ones are discussed by Paulson in [84] and Nordström [77].

Programs in type theory are computed using lazy evaluation. This means that a program is considered to be evaluated if it is on the form

$$c(e_1, \ldots, e_n)$$

where c is a constructor and $e_1, \ldots, e_n$ are expressions. Notice that there is no requirement that the expressions $e_1, \ldots, e_n$ must be evaluated. So, for instance, the expression $\mathsf{succ}(2^{2^{2^2}})$ is considered to be evaluated, although it is not fully evaluated. If a program is on the form

$$s(e_1, \ldots, e_n)$$

where s is a selector, it is usually computed by first computing the value of the first argument. The constructor of this value is then used to decide which of the remaining arguments of s which is used to compute the value of the expression.

When a user wants to derive a correct program from a specification, she uses a programming logic. The activity to derive a program is similar to proving a theorem in mathematics. In the top-down approach, the programmer starts with the task of the program and divides it into subtasks such that the programs solving the subtasks can be combined into a program for the given task. For instance, the problem of finding a program satisfying B can be reduced to finding a program satisfying A and a function taking an arbitrary program satisfying A to a program satisfying B. Similarly, the mathematician starts with the proposition to be proven and divides it into other propositions such that the proofs of them can be combined into a proof of the proposition. For instance, the proposition B is true if we have proofs of the propositions A and $A \supset B$.

Type theory is designed to be a logic for mathematical reasoning, and it is through the computational content of constructive proofs that it can be used as a programming logic (by identifying programs and proof objects). So the logic is rather strong; it is possible to express general mathematical problems and proofs. This is important for a logic which is intended to work in practice. We want to have a language as powerful as possible to reason about programs. The formal system of type theory is inherently open in that it is possible to introduce new type forming operations and their rules. The rules have to be justified using the semantics of type theory.

1.2 Constructive mathematics

Constructive mathematics arose as an independent branch of mathematics out of the foundational crisis in the beginning of this century, mainly developed by Brouwer under the name intuitionism. It did not get much support because of the general belief that important parts of mathematics were impossible to develop constructively. By the work of Bishop, however, this belief has been shown to be wrong. In his book *Foundations of Constructive Analysis* [10], Bishop rebuilds constructively central parts of classical analysis; and he does it in a way that demonstrates that constructive mathematics can be as elegant as classical mathematics. Basic information about the fundamental ideas of intuitionistic mathematics is given in Dummet [33], Heyting [50], and Troelstra and van Dalen [108, 109].

The debate whether mathematics should be built up constructively or not need not concern us here. It is sufficient to notice that constructive mathematics has some fundamental notions in common with computer science, above all the notion of computation. This means that constructive mathematics could be an important source of inspiration for computer science. This was realized already by Bishop in [11]; Constable made a similar proposal in [23].

The notion of function or method is primitive in constructive mathematics and a function from a set A to a set B can be viewed as a program which when applied to an element in A gives an element in B as output. So all functions in constructive mathematics are computable. The notion of constructive proof is also closely related to the notion of computer program. To prove a proposition $(\forall x \in A)(\exists y \in B)P(x, y)$ constructively means to give a function f which when applied to an element a in A gives an element b in B such that $P(a, b)$ holds. So if the proposition $(\forall x \in A)(\exists y \in B)P(x, y)$ expresses a specification, then the function f obtained from the proof is a program satisfying the specification.

A constructive proof could therefore itself be seen as a computer program and the process of computing the value of a program corresponds to the process of normalizing a proof. There is however a small disadvantage of using a constructive proof as a program because the proof contains a lot of computationally irrelevant information. To get rid of this information Goto [45], Paulin-Mohring [80], Sato [93], Takasu [106] and Hayashi [49] have developed different techniques to synthesize a computer program from a constructive proof; this is also the main objective of the subset theory introduced in Part II of this book. Goad has also used the correspondence between proofs and programs to specialize a general program to efficient instantiations [41, 42].

1.3 Different formulations of type theory

One of the basic ideas behind Martin-Löf's type theory is the Curry-Howard interpretation of propositions as types, i.e. in our terminology, propositions as sets. This view of propositions is related both to Heyting's explanation of intuitionistic logic [50] and, on a more formal level, to Kleene's realizability interpretation of intuitionistic arithmetic [59].

Another source for type theory is proof theory. Using the identification of propositions and sets, normalizing a derivation is closely related to computing the value of the proof term corresponding to the derivation. Tait's computability method [105] from 1967 has been used for proving normalization for many different theories; in the *Proceedings of the Second Scandinavian Logic Symposium* [38] Tait's method is exploited in papers by Girard, Martin-Löf and Prawitz. One of Martin-Löf's original aims with type theory was that it could serve as a framework in which other theories could be interpreted. And a normalization proof for type theory would then immediately give normalization for a theory expressed in type theory.

In Martin-Löf's first formulation of type theory [64] from 1971, theories like first order arithmetic, Gödel's T [43], second order logic and simple type theory [22] could easily be interpreted. However, this formulation contained a reflection principle expressed by a universe V and including the axiom $\mathsf{V} \in \mathsf{V}$, which was shown by Girard to be inconsistent. Coquand and Huet's Theory of Constructions [26] is closely related to the type theory in [64]: instead of having a universe V, they have the two types Prop and Type and the axiom $\mathsf{Prop} \in \mathsf{Type}$. If the axiom $\mathsf{Type} \in \mathsf{Type}$ is added to the theory of constructions it would, by Girard's paradox, become inconsistent.

Martin-Löf's formulation of type theory in 1972 *An Intuitionistic Theory of Types* [66] is similar to the polymorphic and intensional set theory in this book. Intensional here means that the judgemental equality is understood as definitional equality; in particular, the equality is decidable. In the formulation used in this book, the judgemental equality $a = b \in A$ depends on the set A and is meaningful only when both a and b are elements in A. In [66], equality is instead defined for two arbitrary terms in a universe of untyped terms. And equality is convertibility in the sense of combinatory logic. A consequence of this approach is that the Church-Rosser property must be proved for the convertibility relation. In contrast to Coquand and Huet's Theory of Constructions, this formulation of type theory is predicative. So, second order logic and simple type theory cannot be interpreted in it.

Although the equality between types in [66] is intensional, the term model obtained from the normalization proof in [66] has an extensional equality on the interpretation of the types. Extensional equality means the same as in ordinary set theory: Two sets are equal if and only if they have the same elements. To remedy this problem, Martin-Löf made several changes of the theory, resulting in the formulation from 1973 in *An Intuitionistic Theory of Types: Predicative Part*[68]. This theory is strongly monomorphic in that a new constant is introduced in each application of a rule. Also, conversion under lambda is not allowed, i.e. the rule of ξ-conversion is abandoned. In this formulation of type theory, type checking is decidable. The concept of model for type theory and definitional equality are discussed in Martin-Löf [67].

The formulation of type theory from 1979 in *Constructive Mathematics and Computer Programming* [69] is polymorphic and extensional. One important difference with the earlier treatments of type theory is that normalization is not obtained by metamathematical reasoning. Instead, a direct semantics is given, based on Tait's computability method. A consequence of the semantics is that a term, which is an element in a set, can be computed to normal form. For the semantics of this theory, lazy evaluation is essential. Because of a strong elimination rule for the set expressing the extensional equality, judgemental equality is not decidable. This theory is also the one in *Intuitionistic Type Theory* [70]. It is treated in this book and is obtained if the equality sets introduced in chapter 8 are expressed by the rules for Eq. It is also the theory used in the Nuprl system [25] and by the group in Groningen [6].

In 1986, Martin-Löf put forward a framework for type theory. The framework is based on the notion of type and one of the primitive types is the type of sets. The resulting set theory is monomorphic and type checking is decidable. The theory of types and monomorphic sets is the topic of part III of this book.

1.4 Implementations of programming logics

Proofs of program correctness and formal derivations of programs soon become very long and tedious. It is therefore very easy to make errors in the derivations. So one is interested in formalizing the proofs in order to be able to mechanically check them and to have computerized tools to construct them.

Several proof checkers for formal logics have been implemented. An early example is the AUTOMATH system [31, 30] which was designed and implemented by de Bruijn et al to check proofs of mathematical theorems. Quite large proofs were checked by the system, for example the proofs in Landau's book: *Grundlagen* [58]. Another system which is more intended as a proof assistant is the Edinburgh (Cambridge) LCF system [44, 85]. In this system a user can construct proofs in Scotts's logic for computable functions. The proofs are constructed in a goal directed fashion, starting from the proposition the user wants to prove and then using tactics to divide it into simpler propositions. The LCF system also introduced the notion of metalanguage (ML) in which the user could implement her own proof strategies. Based on the LCF system, a similar system for Martin-Löf's type theory was implemented in Göteborg 1982 [86]. Another, more advanced system for type theory was developed by Constable et al at Cornell University [25].

In contrast with these systems, which were only suited for one particular logical theory, logical frameworks have been designed and implemented. Harper, Honsell and Plotkin have defined a logical framework called Edinburgh LF [48]. This theory was then implemented, using the Cornell Synthesizer. Paulson has implemented a general logic proof assistant, Isabelle [83], and type theory is one of the logics implemented in this framework. Huet and Coquand at INRIA Paris also have an implementation of their Calculus of Constructions [56].

Chapter 2

The identification of sets, propositions and specifications

The judgement

$$a \in A$$

in type theory can be read in at least the following ways:

- a is an element in the set A.
- a is a proof object for the proposition A.
- a is a program satisfying the specification A.
- a is a solution to the problem A.

The reason for this is that the concepts set, proposition, specification and problem can be explained in the same way.

2.1 Propositions as sets

In order to explain how a proposition can be expressed as a set we will explain the intuitionistic meaning of the logical constants, specifically in the way of Heyting [50]. In classical mathematics, a proposition is thought of as being true or false independently of whether we can prove or disprove it. On the other hand, a proposition is constructively true only if we have a method of proving it. For example, classically the law of excluded middle, $A \vee \neg A$, is true since the proposition A is either true or false. Constructively, however, a disjunction is true only if we can prove one of the disjuncts. Since we have no method of proving or disproving an arbitrary proposition A, we have no proof of $A \vee \neg A$ and therefore the law of excluded middle is not intuitionistically valid.

So, the constructive explanations of propositions are spelled out in terms of proofs and not in terms of a world of mathematical objects existing independently of us. Let us first only consider implication and conjunction.

> A proof of $A \supset B$ is a function (method, program) which to each proof of A gives a proof of B.

For example, in order to prove $A \supset A$ we have to give a method which to each proof of A gives a proof of A; the obvious choice is the method which returns its input as result. This is the identity function $\lambda x.x$, using the λ-notation.

> A proof of $A \,\&\, B$ is a pair whose first component is a proof of A and whose second component is a proof of B.

If we denote the left projection by *fst*, i.e. $\mathit{fst}(\langle a, b\rangle) = a$ where $\langle a, b\rangle$ is the pair of a and b, $\lambda x.\mathit{fst}(x)$ is a proof of $(A \,\&\, B) \supset A$, which can be seen as follows. Assume that

> x is a proof of $A \,\&\, B$

Since x must be a pair whose first component is a proof of A, we get

> $\mathit{fst}(x)$ is a proof of A

Hence, $\lambda x.\mathit{fst}(x)$ is a function which to each proof of $A \,\&\, B$ gives a proof of A, i.e. $\lambda x.\mathit{fst}(x)$ is a proof of $A \,\&\, B \supset A$.

The idea behind propositions as sets is to identify a proposition with the set of its proofs. That a proposition is true then means that its corresponding set is nonempty. For implication and conjunction we get, in view of the explanations above,

> $A \supset B$ is identified with $A \to B$, the set of functions from A to B.

and

> $A \,\&\, B$ is identified with $A \times B$, the cartesian product of A and B.

Using the λ-notation, the elements in $A \to B$ are of the form $\lambda x.b(x)$, where $b(x) \in B$ when $x \in A$, and the elements in set $A \times B$ are of the form $\langle a, b\rangle$ where $a \in A$ and $b \in B$.

These identifications may seem rather obvious, but, in case of implication, it was first observed by Curry [28] but only as a formal correspondence of the types of the basic combinators and the logical axioms for a language only involving implication. This was extended to first order intuitionistic arithmetic by Howard [52] in 1969. Similar ideas also occur in de Bruijn [31] and Lauchli [61]. Scott [97] was the first one to suggest a theory of constructions in which propositions are introduced by types. The idea of using constructions to represent proofs is also related to recursive realizability interpretations, first developed by Kleene [59] for intuitionistic arithmetic and extensively used in metamathematical investigations of constructive mathematics.

These ideas are incorporated in Martin-Löf's type theory, which has enough sets to express all the logical constants. In particular, type theory has function sets and cartesian products which, as we have seen, makes it possible to express implication and conjunction. Let us now see what set forming operations are needed for the remaining logical constants.

A disjunction is constructively true only if we can prove one of the disjuncts. So a proof of $A \vee B$ is either a proof of A or a proof of B together with the information of which of A or B we have a proof. Hence,

$A \vee B$ is identified with $A + B$, the disjoint union of A and B.

The elements in $A+B$ are of the form $\mathsf{inl}(a)$ and $\mathsf{inr}(b)$, where $a \in A$ and $b \in B$.

Using $\equiv$ for definitional equality, we can define the negation of a proposition A as:

$$\neg A \quad \equiv \quad A \supset \bot$$

where $\bot$ stands for absurdity, i.e. a proposition which has no proof. If we let $\emptyset$ denote the empty set, we have

$\neg A$ is identified with the set $A \to \emptyset$

using the interpretation of implication.

For expressing propositional logic, we have only used sets (types) that are available in many programming languages. In order to deal with the quantifiers, however, we need operations defined on families of sets, i.e. sets $B(x)$ depending on elements x in some set A. Heyting's explanation of the existential quantifier is the following.

A proof of $(\exists x \in A)B(x)$ consists of a construction of an element a in the set A together with a proof of $B(a)$.

So, a proof of $(\exists x \in A)B(x)$ is a pair whose first component a is an element in the set A and whose second component is a proof of $B(a)$. The set corresponding to this is the disjoint union of a family of sets, denoted by $(\Sigma x \in A)B(x)$. The elements in this set are pairs $\langle a, b \rangle$ where $a \in A$ and $b \in B(a)$. We get the following interpretation of the existential quantifier.

$(\exists x \in A)B(x)$ is identified with the set $(\Sigma x \in A)B(x)$

Finally, we have the universal quantifier.

A proof of $(\forall x \in A)B(x)$ is a function (method, program) which to each element a in the set A gives a proof of $B(a)$.

The set corresponding to the universal quantifier is the cartesian product of a family of sets, denoted by $(\Pi x \in A)B(x)$. The elements in this set are functions which, when applied to an element a in the set A gives an element in the set $B(a)$. Hence,

$(\forall x \in A)B(x)$ is identified with the set $(\Pi x \in A)B(x)$.

The elements in $(\Pi x \in A)B(x)$ are of the form $\lambda x.b(x)$ where $b(x) \in B(x)$ for $x \in A$.

Except the empty set, we have not yet introduced any sets that correspond to atomic propositions. One such set is the equality set $a =_A b$, which expresses that a and b are equal elements in the set A. Recalling that a proposition is identified with the set of its proofs, we see that this set is nonempty if and only if a and b are equal. If a and b are equal elements in the set A, we postulate that the constant $\mathsf{id}(a)$ is an element in the set $a =_A b$. This is similar to recursive realizability interpretations of arithmetic where one usually lets the natural number 0 realize a true atomic formula.

2.2 Propositions as tasks and specifications of programs

Kolmogorov [60] suggested in 1932 that a proposition could be interpreted as a problem or a task in the following way.

If A and B are tasks then

> $A \& B$ is the task of solving the tasks A and B.
>
> $A \vee B$ is the task of solving at least one of the tasks A and B.
>
> $A \supset B$ is the task of solving the task B under the assumption that we have a solution of A.

He showed that the laws of the constructive propositional calculus can be validated by this interpretation. The interpretation can be used to specify the task of a program in the following way.

> $A \& B$ is a specification of programs which, when executed, yield a pair $\langle a, b \rangle$, where a is a program for the task A and b is a program for the task B.
>
> $A \vee B$ is a specification of programs which, when executed, either yields $\mathsf{inl}(a)$ or $\mathsf{inr}(b)$, where a is a program for A and b is a program for B.
>
> $A \supset B$ is a specification of programs which, when executed, yields $\lambda x.b(x)$, where $b(x)$ is a program for B under the assumption that x is a program for A.

This explanation can be extended to the quantifiers:

> $(\forall x \in A)B(x)$ is a specification of programs which, when executed, yields $\lambda x.b(x)$, where $b(x)$ is a program for $B(x)$ under the assumption that x is an object of A. This means that when a program for the problem $(\forall x \in A)B(x)$ is applied to an arbitrary object x of A, the result will be a program for $B(x)$.
>
> $(\exists x \in A)B(x)$ specifies programs which, when executed, yields $\langle a, b \rangle$, where a is an object of A and b a program for $B(a)$. So, to solve the task $(\exists x \in A)B(x)$ it is necessary to find a method which yields an object a in A and a program for $B(a)$.

To make this into a specification language for a programming language it is of course necessary to add program forms which makes it possible to apply a function to an argument, to compute the components of a pair, to find out how a member of a disjoint union is built up, etc.

Chapter 3

Expressions and definitional equality

This chapter describes a theory of expressions, abbreviations and definitional equality. The theory was developed by Per Martin-Löf and first presented by him at the Brouwer symposium in Holland, 1981; a further developed version of the theory was presented in Siena 1983.

The theory is not limited to type theoretic expressions but is a general theory of expressions in mathematics and computer science. We shall start with an informal introduction of the four different expression forming operations in the theory, then informally introduce arities and conclude with a more formal treatment of the subject.

3.1 Application

In order to see what notions are needed when building up expressions, let us start by analyzing the mathematical expression

$$y + \sin y$$

We can view this expression as being obtained by applying the binary addition operator $+$ on y and $\sin(y)$, where the expression $\sin(y)$ has been obtained by applying the unary function sin on y.

If we use the notation

$$e(e_1, \ldots, e_n)$$

for applying the expression e on $e_1, \ldots, e_n$, the expression above should be written

$$+(y, \sin(y))$$

and we can picture it as a syntax tree:

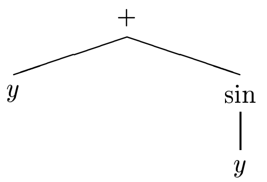

Figure 3.1: Syntax tree for the expression $+(y, \sin(y))$

Similarly, the expression (from ALGOL 68)

```
while x>0 do x:=x-1; f(x) od
```

is analyzed as

```
while(>(x,0),
      ;(:=(x,
           -(x,1)
          ),
        call(f,x)
       )
     )
```

The standard analysis of expressions in Computing Science is to use syntax trees, i.e. to consider expressions being built up from n-ary constants using application. A problem with that approach is the treatment of bound variables.

3.2 Abstraction

In the expression

$$\int_1^x (y + \sin(y))dy$$

the variable y serves only as a placeholder; we could equally well write

$$\int_1^x (u + \sin(u))du \qquad \text{or} \qquad \int_1^x (z + \sin(z))dz$$

The only purpose of the parts dy, du and dz, respectively, is to show what variable is used as the placeholder. If we let $\Box$ denote a place, we could write

$$\int_1^x (\Box + \sin(\Box))$$

for the expression formed by applying the ternary integration operator $\int$ on the integrand $\Box + \sin(\Box)$ and the integration limits 1 and x. The integrand has been obtained by functional abstraction of y from $y + \sin(y)$. We will use the notation

$$(x)e$$

for the expression obtained by functional abstraction of the variable x in e, i.e. the expression obtained from e by looking at all free occurrences of the variable x in e as holes. So, the integral should be written

$$\int(((y) +(y, \sin(y))), 1, x)$$

Since we have introduced syntactical operations for both application and abstraction it is possible to express an object by different syntactical forms. An object which syntactically could be expressed by the expression

$$e$$

could equally well be expressed by

$$((x)e)(x)$$

When two expressions are syntactical synonyms, we say that they are *definitionally*, or *intensionally*, *equal*, and we will use the symbol $\equiv$ for definitional (intensional) equality between expressions. The definitional equality between the expressions above is therefore written:

$$e \quad \equiv \quad ((x)e)(x)$$

Note that definitional equality is a *syntactical* notion and that it has nothing to do with the *meaning* of the syntactical entities.

We conclude with a few other examples of how to analyze common expressions using application and abstraction:

$$\begin{aligned}
\sum_{i=1}^{n} \frac{1}{i^2} \quad &\equiv \quad \sum(1, n, ((i)/(1, \mathit{sqr}(i)))) \\
(\forall x \in \mathsf{N})(x \geq 0) \quad &\equiv \quad \forall(\mathsf{N}, ((x) \geq (x, 0))) \\
\texttt{for } i \texttt{ from } 1 \texttt{ to } n \texttt{ do } S \quad &\equiv \quad \mathit{for}(1, n, ((i)S)))
\end{aligned}$$

3.3 Combination

We have already seen examples of applications where the operator has been applied to more than one expression, for example in the expression $+(y, \sin(y))$. There are several possibilities to syntactically analyze this situation. It is possible to understand the application operation in such a way that an operator in an application may be applied to any number of arguments. Another way is to see such an application just as a notational shorthand for a repeated use of a binary application operation, that is $e(e_1, \ldots, e_n)$ is just a shorthand for $(\ldots((e(e_1))\ldots(e_n))$. A third way, and this is the way we shall follow, is to see the combination of expressions as a separate syntactical operation just as application and abstraction. So if e_1, e_2 $\ldots$ and e_n are expressions, we may form the expression

$$e_1, e_2, \ldots, e_n$$

which we call the *combination* of e_1, e_2, ... and e_n.

Besides its obvious use in connection with functions of several arguments, the combination operation is also used for forming combined objects such as orderings

$$A, \leq$$

where A is a set and $\leq$ is a reflexive, antisymmetric and transitive relation on A, and finite state machines,

$$S, s_0, \Sigma, \delta$$

where S is a finite set of states, $s_0 \in S$ is an initial state, Σ an alphabet and δ a transition/output function.

3.4 Selection

Given an expression, which is a combination, we can use the syntactical operation *selection* to retrieve its components. If e is a combination with n components, then

$$(e).i$$

is an expression that denotes the i'th component of e if $1 \leq i \leq n$. We have the defining equation

$$(e_1, \ldots, e_n).i \quad \equiv \quad e_i$$

where $1 \leq i \leq n$.

3.5 Combinations with named components

The components of the combinations we have introduced so far have been determined by their *position* in the combination. In many situations it is much more convenient to use names to distinguish the components. We will therefore also introduce a variant where we form a combination not only of expressions but also of names that will identify the components. If e_1, e_2 ... and e_n are expressions and i_1, i_2 ... and i_n, $(n > 1)$, are different names, then we can form the expression

$$i_1 : e_1, i_2 : e_2, \ldots, i_n : e_n$$

which we call a *combination with named components*.

To retrieve a component from a combination with named components, the name of the component, of course, is used instead of the position number. So if e is a combination with names i_1, ..., i_n, then

$$(e).i_j$$

(where i_j is one of $i_1, \ldots, i_n$) is an expression that denotes the component with name i_j.

We will not need combinations with named components in this monograph and will not explore them further.

3.6 Arities

From the examples above, it seems perhaps natural to let expressions in general be built up from variables and primitive constants by means of abstraction, application, combination and selection without any restrictions. This is also the analysis, leaving out combinations, made by Church and Curry and their followers in combinatory logic.

However, there are unnatural consequences of this way of defining expressions. One is that you may apply, e.g., the expression succ, representing the successor function, on a combination with arbitrarily many components and form expressions like $\mathsf{succ}(x_1, x_2, x_3)$, although the successor function only has one argument. You may also select a component from an expression which is not a combination, or select the m'th component ($m > n$) from a combination with only n components. Another consequence is that self-application is allowed; you may form expressions like succ(succ). Self-application, together with the defining equation for abstraction:

$$((x)d)(e) \quad \equiv \quad d[x := e]$$

where $d[x := e]$ denotes the result of substituting e for all free occurrences of x in d, leads to expressions in which definitions cannot be eliminated. This is seen by the well-known example

$$((x)x(x))((x)x(x)) \;\equiv\; ((x)x(x))((x)x(x)) \;\equiv\; \ldots$$

From Church [21] we also know that if expressions and definitional equality are analyzed in this way, it will not be decidable whether two expressions are definitionally equal or not. This will have effect on the usage of a formal system of proof rules since it must be mechanically decidable if a proof rule is properly applied. For instance, in Modus Ponens

$$\frac{A \supset B \qquad A}{B}$$

it would be infeasible to require anything but that the implicand of the first premise is definitionally equal to the second premise. Therefore, definitional equality must be decidable and definitions should be eliminable. The analysis given in combinatory logic of these concepts is thus not acceptable for our purposes. Per Martin-Löf has suggested, by going back to Frege [39], that with each expression there should be associated an *arity*, showing the "functionality" of the expression. Instead of just having one syntactical category of expressions, as in combinatory logic, the expressions are divided into different categories according to which syntactical operations are applicable. The arities are similar to the types in typed λ-calculus, at least from a formal point of view.

An expression is either *combined*, in which case it is possible to select components from it, or it is *single*. Another division is between *unsaturated* expressions, which can be operators in applications, and *saturated* expressions, which cannot. The expressions which are both single and saturated have arity $\mathbf{0}$, and neither application nor selection can be performed on such expressions. The unsaturated expressions have arities of the form $(\alpha \twoheadrightarrow \beta)$, where α and β are arities; such expressions may be applied to expressions of arity α and the application gets arity β. For instance, the expression sin has arity $(\mathbf{0} \twoheadrightarrow \mathbf{0})$ and may be applied to a variable x of arity $\mathbf{0}$ to form the expression $\sin(x)$ of arity $\mathbf{0}$. The combined expressions have arities of the form $(\alpha_1 \otimes \ldots \otimes \alpha_n)$, and from expressions of this arity, one may select the i'th component if $1 \leq i \leq n$. The selected component is, of course, of arity α_i. For instance, an ordering $A, \leq$ has arity $(\mathbf{0}\otimes((\mathbf{0}\otimes\mathbf{0})\twoheadrightarrow\mathbf{0}))$.

So we make the definition:

Definition 1 *The* arities *are inductively defined as follows*

1. $\mathbf{0}$ *is an arity; the arity of single, saturated expressions.*
2. *If* $\alpha_1, \ldots, \alpha_n$ ($n \geq 2$) *are arities, then* $(\alpha_1 \otimes \cdots \otimes \alpha_n)$ *is an arity; the arity of a combined expression.*
3. *If* α *and* β *are arities, then* $(\alpha \twoheadrightarrow \beta)$ *is an arity; the arity of unsaturated expressions.*

The inductive clauses generate different arities; two arities are equal only if they are syntactically identical. The arities will often be written without parentheses; in case of conflict, like in

$$\mathbf{0}\twoheadrightarrow\mathbf{0}\otimes\mathbf{0}$$

$\twoheadrightarrow$ will have lower priority than $\otimes$. The arity above should therefore be understood as

$$(\mathbf{0}\twoheadrightarrow(\mathbf{0}\otimes\mathbf{0}))$$

We always assume that every variable and every primitive (predefined) constant has a unique arity associated with it.

The arities of some of the variables and constants we have used above are:

Expression	Arity
y	$\mathbf{0}$
x	$\mathbf{0}$
1	$\mathbf{0}$
sin	$\mathbf{0}\twoheadrightarrow\mathbf{0}$
succ	$\mathbf{0}\twoheadrightarrow\mathbf{0}$
$+$	$\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$
$\int$	$((\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes\mathbf{0}\otimes\mathbf{0})\twoheadrightarrow\mathbf{0}$

From the rules of forming expressions of a certain arity, which we will give, it is easy to derive the arities

Expression	Arity
$\sin(y)$	$\mathbf{0}$
$+(y, \sin(y))$	$\mathbf{0}$
$(y) + (y, \sin(y))$	$\mathbf{0}\twoheadrightarrow\mathbf{0}$
$\int((y) + (y, \sin(y)), 1, x)$	$\mathbf{0}$
$\mathsf{succ}(x)$	$\mathbf{0}$

However, neither $\mathsf{succ}(\mathsf{succ})$ nor $\mathsf{succ}(x)(x)$ can be formed, since succ can only be applied to expressions of arity $\mathbf{0}$ and $\mathsf{succ}(x)$ is a complete expression which can not be applied to any expression whatsoever.

3.7 Definitions

We allow abbreviatory definitions (macros) of the form

$$c \quad \equiv \quad e$$

where c is a unique identifier and e is an expression without free variables. We will often write

$$c(x_1, x_2, \ldots, x_n) \quad \equiv \quad e$$

instead of

$$c \quad \equiv \quad (x_1, x_2, \ldots, x_n)e$$

In a definition, the left hand side is called *definiendum* and the right hand side *definiens.*

3.8 Definition of what an expression of a certain arity is

In the rest of this chapter, we will explain how expressions are built up from variables and primitive constants, each with an arity, and explain when two expressions are (definitionally, intensionally) equal.

1. *Variables.* If x is a variable of arity α, then

   $$x$$

   is an expression of arity α.

2. *Primitive constants.* If c is a primitive constant of arity α, then

   $$c$$

   is an expression of arity α.

3. *Defined constants.* If, in an abbreviatory definition, the definiens is an expression of arity α, then so is the definiendum.

4. *Application.* If d is an expression of arity $\alpha \twoheadrightarrow \beta$ and a is an expression of arity α, then

   $$d(a)$$

   is an expression of arity β.

5. *Abstraction.* If b is an expression of arity β and x a variable of arity α, then

   $$((x)b)$$

   is an expression of arity $\alpha \twoheadrightarrow \beta$. In cases where no ambiguities can occur, we will remove the outermost parenthesis.

6. *Combination.* If a_1 is an expression of arity α_1, a_2 is an expression of arity α_2, ... and a_n is an expression of arity α_n, $2 \leq n$, then

$$a_1, a_2, \dots, a_n$$

is an expression of arity $\alpha_1 \otimes \alpha_2 \otimes \cdots \otimes \alpha_n$.

7. *Selection.* If a is an expression of arity $\alpha_1 \otimes \cdots \otimes \alpha_n$ and $1 \leq i \leq n$, then

$$(a).i$$

is an expression of arity α_i.

3.9 Definition of equality between two expressions

We will use the notation $a : \alpha$ for a is an expression of arity α and $a \equiv b : \alpha$ for a and b are equal expressions of arity α.

1. *Variables.* If x is a variable of arity α, then

$$x \equiv x : \alpha$$

2. *Constants.* If c is a constant of arity α, then

$$c \equiv c : \alpha$$

3. *Definiendum $\equiv$ Definiens.* If a is a definiendum with definiens b of arity α, then

$$a \equiv b : \alpha$$

4. *Application 1.* If $a \equiv a' : \alpha \twoheadrightarrow \beta$ and $b \equiv b' : \alpha$, then

$$a(b) \equiv a'(b') : \beta$$

5. *Application 2. (β-rule).* If x is a variable of arity α, a an expression of arity α and b an expression of arity β, then

$$((x)b)(a) \equiv b[x := a] : \beta$$

provided that no free variables in a becomes bound in $b[x := a]$.

6. *Abstraction 1. (ξ-rule).* If x is a variable of arity α and $b \equiv b' : \beta$, then

$$(x)b \equiv (x)b' : \alpha \twoheadrightarrow \beta$$

7. *Abstraction 2. (α-rule).* If x and y are variables of arity α and $b : \beta$, then

$$(x)b \equiv (y)(b[x := y]) : \alpha \twoheadrightarrow \beta$$

provided that y does not occur free in b.

8. *Abstraction 3. (η-rule).* If x is a variable of arity α and b is an expression of arity $\alpha \twoheadrightarrow \beta$, then

$$(x)(b(x)) \equiv b : \alpha \twoheadrightarrow \beta$$

provided that x does not occur free in b.

9. *Combination 1.* If $a_1 \equiv a_1' : \alpha_1$, $a_2 \equiv a_2' : \alpha_2$, ... and $a_n \equiv a_n' : \alpha_n$, then

$$a_1, a_2, \ldots, a_n \equiv a_1', a_2', \ldots, a_n' : \alpha_1 {\otimes} \alpha_2 {\otimes} \cdots {\otimes} \alpha_n$$

10. *Combination 2.* If $e : \alpha_1 {\otimes} \cdots {\otimes} \alpha_n$ then

$$(e).1, (e).2, \ldots, (e).n \equiv e : \alpha_1 {\otimes} \cdots {\otimes} \alpha_n$$

11. *Selection 1.* If $a \equiv a' : \alpha_1 {\otimes} \cdots {\otimes} \alpha_n$ and $1 \leq i \leq n$, then

$$(a).i \equiv (a').i : \alpha_i$$

12. *Selection 2.* If $a_1 : \alpha_1, \ldots, a_n : \alpha_n$ and $1 \leq i \leq n$ then

$$(a_1, \ldots a_n).i \equiv a_i : \alpha_i$$

13. *Reflexivity.* If $a : \alpha$, then $a \equiv a : \alpha$.

14. *Symmetry.* If $a \equiv b : \alpha$, then $b \equiv a : \alpha$.

15. *Transitivity.* If $a \equiv b : \alpha$ and $b \equiv c : \alpha$, then $a \equiv c : \alpha$.

From a formal point of view, this is similar to typed λ-calculus. The proof of the decidability of equality in typed λ-calculus can be modified to yield a proof of decidability of $\equiv$. It is also possible to define a normal form such that an expression on normal form does not contain any subexpressions of the forms $((x)b)(a)$ and $(a_1, \ldots, a_n).i$. It is then possible to prove that every expression is definitionally equal to an expression on normal form. Such a normalization theorem, leaving out combinations, is proved in Bjerner [14].

A note on the concrete syntax used in this book

When we are writing expressions in type theory we are not going to restrict ourselves to prefix constants but will use a more liberal syntax. We will freely use parentheses for grouping and will in general introduce new syntax by explicit definitions, like

$$(\Pi x \in A)B(x) \quad \equiv \quad \Pi(A, B)$$

If x is a variable of arity $\alpha_1 {\otimes} \cdots {\otimes} \alpha_n$ we will often use a form of pattern matching and write

$$(x_1, \ldots, x_n)e$$

instead of $(x)e$ and, correspondingly, write x_i instead of $x.i$ for occurrences of $x.i$ in the expression e.

Part I

Polymorphic sets

Chapter 4

The semantics of the judgement forms

In the previous chapter, we presented a theory of expressions which is the syntactical basis of type theory. We will now proceed by giving the semantics of the polymorphic set theory. We will do that by explaining the meaning of a judgement of each of the forms

- A is a set
- A_1 and A_2 are equal sets
- a is an element in the set A
- a_1 and a_2 are equal elements in the set A

When reading a set as a proposition, we will also use the judgement forms

- A is a proposition
- A is true,

where the first is the same as the judgement that A is a set and the second one means the same as the judgement a is an element in A, but we do't write down the element a. We will later, in chapter 18 introduce subsets and then separate propositions and sets.

The explanation of the judgement forms is, together with the theory of expressions, the foundation on which type theory is built by the introduction of various individual sets. So, the semantical explanation, as well as the introduction of the particular sets, is independent of and comes before any formal system for type theory. And it is through this semantics that the formal rules we will give later are justified.

The direct semantics will be explained starting from the primitive notion of computation (evaluation); i.e. the purely mechanical procedure of finding the value of a closed saturated expression. Since the semantics of the judgement forms does not depend on what particular primitive constants we have in the language, we will postpone the enumeration of all the constants and the computation rules to later chapters where the individual sets are introduced. A summary of the constants and their arities is also in appendix A.1. Concerning the computation of expressions in type theory, it is sufficient to know that the general strategy is to evaluate them from without, i.e. normal order or lazy evaluation is used.

The semantics is based on the notion of *canonical expression.* The canonical expressions are the values of programs and for each set we will give conditions for how to form a canonical expression of that set. Since canonical expressions represents values, they must be closed and saturated. Examples of expressions, in other programming languages, that correspond to canonical expressions in type theory, are

$$3,\ \mathsf{true},\ \mathsf{cons}(1, \mathsf{cons}(2, \mathsf{nil})) \text{ and } \lambda x.x$$

and expressions that correspond to noncanonical expressions are, for example,

$$3{+}5,\ \mathsf{if}\ 3 = 4\ \mathsf{then}\ \mathit{fst}(\langle 3, 4\rangle)\ \mathsf{else}\ \mathit{snd}(\langle 3, 4\rangle) \text{ and } (\lambda x.x + 1)(12 + 13)$$

Since all primitive constants we use have arities of the form $\alpha_1 \otimes \ldots \otimes \alpha_n \twoheadrightarrow \mathbf{0}$, $n \geq 0$, the normal form of a closed saturated expression is always of the form

$$c(e_1, e_2, \ldots, e_n) \text{ for } n \geq 0$$

where c is a primitive constant and e_1, e_2,... and e_n are expressions. In type theory, the distinction between canonical and noncanonical expressions can always be made from the constant c. It therefore makes sense to divide also the primitive constants into canonical and noncanonical ones. A canonical constant is, of course, one that begins a canonical expression. To a noncanonical constant there will always be associated a computation rule. Since the general strategy for computing expressions is from without, the computation process, for a closed saturated expression, will continue until an expression which starts with a canonical constant is reached. So an expression is considered evaluated when it is of the form

$$c(e_1, e_2, \ldots, e_n)$$

where c is a canonical constant, regardless of whether the expressions $e_1, \ldots, e_n$ are evaluated or not. The expressions

$$\mathsf{true},\ \mathsf{succ}(0),\ \mathsf{succ}(2 + 3) \text{ and } \mathsf{cons}(3, \mathit{append}(\mathsf{cons}(1, \mathsf{nil}), \mathsf{nil}))$$

all begin with a canonical constant and are therefore evaluated. This may seem a little counterintuitive, but the reason is that when variable binding operations are introduced, it may be impossible to evaluate one or several parts of an expression. For example, consider the expression $\lambda((x)b)$, where the part $(x)b$ cannot be evaluated since it is an unsaturated expression. To compute it would be like taking a program which expects input and trying to execute it without any input data.

In order to have a notion that more closely corresponds to what one normally means by a value and an evaluated expression, we will call a closed saturated expression *fully evaluated* when it is evaluated and all its saturated parts are fully evaluated. The expressions

$$\mathsf{true},\ \mathsf{succ}(0) \text{ and } \lambda((x)(x + 1))$$

are fully evaluated, but

$$\mathsf{succ}(2+3) \text{ and } \mathsf{cons}(3, \mathit{append}(\mathsf{cons}(1, \mathsf{nil}), \mathsf{nil}))$$

are not.

Now that we have defined what it means for an expression to be on canonical form, we can proceed with the explanations of the judgement forms:

- A is a set
- A_1 and A_2 are equal sets
- a is an element in the set A
- a_1 and a_2 are equal elements in the set A
- A is a proposition
- A is true

4.1 Categorical judgements

In general, a judgement is made under assumptions, but we will start to explain the categorical judgements, that is, judgements without assumptions.

4.1.1 What does it mean to be a set?

The judgement that A is a set, which is written

$$A \ \mathit{set}$$

is explained as follows:

> To know that A is a set is to know how to form the canonical elements in the set and under what conditions two canonical elements are equal.

A requirement on this is that the equality relation introduced between the canonical elements must be an equivalence relation. Equality on canonical elements must also be defined so that two canonical elements are equal if they have the same form and their parts are equal. So in order to define a set, we must

- Give a prescription of how to form (construct) the canonical elements, i.e. define the syntax of the canonical expressions and the premises for forming them.
- Give the premises for forming two equal canonical elements.

4.1.2 What does it mean for two sets to be equal?

Let A and B be sets. Then, according to the explanation of the first judgement form above, we know how to form the canonical elements together with the equality relation on them. The judgement that A and B are equal sets, which is written

$$A = B$$

is explained as follows:

> To know that two sets, A and B, are equal is to know that a canonical element in the set A is also a canonical element in the set B and, moreover, equal canonical elements of the set A are also equal canonical elements of the set B, and vice versa.

So in order to assert $A = B$ we must know that

- A is a set
- B is a set
- If a is a canonical element in the set A, then it is also a canonical element in the set B.
- If a and a' are equal canonical elements of the set A, then they are also equal canonical elements in the set B.
- If b is a canonical element in the set B, then it is also a canonical element in the set A.
- If b and b' are equal canonical elements in the set B, then they are also equal canonical elements in the set A.

From this explanation of what it means for two sets to be equal, it is clear that the relation of set equality is an equivalence relation.

4.1.3 What does it mean to be an element in a set?

The third judgement form, saying that a is an element in the set A, which is written

$$a \in A$$

is explained as follows:

> If A is a set then to know that $a \in A$ is to know that a, when evaluated, yields a canonical element in A as value.

In order to assert $a \in A$, we must know that A is a set and that the expression a yields a canonical element of A as value.

4.1.4 What does it mean for two elements to be equal in a set?

If A is a set, then we can say what it means for two elements in the set A to be equal. The explanation is:

> To know that a and b are equal elements in the set A, is to know that they yield equal canonical elements in the set A as values.

Since it is an assumption that A is a set, we already know what it means to be a canonical element in the set A and how the equality relation on the canonical elements is defined. Consequently, we know what the judgement that the values of a and b are equal canonical elements in the set A means. The judgement saying that a and b are equal elements in the set A is written

$$a = b \in A$$

4.1.5 What does it mean to be a proposition?

To know that A is a proposition is to know that A is a set.

4.1.6 What does it mean for a proposition to be true?

To know that the proposition A is true is to have an element a in A.

4.2 Hypothetical judgements with one assumption

The next step is to extend the explanations for assumption free judgements to cover also hypothetical ones. The simplest assumption is of the form

$$x \in A$$

where x is a variable of arity $\mathbf{0}$ and A is a set.

Since sets and propositions are identified in type theory, an assumption can be read in two different ways:

1. As a variable declaration, that is, declaring the set which a free variable ranges over, for example, $x \in \mathsf{N}$ and $y \in \mathsf{Bool}$.
2. As an ordinary logical assumption, that is, $x \in A$ means that we assume that the proposition A is true and x is a construction for it.

Being a set, however, may also depend on assumptions. For example, $a =_A b$, which expresses equality on the set A and is defined in chapter 8, is a set only when $a \in A$ and $b \in A$. So we are only interested in assumption lists

$$x_1 \in A_1,\ x_2 \in A_2(x_1),\ \ldots, x_n \in A_n(x_1, x_2, \ldots,\ x_{n-1})$$

where each $A_i(x_1, \ldots, x_{i-1})$ is a set under the preceding assumptions. Such lists are called *contexts* . We limit ourselves here to assumptions whose variables are of arity $\mathbf{0}$; they are sufficient for everything in type theory except for the elimination rule involving the primitive constant $\mathsf{funsplit}$ (chapter 7) and the natural formulation of the elimination rule for well-orderings. A more general kind of assumption is presented in chapter 19.

Now we can extend the semantic explanations to judgements depending on contexts with assumptions of the form described above. The meaning of an arbitrary judgement is explained by induction on the length n of its context. We have already given the meaning of judgements with empty contexts and, as induction hypothesis, we assume that we know what judgements mean when they have contexts of length $n-1$. However, in order not to get the explanations hidden by heavy notation, we will first treat the case with just one assumption.

4.2.1 What does it mean to be a set under an assumption?

To know the judgement

$$A(x)\ set\ \ [x \in C]$$

is to know that for an arbitrary element c in the set C, $A(c)$ is a set. Here it is assumed that C is a set so we already know what $c \in C$ means. We must also know that $A(x)$ is extensional in the sense that if $b = c \in C$ then $A(b) = A(c)$.

4.2.2 What does it mean for two sets to be equal under an assumption?

The second judgement form is explained as follows: To know that

$$A(x) = B(x)\ \ [x \in C]$$

is to know that

$$A(c) = B(c)$$

for an arbitrary element c in the set C. Here it is assumed that the judgements $A(x)\ set\ \ [x \in C]$ and $B(x)\ set\ \ [x \in C]$ hold. Hence, we know what the judgement $A(c) = B(c)$ means, namely that a canonical element in the set $A(c)$ is also a canonical element in the set $B(c)$ and equal canonical elements in the set $A(c)$ are equal canonical elements in the set $B(c)$ and vice versa.

4.2.3 What does it mean to be an element in a set under an assumption?

To know that

$$a(x) \in A(x)\ \ [x \in C]$$

is to know that $a(c) \in A(c)$ for an arbitrary element c in the set C. It is here assumed that the judgement $A(x)\ set\ \ [x \in C]$ holds and hence we know what it means for an expression to be an element in the set $A(c)$. Hence, we know the meaning of $a(c) \in A(c)$. In order to make a judgement of this form, we must also know that $a(x)$ is extensional in the sense that if $b = c \in C$ then $a(b) = a(c) \in A(c)$.

4.2.4 What does it mean for two elements to be equal in a set under an assumption?

To know the judgement

$$a(x) = b(x) \in A(x) \quad [x \in C]$$

is to know that $a(c) = b(c) \in A(c)$ holds for an arbitrary element c in the set C. It is here assumed that the judgements $A(x)$ *set*, $a(x) \in A(x)$ and $b(x) \in A(x)$ hold under the assumption that $x \in C$.

4.2.5 What does it mean to be a proposition under an assumption?

To know that $A(x)$ is a proposition under the assumption that $x \in C$ is to know that $A(x)$ is a set under the assumption that $x \in C$.

4.2.6 What does it mean for a proposition to be true under an assumption?

To know that the proposition $A(x)$ is true under the assumption that $x \in C$ is to have an expression $a(x)$ and know the judgement $a(x) \in A(x) \quad [x \in C]$.

4.3 Hypothetical judgements with several assumptions

We now come to the induction step. The general case of contexts of length n is a straightforward generalization of the case with just one assumption.

4.3.1 What does it mean to be a set under several assumptions?

To know that

$$A(x_1, \ldots, x_n) \ set \ \ [x_1 \in C_1, \ \ldots, x_n \in C_n(x_1, \ldots, x_{n-1})]$$

is to know that

$$A(c, \ldots, x_n) \ set \ [x_2 \in C_2(c), \ \ldots, x_n \in C_n(c, \ldots, x_{n-1})]$$

provided $c \in C_1$. So

$$A(x_1, \ldots, x_n) \ set \ \ [x_1 \in C_1, \ \ldots, x_n \in C_n(x_1, \ldots, x_{n-1})]$$

means that

$$A(c_1, \ldots, c_n) \ set$$

provided

$$\begin{array}{l} c_1 \in C_1 \\ \quad \vdots \\ c_n \in C_n(c_1, \ldots, c_{n-1}) \end{array}$$

It is also inherent in the meaning of a propositional function (family of sets) that it is extensional in the sense that when applied to equal elements in the domain it will yield equal propositions as result. So, if we have that

$$\begin{array}{l} a_1 = b_1 \in C_1 \\ a_2 = b_2 \in C_2(a_1) \\ \qquad\vdots \\ a_n = b_n \in C_n(a_1, \ldots, a_n) \end{array}$$

then it follows from

$$A(x_1, \ldots, x_n)\ set \ \ [x_1 \in C_1,\ \ldots, x_n \in C_n(x_1, \ldots, x_{n-1})]$$

that

$$A(a_1, \ldots, a_n) = A(b_1, \ldots, b_n)$$

4.3.2 What does it mean for two sets to be equal under several assumptions?

Hypothetical judgements of the other forms are defined in a similar way. The second judgement form is explained as follows.

Let $A(x_1, \ldots, x_n)$ and $B(x_1, \ldots, x_n)$ be sets in the context

$$x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})$$

Then to know the judgement

$$A(x_1, \ldots, x_n) = B(x_1, \ldots x_n) \ \ [x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots,\ x_{n-1})]$$

is to know that

$$A(c, \ldots, x_n) = B(c, \ldots, x_n)\ [x_2 \in C_2(c),\ \ldots,\ x_n \in C_n(c, x_2, \ldots, x_{n-1})]$$

provided $c \in C_1$.

4.3.3 What does it mean to be an element in a set under several assumptions?

The third judgement form has the following explanation for a context of length n. Let $A(x_1, \ldots, x_n)$ be a set in the context $x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})$. Then to know the judgement

$$a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n) \ \ [x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})]$$

is to know that

$$a(c, x_2, \ldots, x_n) \in A(c, x_2, \ldots, x_n) \ \ [x_2 \in C_2(c),\ \ldots,\ x_n \in C_n(c_1, \ldots, x_{n-1})]$$

provided $c \in C_1$.

It is also inherent in the meaning of being a functional expression in a set that it is extensional in the sense that if it is applied to equal elements in the domain it will yield equal elements in the range. So, if we have

$$\begin{array}{l} a_1 = b_1 \in C_1 \\ a_2 = b_2 \in C_2(a_1) \\ \qquad\vdots \\ a_n = b_n \in C_n(a_1, \ldots, a_n) \end{array}$$

then it follows from

$$a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n) \quad [x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})]$$

that

$$a(a_1, \ldots, a_n) = a(b_1, \ldots, b_n) \in A(a_1, \ldots, a_n).$$

4.3.4 What does it mean for two elements to be equal in a set under several assumptions?

The fourth judgement form is explained as follows. Let $a(x_1, \ldots, x_n)$ and $b(x_1, \ldots, x_n)$ be elements in the set $A(x_1, \ldots, x_n)$ in the context

$$x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1}).$$

Then to know that

$$a(x_1, \ldots, x_{n)} = b(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)\ [x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})]$$

is to know that

$$a(c, x_2, \ldots, x_n) = b(c, x_2, \ldots, x_n) \in A(c, x_2, \ldots, x_n)\ [x_2 \in C_2(c),\ \ldots,\ x_n \in C_n(c, x_2, \ldots, x_{n-1})]$$

provided $c \in C_1$.

4.3.5 What does it mean to be a proposition under several assumptions?

To know

$$A(x_1, \ldots, x_n)\ \mathit{prop}\ \ [x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})]$$

is to know that

$$A(x_1, \ldots, x_n)\ \mathit{set}\ \ [x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})]$$

4.3.6 What does it mean for a proposition to be true under several assumptions?

To know

$$A(x_1, \ldots, x_n)\ \mathit{true}\ \ [x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})]$$

is to have an expression $a(x_1, \ldots, x_n)$ and know the judgement

$$a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)\ \mathit{set}\ \ [x_1 \in C_1,\ \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})]$$

Chapter 5

General rules

In a formal system for type theory there are first some general rules concerning equality and substitution. These rules can be justified from the semantical explanations given in the previous chapter. Then, for each set forming operation there are rules for reasoning about the set and its elements.

For each set forming operation there are four kinds of rules.

- The *formation rules* for A describe under which conditions we may infer that A is a set and when two sets A and B are equal.

- The *introduction rules* define the set A in that they prescribe how the canonical elements are formed and when two canonical elements are equal. The constructors for the set are introduced in these rules.

- The *elimination rules* show how to prove a proposition about an arbitrary element in the set. These rules are a kind of structural induction rules in that they state that to prove that an arbitrary element p in the set A has a property $C(p)$ it is enough to prove that an arbitrary canonical element in the set has that property. The selector, which is a primitive noncanonical constant associated with the set is introduced in this kind of rule. It is the selector which makes it possible to do pattern-matching and primitive recursion over the elements in the set.

- The *equality rules* describe the equalities which are introduced by the the computation rules for the selector associated with the set.

In this chapter we will present the general rules, and in later chapters set forming operations and their rules.

We will present the rules in a natural deduction style

$$\frac{P_1 \qquad P_2 \qquad \cdots \qquad P_n}{C}$$

where the premises P_1, P_2, $\ldots$, P_n and the conclusion C in general are hypothetical judgements. When all the premises do not fit on one line, we write the rule with one premise on each line:

$$\frac{\begin{array}{c} P_1 \\ P_2 \\ \vdots \\ P_n \end{array}}{C}$$

When we write a rule, we will only present those assumptions that are discharged by the rule. The formation rule for Π will, for instance, be written

$$\frac{A \ set \qquad B(x) \ set \ \ [x \in A]}{\Pi(A, B) \ set}$$

The full form of this rule with assumption lists Γ and Δ is

$$\frac{A \ set \ \ [\Gamma] \qquad B(x) \ set \ \ [\Delta, x \in A]}{\Pi(A, B) \ set \ \ [\Gamma, \Delta]}$$

A rule like this one is applicable to form a proof of the conclusion if we have proofs of the two judgements

- $A \ set \ \ [\Gamma']$
- $B(x) \ set \ \ [\Delta']$

and the assumption lists Γ' and Δ' in those judgements have the following properties

- Γ' must not contain an assumption for the variable x.
- If there are assumptions for the same variable in Γ' and Δ' the sets in the assumptions must be identical, i.e., definitionally equal.
- If there is an assumption for the variable x in Δ' it must be the last assumption and the set must be A.

The assumption list $[\Gamma, \Delta]$, in the rule above, consists of the assumptions in Γ followed by those assumptions in Δ which do not occur in Γ.

If a rule has a premise of the form $a \in A$, we will often exclude the premise $A \ set$ and if a premise has the form $A = B$ we will often exclude the premises $A \ set$ and $B \ set$. And similarly, if the premise is of the form $a = b \in A$, we will often exclude the premises $A \ set$, $a \in A$ and $b \in A$. We also extend this to families of sets, so if we have a premise of the form $a(x) \in B(x) \ \ [x \in A]$ we exclude the premises $A \ set$ and $B(x) \ set \ \ [x \in A]$. That these premises are required follows from the explanation of $a \in A$, $A = B$ and $a = b \in A$. The full form of the introduction rule for $\rightarrow$

$$\frac{b(x) \in B \ \ [x \in A]}{\lambda(b) \in A \rightarrow B}$$

is therefore

$$\frac{A \ set \ \ [\Gamma] \qquad B \ set \ \ [\Delta] \qquad b(x) \in B \ \ [\Theta, x \in A]}{\lambda(b) \in A \rightarrow B \ \ [\Gamma, \Delta, \Theta]}$$

where A, B and b may have occurrences of the variables that are introduced in the assumption lists Γ, Δ and Θ respectively.

5.1 Assumptions

The first rule we give is the one which makes it possible to introduce assumptions.

Assumption

$$\frac{A \ \mathit{set}}{x \in A \ \ [x \in A]}$$

This rule says that if A is a set, then we can introduce a variable x of that set.

By the correspondence between propositions and sets, and the interpretation of true propositions as nonempty sets, the assumption $x \in A$ also serves as the assumption that the proposition A is true. An assumption of the form A *true* is therefore an abbreviation of an assumption $x \in A$ where x is a new variable.

Applying the assumption rule on the premise A *set* gives us the judgement $x \in A \ \ [x \in A]$. We can see the variable x as a name of an indeterminate proof-element of the proposition A. One way to discharge the assumption $x \in A$ is to find an element a in the set A and substitute it for all free occurrences of x. Formally this is done by applying one of the substitution rules that are introduced in section 5.5.

5.2 Propositions as sets

If we have an element in a set, then we will interpret that set as a true proposition. We have the rule:

Proposition as set

$$\frac{a \in A}{A \ \mathit{true}}$$

5.3 Equality rules

We have the following general equality rules:

Reflexivity

$$\frac{a \in A}{a = a \in A} \qquad \frac{A \ \mathit{set}}{A = A}$$

Symmetry

$$\frac{a = b \in A}{b = a \in A} \qquad \frac{A = B}{B = A}$$

Transitivity

$$\frac{a = b \in A \qquad b = c \in A}{a = c \in A} \qquad \frac{A = B \qquad B = C}{A = C}$$

The rules concerning equality between elements can be justified from the fact that they hold for canonical elements. For instance, the symmetry rule can be justified in the following way: That $a = b \in A$ means that $a' = b' \in A$, where a' is the value of a and b' is the value of b. Since equality between canonical elements is symmetric we have $b' = a' \in A$, which gives that $b = a \in A$.

The other rules are also easily justified, for example the rule concerning symmetry of equality between sets: The meaning of $A = B$ is that canonical elements in A are canonical in B and equal canonical elements in A are equal canonical elements in B. The judgement also means that canonical elements in B are canonical in A and that equal canonical elements in B are equal canonical elements in A. By just changing the order of these two sentences we get the definition of what $B = A$ means.

5.4 Set rules

The meanings of the judgement forms $A = B$, $a \in A$ and $a = b \in A$ immediately justify the following rules:

Set equality

$$\frac{a \in A \qquad A = B}{a \in B} \qquad\qquad \frac{a = b \in A \qquad A = B}{a = b \in B}$$

5.5 Substitution rules

The meanings of the four judgement forms when they depend on a nonempty context yield four sets of substitution rules. The judgement

$$C(x)\ set\ \ [x \in A]$$

means that $C(a)$ is a set, provided $a \in A$, and that $C(a) = C(b)$ whenever $a = b \in A$. This explanation immediately gives us the rules:

Substitution in sets

$$\frac{C(x)\ set\ \ [x \in A] \qquad a \in A}{C(a)\ set} \qquad\qquad \frac{C(x)\ set\ \ [x \in A] \qquad a = b \in A}{C(a) = C(b)}$$

The judgement

$$c(x) \in C(x)\ \ [x \in A]$$

means that $c(a) \in C(a)$ if $a \in A$ and that $c(a) = c(b) \in C(a)$ if $a = b \in A$. This justifies the rules:

Substitution in elements

$$\frac{c(x) \in C(x)\ \ [x \in A] \qquad a \in A}{c(a) \in C(a)} \qquad\qquad \frac{c(x) \in C(x)\ \ [x \in A] \qquad a = b \in A}{c(a) = c(b) \in C(a)}$$

If we read $C(x)$ as a proposition, and consequently $c(x)$ as a proof-element of the proposition, these rules can be used to discharge an assumption. When a judgement depends on the assumption that x is a proof-element of the proposition A, we can substitute an actual proof-element for the indeterminate proof-element x and discharge the assumption $x \in A$.

The meaning of the hypothetical judgement

$$B(x) = C(x) \quad [x \in A]$$

is that $B(a)$ and $C(a)$ are equal sets for any element a in A. Therefore we have the rule

Substitution in equal sets

$$\frac{B(x) = C(x) \quad [x \in A] \qquad a \in A}{B(a) = C(a)}$$

Finally, we have the hypothetical judgement

$$b(x) = c(x) \in B(x) \quad [x \in A]$$

which means that $b(a)$ and $c(a)$ are equal elements in $B(a)$, provided that $a \in A$. This justifies the rule

Substitution in equal elements

$$\frac{b(x) = c(x) \in B(x) \ [x \in A] \qquad a \in A}{b(a) = c(a) \in B(a)}$$

These rules for substitution are not sufficient because if we have a judgement

$$C(x, y) \ set \quad [x \in A, y \in B(x)]$$

and want to substitute $a \in A$ for x and $b \in B(a)$ for y we cannot use the rules given above since they cannot handle the case with simultaneous substitution of several variables. We therefore extend the substitution rules to n simultaneous substitutions. We present only the rule for substitution in equal sets.

Substitution in equal sets of n variables

$$\frac{\begin{array}{l} B(x_1, \ldots, x_n) = C(x_1, \ldots, x_n) \ [x_1 \in A_1, \ldots, \ x_n \in A_n(x_1, \ldots, x_{n-1}] \\ \qquad\qquad a_1 \in A_1 \\ \qquad\qquad\quad \vdots \\ \qquad\qquad a_n \in A_n(a_1, \ldots, a_{n-1} \end{array}}{B(a_1, \ldots, a_n) = C(a_1, \ldots, a_n)}$$

The rule is justified from the meaning of a hypothetical judgement with several assumptions.

Another way to achieve the same effect is to allow substitution in the middle of a context. For example if we have a judgement

$$C(x, y) \ set \quad [x \in A, y \in B(x)]$$

we could first substitute $a \in A$ for x obtaining the judgement

$$C(a, y) \ set \quad [y \in B(a)]$$

then substitute $b \in B(a)$ for y. When using type theory to do formal proofs, it is convenient to have substitution rules of this form.

Chapter 6

Enumeration sets

Given n canonical constants $i_1, \ldots, i_n$, each of arity $\mathbf{0}$, we want to be able to introduce the enumeration set $\{i_1, \ldots, i_n\}$. So, we introduce a constant $\{i_1, \ldots, i_n\}$ of arity $\mathbf{0}$. It must be immediate from each identifier i_k to which enumeration set it belongs and what position (index) it has. The convention we will follow is that an identifier can only belong to one enumeration set and the first occurrence of the set decides the index of the elements. We have the following formation rule:

$\{i_1, \ldots, i_n\}$ – formation

$$\{i_1, \ldots, i_n\} \ set$$

The canonical elements of $\{i_1, ..., i_n\}$ are i_1, i_2, $\ldots$ and i_n which gives the following n introduction rules ($n \geq 0$):

$\{i_1, \ldots, i_n\}$ – introduction 1

$$i_1 \in \{i_1, \ldots, i_n\} \quad \ldots \quad i_n \in \{i_1, \ldots, i_n\}$$

Two canonical elements of $\{i_1, \ldots, i_n\}$ are equal only if they are the same canonical constants:

$\{i_1, \ldots, i_n\}$ – introduction 2

$$i_1 = i_1 \in \{i_1, \ldots, i_n\} \quad \ldots \quad i_n = i_n \in \{i_1, \ldots, i_n\}$$

The selector expression for $\{i_1, \ldots, i_n\}$ is the expression

$$\mathsf{case}_{\{i_1, \ldots, i_n\}}(a, b_1, \ldots, b_n)$$

where $\mathsf{case}_{\{i_1, \ldots, i_n\}}$ is a constant of arity $\mathbf{0}\otimes\cdots\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$. The notation for the expression $\mathsf{case}_{\{i_1, \ldots, i_n\}}(a, b_1, \ldots, b_n)$ in ML is

$$\begin{array}{l} \texttt{case a of } i_1 \texttt{ => } b_1 \\ \qquad\qquad \vdots \\ \qquad\quad \texttt{| } i_n \texttt{ => } b_n \end{array}$$

We will usually drop the index in $\mathsf{case}_{\{i_1, \ldots, i_n\}}$ since it is often clear from the context. The case-expression is computed in the following way:

1. First evaluate a.
2. If the value of a is i_k $(1 \leq k \leq n)$ then the value of the case expression is the value of b_k.

We have the following elimination rules:

$\{i_1, \ldots, i_n\}$ – elimination 1

$$\frac{\begin{array}{l} a \in \{i_1, \ldots, i_n\} \\ C(x)\ set\ \ [x \in \{i_1, \ldots, i_n\}] \\ b_1 \in C(i_1) \\ \qquad \vdots \\ b_n \in C(i_n) \end{array}}{\mathsf{case}(a, b_1, \ldots, b_n) \in C(a)}$$

$\{i_1, \ldots, i_n\}$ – elimination 2

$$\frac{\begin{array}{l} a = a' \in \{i_1, \ldots, i_n\} \\ C(x)\ set\ \ [x \in \{i_1, \ldots, i_n\}] \\ b_1 = b_1' \in C(i_1) \\ \qquad \vdots \\ b_n = b_n' \in C(i_n) \end{array}}{\mathsf{case}(a, b_1, \ldots, b_n) = \mathsf{case}(a', b_1', \ldots, b_n') \in C(a)}$$

The first elimination rule is justified in the following way. Assume the premises of the rule. We have to show that

$$\mathsf{case}(a, b_1, \ldots, b_n) \in C(a)$$

which means that we have to show that the value of $\mathsf{case}(a, b_1, \ldots, b_n)$ is a canonical element in $C(a)$. This program is computed by first computing the value of a. From the first premise we know that the value of a is a canonical element in $\{i_1, \ldots, i_n\}$, so the value must be i_j for some j, $1 \leq j \leq n$. The value of the case-expression is then the value of b_j, according to the computation rule for case. From one of the premises, we know that the value of b_j is a canonical element in $C(i_j)$. So we have shown that the value of the case-expression is a canonical value in $C(i_j)$. But this set is equal to the set $C(a)$. This follows from the meaning of the second premise. That $C(x)$ *set* $[x \in \{i_1, \ldots, i_n\}]$ gives that $C(a) = C(i_j)$. From the meaning of two sets being equal it follows that the value of the program $\mathsf{case}(a, b_1, \ldots, b_n)$ being a canonical element in $C(i_j)$ is also a canonical element in $C(a)$.

The second elimination rule can be justified in a similar way, using the computation rule for the case-expression and the meaning of the different forms of judgements. Furthermore, the computation rule justifies n equality rules. For each k, $1 \leq k \leq n$, we get the rule:

$\{i_1, \ldots, i_n\}$ – equality

$$\frac{C(x)\ set\ \ [x \in \{i_1, \ldots, i_n\}] \qquad b_1 \in C(i_1) \quad \ldots \quad b_n \in C(i_n)}{\mathsf{case}(i_k, b_1, \ldots, b_n) = b_k \in C(i_k)}$$

6.1 Absurdity and the empty set

If $n = 0$ we get the empty set $\{\}$ which, of course, has no introduction rule. The $\{\}$ – elimination rule becomes:

$\{\}$ – elimination 1

$$\frac{a \in \{\} \qquad C(x)\ set\ \ [x \in \{\}]}{\mathsf{case}(a) \in C(a)}$$

$\{\}$ – elimination 2

$$\frac{a = a' \in \{\} \qquad C(x)\ set\ \ [x \in \{\}]}{\mathsf{case}(a) = \mathsf{case}(a') \in C(a)}$$

In the following we will not give rules like the second elimination rule above. The general shape of these rules is that sets or elements are equal if their form is identical and their parts are equal. For the monomorphic type theory (see chapter 19) these rules follows immediately from substitution in objects on the type level.

We will sometimes use the definition

$$\emptyset \quad \equiv \quad \{\}$$

Viewing sets as propositions, the empty set corresponds to absurdity, i.e. the proposition $\perp$ which has no proof. So, making the definition

$$\perp \quad \equiv \quad \{\}$$

we get, from the elimination rule for $\{\}$ by omitting some of the constructions, the natural deduction rule for absurdity:

$\perp$ – elimination

$$\frac{\perp\ true \qquad C\ prop}{C\ true}$$

where C is an arbitrary proposition (set). That this rule is correct is a direct consequence of the semantics of type theory. If $\perp$ is true then we have an element a in $\perp$ and then we can use the rule $\{\}$ – elimination 1 to conclude that $\mathsf{case}(a) \in C$ and hence that C is true.

6.2 The one-element set and the true proposition

There are many sets which are non-empty and thus can be used to represent the true proposition T (truth). We make the following definition:

$$\mathsf{T} \equiv \{\mathsf{tt}\}$$

where tt is a new primitive constant of arity $\mathbf{0}$. From the general rules for the enumeration set, we get the following rules:

T – formation

$$\mathsf{T}\ set$$

T – introduction

$$\mathsf{tt} \in \mathsf{T}$$

T – elimination

$$\frac{a \in \mathsf{T} \qquad C(x)\ set\ \ [x \in \mathsf{T}] \qquad b \in C(\mathsf{tt})}{\mathsf{case}(a, b) \in C(a)}$$

T – equality

$$\frac{C(x)\ set\ \ [x \in \mathsf{T}] \qquad b \in C(\mathsf{tt})}{\mathsf{case}(\mathsf{tt}, b) = b \in C(\mathsf{tt})}$$

We also get the natural deduction rules for truth:

T – introduction

$$\mathsf{T}\ true$$

T – elimination

$$\frac{\mathsf{T}\ true \qquad C\ true}{C\ true}$$

These two rules are usually not formulated in systems of natural deduction. The last one is for obvious reasons never used.

6.3 The set Bool

In order to form the set of boolean values, we introduce the two constants true and false, both of arity $\mathbf{0}$, and make the definitions

$$\begin{array}{rcl} \mathsf{Bool} & \equiv & \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ d & \equiv & \mathsf{case}(b, c, d) \end{array}$$

As special cases of the rules for enumeration sets, we get

Bool – formation

$$\mathsf{Bool}\ set$$

Bool – introduction

$$\mathsf{true} \in \mathsf{Bool} \qquad \mathsf{false} \in \mathsf{Bool}$$

Bool – elimination

$$\frac{b \in \mathsf{Bool} \qquad C(v)\ set\ \ [v \in \mathsf{Bool}] \qquad c \in C(\mathsf{true}) \qquad d \in C(\mathsf{false})}{\mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ d \in C(b)}$$

Bool – equality

$$\frac{C(v)\ set\ \ [v \in \mathsf{Bool}] \qquad c \in C(\mathsf{true}) \qquad d \in C(\mathsf{false})}{\mathsf{if\ true\ then}\ c\ \mathsf{else}\ d = c \in C(\mathsf{true})}$$

$$\frac{C(v)\ set\ \ [v \in \mathsf{Bool}] \qquad c \in C(\mathsf{true}) \qquad d \in C(\mathsf{false})}{\mathsf{if\ false\ then}\ c\ \mathsf{else}\ d = d \in C(\mathsf{false})}$$

Note the difference of true being an element in the set Bool and the judgement C *true* which abbreviates that the set C is non-empty. The judgement C is true means that we have a proof of the *proposition* C, so C is really true since we have proven it. The judgement $c = \mathsf{true} \in \mathsf{Bool}$ means only that if we compute the *program* c we get the canonical element true as a result. This has nothing to do with truth; we only use true as a convenient name for this canonical element. Some programming languages use other names, for instance 0 and 1 are also used. Many years of programming practice have shown that it is convenient to use the names true and false for the canonical elements in the set with two elements. There is, however, something arbitrary in this choice.

In type theory with a universe (see chapter 14) it is possible to prove that

$$\neg(\mathsf{true} =_{\mathsf{Bool}} \mathsf{false})$$

where $(\mathsf{true} =_{\mathsf{Bool}} \mathsf{false})$ is the proposition, to be introduced in chapter 8, which is true if true is equal to false .

Chapter 7

Cartesian product of a family of sets

The members of a cartesian product of a family of sets are functions. But a cartesian product is more general than the usual set of functions $A \to B$, since the result of applying a function to an argument is in a set which may depend on the value to which the function is applied. If f is an element in a cartesian product and a and b are expressions, it is, for instance, possible that f applied to a is a member of N, the set of natural numbers, and f applied to b is a member of Bool. This means that type theory contains functions which are not definable in typed programming languages like ML and Pascal. One reason for this generality is that it is needed in the definition of the universal quantifier. It is also needed when we use sets to specify programs. A specification of a program has often the following form: find a function f which for any argument a from the set A yields a value in the set $B(a)$. For instance a sorting program takes an argument a from the set of integer lists and outputs an ordered permutation of a, so the output is in the set $Op(a)$, the set of all ordered permutations of a. It is here essential that we can give a specification that expresses how the type of the result of the function depends on the value of the argument.

In order to form a cartesian product of a family of sets we must have a set A and a family B of sets on A , i.e.

$$A \ set$$

and

$$B(x) \ set \ \ [x \in A]$$

We will use the primitive constant Π of arity $\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$ when forming a cartesian product. So

$$\Pi(A, B)$$

denotes the cartesian product of A and B. The following explicit definition is used:

$$(\Pi x \in A)B(x) \quad \equiv \quad \Pi(A, B)$$

We have to define the canonical elements in $\Pi(A, B)$ and define what it means for two canonical elements to be equal. The elements in $\Pi(A, B)$ are functions and we will use the lambda notation for expressing them. So we introduce the primitive constant λ of arity $(\mathbf{0}{\twoheadrightarrow}\mathbf{0}){\twoheadrightarrow}\mathbf{0}$. The basic notion of function is an expression formed by abstraction. Therefore the canonical elements in $\Pi(A, B)$ will be formed by applying the λ on an abstraction b such that $b(x)$ is an element of $B(x)$ when $x \in A$:

> $\lambda(b)$ is a canonical element in $\Pi(A, B)$ if $b(x) \in B(x) \quad [x \in A]$.

The equality between two canonical elements $\lambda(b_1)$ and $\lambda(b_2)$ of $\Pi(A, B)$ is derived from the equality on the family $B(x)$ on A :

> $\lambda(b_1)$ and $\lambda(b_2)$ are equal canonical elements in $\Pi(A, B)$ provided that $b_1(x) = b_2(x) \in B(x) \quad [x \in A]$.

The primitive non-canonical constant for the Π-set is apply of arity $\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}$. It is the constant used for applying an element in $\Pi(A, B)$ to an element in A. Hence, it has the following computation rule:

1. $\mathsf{apply}(f, a)$ is evaluated by first evaluating f.
2. If f has value $\lambda(b)$ then the value of $\mathsf{apply}(f, a)$ is the value of $b(a)$.

We will later, in section 7.2, give an alternative non-canonical constant for the Π-set.

One of the main reasons for introducing the Π-set is that it is needed when interpreting the universal quantifier, which has the following Heyting interpretation:

> $(\forall x \in A)B(x)$ is true if we can construct a function which when applied to an element a in the set A, yields a proof of $B(a)$.

If we identify the proposition $B(x)$ with the family of sets $B(x) \quad [x \in A]$, and if we let the proofs of $B(x)$ be represented by the elements in the set $B(x) \quad [x \in A]$, then the elements in the set $\Pi(A, B)$ are exactly the functions mentioned in the Heyting interpretation. The elements in $\Pi(A, B)$ therefore represent the proofs of $(\forall x \in A)B(x)$. So we see, that in order to cope with the universal quantifier, it is necessary to have this kind of generalized function set.

Other examples of sets (propositions) that are defined as special cases of the cartesian product are:

1. the restricted set of functions $A \to B$, where the set B does not depend on the argument $x \in A$
2. the implication $A \supset B$.
3. the record type former in Pascal is a set $(\Pi x \in \{i_1, \ldots, i_n\})B(x)$, the members of which are tuples. The component of the tuple with the name j is in the set $B(j)$. In Pascal the application $\mathsf{apply}(f, j)$ is written $f.j$.

The last example shows that a cartesian product of a family of sets is a generalization of a cartesian product of a finite number of sets.

It is important to distinguish between the two different notions of function we have used. The first is the fundamental syntactical notion of function as an expression with holes in it, i.e. an expression which is not saturated. The second is the notion of function as an element in the cartesian product. When there is a risk of confusion between these two notions, we will use the word *abstraction* for the syntactic notion and *function element* for the second. The syntactical notion of function is more basic; we use it already when we write down the sets $\Pi(A, B)$ and $A \to C$, in these expressions B, Π and $\to$ are abstractions.

Examples of canonical elements in different Π-sets are:

$$\lambda((x)x) \in \Pi(\mathsf{Bool}, (x)\mathsf{Bool})$$
$$\lambda(\mathsf{succ}) \in \Pi(\mathsf{N}, (x)\mathsf{N})$$
$$\lambda((x)\lambda((y)x + y)) \in \Pi(\mathsf{N}, (x)\Pi(\mathsf{N}, (y)\mathsf{N}))$$

where N is the set of natural numbers and succ and $+$ the usual arithmetical operations, to be introduced in chapter 9. These expressions can also be written:

$$\lambda x.x \in (\Pi x \in \mathsf{Bool})\mathsf{Bool}$$
$$\lambda x.\mathsf{succ}(x) \in (\Pi x \in \mathsf{N})\mathsf{N}$$
$$\lambda x.\lambda y.x + y \in (\Pi x \in \mathsf{N})(\Pi x \in \mathsf{N})\mathsf{N}$$

An example of a non-canonical expression is:

$$\mathsf{apply}(\lambda x.x, \mathsf{false}) \in \mathsf{Bool}$$

The computation rule for apply justifies the equality

$$\mathsf{apply}(\lambda(b), a) = b(a) \in B(a)$$

For example,

$$\mathsf{apply}(\lambda x.x, \mathsf{false}) = \mathsf{false} \in \mathsf{Bool}$$

and

$$\mathsf{apply}(\lambda x.\mathsf{if}\ x\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{false}), \mathsf{true}) = \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{false} \in \mathsf{N}$$

which can be further evaluated to $\mathsf{0}$.

7.1 The formal rules and their justification

As defined previously, the canonical elements in $\Pi(A, B)$ are of the form $\lambda(b)$, where $b(x) \in B(x)$ when $x \in A$. We also defined two canonical elements $\lambda(b_1)$ and $\lambda(b_2)$ in $\Pi(A, B)$ to be equal if $b_1(x) = b_2(x) \in B(x)$ when $x \in A$. In order to see that $\Pi(A, B)$ is a set it only remains to verify that the equality on $\Pi(A, B)$ is extensional. But this is obvious since the free variables in $\lambda(b_1)$ and $\lambda(b_2)$ are also free in $b_1(x)$ and $b_2(x)$ and the equality on the family $B(x)$ over A is required to be extensional.

Therefore, $\Pi(A, B)$ is a set if A is a set and if $B(x)$ is a set under the assumption that $x \in A$. Hence, the formation rule is:

Π – formation

$$\frac{A\ set \qquad B(x)\ set\ \ [x \in A]}{\Pi(A, B)\ set}$$

Since the canonical elements in the set $\Pi(A,B)$ are of the form $\lambda(b)$ where $b(x) \in B(x)$ under the assumption that $x \in A$, we get

Π – introduction

$$\frac{b(x) \in B(x) \quad [x \in A]}{\lambda(b) \in \Pi(A,B)}$$

As mentioned earlier, the primitive non-canonical constant for the cartesian product is

$$\mathsf{apply}$$

of arity $\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$. We also introduce an infix form of apply by the definition

$$x \cdot y \quad \equiv \quad \mathsf{apply}(x,y)$$

The rule associated with apply is:

Π – elimination 1

$$\frac{f \in \Pi(A,B) \qquad a \in A}{\mathsf{apply}(f,a) \in B(a)}$$

We have to convince ourselves, from the way $\mathsf{apply}(f,a)$ is computed and the semantics of the judgement forms, that this rule is correct. That $f \in \Pi(A,B)$ means that

$$f \text{ has a value of the form } \lambda(b) \tag{1}$$

where

$$b(x) \in B(x) \quad [x \in A] \tag{2}$$

since it must have a canonical value in the set $\Pi(A,B)$ and all canonical values of $\Pi(A,B)$ have this form. By the definition of how $\mathsf{apply}(f,a)$ is computed and (1), we get that

$$\mathsf{apply}(f,a) \text{ is computed by computing } b(a). \tag{3}$$

Since $a \in A$, we get from (2) that

$$b(a) \in B(a) \tag{4}$$

(3) and (4) finally give us

$$\mathsf{apply}(f,a) \in B(a)$$

and thereby the elimination rule is justified.

The way $\mathsf{apply}(f,a)$ is computed gives the rule:

Π – equality 1

$$\frac{b(x) \in B(x) \quad [x \in A] \qquad a \in A}{\mathsf{apply}(\lambda(b),a) = b(a) \in B(a)}$$

since $b(x) \in B(x) \quad [x \in A]$ and $a \in A$ give that $b(a) \in B(a)$.

7.2 An alternative primitive non-canonical form

As an example of how the semantics can justify the introduction of a different non-canonical form, we will introduce an alternative to the selector apply in the Π-set.

For most sets, the non-canonical forms and their computation rules are based on the principle of structural induction. This principle says, that to prove that a property $B(a)$ holds for an arbitrary element a in the set A, prove that the property holds for each of the canonical elements in A. Similarly, to construct a program for an arbitrary element a in the set A, construct a program for each of the canonical forms of A. The computation rule for the non-canonical form in the Π-set does not follow this principle. It is chosen because the rule is well-known from the λ-calculus (β-reduction). The alternative non-canonical form is based on the principle of structural induction. We define the new non-canonical form as follows:

Introduce the constant $\mathsf{funsplit}$ of arity $(\mathbf{0}\otimes((\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}))\twoheadrightarrow\mathbf{0}$ and let the expression $\mathsf{funsplit}(f, d)$ be computed by the following rule:

1. Compute f.
2. If the value of f is $\lambda(b)$, then the value of $\mathsf{funsplit}(f, d)$ is the value of $d(b)$.

The expression f is to be an arbitrary element in the set $\Pi(A, B)$ and $d(y)$ is a program in the set $C(\lambda(y))$ under the assumption that $y(x) \in B(x)\ \ [x \in A]$. Notice that this is a higher order assumption, an assumption in which an assumption is made. The variable y is of arity $\mathbf{0}\twoheadrightarrow\mathbf{0}$, i.e. it is a function variable, i.e. a variable standing for an abstraction. Note that a function variable is something quite different from an element variable ranging over a Π set.

The alternative elimination rule becomes:

Π – elimination 2

$$\frac{\begin{array}{l} f \in \Pi(A, B) \\ C(v)\ set\ \ [v \in \Pi(A, B)] \\ d(y) \in C(\lambda(y))\ \ [y(x) \in B(x)\ \ [x \in A]] \end{array}}{\mathsf{funsplit}(f, d) \in C(f)}$$

We can justify Π-elimination 2 in the following way: If $f \in \Pi(A, B)$ it follows from the meaning of this judgement form that f must have a canonical element as value. The canonical elements in the Π set are of the form $\lambda(b)$, so f has a value of the form $\lambda(b)$ and

$$f = \lambda(b) \in \Pi(A, B) \tag{1}$$

where

$$b(x) \in B(x)\ \ [x \in A] \tag{2}$$

Since we know that $d(y) \in C(\lambda(y))$ whenever $y(x) \in B(x)\ \ [x \in A]$ and $b(x) \in B(x)\ \ [x \in A]$, we get

$$d(b) \in C(\lambda(b)) \tag{3}$$

From the computation rule for $\mathsf{funsplit}$ and from (1) we can conclude that $\mathsf{funsplit}(f, d)$ is computed by computing $d(b)$ and from (3) it follows that

$$\mathsf{funsplit}(f, d) \in C(\lambda(b)) \tag{4}$$

From the premise that $C(v)$ is a set under the assumption that $v \in \Pi(A, B)$ and from (1) it follows that

$$C(f) = C(\lambda(b)) \tag{5}$$

and now from (4) and (5) and the meaning of the judgement form $A = B$, it immediately follows that

$$\mathsf{funsplit}(f, b) \in C(f)$$

Hence, the first elimination rule is justified.

The computation rule for $\mathsf{funsplit}(\lambda(b), b)$ gives the equality rule:

Π – equality 2

$$\frac{\begin{array}{l} b(x) \in B(x) \;\; [x \in A] \\ C(v) \; set \;\; [v \in \Pi(A, B)] \\ d(y) \in C(\lambda(y)) \;\; [y(w) \in B(w) \;\; [w \in A]] \end{array}}{\mathsf{funsplit}(\lambda(b), d) = d(b) \in C(\lambda(b))}$$

since $b(x) \in B(x) \;\; [x \in A]$ and $d(y) \in C(\lambda(y)) \;\; [y(w) \in B(w) \;\; [w \in A]]$ give $d(b) \in C(\lambda(b))$.

Now we can reintroduce the constant apply of arity $\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}$ by making an explicit definition

$$\mathsf{apply}(f, a) \quad \equiv \quad \mathsf{funsplit}(f, (x)(x(a)))$$

If we have defined apply in this way, the expression $\mathsf{apply}(f, a)$ will be computed in the following way. The program $\mathsf{apply}(f, a)$ is definitionally equal to $\mathsf{funsplit}(f, (x)(x(a)))$ which is computed by first computing the value of f. If the value is $\lambda(b)$ then continue to compute the value of the program $((x)(x(a)))(b)$, a program which is definitionally equal to $b(a)$.

We can also prove a counterpart to the first Π-elimination rule:

Theorem If $a \in A$ and $f \in \Pi(A, B)$, then $\mathsf{apply}(f, a) \in B(a)$.

Proof: Assume that $a \in A$ and $f \in \Pi(A, B)$. For some expression b, f must be equal to $\lambda(b)$ where

$$b(x) \in B(x) \;\; [x \in A] \tag{1}$$

Using the definition of apply, we get that $\mathsf{apply}(f, a)$ is computed by computing $\mathsf{funsplit}(\lambda(b), (x)x(a))$. The computation rule for $\mathsf{funsplit}$ gives that $\mathsf{apply}(f, a)$ is equal to $b(a)$. From (1) we get

$$b(a) \in B(a)$$

Hence,

$$\mathsf{apply}(f, a) \in B(a)$$

$\square$

7.3 Constants defined in terms of the Π set

7.3.1 The universal quantifier ($\forall$)

In order to define the universal quantifier, we introduce a new constant $\forall$ of arity $\mathbf{0}{\otimes}(\mathbf{0}{\twoheadrightarrow}\mathbf{0}){\twoheadrightarrow}\mathbf{0}$ and then make the explicit definition

$$\forall \quad \equiv \quad \Pi$$

Instead of using the somewhat unusual notation $\forall(A, B)$ for the universal quantifier, we will write $(\forall x \in A)B(x)$. The rules for the universal quantifier follow directly from the rules for the Π-set by reading $B(x)$ as a family of propositions and $(\forall x \in A)\,B(x)$ as a proposition. We get the following rules for the universal quantifier.

$\forall$ – formation

$$\frac{A\ \textit{prop} \qquad B(x)\ \textit{prop}\ \ [x \in A]}{(\forall x \in A)B(x)\ \textit{prop}}$$

$\forall$ – introduction

$$\frac{B(x)\ \textit{true}\ \ [x \in A]}{(\forall x \in A)B(x)\ \textit{true}}$$

$\forall$ – elimination 1

$$\frac{(\forall x \in A)B(x)\ \textit{true} \qquad a \in A}{B(a)\ \textit{true}}$$

The alternative elimination rule becomes

$\forall$ – elimination 2

$$\frac{(\forall x \in A)\,B(x)\ \textit{true} \qquad C\ \textit{prop} \qquad C\ \textit{true}\ \ [B(x)\ \textit{true}\ \ [x \in A]]}{C\ \textit{true}}$$

7.3.2 The function set ($\rightarrow$)

As we have already remarked, the cartesian product is a generalization of the formation of the set of functions from a set A to a set B, which we now get in the following way. We introduce a new constant $\rightarrow$ of arity $\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}$ and make the definition

$$\rightarrow(A, B) \quad \equiv \quad \Pi(A, (x)B)$$

Instead of $\rightarrow (A, B)$, we shall write $A \rightarrow B$. From the rules for Π we get, as special cases:

$\rightarrow$ – formation

$$\frac{A\ \textit{set} \qquad B\ \textit{set}\ \ [x \in A]}{A \rightarrow B\ \textit{set}}$$

where x must not occur free in B

$\rightarrow$ – introduction

$$\frac{b(x) \in B \quad [x \in A]}{\lambda(b) \in A \rightarrow B}$$

where x must not occur free in B

$\rightarrow$ – elimination

$$\frac{f \in A \rightarrow B \qquad a \in A}{\mathsf{apply}(f, a) \in B}$$

$\rightarrow$ – equality

$$\frac{b(x) \in B \quad [x \in A] \qquad a \in A}{\mathsf{apply}(\lambda(b), a) = b(a) \in B}$$

where x must not occur free in B or f

7.3.3 Implication ($\supset$)

The Heyting interpretation of implication is

> The implication $A \supset B$ is true if we can construct a function which when applied to a proof of A, yields a proof of B.

If we let the elements in the set A represent the proofs of the proposition A and similarly for the set (proposition) B, then we can see that the elements (functions) of $A \rightarrow B$ are exactly the constructions we require in the Heyting interpretation to prove $A \supset B$. So we get the implication $A \supset B$ simply by introducing a new constant $\supset$ of arity $\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$ and making the explicit definition

$$\supset \;\equiv\; \rightarrow$$

The rules for implication immediately follow from the rules for $\rightarrow$. By omitting the proof elements in the rules for implication we get the natural deduction rules:

$\supset$ – formation

$$\frac{A\ prop \qquad B\ prop\ \ [A\ true]}{A \supset B\ prop}$$

$\supset$ – introduction

$$\frac{B\ true\ \ [A\ true]}{A \supset B\ true}$$

$\supset$ – elimination

$$\frac{A \supset B\ true \qquad A\ true}{B\ true}$$

The alternative elimination rule becomes:

$$\frac{A \supset B\ true \qquad C\ prop \qquad C\ true\ \ [B\ true\ \ [A\ true]]}{C\ true}$$

Notice that the second premise of the formation rule is weaker than in the traditional rule. To show that $A \supset B$ is a proposition it is enough to show that A is a proposition and that B is a proposition under the assumption that A is true This rule has been suggested by Schroeder-Heister [96].

Example. Changing the order of universal quantifiers

From a constructive proof in natural deduction, it is always possible to obtain, by filling in the omitted constructions, a proof in type theory. Consider, for instance, the following proof in natural deduction:

Assume

$$(\forall x \in \mathsf{N})(\forall y \in \mathsf{Bool})\, Q(x, y)$$

$\forall$-elimination used twice, gives

$$Q(x, y) \quad [x \in \mathsf{N}, y \in \mathsf{Bool}]$$

By $\forall$-introduction (twice) we get

$$(\forall y \in \mathsf{Bool})(\forall x \in \mathsf{N})\, Q(x, y)$$

Finally by $\supset$-introduction

$$(\forall x \in \mathsf{N})(\forall y \in \mathsf{Bool})\, Q(x, y) \supset (\forall y \in \mathsf{Bool})(\forall x \in \mathsf{N})\, Q(x, y)$$

With the proof elements present, this proof becomes:

Assume

$$w \in (\Pi x \in \mathsf{N})(\Pi y \in \mathsf{Bool})\, Q(x, y)$$

By Π-elimination (twice) we get

$$\mathsf{apply}_2(w, x, y) \in Q(x, y) \quad [x \in \mathsf{N},\ y \in \mathsf{Bool}]$$

where

$$\mathsf{apply}_2(x, y, z) \quad \equiv \quad \mathsf{apply}(\mathsf{apply}(x, y), z)$$

and then by Π-introduction (twice)

$$\lambda y.\lambda x.\mathsf{apply}_2(w, x, y) \in (\Pi y \in \mathsf{Bool})(\Pi x \in \mathsf{N})\, Q(x, y)$$

Finally, by $\rightarrow$-introduction

$$\begin{array}{l} \lambda w.\lambda y.\lambda x.\mathsf{apply}_2(w, x, y) \in \\ \qquad (\Pi x \in \mathsf{N})(\Pi y \in \mathsf{Bool}) Q(x, y) \rightarrow (\Pi y \in \mathsf{Bool})(\Pi x \in \mathsf{N}) Q(x, y) \end{array}$$

Chapter 8

Equality sets

We have seen how to use set-forming operations to build up complex propositions from simpler ones, but so far we have only introduced the elementary propositions T (the truth) and $\perp$ (the absurdity). Since the judgemental equality cannot be used when building propositions, it is necessary to have an elementary proposition expressing that two elements are equal. Beside the equality sets, it is the universe and general trees, which are introduced later, which make it possible to have dependent sets.

We will introduce two different sets to express that a and b are equal elements of a set A. The first one, which we denote by $\mathsf{Id}(A, a, b)$ and which we will call intensional equality, will have an elimination rule which expresses an induction principle. The second one, which we denote by $\mathsf{Eq}(A, a, b)$, will have a strong elimination rule of a different form than the elimination rules for the other sets. With this set, judgemental equality will no longer be decidable and we will therefore avoid this equality when possible. It is only in the chapters on well-orderings and general trees we must use it. In the chapter on cartesian product of two sets, we will show that extensionally equal functions are equal in the sense of Eq. Hence, we will call these kind of equalities extensional equalities.

8.1 Intensional equality

The set $\mathsf{Id}(A, a, b)$, where Id is a primitive constant of arity $\mathbf{0}{\otimes}\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}$, will represent the judgement $a = b \in A$ as a set.

Id – formation

$$\frac{A\ set \qquad a \in A \qquad b \in A}{\mathsf{Id}(A, a, b)\ set}$$

The set $\mathsf{Id}(A, a, a)$ will have the member $\mathsf{id}(a)$ where $a \in A$ and id is a primitive constant of arity $\mathbf{0}{\twoheadrightarrow}\mathbf{0}$. So we have

Id – introduction

$$\frac{a \in A}{\mathsf{id}(a) \in \mathsf{Id}(A, a, a)}$$

By using Substitution in sets on $a = b \in A$ and $\mathsf{Id}(A, a, x)\ set\ \ [x \in A]$ we obtain

$\mathsf{Id}(A, a, a) = \mathsf{Id}(A, a, b)$. So, by Id –introduction 1 and Set equality we get the derived rule

Id – introduction'

$$\frac{a = b \in A}{\mathsf{id}(a) \in \mathsf{Id}(A, a, b)}$$

The primitive non-canonical constant of the equality set is idpeel of arity

$$(\mathbf{0} \otimes (\mathbf{0} \twoheadrightarrow \mathbf{0})) \twoheadrightarrow \mathbf{0}$$

The expression $\mathsf{idpeel}(c, d)$ is computed as follows:

1. $\mathsf{idpeel}(c, d)$ is evaluated by first evaluating c.
2. If c has value $\mathsf{id}(a)$ then the value of $\mathsf{idpeel}(c, d)$ is the value of $d(a)$.

The way a canonical element is introduced in an equality set and the computation rule for idpeel justifies the elimination rule:

Id – elimination

$$\frac{\begin{array}{l} a \in A \\ b \in A \\ c \in \mathsf{Id}(A, a, b) \\ C(x, y, z)\ set\ \ [x \in A,\ y \in A,\ z \in \mathsf{Id}(A, x, y)] \\ d(x) \in C(x, x, \mathsf{id}(x))\ \ [x \in A] \end{array}}{\mathsf{idpeel}(c, d) \in C(a, b, c)}$$

As for the other sets, the elimination rule expresses a principle of structural induction on an equality set, but the importance of the elimination rule in this case is more in that it is a substitution rule for elements which are equal in the sense of an equality set.

The way $\mathsf{idpeel}(c, d)$ is computed gives the rule:

Id – equality

$$\frac{\begin{array}{l} a \in A \\ C(x, y, z)\ set\ \ [x \in A,\ y \in A,\ z \in \mathsf{Id}(A, x, y)] \\ d(x) \in C(x, x, \mathsf{id}(x))\ \ [x \in A] \end{array}}{\mathsf{idpeel}(\mathsf{id}(a), d) = d(a) \in C(a, a, \mathsf{id}(a))}$$

Instead of $\mathsf{Id}(A, a, b)$ we will often write $a =_A b$.

Example. Symmetry and transitivity of equality

Let A be a set and a and b elements of A. Assume that

$$d \in \mathsf{Id}(A, a, b) \tag{8.1}$$

In order to prove symmetry, we must construct an element in $\mathsf{Id}(A, b, a)$. By putting $C \equiv (x, y, z)\mathsf{Id}(A, y, x)$ in Id-elimination we get, by Id-introduction,

$$\mathsf{idpeel}(d, \mathsf{id}) \in \mathsf{Id}(A, b, a)$$

so we have proved symmetry. Hence, we have the following derived rule:

Symmetry of propositional equality

$$\frac{d \in [a =_A b]}{\mathit{symm}(d) \in [b =_A a]}$$

where

$$\mathit{symm}(d) \;\equiv\; \mathsf{idpeel}(d, \mathsf{id})$$

To prove transitivity, we assume

$$e \in \mathsf{Id}(A, b, c) \tag{8.2}$$

where c is an element in A. We then have to construct an element in $\mathsf{Id}(A, a, c)$. Using Id-elimination with $C \;\equiv\; (x, y, z)(\mathsf{Id}(A, y, c) \to \mathsf{Id}(A, x, c))$ we get from $d \in \mathsf{Id}(A, a, b)$, by Π-introduction,

$$\mathsf{idpeel}(d, (x)\lambda y.y) \in \mathsf{Id}(A, b, c) \to \mathsf{Id}(A, a, c) \tag{8.3}$$

(8.2), (8.3) and Π-elimination give

$$\mathsf{apply}(\mathsf{idpeel}(d, (x)\lambda y.y), e) \in \mathsf{Id}(A, a, c)$$

and, hence, we have transitivity. So we have the following derived rule:

Transitivity of propositional equality

$$\frac{d \in [a =_A b] \qquad e \in [b =_A c]}{\mathit{trans}(d, e) \in [a =_A c]}$$

where

$$\mathit{trans}(d, e) \;\equiv\; \mathsf{apply}(\mathsf{idpeel}(d, (x)\lambda y.y), e)$$

Example. Substitution with equal elements

Assume that we have a set A and elements a and b of A. Assume also that $c \in \mathsf{Id}(A, a, b)$, $P(x)$ *set* $[x \in A]$ and $p \in P(a)$. By Π-introduction we get

$$\lambda x.x \in P(x) \to P(x)$$

Putting $C \;\equiv\; (x, y, z)(P(x) \to P(y))$ in Id-elimination we then get

$$\mathsf{idpeel}(c, (x)\lambda x.x) \in P(a) \to P(b)$$

from which we obtain, by Π-elimination,

$$\mathsf{apply}(\mathsf{idpeel}(c, (x)\lambda x.x), p) \in P(b)$$

So we have the derived rule

$$\frac{P(x)\ \mathit{set}\ \ [x \in A] \qquad a \in A \qquad b \in A \qquad c \in \mathsf{Id}(A, a, b) \qquad p \in P(a)}{\mathit{subst}(c, p) \in P(b)}$$

where

$$subst(c, p) \equiv \mathsf{apply}(\mathsf{idpeel}(c, (x)\lambda x.x), p)$$

If we suppress the proof-objects we get the rule

$$\frac{P(x)\ set\ \ [x \in A] \qquad a \in A \qquad b \in A \qquad \mathsf{Id}(A, a, b)\ true \qquad P(a)\ true}{P(b)\ true}$$

which corresponds to the usual substitution rule in predicate logic with equality.

Example. An equality involving the conditional expression

In this example we will prove, that for any set A

$$\mathsf{Id}(A, \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c, c)\ [b \in \mathsf{Bool}, c \in A]$$

is inhabited. We start by assuming that $c \in A$ and $b \in \mathsf{Bool}$. and will show that there is an element in $\mathsf{Id}(A, \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c, c)$ by case analysis on b.

1. $b = \mathsf{false}$: The Bool – equality rule gives

   $$\mathsf{if\ false\ then}\ c\ \mathsf{else}\ c = c \in A$$

   which, using Id – introduction, gives

   $$\mathsf{id}(c) \in \mathsf{Id}(A, \mathsf{if\ false\ then}\ c\ \mathsf{else}\ c, c)$$

2. $b = \mathsf{true}$: In the same way as above, we first get

   $$\mathsf{if\ true\ then}\ c\ \mathsf{else}\ c = c \in A$$

   by one of the Bool – equality rules, and then

   $$\mathsf{id}(c) \in \mathsf{Id}(A, \mathsf{if\ true\ then}\ c\ \mathsf{else}\ c, c)$$

   by Id – introduction.

Applying the Bool – elimination rule on the two cases, we finally get

$$\mathsf{if}\ b\ \mathsf{then}\ \mathsf{id}(c)\ \mathsf{else}\ \mathsf{id}(c) \in \mathsf{Id}(A, \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c, c)$$

8.2 Extensional equality

We will now give an alternative formulation of equality sets which will have a strong elimination rule of a different form than all the other sets.

In the semantics we have given, following [69, 70], the judgemental equality is more general than convertibility; we have only required that it should be an equivalence relation which is extensional with respect to substitution. The rules for the equality sets given in [69, 70] are different from those we are using. The formation rule is

Eq – formation

$$\frac{A\ set \qquad a \in A \qquad b \in A}{\mathsf{Eq}(A, a, b)\ set}$$

where Eq is a primitive constant of arity $\mathbf{0}{\otimes}\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}$.

There is at most one canonical element in an Eq-set:

Eq – introduction

$$\frac{a = b \in A}{\mathsf{eq} \in \mathsf{Eq}(A, a, b)}$$

which differs from the introduction rule for Id-sets in that eq is of arity $\mathbf{0}$ and, hence, a canonical element of $\mathsf{Eq}(A, a, b)$ does not depend on an element in A. The crucial difference, however, is the elimination rule:

Strong Eq – elimination

$$\frac{c \in \mathsf{Eq}(A, a, b)}{a = b \in A}$$

Unlike the elimination rules for the other sets, this elimination rule is not a structural induction principle.

We also need an elimination rule by which we can deduce that all elements in an Eq are equal to eq:

Eq – elimination 2

$$\frac{c \in \mathsf{Eq}(A, a, b)}{c = \mathsf{eq} \in \mathsf{Eq}(A, a, b)}$$

Using the two elimination rules for Eq, we can derive an induction rule for Eq, corresponding to Id-elimination,

$$\frac{\begin{array}{l} a \in A \\ b \in A \\ c \in \mathsf{Eq}(A, a, b) \\ C(x, y, z)\ set\ \ [x \in A,\ y \in A,\ z \in \mathsf{Eq}(A, x, y)] \\ d(x) \in C(x, x, \mathsf{eq})\ \ [x \in A] \end{array}}{d(a) \in C(a, b, c)}$$

To prove this rule, we assume the premises of the rule. By strong Eq-elimination and $c \in \mathsf{Eq}(A, a, b)$, we get

$$a = b \in A \tag{8.1}$$

From $a \in A$ and $d(x) \in C(x, x, \mathsf{eq})\ \ [x \in A]$ we obtain, by substitution,

$$d(a) \in C(a, a, \mathsf{eq}) \tag{8.2}$$

(1), Eq-elimination 2, (2) and substitution, finally give

$$d(a) \in C(a, b, c) \tag{8.3}$$

If we do not have sets formed by Eq in our formal theory it is possible to show, by metamathematical reasoning, that if $a = b \in A$ is derivable then a converts to b. That a converts to b is then understood in the usual way of combinatory logic with our computational rules for the noncanonical constants as reduction rules; in particular, it is not necessary to have lazy evaluation. The proof is by induction on the length of the derivation of $a = b \in A$. It is also possible to show that if $c \in \mathsf{Id}(A, a, b)$ is derivable and does not depend on any assumptions, then a converts to b; this is the reason why we call equalities formed by Id intensional. This result can be proved by normalization; such a proof is complicated but can be done, using standard techniques.

If we express propositional equalities by Eq it is no longer possible to understand judgemental equality as convertibility, because it is then possible to prove a judgemental equality by reasoning using propositions. So we may e.g. use induction when proving a judgement of the form $a(x) = b(x) \in A \ \ [x \in \mathsf{N}]$ by first proving $\mathsf{Eq}(A, a(x), b(x)) \ \ [x \in \mathsf{N}]$ and then applying the strong Eq-elimination rule.

8.3 η-equality for elements in a Π set

We have not formulated any judgemental rule corresponding to η-conversion, that is, we have no rule by which we can conclude

$$\lambda((x)\mathsf{apply}(f, x)) = f \in \Pi(A, B) \ \ [f \in \Pi(A, B)]$$

Although we do not have this judgemental equality we can prove, by using Π-elimination 3, that the corresponding Id judgement holds:

$$\mathsf{Id}(\Pi(A, B), \lambda((x)\mathsf{apply}(f, x)), f) \ \mathit{true} \ \ [f \in \Pi(A, B)] \tag{1}$$

(1) can be derived in the following way. By Π-equality we obtain

$$\lambda((x)\mathsf{apply}(\lambda(y), x)) = \lambda(y) \in \Pi(A, B) \ \ [y(x) \in B(x) \ \ [x \in A]]$$

from which we get, by Id-introduction,

$$\mathsf{id}(\lambda(y)) \in \mathsf{Id}(\Pi(A, B), \lambda((x)\mathsf{apply}(\lambda(y), x)), \lambda(y)) \ \ [y(x) \in B(x) \ \ [x \in A]] \tag{2}$$

Putting

$$D(\lambda(y)) \ \equiv \ \mathsf{Id}(\Pi(A, B), \lambda((x)\mathsf{apply}(\lambda(y), x)), \lambda(y))$$

in Π-elimination 3, we obtain from (2)

$$\mathsf{funsplit}(f, (y)\mathsf{id}(\lambda(y))) \in \mathsf{Id}(\Pi(A, B), \lambda((x)\mathsf{apply}(f, x)), f) \ \ [f \in \Pi(A, B)]$$

which shows that the judgement (1) holds.

A similar proof for Eq instead of Id gives a term t such that

$$t \in \mathsf{Eq}(\Pi(A, B), \lambda((x)\mathsf{apply}(f, x)), f) \ \ [f \in \Pi(A, B)]$$

By strong Eq-elimination, we then obtain

$$\lambda((x)\mathsf{apply}(f, x)) = f \in \ \ [f \in \Pi(A, B)]$$

So in the theory with Eq-sets, we have η-conversion on the judgemental level.

Chapter 9

Natural numbers

The constant N of arity $\mathbf{0}$ denotes the set of natural numbers. The rule for forming this set is simply

N – formation

$$\mathsf{N} \ set$$

The canonical constants $\mathsf{0}$ and succ of arities $\mathbf{0}$ and $\mathbf{0} \twoheadrightarrow \mathbf{0}$ respectively, are used for expressing the canonical elements in N. The object $\mathsf{0}$ is a canonical element in N and if a is an element in N then $\mathsf{succ}(a)$ is a canonical element in N. This is reflected in the following introduction rules:

N – introduction 1

$$\mathsf{0} \in \mathsf{N}$$

N – introduction 2

$$\frac{a \in \mathsf{N}}{\mathsf{succ}(a) \in \mathsf{N}}$$

We will often use the numerals $1, 2, \ldots$ to denote canonical elements in N.

If a and b are equal elements in N then $\mathsf{succ}(a)$ and $\mathsf{succ}(b)$ are equal canonical elements in N.

The basic way of proving that a proposition holds for all natural numbers is by mathematical induction: From $P(\mathsf{0})$ and that $P(x)$ implies $P(\mathsf{succ}(x))$ you may conclude that $P(n)$ holds for all natural numbers n. In order to be able to prove properties by induction on natural numbers in type theory, we introduce the selector natrec of arity $\mathbf{0} \otimes \mathbf{0} \otimes (\mathbf{0} \otimes \mathbf{0} \twoheadrightarrow \mathbf{0}) \twoheadrightarrow \mathbf{0}$. From a computational point of view, natrec makes it possible to make definitions by primitive recursion. The expression $\mathsf{natrec}(a, d, e)$ is computed as follows.

1. Evaluate a to canonical form.

2a. If the result of evaluating a is $\mathsf{0}$ then the value of the expression is the value of d.

2b. If the result of evaluating a is $\mathsf{succ}(b)$ then the value of the expression is the value of $e(b, \mathsf{natrec}(b, d, e))$.

So, defining a function f by the primitive recursion

$$\begin{cases} f(0) & = & d \\ f(n\oplus 1) & = & e(n, f(n)) \end{cases}$$

is in type theory expressed by the definition

$$f \equiv (n)\mathsf{natrec}(n, d, e)$$

For example, using natrec, we can define the constants $\oplus$ and $*$ of arity $\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$ by the explicit definitions

$$\begin{array}{rcl} \oplus(x,y) & \equiv & \mathsf{natrec}(x, y, (u,v)\,\mathsf{succ}(v)) \\ *(x,y) & \equiv & \mathsf{natrec}(x, \mathsf{0}, (u,v)\oplus(y,v)) \end{array}$$

expressing addition and multiplication, respectively. We will use the infix format and the ordinary precedence rules for $\oplus$ and $*$. These definitions correspond exactly to the usual definitions of addition and multiplication by primitive recursion.

The elimination rule for the natural numbers is:

N – elimination

$$\frac{\begin{array}{l} a \in \mathsf{N} \\ d \in C(\mathsf{0}) \\ C(v)\ set\ \ [v \in \mathsf{N}] \\ e(x,y) \in C(\mathsf{succ}(x))\ \ [x \in \mathsf{N},\ y \in C(x)] \end{array}}{\mathsf{natrec}(a, d, e) \in C(a)}$$

In order to justify N-elimination we assume the premises $a \in \mathsf{N}$, $d \in C(\mathsf{0})$ and $e(x,y) \in C(\mathsf{succ}(x))\ \ [x \in \mathsf{N},\ y \in C(x)]$. We want to convince ourselves that the conclusion is correct, i.e. that the value of $\mathsf{natrec}(a,d,e)$ is a canonical element in $C(a)$

1. If the value of a is $\mathsf{0}$ then the value of $\mathsf{natrec}(a,d,e)$ is the value of d which by the second premise is a canonical element in $C(\mathsf{0})$. From the extensionality of the family C it follows that $C(a) = C(\mathsf{0})$ and, hence, that the value of $\mathsf{natrec}(a,d,e)$ is a canonical element in $C(a)$.

2. If the value of a is $\mathsf{succ}(b)$, where $b \in \mathsf{N}$, then the value of $\mathsf{natrec}(a,d,e)$ is the value of
$$e(b, \mathsf{natrec}(b,d,e)) \qquad (1)$$
It now remains to show that $\mathsf{natrec}(b,d,e) \in C(b)$. Then it follows from the meaning of the last premise that the value of (1) is a canonical element in $C(\mathsf{succ}(b))$ which by the extensionality of C is also a canonical element in $C(a)$. To show that $\mathsf{natrec}(b,d,e) \in C(b)$ we compute the value of $\mathsf{natrec}(b,d,e)$ by first computing b. The value of b is either $\mathsf{0}$ or $\mathsf{succ}(c)$, where $c \in \mathsf{N}$.

   (a) If the value of b is $\mathsf{0}$ then by a similar reasoning as in (1) we conclude that the value of $\mathsf{natrec}(b,d,e)$ is a canonical element in $C(b)$.

(b) Otherwise, if the value of b is $\mathsf{succ}(c)$, where $c \in \mathsf{N}$, then we proceed as in (2) to show that the value of $\mathsf{natrec}(b, d, e)$ is a canonical element in $C(b)$. This method will terminate since all natural numbers are obtained by applying the successor function to $\mathsf{0}$ a finite number of times.

If some of the constructions in the elimination rule are omitted, Peano's fifth axiom is obtained:

$$\frac{a \in \mathsf{N} \qquad C(v)\ \mathit{prop}\ \ [v \in \mathsf{N}] \qquad C(\mathsf{0})\ \mathit{true} \qquad C(\mathsf{succ}(x))\ \mathit{true}\ \ [C(x)\ \mathit{true}]}{C(a)\ \mathit{true}}$$

Notice that the justification of the induction rule comes from N-elimination which was justified by using mathematical induction on the semantical level. Of course, neither N-elimination nor Peano's fifth axiom can be justified without the knowledge that N is well-founded, which is something which we must understand from the inductive definition of the canonical elements in N, that is, from the introduction rules for N.

Finally we have the equality rules, which are justified from the computation rule for natrec.

N – equality 1

$$\frac{\begin{array}{l} C(v)\ \mathit{set}\ \ [v \in \mathsf{N}] \\ d \in C(\mathsf{0}) \\ e(x,y) \in C(\mathsf{succ}(x))\ \ [x \in \mathsf{N},\ y \in C(x)] \end{array}}{\mathsf{natrec}(\mathsf{0}, d, e) = d \in C(\mathsf{0})}$$

N – equality 2

$$\frac{\begin{array}{l} C(v)\ \mathit{set}\ \ [v \in \mathsf{N}] \\ a \in \mathsf{N} \\ d \in C(\mathsf{0}) \\ e(x,y) \in C(\mathsf{succ}(x))\ \ [x \in \mathsf{N},\ y \in C(x)] \end{array}}{\mathsf{natrec}(\mathsf{succ}(a), d, e) = e(a, \mathsf{natrec}(a, d, e)) \in C(\mathsf{succ}(a))}$$

The proposition in type theory corresponding to Peano's fourth axiom needs the Universe set to be proved, so we have to postpone this until later.

Example. The typing of the $\oplus$ -operator

The constant $\oplus$ was defined by

$$\oplus(x, y) \ \equiv\ \mathsf{natrec}(x, y, (u, v)\,\mathsf{succ}(v))$$

We will now formally show that

$$\oplus(x, y) \in \mathsf{N}\ \ [x \in \mathsf{N},\ y \in \mathsf{N}]$$

By the rule of assumption we get

$$x \in \mathsf{N}\ \ [x \in \mathsf{N}] \tag{9.1}$$
$$y \in \mathsf{N}\ \ [y \in \mathsf{N}] \tag{9.2}$$

Assumption and N-introduction 2 give

$$\mathsf{succ}(v) \in \mathsf{N} \quad [v \in \mathsf{N}] \tag{9.3}$$

By applying N-elimination on (9.1), (9.2) and (9.3) we get

$$\mathsf{natrec}(x, y, (u, v)\, \mathsf{succ}(v)) \in \mathsf{N} \quad [x \in \mathsf{N},\ y \in \mathsf{N}]$$

that is, by definition,

$$\oplus(x, y) \in \mathsf{N} \quad [x \in \mathsf{N},\ y \in \mathsf{N}]$$

Example. Peano's third axiom

Peano's third axiom is that if the successor of two natural numbers are equal then the natural numbers are equal. We can formulate this in type theory as a derived rule:

$$\frac{m \in \mathsf{N} \qquad n \in \mathsf{N} \qquad \mathsf{succ}(m) = \mathsf{succ}(n) \in \mathsf{N}}{m = n \in \mathsf{N}}$$

In the derivation of this rule we will use the predecessor function *pred*, which is defined by

$$\mathit{pred} \ \equiv \ (x)\mathsf{natrec}(x, \mathsf{0}, (u, v)u)$$

Since $\mathsf{0} \in \mathsf{N}$ and $u \in \mathsf{N}$ $[u \in \mathsf{N}]$, the definition of *pred* and N-elimination give

$$\mathit{pred}(x) \in \mathsf{N}\ [x \in \mathsf{N}] \tag{9.1}$$

Let $m \in \mathsf{N}$, $n \in \mathsf{N}$ and

$$\mathsf{succ}(m) = \mathsf{succ}(n) \in \mathsf{N} \tag{9.2}$$

By (9.1), (9.2) and Substitution in equal elements, we get

$$\mathit{pred}(\mathsf{succ}(m)) = \mathit{pred}(\mathsf{succ}(n)) \in \mathsf{N} \tag{9.3}$$

The definition of *pred* and N-equality 2 give

$$\mathit{pred}(\mathsf{succ}(m)) \ = \ m \ \in \mathsf{N} \tag{9.4}$$

$$\mathit{pred}(\mathsf{succ}(n)) \ = \ n \ \in \mathsf{N} \tag{9.5}$$

Using symmetry and transitivity of judgemental equality on (9.3) – (9.5), we finally obtain

$$m = n \in \mathsf{N}$$

and, hence we have Peano's third axiom as a derived rule.

Instead of formulating Peano's third axiom as a derived rule, we could express it as a proposition, using an equality set:

$$(\forall x \in \mathsf{N})(\forall y \in \mathsf{N})(\mathsf{Id}(\mathsf{N}, \mathsf{succ}(x), \mathsf{succ}(y)) \supset \mathsf{Id}(\mathsf{N}, x, y))$$

This proposition can be proved in a similar way as the derived rule, using the rules for Id instead of the rules for judgemental equality. Note that these two formulations of Peano's third axiom are inherently different: the first formulation is about judgements but the second is a proposition.

Chapter 10

Lists

In order to form the set of lists of elements in a set A, we introduce three new constants: List of arity $\mathbf{0}{\twoheadrightarrow}\mathbf{0}$, nil of arity $\mathbf{0}$ and cons of arity $\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}$. If A is a set, then the canonical elements in $\mathsf{List}(A)$ are nil and $\mathsf{cons}(a, l)$ where a is an element in A and l is an element in $\mathsf{List}(A)$. If $a = a' \in A$ and $l = l' \in \mathsf{List}(A)$ then $\mathsf{cons}(a, l)$ and $\mathsf{cons}(a', l')$ are equal canonical elements in $\mathsf{List}(a)$.

We have the following rule for forming list sets.

List – formation

$$\frac{A \;\; set}{\mathsf{List}(A) \;\; set}$$

In order to be able to use infix notation when constructing lists, we make the definition

$$a.l \;\equiv\; \mathsf{cons}(a, l)$$

The introduction rules are:

List – introduction

$$\mathsf{nil} \in \mathsf{List}(A) \qquad\qquad \frac{a \in A \qquad l \in \mathsf{List}(A)}{a.l \in \mathsf{List}(A)}$$

The primitive non-canonical constant $\mathsf{listrec}$ of arity $\mathbf{0}{\otimes}\mathbf{0}{\otimes}(\mathbf{0}{\otimes}\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}){\twoheadrightarrow}\mathbf{0}$ is introduced in order to express recursion on lists. The expression $\mathsf{listrec}(l, c, e)$ is computed as follows:

1. First compute l.

2a. If the value of l is nil, then the value of $\mathsf{listrec}(l, c, e)$ is the value of c.

2b. If the value of l is $a.l_1$ then the value of $\mathsf{listrec}(l, c, e)$ is the value of $e(a, l_1, \mathsf{listrec}(l_1, c, e))$.

The following rules are justified in the same way as the corresponding rules for natural numbers:

List – elimination

$$\frac{\begin{array}{l} l \in \mathsf{List}(A) \\ C(v)\ set\ \ [v \in \mathsf{List}(A)] \\ c \in C(\mathsf{nil}) \\ e(x,y,z) \in C(x.y)\ \ [x \in A,\ y \in \mathsf{List}(A),\ z \in C(y)] \end{array}}{\mathsf{listrec}(l,c,e) \in C(l)}$$

List – equality 1

$$\frac{\begin{array}{l} C(v)\ set\ \ [v \in \mathsf{List}(A)] \\ c \in C(\mathsf{nil}) \\ e(x,y,z) \in C(x.y)\ \ [x \in A,\ y \in \mathsf{List}(A),\ z \in C(y)] \end{array}}{\mathsf{listrec}(\mathsf{nil},c,e) = c \in C(\mathsf{nil})}$$

List – equality 2

$$\frac{\begin{array}{l} a \in A \\ l \in \mathsf{List}(A) \\ C(v)\ set\ \ [v \in \mathsf{List}(A)] \\ c \in C(\mathsf{nil}) \\ e(x,y,z) \in C(x.y))\ \ [x \in A,\ y \in \mathsf{List}(A),\ z \in C(y)] \end{array}}{\mathsf{listrec}(a.l,c,e) = e(a,l,\mathsf{listrec}(l,c,e)) \in C(a.l)}$$

Example. Associativity of append

The function *append* concatenates two lists and is defined by

$$append(l_1, l_2)\ \equiv\ \mathsf{listrec}(l_1, l_2, (x,y,z)\,x.z))$$

We will use the binary infix operator @ for *append*,

$$l_1 @ l_2\ \equiv\ append(l_1, l_2)$$

From the List-elimination rule, it follows directly that

$$l_1 @ l_2 \equiv \mathsf{listrec}(l_1, l_2, (x,y,z)\,x.z) \in \mathsf{List}(A)\ \ [l_1 \in \mathsf{List}(A), l_2 \in \mathsf{List}(A)]$$

By applying List-equality to the definition of $l_1 @ l_2$ we get the following equalities

$$\begin{cases} \mathsf{nil} @ l_2 & = & l_2 \in \mathsf{List}(A) \\ a.l_1 @ l_2 & = & a.(l_1 @ l_2) \in \mathsf{List}(A) \end{cases}$$

which are the usual defining equations for *append*.

As a simple example, we are going to show how to formally prove that @ is associative, i.e. if $p, q, r \in \mathsf{List}(A)$ then

$$p@(q@r) =_{\mathsf{List}(A)} (p@q)@r$$

is a true proposition. We will write L instead of $\mathsf{List}(A)$. We first give the informal proof and then translate it to a proof in type theory.

We sometimes use the following notation, introduced by Dijkstra, for informal proofs:

$$
\begin{array}{ll}
 & t_1 \\
= & \{ \text{ informal argument why } t_1 = t_2 \text{ } \} \\
 & t_2 \\
= & \{ \text{ informal argument why } t_2 = t_3 \text{ } \} \\
 & t_3
\end{array}
$$

This is sometimes generalized from equality to another transitive operator.

The proof proceeds by induction on the list p. For the base case, we have to show that $\mathsf{nil}@(q@r) =_L (\mathsf{nil}@q)@r$, which is done by simplifying the two sides of the equation:

$$
\begin{array}{ll}
 & \mathsf{nil}@(q@r) \\
= & \{ \text{ definition of } @ \text{ } \} \\
 & q@r
\end{array}
$$

$$
\begin{array}{ll}
 & (\mathsf{nil}@q)@r \\
= & \{ \text{ definition of } @\text{, substitution } \} \\
 & q@r
\end{array}
$$

The induction step starts in a similar way and ends in using the induction hypothesis. We are going to show that $(x.y)@(q@r) =_L ((x.y)@q)@r$ from the assumption that $y@(q@r) =_L (y@q)@r$. First, the left hand side:

$$
\begin{array}{ll}
 & (x.y)@(q@r) \\
= & \{ \text{ definition of } @ \text{ } \} \\
 & x.(y@(q@r))
\end{array}
$$

Then the right hand side:

$$
\begin{array}{ll}
 & ((x.y)@q)@r \\
= & \{ \text{ definition of } @\text{, substitution } \} \\
 & (x.(y@q))@r \\
= & \{ \text{ definition of } @ \text{ } \} \\
 & x.((y@q)@r) \\
=_L & \{ \text{ induction assumption, substitution } \} \\
 & x.(y@(q@r))
\end{array}
$$

The proof is by induction on the list p, so in type theory we use List-elimination. We have to prove the three premises

1. $p \in L$, which we already have assumed.
2. Find an element in $[\mathsf{nil}@(q@r) =_L (\mathsf{nil}@q)@r]$.
3. Under the assumptions that $x \in A$, $y \in L$ and $z \in [y@(q@r) =_L (y@q)@r]$ find an element in $[(x.y)@(q@r) =_L ((x.y)@q)@r]$.

The following is a formal proof of the two parts in the base step. First we have the simplification of the left hand side:

$$\dfrac{\dfrac{q \in L \qquad r \in L}{q@r \in L} \qquad \dfrac{x \in A \qquad z \in L}{x.z \in L}\ \mathsf{List}_{-\text{intro}}}{\underbrace{\mathsf{listrec}(\mathsf{nil}, (q@r), (x,y,z)x.z)}_{\mathsf{nil}@(q@r)} = q@r \in L}\ \mathsf{List}_{-\text{equality}}$$

And then we have the simplification of the right hand side:

$$\dfrac{\dfrac{\mathsf{nil} \in L \qquad \dfrac{x \in A \qquad z \in L}{x.z \in L}\ \mathsf{List}_{-\text{intro}}}{\mathsf{nil}@q = q \in L}\ \mathsf{List}_{-\text{equality}} \qquad \dfrac{u \in L \qquad r \in L}{u@r \in L}}{(\mathsf{nil}@q)@r = q@r \in L}\ \text{subst}$$

These two steps are combined using symmetry and transitivity of equality to obtain the conclusion

$$\mathsf{nil}@(q@r) = (\mathsf{nil}@q)@r \in L$$

and hence, using Id-introduction, we get

$$\mathsf{id}(\mathsf{nil}@(q@r)) \in [\mathsf{nil}@(q@r) =_L (\mathsf{nil}@q)@r]$$

The induction step is formalized in almost the same way, the only complication is in the last step where the induction assumption is used. Here we must switch from definitional equality to propositional equality, and therefore we will use the derived rules for substitution and transitivity from chapter 8.

In the first part of the induction step we have shown that

$$(x.y)@(q@r) = x.(y@(q@r)) \in L$$

and in the second part (except for the last step)

$$((x.y)@q)@r = x.((y@q)@r) \in L$$

Id-introduction then gives

$$\mathsf{id}((x.y)@(q@r)) \in [(x.y)@(q@r) =_L x.(y@(q@r))] \tag{10.1}$$

and

$$\mathsf{id}(x.((y@q)@r)) \in [x.((y@q)@r) =_L ((x.y)@q)@r] \tag{10.2}$$

We then apply the substitution rule for propositional equality on the induction assumption and the family

$$P(u) \equiv [x.(y@(q@r)) =_L x.u]$$

and obtain

$$\mathit{subst}(z, \mathsf{id}(x.(y@(q@r)))) \in [x.(y@(q@r)) =_L x.((y@q)@r)] \tag{10.3}$$

We can now use transitivity of propositional equality twice on (10.1), (10.3) and (10.2) to get

$$
\begin{array}{l}
\mathit{trans}(\mathit{trans}(\mathsf{id}((x.y)@(q@r)),\\
\qquad\qquad \mathit{subst}(z, \mathsf{id}(x.(y@(q@r))))\\
\qquad\qquad),\\
\qquad \mathsf{id}(x.((y@q)@r))\\
\qquad) \in [(x.y)@(q@r) =_L ((x.y)@q)@r]
\end{array}
$$

We can now combine the solution of the base step and the induction step, using List-elimination:

$$
\begin{array}{l}
\mathsf{listrec}(p,\\
\qquad \mathsf{id}(\mathsf{nil}@(q@r)),\\
\qquad (x, y, u)\mathit{trans}(\mathit{trans}(\mathsf{id}((x.y)@(q@r)),\\
\qquad\qquad\qquad\qquad \mathit{subst}(z, \mathsf{id}(x.(y@(q@r))))\\
\qquad\qquad\qquad\qquad),\\
\qquad\qquad\qquad \mathsf{id}(x.((y@q)@r))\\
\qquad\qquad\qquad)\\
\qquad) \in [p@(q@r) =_L (p@q)@r]
\end{array}
$$

which concludes the proof. This example shows the practical importance of using the judgement form A *true*. The explicit element we have found in the set $[p@(q@r) =_L (p@q)@r]$ is not a very interesting program. A more elaborate example is found in [99].

Chapter 11

Cartesian product of two sets

If A and B are sets, then the cartesian product

$$A \times B$$

can be formed. The canonical elements of this set are pairs

$$\langle a, b \rangle$$

where $a \in A$ and $b \in B$. The primitive noncanonical constant for the cartesian product is split of arity $\mathbf{0}\otimes(\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$. If $p \in A \times B$ and $e(x, y) \in C(\langle x, y \rangle)$ under the assumptions that $x \in A$ and $y \in C$, then

$$\mathsf{split}(p, e) \in C(p)$$

which is evaluated as follows:

1. $\mathsf{split}(p, e)$ is evaluated by first evaluating p.
2. If p has value $\langle a, b \rangle$ then the value of $\mathsf{split}(p, e)$ is the value of $e(a, b)$.

The split expression is similar to a let expression in ML of the form

```
case p of (x,y) => e(x,y)
```

The ordinary projection operators are defined by:

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

We will later see that the cartesian product $A \times B$ is a special case of the disjoint union $(\Sigma x \in A)B$.

11.1 The formal rules

In order to define $A \times B$, we have to introduce a new constant $\times$ of arity $\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$. We will write $A \times B$ instead of $\times(A, B)$. The set $A \times B$ is introduced by the rule

$\times$ – formation

$$\frac{A \; set \qquad B \; set}{A \times B \; set}$$

In order to explain the set $A \times B$, we must explain what a canonical element in the set is and what it means for two canonical elements to be equal. For this purpose, we introduce a new constant $\langle\rangle$ of arity $\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}$. Instead of writing $\langle\rangle(a, b)$, we will write $\langle a, b\rangle$. The canonical elements in the set $A \times B$ are given by the following rule:

$\times$ – introduction

$$\frac{a \in A \qquad b \in B}{\langle a, b\rangle \in A \times B}$$

So the canonical elements in the set $A \times B$ are of the form $\langle a, b\rangle$, where $a \in A$ and $b \in B$.

The elimination rule for the cartesian product is:

$\times$ – elimination

$$\frac{p \in A \times B \qquad C(v) \; set \;\; [v \in A \times B] \qquad e(x, y) \in C(\langle x, y\rangle) \;\; [x \in A, y \in B]}{\mathsf{split}(p, e) \in C(p)}$$

We can justify this rule, using the computation rule for split and the semantical explanations, in the following way.

The premise that $p \in A \times B$ means that the value of p is a canonical element in the set $A \times B$, which by the introduction rule is of the form $\langle a, b\rangle$, where $a \in A$ and $b \in B$. We are going to show that

$$\mathsf{split}(p, e) \in C(p)$$

i.e. that the value of $\mathsf{split}(p, e)$ is a canonical element in $C(p)$. It follows from the computation rule for split that the value of $\mathsf{split}(p, e)$ is the value of $e(a, b)$. The meaning of the second premise gives that

$$e(a, b) \in C(\langle a, b\rangle)$$

i.e. the value of $\mathsf{split}(p, e)$ is a canonical element in $C(\langle a, b\rangle)$.

From the premise

$$C(v) \; set \;\; [v \in A \times B]$$

it follows that

$$C(\langle a, b\rangle) = C(p)$$

since $\langle a, b\rangle = p \in A \times B$. Hence, canonical elements in $C(\langle a, b\rangle)$ are also canonical elements in $C(p)$, in particular the value of $\mathsf{split}(p, e)$ is a canonical element in $C(p)$.

The computation rule also justifies the equality rule

$\times$ – equality

$$\frac{a \in A \qquad b \in B \qquad e(x, y) \in C(\langle x, y\rangle) \;\; [x \in A, y \in B]}{\mathsf{split}(\langle a, b\rangle, e) = e(a, b) \in C(\langle a, b\rangle)}$$

We can define logical conjunction by

$$\& \;\equiv\; \times$$

and we get the usual natural deduction rules for conjunction by omitting the constructions in the rules above:

& – formation

$$\frac{A \; prop \qquad B \; prop}{A \,\&\, B \; prop}$$

& – introduction

$$\frac{A \; true \qquad B \; true}{A \,\&\, B \; true}$$

& – elimination

$$\frac{A \,\&\, B \; true \qquad C \; prop \qquad C \; true \;\; [A \; true, \; B \; true]}{C \; true}$$

It is also convenient to have a constant for logical equivalence:

$$A \Leftrightarrow B \;\equiv\; (A \supset B) \,\&\, (B \supset A)$$

Example. Projection is the inverse of pairing

In the lambda-calculus it is not possible to define pairing and projection so that $\langle \mathit{fst}(z), \mathit{snd}(z) \rangle$ converts to z. In type theory we have only defined the computation rules for closed expressions. However, we can prove

$$(z =_{A \times B} \langle \mathit{fst}(z), \mathit{snd}(z) \rangle) \; true \;\; [z \in A \times B] \tag{1}$$

in the following way. By $\times$ – equality and the definitions of *fst* and *snd* we get

$$\mathit{fst}(\langle x, y \rangle) = x \in A \;\; [x \in A, \; y \in B]$$

and

$$\mathit{snd}(\langle x, y \rangle) = y \in B \;\; [x \in A, \; y \in B]$$

$\times$-introduction 2 then gives

$$\langle \mathit{fst}(\langle x, y \rangle), \mathit{snd}(\langle x, y \rangle) \rangle = \langle x, y \rangle \in A \times B \;\; [x \in A, \; y \in B]$$

We can now apply symmetry and Id-introduction to the last equation to get

$$\mathsf{id}(\langle x, y \rangle) \in (\langle x, y \rangle =_{A \times B} \langle \mathit{fst}(\langle x, y \rangle), \mathit{snd}(\langle x, y \rangle) \rangle) \;\; [x \in A, \; y \in B]$$

from which we get, by $\times$-elimination,

$$\mathsf{split}(z, (x, y)\mathsf{id}(\langle x, y \rangle)) \in (z =_{A \times B} \langle \mathit{fst}(z), \mathit{snd}(z) \rangle) \;\; [z \in A \times B]$$

Hence, we have proved (1).

11.2 Extensional equality on functions

That two functions f and g in a cartesian product $\Pi(A,B)$ are extensionally equal means that

$$(\forall x \in A)\, \mathsf{Id}(B(x), \mathsf{apply}(f,x), \mathsf{apply}(g,x))$$

is true. We cannot expect the equality expressed by Id to be extensional, i.e. we cannot expect

$$(\forall x \in A)\, \mathsf{Id}(B(x), \mathsf{apply}(f,x), \mathsf{apply}(g,x)) \quad \Leftrightarrow \quad \mathsf{Id}(\Pi(A,B), f, g)$$

to hold in general. Informally, we can see that in the following way. Since the set $\mathsf{Id}(\Pi(A,B), f, g)$ does not depend on any assumptions, it is nonempty if and only if f and g are convertible; this follows from a result mentioned in section 8.2. Hence, it is decidable whether $\mathsf{Id}(\Pi(A,B), f, g)$ holds or not. But we cannot even expect

$$(\forall x \in \mathsf{N})\mathsf{Id}(\mathsf{N}, \mathsf{apply}(f,x), \mathsf{apply}(g,x))$$

to be decidable. However, Eq is extensional on a cartesian product:

Theorem Under the assumptions $f \in \Pi(A,B)$ and $g \in \Pi(A,B)$ it holds that

$$(\forall x \in A)\, \mathsf{Eq}(B(x), \mathsf{apply}(f,x), \mathsf{apply}(g,x)) \quad \Leftrightarrow \quad \mathsf{Eq}(\Pi(A,B), f, g)$$

Proof: We first prove the implication from right to left. So let us assume $\mathsf{Eq}(\Pi(A,B), f, g)$. By the strong Eq-elimination rule, we then obtain

$$f = g \in \Pi(A,B)$$

which, by equality rules, gives

$$\mathsf{apply}(f,x) = \mathsf{apply}(g,x) \in B(x) \;\; [x \in A]$$

Hence, by Eq-introduction,

$$\mathsf{eq} \in \mathsf{Eq}(B(x), \mathsf{apply}(f,x), \mathsf{apply}(g,x))$$

which, by Π-introduction, gives

$$\lambda((x)\mathsf{eq}) \in (\forall x \in A)\, \mathsf{Eq}(B(x), \mathsf{apply}(f,x), \mathsf{apply}(g,x))$$

as desired.

For the proof of the implication from left to right, assume

$$(\forall x \in A)\, \mathsf{Eq}(B(x), \mathsf{apply}(f,x), \mathsf{apply}(g,x))$$

By Π-elimination and the strong Eq-elimination rule, we then obtain

$$\mathsf{apply}(f,x) = \mathsf{apply}(g,x) \in B(x) \;\; [x \in A]$$

which, by equality rules, gives

$$\lambda((x)\mathsf{apply}(f,x)) = \lambda((x)\mathsf{apply}(g,x)) \in \Pi(A,B)$$

By η-conversion, which we have in the theory with Eq-sets, we then obtain

$$f = g \in \Pi(A, B)$$

Hence, by Eq-introduction,

$$\mathsf{eq} \in \mathsf{Eq}(\Pi(A, B), f, g)$$

$\square$

Chapter 12

Disjoint union of two sets

We introduce the constant $+$ of arity $\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$ to represent the disjoint union of two sets. We will often use infix notation instead of the standard prefix one, and, therefore, introduce the definition:

$$A + B \equiv +(A, B)$$

To form $A + B$ we have the rule

$+$ – formation

$$\frac{A \; set \qquad B \; set}{A + B \; set}$$

In order to form elements in a disjoint union of two sets, we introduce the canonical constants inl and inr, both of arity $\mathbf{0}\twoheadrightarrow\mathbf{0}$.

Let A and B be sets. The canonical elements in $A + B$ are given by the following introduction rules

$+$ – introduction

$$\frac{a \in A \qquad B \; set}{\mathsf{inl}(a) \in A + B} \qquad \frac{A \; set \qquad b \in B}{\mathsf{inr}(b) \in A + B}$$

The selector for $A+B$ is the constant when of arity $\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$. The expression $\mathsf{when}(c, d, e)$ is computed in the following way:

1. Evaluate c to canonical form.

2a. If the value of c is of the form $\mathsf{inl}(a)$, then continue by evaluating $d(a)$.

2b. If the value of c is of the form $\mathsf{inr}(b)$, then continue by evaluating $e(b)$.

From this computation rule, we get the elimination rule:

$+$ – elimination

$$\frac{\begin{array}{l} c \in A + B \\ C(v) \; set \;\; [v \in A + B] \\ d(x) \in C(\mathsf{inl}(x)) \;\; [x \in A] \\ e(y) \in C(\mathsf{inr}(y)) \;\; [y \in B] \end{array}}{\mathsf{when}(c, d, e) \in C(c)}$$

We also get the equality rules:

+ – equality

$$\frac{\begin{array}{l} a \in A \\ C(v)\ set\ \ [v \in A+B] \\ d(x) \in C(\mathsf{inl}(x))\ \ [x \in A] \\ e(y) \in C(\mathsf{inr}(y))\ \ [y \in B] \end{array}}{\mathsf{when}(\mathsf{inl}(a), d, e) = d(a) \in C(\mathsf{inl}(a))}$$

$$\frac{\begin{array}{l} b \in B \\ C(v)\ set\ \ [v \in A+B] \\ d(x) \in C(\mathsf{inl}(x))\ \ [x \in A] \\ e(y) \in C(\mathsf{inr}(y))\ \ [y \in B] \end{array}}{\mathsf{when}(\mathsf{inr}(b), d, e) = e(b) \in C(\mathsf{inr}(b))}$$

Having defined disjoint union, we can introduce disjunction by the definition:

$$A \vee B \equiv A + B$$

and from the rules for $+$, we get the natural deduction rules for $\vee$:

$\vee$ – formation

$$\frac{A\ prop \qquad A\ prop}{A \vee B\ prop}$$

$\vee$ – introduction

$$\frac{A\ true}{A \vee B\ true} \qquad \frac{B\ true}{A \vee B\ true}$$

$\vee$ – elimination

$$\frac{A \vee B\ true \qquad C\ prop \qquad C\ true\ \ [A\ true] \qquad C\ true\ \ [B\ true]}{C\ true}$$

Chapter 13

Disjoint union of a family of sets

In order to be able to deal with the existential quantifier, we will now generalize the cartesian product of two sets to disjoint union on a family of sets. We therefore introduce a new constant Σ of arity $\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$. Let A be a set and B a family of sets over A, i.e.

$$B(x)\ set\ \ [x \in A]$$

then we may conclude that $\Sigma(A, B)$ is a set. So we have the formation rule

Σ – formation

$$\frac{A\ set \qquad B(x)\ set\ \ [x \in A]}{\Sigma(A, B)\ set}$$

A canonical element in the set $\Sigma(A, B)$ is of the form $\langle a, b\rangle$ where a is an element in the set A and b an element in the set $B(a)$. Two canonical elements $\langle a, b\rangle$ and $\langle a', b'\rangle$ are equal if $a = a' \in A$ and $b = b' \in B(a)$. So we have the introduction rule

Σ – introduction

$$\frac{a \in A \qquad B(x)\ set\ \ [x \in A] \qquad b \in B(a)}{\langle a, b\rangle \in \Sigma(A, B)}$$

We get the cartesian product of two sets if we make the following definition:

$$A \times B \equiv \Sigma(A, (x)B)$$

In the chapter on cartesian product of two sets, we introduced the non-canonical constant split. The computation rules for split justify the elimination rule

Σ – elimination

$$\frac{\begin{array}{l} c \in \Sigma(A, B) \\ C(v)\ set\ \ [v \in \Sigma(A, B)] \\ d(x, y) \in C(\langle x, y\rangle)\ \ [x \in A,\ y \in B(x)] \end{array}}{\mathsf{split}(c, d) \in C(c)}$$

and the equality rule

Σ – equality

$$\frac{\begin{array}{l} a \in A \\ b \in B(a) \\ C(v)\ set\ \ [v \in \Sigma(A, B)] \\ d(x, y) \in C(\langle x, y\rangle)\ \ [x \in A,\ y \in B(x)] \end{array}}{\mathsf{split}(\langle a, b\rangle, d) = d(a, b) \in C(\langle a, b\rangle)}$$

We can show that the elimination rule is correct by assuming the premises $c \in \Sigma(A, B)$ and $d(x, y) \in C(\langle x, y\rangle)\ \ [x \in A, y \in B(x)]$. The value of $\mathsf{split}(c, d)$ is computed by first computing c. By the meaning of the first premise, the value of c is $\langle a, b\rangle$ where $a \in A$ and $b \in B(a)$. The value of $\mathsf{split}(c, d)$ is then the value of $d(a, b)$ which, by the meaning of the second premise and the extensionality of C, is a canonical element in $C(c)$.

The equality rule is immediately justified from the way $\mathsf{split}(\langle a, b\rangle, d)$ is computed.

In order to use a notation which is more similar to the existential quantifier, we make the definition

$$(\Sigma x \in A)B(x)\ \equiv\ \Sigma(A, B)$$

We can now introduce the existential quantifier:

$$(\exists x \in A)B(x)\ \equiv\ (\Sigma x \in A)B(x)$$

By omitting some of the constructions in the rules for the Σ-set, we get the natural deduction rules for the existential quantifier:

$\exists$ – introduction

$$\frac{a \in A \qquad B(a)\ true}{(\exists x \in A)B(x)\ true}$$

$\exists$ – elimination

$$\frac{(\exists x \in A)B(x)\ true \qquad C\ prop \qquad C\ true\ \ [x \in A,\ B(x)\ true]}{C\ true}$$

Example. All elements in a Σ set are pairs

We will prove that the proposition

$$(\forall p \in \Sigma(A, B))(\exists a \in A)(\exists b \in B(a))\,(p =_{\Sigma(A,B)} \langle a, b\rangle)$$

is true for an arbitrary set A and an arbitrary family B of sets over A.

Assume that $p \in \Sigma(A, B)$. We will prove that the proposition

$$(\exists a \in A)(\exists b \in B(a))\,(p =_{\Sigma(A,B)} \langle a, b\rangle)$$

is true by Σ-elimination. So, we assume that $x \in A$ and $y \in B(x)$ and try to prove $(\exists a \in A)(\exists b \in B(a))\,(\langle x, y\rangle =_{\Sigma(A,B)} \langle a, b\rangle)$. But this is immediate from the facts that $x \in A$ and $y \in B(x)$, since then we get that $\langle x, y\rangle =_{\Sigma(A,B)} \langle x, y\rangle$ is true by Id-introduction. And then we can use $\exists$-introduction twice to conclude that $(\exists a \in A)(\exists b \in B(a))\,\langle x, y\rangle =_{\Sigma(A,B)} \langle a, b\rangle$. Finally, we get the desired result by an $\forall$-introduction.

Chapter 14

The set of small sets (The first universe)

14.1 Formal rules

The idea behind the set of small sets, i.e. the first universe, is to reflect the set structure on the object level. In programming we need it for many specifications when the most natural way of expressing a proposition is to use recursion or conditionals. We also need it in order to prove inequalities such as $0 \neq_{\mathsf{N}} \mathsf{succ}(0)$ (see later in this section). It is also necessary when defining abstract data types in type theory (see chapter 23).

We shall first introduce a set U of small sets, where U is a primitive constant of arity $\mathbf{0}$, which has constructors corresponding to the set forming operations $\{i_1, ..., i_n\}$, N, List, Id, $+$, Π, Σ, and W. The set forming operation W is used to represent well-orderings in type theory and is introduced in chapter 15. We start by introducing the following primitive constants: $\widehat{\{i_1, ..., i_n\}}$ and $\widehat{\mathsf{N}}$ of arity $\mathbf{0}$, $\widehat{\mathsf{List}}$ of arity $\mathbf{0}\twoheadrightarrow\mathbf{0}$, $\widehat{\mathsf{Id}}$ of arity $\mathbf{0}\otimes\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$, $\widehat{+}$ of arity $\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$ and $\widehat{\Pi}$, $\widehat{\Sigma}$ and $\widehat{\mathsf{W}}$ of arity $\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$.

A problem with the set U is that, because of the enumeration sets, the number of constructors is not fixed; this makes it impossible to formulate an induction principle for U. We will therefore, in section 14.2, change the set structure and the set of small sets in order to justify an elimination rule for the universe. One motivation for this is to introduce a selector urec, which is necessary for doing computations with the elements in the set of small sets.

The set of small sets is defined by giving its canonical elements and their equality relation. The idea is to let each canonical element represent (code) a set formed by using the set forming operations mentioned earlier. Simultaneously with the definition of the canonical elements, we will define a family of sets $\mathsf{Set}(x)$ *set* $[x \in \mathsf{U}]$ which decodes the elements in the universe to the set they represent. The canonical elements are given by the introduction rules.

U – formation

$$\mathsf{U}\ \mathit{set}$$

U – introduction 1

$$\widehat{\{i_1,...,i_n\}} \in \mathsf{U}$$

Set– introduction 1

$$\mathsf{Set}(\widehat{\{i_1,...,i_n\}}) = \{i_1,...,i_n\}$$

U – introduction 2

$$\widehat{\mathsf{N}} \in \mathsf{U}$$

Set – introduction 2

$$\mathsf{Set}(\widehat{\mathsf{N}}) = \mathsf{N}$$

U – introduction 3

$$\frac{A \in \mathsf{U}}{\widehat{\mathsf{List}}(A) \in \mathsf{U}}$$

Set – introduction 3

$$\frac{A \in \mathsf{U}}{\mathsf{Set}(\widehat{\mathsf{List}}(A)) = \mathsf{List}(\mathsf{Set}(A))}$$

U – introduction 4

$$\frac{A \in \mathsf{U} \qquad a \in \mathsf{Set}(A) \qquad b \in \mathsf{Set}(A)}{\widehat{\mathsf{Id}}(A,a,b) \in \mathsf{U}}$$

Set – introduction 4

$$\frac{A \in \mathsf{U} \qquad a \in \mathsf{Set}(A) \qquad b \in \mathsf{Set}(A)}{\mathsf{Set}(\widehat{\mathsf{Id}}(A,a,b)) = \mathsf{Id}(\mathsf{Set}(A),a,b)}$$

U – introduction 5

$$\frac{A \in \mathsf{U} \qquad B \in \mathsf{U}}{A \hat{+} B \in \mathsf{U}}$$

Set – introduction 5

$$\frac{A \in \mathsf{U} \qquad B \in \mathsf{U}}{\mathsf{Set}(A \hat{+} B) = \mathsf{Set}(A) + \mathsf{Set}(B)}$$

U – introduction 6

$$\frac{A \in \mathsf{U} \qquad B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)]}{\widehat{\Pi}(A,B) \in \mathsf{U}}$$

Set – introduction 6

$$\frac{A \in \mathsf{U} \qquad B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)]}{\mathsf{Set}(\widehat{\Pi}(A,B)) = \Pi(\mathsf{Set}(A),(x)\mathsf{Set}(B(x)))}$$

U – introduction 7

$$\frac{A \in \mathsf{U} \qquad B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)]}{\widehat{\Sigma}(A, B) \in \mathsf{U}}$$

Set – introduction 7

$$\frac{A \in \mathsf{U} \qquad B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)]}{\mathsf{Set}(\widehat{\Sigma}(A, B)) = \Sigma(\mathsf{Set}(A), (x)\mathsf{Set}(B(x)))}$$

U – introduction 8

$$\frac{A \in \mathsf{U} \qquad B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)]}{\widehat{\mathsf{W}}(A, B) \in \mathsf{U}}$$

Set – introduction 8

$$\frac{A \in \mathsf{U} \qquad B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)]}{\mathsf{Set}(\widehat{\mathsf{W}}(A, B)) = \mathsf{W}(\mathsf{Set}(A), (x)\mathsf{Set}(B(x)))}$$

The formation rules for the set of small sets are justified by the way the canonical elements and their equality relation were introduced. The formation rules are:

Set – formation 1

$$\frac{A \in \mathsf{U}}{\mathsf{Set}(A)\ \mathit{set}}$$

Set – formation 2

$$\frac{A = B \in \mathsf{U}}{\mathsf{Set}(A) = \mathsf{Set}(B)}$$

The premise $A \in \mathsf{U}$ means that the value of A is a canonical element in the set U, and since $\mathsf{Set}(x)$ is defined to be equal to a set whenever x is a canonical element in the set U, we may conclude that $\mathsf{Set}(x)$ is a set. And, similarly, $A = B \in \mathsf{U}$ means that A and B have equal canonical elements in the set U as values. The corresponding sets must therefore be equal, since the equality relation between the canonical elements in the set U exactly corresponds to the set equality relation.

We shall often use the same notation for the elements in the set U and the sets they represent. From the context, it is always possible to reconstruct the correct notation for the expressions. For example, instead of

$$\mathsf{Set}(\mathsf{natrec}(n, \widehat{\mathsf{Bool}}, (x, Y)\widehat{\mathsf{Bool}\rightarrow} Y))$$

we write

$$\mathsf{natrec}(n, \mathsf{Bool}, (x, Y)\mathsf{Bool} \rightarrow Y)$$

Example. Peano's fourth axiom

When we have introduced the universe set we are able to prove that the proposition

$$0 \neq_{\mathsf{N}} \mathsf{succ}(n)$$

is true for an arbitrary $n \in \mathsf{N}$. That is, if we express it in terms of sets, we can construct an element *peano4* in the set

$$\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(n)) \to \{\}$$

We will do this by assuming that the set $\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(n))$ is nonempty and show that we then can construct an element in the empty set. We will use substitutivity of propositional equality on a predicate over the natural numbers which is true only for the number zero.

We start by assuming $n \in \mathsf{N}$ and $x \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(n))$. By using N-elimination, we get

$$\mathsf{natrec}(m, \widehat{\mathsf{T}}, (y, z)\widehat{\{\}}) \in \mathsf{U}\ [m \in \mathsf{N}]$$

We make the definition

$$\mathit{Is\_zero}(m) \equiv \mathsf{Set}(\mathsf{natrec}(m, \widehat{\mathsf{T}}, (y, z)\widehat{\{\}}))$$

From N-equality and Set-formation we get the set equalities

$$\mathit{Is\_zero}(0) = \mathsf{Set}(\widehat{\mathsf{T}}) = \mathsf{T}$$

$$\mathit{Is\_zero}(\mathsf{succ}(n)) = \mathsf{Set}(\widehat{\{\}}) = \{\}$$

Using substitutivity of propositional equality we get that

$$\mathit{subst}(x, \mathsf{tt}) \in \mathit{Is\_zero}(\mathsf{succ}(n))$$

which by Set-equality yields

$$\mathit{subst}(x, \mathsf{tt}) \in \{\}$$

Finally, by $\to$-introduction, we discharge the second assumption and obtain

$$\lambda((x)\mathit{subst}(x, \mathsf{tt})) \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(n)) \to \{\}\ [n \in \mathsf{N}]$$

So we may put

$$\mathit{peano4} \ \equiv\ \lambda((x)\mathit{subst}(x, \mathsf{tt}))$$

and we have a proof of Peano's fourth axiom.

In [101] it is shown that Peano's fourth axiom cannot be derived in type theory without universes. The proof is based on interpreting set theory without a universe in a domain with only two elements. So, a truth valued function φ is defined on the sets and, intuitively, $\varphi(A) = \top$ means that the interpretation of the set A is a set with one element and $\varphi(A) = \bot$ means that A is interpreted as the empty set. φ is defined for each set expression $A(x_1, \ldots, x_n)$ by recursion on the length of the derivation of $A(x_1, \ldots, x_n)$ *set* $[x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})]$, using the clauses

$$\begin{array}{rcl}
\varphi(\{\}) & = & \bot \\
\varphi(\{i_1, \ldots, i_n\}) & = & \top \\
\varphi(\mathsf{N}) & = & \top \\
\varphi(\mathsf{Id}(A, a, b)) & = & \varphi(A) \\
\varphi(A + B) & = & \varphi(A) \vee \varphi(B) \\
\varphi((\Pi x \in A)B(x)) & = & \varphi(A) \to \varphi(B(x)) \\
\varphi((\Sigma x \in A)B(x)) & = & \varphi(A) \wedge \varphi(B(x)) \\
\varphi((\mathsf{W} x \in A)B(x)) & = & \varphi(A) \wedge (\neg\varphi(B(x))) \\
\varphi(\{x \in A \mid B(x)\}) & = & \varphi(A) \wedge \varphi(B(x))
\end{array}$$

Here $\wedge$, $\vee$, $\to$, and $\neg$ denote the usual boolean operations.

That φ really interprets set theory in the intended way is the content of the following theorem, which is proved in [101].

Theorem Let $a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)$ be derivable in set theory without universes under the assumptions $x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})$. Then $\varphi(A(x_1, \ldots, x_n)) = \top$ if $\varphi(A_1) = \cdots = \varphi(A_n(x_1, \ldots, x_{n-1})) = \top$.

By the interpretation we can now see that for no type A and terms a and b does there exist a closed term t such that

$$t \in \neg\mathsf{Id}(A, a, b) \qquad (*)$$

is derivable in type theory without universes. Assume that $(*)$ holds. Then there must exist a derivation of $\mathsf{Id}(A, a, b)$ *set* and, hence, also a derivation of $a \in A$. So, by the theorem, $\varphi(A) = \top$ which, together with the definitions of φ and $\neg$, gives

$$\varphi(\neg\mathsf{Id}(A, a, b)) = \varphi(\mathsf{Id}(A, a, b) \to \{\}) = \varphi(\mathsf{Id}(A, a, b)) \to \varphi(\{\}) = \\ \varphi(A) \to \bot = \bot$$

Hence, by the theorem, $\neg\mathsf{Id}(A, a, b)$ cannot be derived in type theory without universes.

Assume that Peano's fourth axiom can be derived, that is, that we, for some closed term s, have a derivation of

$$s \in (\Pi x \in \mathsf{N})\neg\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(x))$$

By Π-elimination we get $\mathsf{apply}(s, 0) \in \neg\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0))$ which is of the form $(*)$ and therefore impossible to derive in type theory without universes.

Example. The tautology function

A disadvantage with many type systems in programming languages is that some expressions, although perfectly reasonable, can not be assigned a type. The type systems are not well suited to express some properties needed for a safe evaluation of the expression. As an example, take the tautology function from the SASL manual [110]. It determines if a boolean expression of n variables (represented as a curried function of n arguments) is a tautology or not. The function is defined, using SASL-notation, as:

$$\begin{aligned} taut\, 0\; f &= f \\ taut\, n\; f &= taut(n-1)\,(f\ \mathsf{true})\ and\ taut(n-1)\,(f\ \mathsf{false}) \end{aligned}$$

Since SASL is untyped, the function is not assigned a type and for most other typed languages the definition causes a type error. Informally the type of *taut* is

$$(\Pi n \in \mathsf{N})((\mathsf{Bool} \to^n \mathsf{Bool}) \to \mathsf{Bool})$$

where $(\mathsf{Bool} \to^n \mathsf{Bool})$ is defined by the equations

$$\begin{aligned} \mathsf{Bool} \to^0 \mathsf{Bool} &= \mathsf{Bool} \\ \mathsf{Bool} \to^{k+1} \mathsf{Bool} &= \mathsf{Bool} \to (\mathsf{Bool} \to^k \mathsf{Bool}) \end{aligned}$$

So, for example,

$$\begin{aligned} taut\, 0 &\in \mathsf{Bool} \to \mathsf{Bool} \\ taut\, 3 &\in (\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \end{aligned}$$

and we can see that the type of the second argument depends on the value of the first.

The type of *taut* can be expressed using the set U in type theory. Make the following definitions:

$$\begin{aligned} and(x,y) &\equiv \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else\ false} \\ F(n) &\equiv \mathsf{natrec}(n, \mathsf{Bool}, (x,Y)\mathsf{Bool} \to Y) \\ taut(n) &\equiv \mathsf{natrec}(n, \\ &\qquad\qquad \lambda((f)f), \\ &\qquad\qquad (x,y)\lambda((f)and(y \cdot (f \cdot \mathsf{true}), \\ &\qquad\qquad\qquad\qquad y \cdot (f \cdot \mathsf{false})))) \end{aligned}$$

Notice that we have used the infix version of the constant apply,

$$x \cdot y \equiv \mathsf{apply}(x,y)$$

From these definitions, it immediately follows that

$$and(x,y) \in \mathsf{Bool}\ [x \in \mathsf{Bool}, y \in \mathsf{Bool}] \tag{14.1}$$
$$F(0) = \mathsf{Bool} \in \mathsf{U} \tag{14.2}$$
$$F(\mathsf{succ}(x)) = \mathsf{Bool} \to F(x) \in \mathsf{U}\ [x \in \mathsf{N}] \tag{14.3}$$

Using Set-formation on (14.2) and (14.3), we get the set equalities

$$F(0) = \mathsf{Bool} \tag{14.4}$$
$$F(\mathsf{succ}(x)) = \mathsf{Bool} \to F(x)\ [x \in \mathsf{N}] \tag{14.5}$$

The goal is to prove:

$$\lambda((n)taut(n)) \in (\Pi n \in \mathsf{N})(F(n) \to \mathsf{Bool})$$

so we start by assuming that

$$n \in \mathsf{N}$$

and then prove $taut(n) \in F(n) \to \mathsf{Bool}$ by induction on n. We first have the base case. It is easy to see that

$$\lambda((f)f) \in \mathsf{Bool} \to \mathsf{Bool}$$

and, since we from (14.4) and $\to$-formation get the set equality

$$F(0) \to \mathsf{Bool} = \mathsf{Bool} \to \mathsf{Bool}$$

we can conclude that

$$\lambda((f)f) \in F(0) \to \mathsf{Bool} \tag{14.6}$$

For the induction step, we make the assumptions

$$\begin{aligned} &x \in \mathsf{N} \\ &y \in F(x) \to \mathsf{Bool} \end{aligned}$$

The goal is to prove

$$\lambda((f)and(y \cdot (f \cdot \mathsf{true}), y \cdot (f \cdot \mathsf{false}))) \in F(\mathsf{succ}(x)) \to \mathsf{Bool}$$

We therefore make the assumption

$$f \in F(\mathsf{succ}(x)) \tag{14.7}$$

From (14.7) and the set equality (14.4), we get

$$f \in \mathsf{Bool} \to F(x)$$

and then by $\to$-elimination

$$\begin{aligned} &f \cdot \mathsf{true} \in F(x) \\ &f \cdot \mathsf{false} \in F(x) \end{aligned}$$

and furthermore by using the induction hypothesis

$$\begin{aligned} &y \cdot (f \cdot \mathsf{true}) \in \mathsf{Bool} \\ &y \cdot (f \cdot \mathsf{false}) \in \mathsf{Bool} \end{aligned}$$

By substituting these elements into (14.1), we obtain

$$and(y \cdot (f \cdot \mathsf{true}), y \cdot (f \cdot \mathsf{false})) \in \mathsf{Bool}$$

By $\to$-introduction, we discharge assumption (14.7) and get

$$\lambda((f)and(y \cdot (f \cdot \mathsf{true}), y \cdot (f \cdot \mathsf{false}))) \in F(\mathsf{succ}(x)) \to \mathsf{Bool} \tag{14.8}$$

We can now use N-elimination on (14.6) and (14.8) to obtain

$$taut(n) \in F(n) \to \mathsf{Bool}$$

and finally, by Π-introduction, we get the desired result

$$\lambda((n)taut(n)) \in (\Pi n \in \mathsf{N})(F(n) \to \mathsf{Bool})$$

Example. An expression without normal form in the theory with extensional equality

A canonical element in the set $(\Pi x \in A)B(x)$ is of the form $\lambda(b)$ where $b(x) \in B(x)\ [x \in A]$ and the expression $b(x)$ is not further evaluated. We have already remarked that evaluating $b(x)$ would be the same as trying to execute a program which expects an input without giving any input. Using the extensional equality Eq and the universe in a crucial way, we will now give an example of a lambda-expression $\lambda(b)$ in the set $\{\} \to A$, where, by regarding the evaluation rules as reduction rules, $b(x)$ does not even terminate.

By the use of the set of small sets, we will show that

$$\mathsf{Set}(A) = \mathsf{Set}(B)\ [A \in \mathsf{U},\ B \in \mathsf{U},\ x \in \{\}] \tag{14.1}$$

Assume

$$x \in \{\}$$

Since $\mathsf{Eq}(\mathsf{U}, A, B)$ is a set, we get by $\{\}$-elimination that

$$\mathsf{case}_0(x) \in \mathsf{Eq}(\mathsf{U}, A, B)\ [A \in \mathsf{U},\ B \in \mathsf{U},\ x \in \{\}]$$

and by strong Eq-elimination it follows that

$$A = B \in \mathsf{U}\ [A \in \mathsf{U},\ B \in \mathsf{U},\ x \in \{\}] \tag{14.2}$$

Set-formation 2 and (2) gives

$$\mathsf{Set}(A) = \mathsf{Set}(B)\ [A \in \mathsf{U},\ B \in \mathsf{U},\ x \in \{\}]$$

and, hence, we have a derivation of (1).

Now assume

$$x \in \{\} \tag{14.3}$$

By choosing A to be $\widehat{\mathsf{N}}$ and B to be $\widehat{\mathsf{N}} \widehat{\to} \widehat{\mathsf{N}}$, we get from (1)

$$\mathsf{N} = \mathsf{N} \to \mathsf{N} \tag{14.4}$$

Assume

$$y \in \mathsf{N} \tag{14.5}$$

One of the rules for set equality applied on (4) and (5) gives

$$y \in \mathsf{N} \to \mathsf{N} \tag{14.6}$$

From (5) and (6) we get, by $\to$-elimination,

$$\mathsf{apply}(y, y) \in \mathsf{N} \tag{14.7}$$

and from (7) we get, by $\to$-introduction,

$$\lambda y.\mathsf{apply}(y, y) \in \mathsf{N} \to \mathsf{N} \tag{14.8}$$

thereby discharging the assumption (5). (6) and (8) give

$$\lambda y.\mathsf{apply}(y, y) \in \mathsf{N} \tag{14.9}$$

We can now apply $\rightarrow$-elimination on (8) and (9) to get

$$\mathsf{apply}(\lambda y.\mathsf{apply}(y, y), \lambda y.\mathsf{apply}(y, y)) \in \mathsf{N}$$

and $\rightarrow$-introduction finally gives

$$\lambda x.\mathsf{apply}(\lambda y.\mathsf{apply}(y, y), \lambda y.\mathsf{apply}(y, y)) \in \{\} \rightarrow \mathsf{N}$$

thereby discharging the assumption (3). The expression

$$\mathsf{apply}(\lambda y.\mathsf{apply}(y, y), \lambda y.\mathsf{apply}(y, y))$$

is the well-known example from combinatory logic of an expression which reduces to itself. Since this expression is not on canonical form, we have an example of a lambda-expression which is an element of a Π-set and whose body does not terminate. Notice that there is no violation of the arity rules when forming $\mathsf{apply}(y, y)$ because apply is of arity $\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}$ and y is a variable of arity $\mathbf{0}$.

14.2 Elimination rule

With a set of small sets that reflects a set structure with infinitely many set forming operations, it is impossible to justify a structural induction rule on the set. In order to be able to introduce such an induction rule, the small enumeration sets, i.e the sets $\{i_1, ..., i_n\}$, must be generated from finitely many basic enumeration sets. We shall therefore modify the system of set forming operations, and consequently also the set of small sets, to make room for an induction rule on the elements of the universe. The modified system will only contain two basic enumeration sets, the empty set and a set with one element (see the section on enumeration sets); the other enumeration sets are generated from these two sets by means of the disjoint union. With a set structure with only these two enumeration sets, we get a set of small sets where the first U-introduction rule is replaced by the rules:

U – introduction 1a

$$\widehat{\emptyset} \in \mathsf{U}$$
$$\mathsf{Set}(\widehat{\emptyset}) = \emptyset$$

and

U – introduction 1b

$$\widehat{\mathsf{T}} \in \mathsf{U}$$
$$\mathsf{Set}(\widehat{\mathsf{T}}) = \mathsf{T}$$

An enumeration set with more than one element is formed by repeated use of the T set and the disjoint union. We introduce the function constant N' of arity $\mathbf{0}$ by the definition:

$$\mathsf{N}'(x) \quad \equiv \quad \mathsf{natrec}(x, \widehat{\emptyset}, (u, v)\mathsf{S}'(v))$$

where

$$\mathsf{S}'(x) \quad \equiv \quad \widehat{\mathsf{T}}\widehat{+}x$$

So

$$\mathsf{Set}(\mathsf{S}'(A)) \ \mathit{set} \ [A \in \mathsf{U}] \tag{14.10}$$

and N' applied to a natural number n gives an element in U, which corresponds to an enumeration set with n elements. We can now prove that

$$\mathsf{N}'(x) \in \mathsf{U} \ [x \in \mathsf{N}] \tag{14.11}$$

$$\mathsf{N}'(\mathsf{succ}(x)) = \mathsf{S}'(\mathsf{N}'(x)) \in \mathsf{U} \ [x \in \mathsf{N}] \tag{14.12}$$

From 14.11, we get, by Set-formation,

$$\mathsf{Set}(\mathsf{N}'(x)) \ \mathit{set} \ [x \in \mathsf{N}]$$

Moreover, simplification gives us:

$$\begin{aligned} \mathsf{Set}(\mathsf{N}'(0)) &= \emptyset \\ \mathsf{Set}(\mathsf{N}'(1)) &= \mathsf{T} + \emptyset \end{aligned}$$

with the element $\mathsf{inl}(\mathsf{tt})$, and

$$\mathsf{Set}(\mathsf{N}'(2)) \quad = \quad \mathsf{T}+(\mathsf{T}+\emptyset)$$

with elements $\mathsf{inl}(\mathsf{tt})$ and $\mathsf{inr}(\mathsf{inl}(\mathsf{tt}))$, and so on. If the enumeration sets defined here are compared with the enumeration sets N_k in [69] then $\mathsf{Set}(\mathsf{N}'(k))$ corresponds to N_k, $\mathsf{inl}(\mathsf{tt})$ corresponds to 0_k and $\mathsf{inr}(\mathsf{inr}(\ldots\ \mathsf{inr}(\mathsf{inl}(\mathsf{tt}))\ldots))$ corresponds to n_k, with n being the number of 'inr'-applications.

By making the definitions:

$$\begin{aligned} \mathsf{o}' &\equiv \mathsf{inl}(\mathsf{tt}) \\ \mathsf{s}'(x) &\equiv \mathsf{inr}(x) \\ \mathsf{scase}'(x,y,z) &\equiv \mathsf{when}(x,(w)y,z) \end{aligned}$$

where o', s' and scase' are constants of arity $\mathbf{0}$, $\mathbf{0}{\twoheadrightarrow}\mathbf{0}$ and $\mathbf{0}{\otimes}\mathbf{0}{\otimes}(\mathbf{0}{\twoheadrightarrow}\mathbf{0}){\twoheadrightarrow}\mathbf{0}$ respectively, we can prove the judgements

$$\mathsf{o}' \in \mathsf{Set}(\mathsf{S}'(A)) \ [A \in \mathsf{U}] \tag{14.13}$$

$$\mathsf{s}'(x) \in \mathsf{Set}(\mathsf{S}'(A)) \ [A \in \mathsf{U},\ x \in \mathsf{Set}(A)] \tag{14.14}$$

$$\begin{array}{l} \mathsf{scase}'(x,y,z) \in \mathsf{Set}(C(x)) \\ {[A \in \mathsf{U},\ x \in \mathsf{Set}(\mathsf{S}'(A)),\ C(u) \in \mathsf{U}\ [u \in \mathsf{Set}(\mathsf{S}'(A))],} \\ \ y \in \mathsf{Set}(C(\mathsf{o}')),\ z(v) \in C(\mathsf{s}'(v))\ [v \in \mathsf{Set}(A)]] \end{array} \tag{14.15}$$

$$\begin{array}{l} \mathsf{scase}'(\mathsf{o}',y,z) = y \in \mathsf{Set}(C(\mathsf{o}')) \\ {[A \in \mathsf{U},\ C(u) \in \mathsf{U}\ [u \in \mathsf{Set}(\mathsf{S}'(A))],} \\ \ y \in \mathsf{Set}(C(\mathsf{o}')),\ z(v) \in C(\mathsf{s}'(v))\ [v \in \mathsf{Set}(A)]] \end{array} \tag{14.16}$$

$$\begin{array}{l} \mathsf{scase}'(\mathsf{s}'(x),y,z) = z(x) \in \mathsf{Set}(C(\mathsf{s}'(x))) \\ {[A \in \mathsf{U},\ x \in \mathsf{Set}(A),\ C(u) \in \mathsf{U}\ [u \in \mathsf{Set}(\mathsf{S}'(A))],} \\ \ y \in \mathsf{Set}(C(\mathsf{o}')),\ z(v) \in C(\mathsf{s}'(v))\ [v \in \mathsf{Set}(A)]] \end{array} \tag{14.17}$$

Per Martin-Löf has given a more direct formulation of the enumeration sets by introducing the set former S as a primitive constant with the following rules (compare with the theorems (14.10), (14.13), (14.14), (14.15), (14.16) and (14.17) above):

S– formation

$$\frac{A\ set}{\mathsf{S}(A)\ set}$$

S– introduction

$$\mathsf{o} \in \mathsf{S}(A) \qquad \frac{a \in A}{\mathsf{s}(a) \in \mathsf{S}(A)}$$

S– elimination

$$\frac{a \in \mathsf{S}(A) \qquad b \in C(\mathsf{o}) \qquad c(x) \in C(\mathsf{s}(x))\ [x \in A]}{\mathsf{scase}(a, b, c) \in C(a)}$$

S– equality

$$\frac{b \in C(\mathsf{o}) \qquad c(x) \in C(\mathsf{s}(x))\ [x \in A]}{\mathsf{scase}(\mathsf{o}, b, c) = b \in C(\mathsf{o})}$$

$$\frac{a \in \mathsf{S}(A) \qquad b \in C(\mathsf{o}) \qquad c(x) \in C(\mathsf{s}(x))\ [x \in A]}{\mathsf{scase}(\mathsf{s}(a), b, c) = c(a) \in C(\mathsf{s}(a))}$$

Given the reformulated set of small sets, we can now justify a structural induction rule, which is introduced as follows. First we introduce urec as a constant of arity

$$\begin{array}{l} \mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0} \otimes \\ \qquad (\mathbf{0} \otimes \mathbf{0} \twoheadrightarrow \mathbf{0}) \otimes \\ \qquad (\mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0} \twoheadrightarrow \mathbf{0})) \otimes \\ \qquad (\mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0} \twoheadrightarrow \mathbf{0})) \otimes \\ \qquad (\mathbf{0} \otimes (\mathbf{0} \twoheadrightarrow \mathbf{0}) \otimes \mathbf{0} \otimes (\mathbf{0} \twoheadrightarrow \mathbf{0}) \twoheadrightarrow \mathbf{0}) \otimes \\ \qquad (\mathbf{0} \otimes (\mathbf{0} \twoheadrightarrow \mathbf{0}) \otimes \mathbf{0} \otimes (\mathbf{0} \twoheadrightarrow \mathbf{0}) \twoheadrightarrow \mathbf{0}) \otimes \\ \qquad (\mathbf{0} \otimes (\mathbf{0} \twoheadrightarrow \mathbf{0}) \otimes \mathbf{0} \otimes (\mathbf{0} \twoheadrightarrow \mathbf{0}) \twoheadrightarrow \mathbf{0}) \\ \twoheadrightarrow \mathbf{0} \end{array}$$

and we then define how $\mathsf{urec}(A, a_1, \ldots, a_9)$ is computed by the following rules ($a \Rightarrow b$ means that b is the value of a).

$$\frac{a \Rightarrow \widehat{\emptyset} \qquad a_1 \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9) \Rightarrow b}$$

$$\frac{a \Rightarrow \widehat{\mathsf{T}} \qquad a_2 \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, ..., a_9) \Rightarrow b}$$

$$\frac{a \Rightarrow \widehat{\mathsf{N}} \qquad a_3 \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, ..., a_9) \Rightarrow b}$$

$$\frac{a \Rightarrow \widehat{\mathsf{List}}(A) \qquad a_4(A, \mathsf{urec}(A, a_1, a_2, ..., a_9)) \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, ..., a_9) \Rightarrow b}$$

$$\frac{a \Rightarrow \widehat{\mathsf{Id}}(A, c, d) \qquad a_5(A, c, d, \mathsf{urec}(A, a_1, ..., a_9)) \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, ..., a_9) \Rightarrow b}$$

$$\frac{a \Rightarrow A \widehat{+} B \qquad a_6(A, B, \mathsf{urec}(A, a_1, ..., a_9), \mathsf{urec}(B, a_1, ..., a_9)) \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, ..., a_9) \Rightarrow b}$$

$$\frac{a \Rightarrow \widehat{\Pi}(A, B) \qquad a_7(A, B, \mathsf{urec}(A, a_1, ..., a_9), (w)\mathsf{urec}(B(w), a_1, ..., a_9)) \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, ..., a_9) \Rightarrow b}$$

$$\frac{a \Rightarrow \widehat{\Sigma}(A, B) \qquad a_8(A, B, \mathsf{urec}(A, a_1, ..., a_9), (w)\mathsf{urec}(B(w), a_1, ..., a_9)) \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, ..., a_9) \Rightarrow b}$$

$$\frac{a \Rightarrow \widehat{\mathsf{W}}(A, B) \qquad a_9(A, B, \mathsf{urec}(A, a_1, ..., a_9), (w)\mathsf{urec}(B(w), a_1, ..., a_9)) \Rightarrow b}{\mathsf{urec}(a, a_1, a_2, ..., a_9) \Rightarrow b}$$

A restriction in these rules is that w must not occur free in B, a_1, ... , a_8 or a_9. It would otherwise be bound in $(w)\mathsf{urec}(B(w), a_1, ..., a_9)$.

The computation rule for urec justifies the following elimination rule for the set of small sets:

U- elimination

$$\frac{\begin{array}{l} a \in \mathsf{U} \\ C(v)\ set\ [v \in \mathsf{U}] \\ a_1 \in C(\widehat{\emptyset}) \\ a_2 \in C(\widehat{\mathsf{T}}) \\ a_3 \in C(\widehat{\mathsf{N}}) \\ a_4(x, y) \in C(\widehat{\mathsf{List}}(x))\ [x \in \mathsf{U},\ y \in C(x)] \\ a_5(x, y, z, u) \in C(\widehat{\mathsf{Id}}(x, y, z))\ [x \in \mathsf{U},\ y \in \mathsf{Set}(x),\ z \in \mathsf{Set}(x),\ u \in C(x)] \\ a_6(x, y, z, u) \in C(x \widehat{+} y)\ [x \in \mathsf{U},\ y \in \mathsf{U},\ z \in C(x),\ u \in C(y)] \\ a_7(x, y, z, u) \in C(\widehat{\Pi}(x, y))\ [x \in \mathsf{U},\ y(v) \in \mathsf{U}[v \in \mathsf{Set}(x)], \\ \qquad\qquad\qquad\qquad\qquad z \in C(x),\ u(v) \in C(y(v))[v \in \mathsf{Set}(x)]] \\ a_8(x, y, z, u) \in C(\widehat{\Sigma}(x, y))\ [x \in \mathsf{U},\ y(v) \in \mathsf{U}[v \in \mathsf{Set}(x)], \\ \qquad\qquad\qquad\qquad\qquad z \in C(x),\ u(v) \in C(y(v))[v \in \mathsf{Set}(x)]] \\ a_9(x, y, z, u) \in C(\widehat{\mathsf{W}}(x, y))\ [x \in \mathsf{U},\ y(v) \in \mathsf{U}[v \in \mathsf{Set}(x)], \\ \qquad\qquad\qquad\qquad\qquad z \in C(x),\ u(v) \in C(y(v))[v \in \mathsf{Set}(x)]] \end{array}}{\mathsf{urec}(a, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9) \in C(a)}$$

Here x, y, z and u must not occur free in the abstraction C. In the following rules we will not write down the premise $C(v)\ set\ [v \in \mathsf{U}]$.

We also have an elimination rule where the premises and conclusion are of the form $a = b \in A$. Furthermore, the computation rule for urec justifies the following equality rules. The last 9 premises of all the equality rules are the same as the last 9 premises of the elimination rule above.

U- equality 1

$$\frac{a_1 \in C(\widehat{\emptyset}) \quad a_2 \in C(\widehat{\mathsf{T}}) \quad \ldots \quad a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots]}{\mathsf{urec}(\widehat{\emptyset}, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9) = a_1 \in C(\widehat{\emptyset})}$$

U- equality 2

$$\frac{a_1 \in C(\widehat{\emptyset}) \quad a_2 \in C(\widehat{\mathsf{T}}) \quad \ldots \quad a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots]}{\mathsf{urec}(\widehat{\mathsf{T}}, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9) = a_2 \in C(\widehat{\mathsf{T}})}$$

U- equality 3

$$\frac{a_1 \in C(\widehat{\emptyset}) \quad a_2 \in C(\widehat{\mathsf{T}}) \quad \ldots \quad a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots]}{\mathsf{urec}(\widehat{\mathsf{N}}, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9) = a_3 \in C(\widehat{\mathsf{N}})}$$

U- equality 4

$$\frac{A \in \mathsf{U} \quad a_1 \in C(\widehat{\emptyset}) \quad \ldots \quad a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots]}{\mathsf{urec}(\widehat{\mathsf{List}}(A), a_1, \ldots, a_9) = a_4(A, \mathsf{urec}(A, a_1, \ldots, a_9)) \in C(\widehat{\mathsf{List}}(A))}$$

U- equality 5

$$\frac{\begin{array}{ll} A \in \mathsf{U} & \\ c \in \mathsf{Set}(A) & \\ d \in \mathsf{Set}(A) & a_1 \in C(\widehat{\emptyset}) \\ \quad\vdots & \\ a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots] & \end{array}}{\begin{array}{l} \mathsf{urec}(\widehat{\mathsf{Id}}(A,c,d), a_1, \ldots, a_9) = \\ a_5(A, c, d, \mathsf{urec}(A, a_1, \ldots, a_9)) \in C(\widehat{\mathsf{Id}}(A,c,d)) \end{array}}$$

U- equality 6

$$\frac{\begin{array}{l} A \in \mathsf{U} \\ B \in \mathsf{U} \\ a_1 \in C(\widehat{\emptyset}) \\ \quad\vdots \\ a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots] \end{array}}{\begin{array}{l} \mathsf{urec}(A \mathbin{\widehat{+}} B, a_1, \ldots, a_9) = \\ a_6(A, B, \mathsf{urec}(A, a_1 \ldots, a_9), \mathsf{urec}(B, a_1, \ldots, a_9)) \in C(A \mathbin{\widehat{+}} B) \end{array}}$$

U- equality 7

$$\frac{\begin{array}{l} A \in \mathsf{U} \\ B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)] \\ a_1 \in C(\widehat{\emptyset}) \\ \quad\vdots \\ a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots] \end{array}}{\begin{array}{l} \mathsf{urec}(\widehat{\Pi}(A,B), a_1, \ldots, a_9) = \\ a_7(A, B, \mathsf{urec}(A, a_1, \ldots, a_9), (w)\mathsf{urec}(B(w), a_1, \ldots, a_9)) \in C(\widehat{\Pi}(A,B)) \end{array}}$$

U- equality 8

$$\frac{\begin{array}{l} A \in \mathsf{U} \\ B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)] \\ a_1 \in C(\widehat{\emptyset}) \\ \quad\vdots \\ a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots] \end{array}}{\begin{array}{l} \mathsf{urec}(\widehat{\Sigma}(A,B), a_1, \ldots, a_9) = \\ a_8(A, B, \mathsf{urec}(A, a_1, \ldots, a_9), (w)\mathsf{urec}(B(w), a_1, \ldots, a_9)) \in C(\widehat{\Sigma}(A,B)) \end{array}}$$

U- equality 9

$$\frac{\begin{array}{l} A \in \mathsf{U} \\ B(x) \in \mathsf{U}\ [x \in \mathsf{Set}(A)] \\ a_1 \in C(\widehat{\emptyset}) \\ \quad\vdots \\ a_9(x,y,z,u) \in C(\widehat{\mathsf{W}}(x,y))\ [\ldots] \end{array}}{\begin{array}{l} \mathsf{urec}(\widehat{\mathsf{W}}(A,B), a_1, \ldots, a_9) = \\ a_9(A, B, \mathsf{urec}(A, a_1, \ldots, a_9), (w)\mathsf{urec}(B(w), a_1, \ldots, a_9)) \in C(\widehat{\mathsf{W}}(A,B)) \end{array}}$$

The variables x, y, z and u must not occur free in C, and there must be no free occurrences of w in B, a_1, $\ldots$ or a_9.

Chapter 15

Well-orderings

In order to introduce the well-ordering set constructor (or well-founded tree set constructor) we introduce the primitive constants

$$\begin{array}{rll} \mathsf{W} & \text{of arity} & (\mathbf{0}{\otimes}(\mathbf{0}{\twoheadrightarrow}\mathbf{0})){\twoheadrightarrow}\mathbf{0} \\ \mathsf{sup} & \text{of arity} & (\mathbf{0}{\otimes}(\mathbf{0}{\twoheadrightarrow}\mathbf{0})){\twoheadrightarrow}\mathbf{0} \\ \mathsf{wrec} & \text{of arity} & \mathbf{0}{\otimes}(\mathbf{0}{\otimes}(\mathbf{0}{\twoheadrightarrow}\mathbf{0}){\otimes}(\mathbf{0}{\twoheadrightarrow}\mathbf{0}){\twoheadrightarrow}\mathbf{0}){\twoheadrightarrow}\mathbf{0} \end{array}$$

With the well-order set constructor we can construct many different sets of trees and to characterize a particular set we must provide information about two things:

- the different ways the trees may be formed, and
- for each way to form a tree which parts it consists of.

To provide this information, the well-order set constructor W has two arguments:

1. The *constructor set* A.
2. The *selector family* B.

Given a constructor set A and selector family B on A, we can form a well-order $\mathsf{W}(A, B)$ (two other notations are $(\mathsf{W}x \in A)B(x)$ and $\mathsf{W}_{x\in A}B(x)$). The formation rule therefore has the following form:

W - formation

$$\frac{A \ set \qquad B(x) \ set \ \ [x \in A]}{\mathsf{W}(A, B) \ set}$$

The elements in the set A represents the different ways to form an element in $\mathsf{W}(A, B)$ and $B(x)$ represents the parts of a tree formed by x.

The elements of a well-order $\mathsf{W}(A, B)$ can, as we already mentioned, be seen as well-founded trees and to form a particular element of $\mathsf{W}(A, B)$ we must say which way the tree is formed and what the parts are. If we have an element a in the set A, that is, if we have a particular form we want the tree to have, and if we have a function from $B(a)$ to $\mathsf{W}(A, B)$, that is if we have a collection of subtrees, we may form the tree $\mathsf{sup}(a, b)$. We visualize this element in figure 15.1. The introduction rule has the form

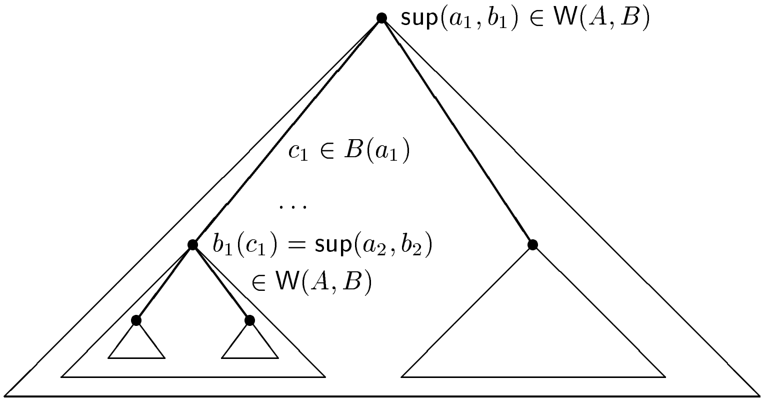

Figure 15.1: An element of a well-order

W - introduction

$$\frac{a \in A \qquad b(x) \in \mathsf{W}(A, B) \;\; [x \in B(a)]}{\mathsf{sup}(a, b) \in \mathsf{W}(A, B)}$$

It may seem strange that we do not have a particular introduction rule for the leaves, but we get the same effect if we choose $B(x)$ to be the empty set for some $x \in A$. In the introduction rule we can see that we must provide a function from $B(x)$ to $\mathsf{W}(A, B)$ in order to form an element $\mathsf{sup}(a, b)$. In the case when $B(x)$ is the empty set, we use a small "trick" to provide such a function. From the assumption $x \in \{\}$, we can, by using the $\{\}$-elimination rule, conclude that $\mathsf{case}_{\{\}}(x)$ is an element of an arbitrary set, and in this case we of course choose $\mathsf{W}(A, B)$. So if $B(a)$ is empty, then $(x)\mathsf{case}_{\{\}}(x) \equiv \mathsf{case}_{\{\}}$ is a function from $B(a)$ to $\mathsf{W}(A, B)$ and $\mathsf{sup}(a, \mathsf{case}_{\{\}})$ is an element of $\mathsf{W}(A, B)$.

Let us take a simple example. We want to construct a well-order set to represent simple binary trees which, for example, could be defined in ML [72] by

$$\textbf{datatype}\ \mathit{BinTree} = \mathit{leaf} \ \mid\ \mathit{node}\ \textbf{of}\ \mathit{BinTree} * \mathit{BinTree}$$

There are two different ways of constructing a binary tree, one to construct a leaf and one to construct a compound tree. The constructor set A must therefore contain two elements, and we can for example use the enumeration set $\{\mathit{leaf}, \mathit{node}\}$. A leaf does not have any parts, so $B(\mathit{leaf})$ must be the empty set, and a compound tree has two parts, so we can choose $B(\mathit{node})$ as the set $\{\mathit{left}, \mathit{right}\}$. Putting this together, we get a well-order set

$$\mathit{BinTree} \ \equiv\ \mathsf{W}(\{\mathit{leaf}, \mathit{node}\}, (x)\mathsf{Set}(\mathsf{case}_{\{leaf,node\}}(x, \widehat{\{\}}, \widehat{\{\mathit{left}, \mathit{right}\}})))$$

which has representations of all binary trees as elements. Notice that we must use the universe set to construct the family B. The elements of this well-order are always of one of the forms

$$\mathsf{sup}(\mathit{leaf}, \mathsf{case}_{\{\}}) \qquad\qquad \mathsf{sup}(\mathit{node}, (x)\mathsf{case}_{\{left,right\}}(x, t', t''))$$

where t' and t'' are two elements in $\mathsf{W}(A,B)$. By introducing definitions

$$\begin{aligned} \mathit{leaf}' &\equiv \mathsf{sup}(\mathit{leaf}, \mathsf{case}) \\ \mathit{node}'(t',t'') &\equiv \mathsf{sup}(\mathit{node}, (x)\mathsf{case}(x,t',t'')) \end{aligned}$$

we get expressions for the elements that look just like the corresponding ML expressions.

The non-canonical constant in a well-ordering is wrec and the expression $\mathsf{wrec}(a,b)$ is computed as follows:

1. Compute the value of a.
2. If the value is $\mathsf{sup}(d,e)$, then the value of $\mathsf{wrec}(a,b)$ is the value of $b(d,e,(x)\mathsf{wrec}(e(x),b))$.

The computation rule for wrec justifies the following elimination rule:

W – elimination

$$\frac{\begin{array}{l} a \in \mathsf{W}(A,B) \\ C(v)\ \mathit{set}\ \ [v \in \mathsf{W}(A,B)] \\ b(y,z,u) \in C(\mathsf{sup}(y,z)) \\ \qquad [y \in A, z(x) \in \mathsf{W}(A,B)\ \ [x \in B(y)],\ u(x) \in C(z(x))\ \ [x \in B(y)]] \end{array}}{\mathsf{wrec}(a,b) \in C(a)}$$

and the following equality rule

W - equality

$$\frac{\begin{array}{l} d \in A \\ e(x) \in \mathsf{W}(A,B)\ \ [x \in B(d)] \\ C(v)\ \mathit{set}\ \ [v \in \mathsf{W}(A,B)] \\ b(y,z,u) \in C(\mathsf{sup}(y,z)) \\ \qquad [y \in A, z(x) \in \mathsf{W}(A,B)\ \ [x \in B(y)],\ u(x) \in C(z(x))\ \ [x \in B(y)]] \end{array}}{\mathsf{wrec}(\mathsf{sup}(d,e),b) = b(d,e,(x)\mathsf{wrec}(e(x),b)) \in C(\mathsf{sup}(d,e))}$$

As an example of how the non-canonical constant can be used, we define the function that counts the number of nodes in a binary tree and which in ML could be defined by:

$$\begin{array}{lll} \textbf{fun} & \mathit{nrofnodes}(\mathit{leaf}) & = 1 \\ \mid & \mathit{nrofnodes}(\mathit{node}(t',t'')) & = \mathit{nrofnodes}(t') + \mathit{nrofnodes}(t'') \end{array}$$

In type theory this function could be defined by

$$\mathit{nrofnodes}(x) \equiv \mathsf{wrec}(x, (y,z,u)\mathsf{case}(y, 1, u(\mathit{left}) + u(\mathit{right})))$$

Using the elimination rule, we immediately see that

$$\mathit{nrofnodes}(x) \in \mathsf{N}\ \ [x \in \mathit{BinTree}]$$

and using the equality rule, we immediately get the equalities that correspond to the ML definition.

In the same way as we above introduced defined constants to get a nicer syntax for the elements of the type *BinTree*, we can make a definition and get a constant that behaves just like a recursion operator on binary trees.

$$\mathit{trec}'(t,a,b) \equiv \mathsf{wrec}(t, (x,y,z)\mathsf{case}(x, a, b(y(\mathit{left}), y(\mathit{right}), z(\mathit{left}), z(\mathit{right}))))$$

The equality rule for wrec corresponds to the equalities:

$$\begin{array}{rcl} \mathit{trec}'(\mathit{leaf}', a, b) & = & a \\ \mathit{trec}'(\mathit{node}'(t', t''), a, b) & = & b(t', t'', \mathit{trec}'(t', a, b), \mathit{trec}'(t'', a, b)) \end{array}$$

And the function counting the number of nodes, which we defined above, can then be defined as

$$\mathit{nrofnodes} \quad \equiv \quad \mathit{trec}'(x, 1, (t', t'', z', z'')\, z' \oplus z'')$$

Example. Defining the natural numbers as a well-ordering

It is not difficult to see that the set of natural numbers can be defined by the following abbreviations:

$$\begin{array}{rcl} \mathsf{N} & \equiv & (\mathsf{W}x \in \{\mathit{zero}, \mathit{succ}\})\, \mathsf{Set}(\mathsf{case}(x, \widehat{\{\}}, \widehat{\mathsf{T}})) \\ 0 & \equiv & \mathsf{sup}(\mathit{zero}, \mathsf{case}) \\ \mathsf{succ}(a) & \equiv & \mathsf{sup}(\mathit{succ}, (x)a) \\ \mathsf{natrec}(a, b, c) & \equiv & \mathsf{wrec}(a, (y, z, u)\mathsf{case}(y, b, c(z(\mathsf{tt}), u(\mathsf{tt})))) \end{array}$$

The idea is to let the n:th natural number be represented by a thin tree of height n. We immediately see from the W-formation rule that

$$(\mathsf{W}x \in \{\mathit{zero}, \mathit{succ}\})\, \mathsf{Set}(\mathsf{case}(x, \widehat{\{\}}, \widehat{\mathsf{T}}))\ \mathit{set}$$

and therefore, using the definition of N, that the formation rule for the natural numbers can be proved. We can also see, by using the W-introduction rule, that

$$\mathsf{sup}(\mathit{zero}, \mathsf{case}) \in (\mathsf{W}x \in \{\mathit{zero}, \mathit{succ}\})\mathsf{Set}(\mathsf{case}(x, \widehat{\{\}}, \widehat{\mathsf{T}}))$$

and hence, using the abbreviations, that the first N-introduction rule, $0 \in \mathsf{N}$, holds. The second introduction rule, $\mathsf{succ}(x) \in \mathsf{N}\ \ [x \in \mathsf{N}]$, corresponds to the judgement

$$\begin{array}{l} \mathsf{sup}(\mathit{succ}, (y)x) \in (\mathsf{W}x \in \{\mathit{zero}, \mathit{succ}\})\mathsf{Set}(\mathsf{case}(x, \widehat{\{\}}, \widehat{\mathsf{T}})) \\ \qquad\qquad\qquad\qquad [x \in (\mathsf{W}x \in \{\mathit{zero}, \mathit{succ}\})\mathsf{Set}(\mathsf{case}(x, \widehat{\{\}}, \widehat{\mathsf{T}}))] \end{array}$$

which also is proved directly from the W-introduction rule.

Unfortunately the N-elimination rule and the N-equality rule can not be proved using the *intensional* equality in type theory. The reason for this is that there are more elements in the well-order representing the natural numbers than one expect at first. An element $\mathsf{sup}(a, b)$ of a well-order has a functional component b and the intensional equality means that two functions are equal only if they convert to each other. So the two functions

$$(x)\,0 \quad \text{and} \quad (x)\,1$$

which maps elements in the empty set to natural numbers are not equal even if they give the same result for all elements in the domain. The consequence of this for the representation of natural numbers is that there are elements in the well-order that do not represent any natural number. With an extensional equality this problem never occurs.

15.1 Representing inductively defined sets by well-orderings

Most programming languages have some construction for defining types by inductive definitions. “Old” languages use pointers and records and “modern” languages use more sophisticated constructions, see for example [51] and [72]. In type theory the well-order set constructor can be used for representing many inductively defined sets. But as we remarked above, we must have an extensional equality in order to get the correct elimination and equality rules.

We have shown above how one could represent binary trees and natural numbers by well-orders. Let us also show how one can define an inductively defined set which uses another set in its definition. Consider the set of binary trees with natural numbers in its nodes and defined by the following ML definition

$$\textbf{datatype}\ \mathit{BinTree} = \mathit{leaf}\ \textbf{of}\ \mathsf{N}\ \mid\ \mathit{node}\ \textbf{of}\ \mathsf{N} * \mathit{BinTree} * \mathit{BinTree}$$

In order to represent this set by a well-order one must consider the natural number as part of the constructor of the tree and instead of having a two element set as the set of constructors, we now need $\mathsf{N} \times \mathsf{N}$. The selectors for $\mathsf{inl}(n)$ is the empty set and for $\mathsf{inr}(n)$ the set $\{\mathit{left}, \mathit{right}\}$. So

$$\mathsf{W}(\mathsf{N}+\mathsf{N}, (x)\mathsf{Set}(\mathsf{when}(x, (n)\widehat{\{\}}, (n)\widehat{\{\mathit{left}, \mathit{right}\}})))$$

is a well-ordering that represents the type of binary trees with natural numbers in its nodes. The elements are of the form

$$\mathsf{sup}(\mathsf{inl}(n), \mathsf{case}) \qquad \text{and} \qquad \mathsf{sup}(\mathsf{inr}(n), (x)\mathsf{case}(x, t', t''))$$

where n is a natural number and t' and t'' are two elements in $\mathsf{W}(A, B)$. To get a better syntax, we can introduce three definitions:

$$\begin{array}{rcl}
\mathit{leaf}''(n) & \equiv & \mathsf{sup}(\mathsf{inl}(n), \mathsf{case}) \\
\mathit{node}''(n, t', t'') & \equiv & \mathsf{sup}(\mathsf{inr}(n), (x)\mathsf{case}(x, t', t'')) \\
\mathit{trec}''(t, a, b) & \equiv & \mathsf{wrec}(t, \\
& & \quad (y, z, u)\mathsf{when}(y, a, (n)b(n, \\
& & \qquad\qquad z(\mathit{left}), \\
& & \qquad\qquad z(\mathit{right}), \\
& & \qquad\qquad u(\mathit{left}), u(\mathit{right}))))
\end{array}$$

The function that adds all the numbers in a tree could in type theory be defined by

$$\begin{array}{rcl}
\mathit{addnum}(x) & \equiv & \mathit{trec}''(x, (n)n, (n, y, z, u, v)n + u + v) \\
& \equiv & \mathsf{wrec}(x, (y, z, u)\mathsf{when}(y, (n)n, (n)n + u(\mathit{left}) + u(\mathit{right}))
\end{array}$$

Chapter 16

General trees

When we introduced the well-order set constructor in the previous chapter, we said that many inductively defined sets could be represented by well-orders and that the elements of a well-order could be seen as well-founded trees. The well-order set constructor, however, is not easy to use when we want to define a family of mutually dependent inductive sets, or mutually dependent families of trees.

For example if we want to represent the types defined in ML by

$$\begin{array}{ll} \textbf{datatype} & Odd \ = sO \ \textbf{of} \ Even \\ \textbf{and} & Even = zeroE \mid sE \ \textbf{of} \ Odd; \end{array}$$

it is possible but quite complicated to do this by using well-orders. We therefore introduce a set constructor, Tree, which could be used for representing such sets in a more direct way. Notice that we must have an extensional equality to get the correct elimination and equality rule when we represent inductively defined types by well-orders and general trees. The set constructor for general trees was first introduced in [88] on which the following chapter is based.

The constructor should produce a family of sets instead of one set as the well-order set constructor does. In order to do this, we introduce a *name set*, which is a set of names of the mutually defined sets in the inductive definition. A suitable choice of name set for the example above would be $\{Odd, Even\}$. Instead of having one set of constructors B and one index family C over B, as in the well-order case, we now have one constructor set and one selector family for each element in A. The constructors form a family of sets B, where $B(x)$ is a set for each x in A and the selector family forms a family of sets C where $C(x,y)$ is a set for each x in A and y in $B(x)$. Furthermore, since the parts of a tree now may come from different sets, we introduce a function d which provides information about this; $d(x,y,z)$ is an element of A if $x \in A$, $y \in B(x)$ and $z \in C(x,y)$. We call this element the *component set name*.

The family of sets $\mathsf{Tree}(A,B,C,d)$ is a representation of the family of sets introduced by a collection of inductive definitions, for example an ML data type definition. It could also be seen as a solution to the equation

$$\mathcal{T} \cong (x)(\Sigma y \in B(x))(\Pi z \in C(x,y))\mathcal{T}(d(x,y,z))$$

where $\mathcal{T}$ is a family of sets over A and $x \in A$. This equation could be interpreted as a possibly infinite collection of ordinary set equations, one for each $a \in A$.

$$\begin{array}{rcl} \mathcal{T}(a_1) & \cong & (\Sigma y \in B(a_1))(\Pi z \in C(a_1,y))\,\mathcal{T}(d(a_1,y,z)) \\ \mathcal{T}(a_2) & \cong & (\Sigma y \in B(a_2))(\Pi z \in C(a_2,y))\,\mathcal{T}(d(a_2,y,z)) \\ & \vdots & \end{array}$$

Or, if we want to express the tree set constructor as the least fixed point of a set function operator.

$$\mathsf{Tree}(A,B,C,d) \;\cong\; \mathsf{FIX}((\mathcal{T})(x)(\Sigma y \in B(x))(\Pi z \in C(x,y))\,\mathcal{T}(d(x,y,z)))$$

Comparing this equation with the equation for the well-order set constructor

$$\mathsf{W}(B,C) \;\cong \mathsf{FIX}((\mathcal{X})(\Sigma y \in B)C(y) \to \mathcal{X})$$

we can see that it is a generalization in that the non-dependent function set, "$\rightarrow$", has become a set of dependent functions, Π. This is a natural generalization since we are now defining a family of sets instead of just one set and every instance of the family could be defined in terms of every one of the other instances. It is the function d that expresses this relation.

16.1 Formal rules

In order to be able to formulate the rules for the set constructor for trees, we introduce the primitive constant Tree which has the arity

$$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\otimes\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0}$$

tree of arity $\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$ and finally $\mathsf{treerec}$ of arity

$$\mathbf{0}\otimes(\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$$

The formation rule for the set of trees is:

Tree – formation

$$\frac{\begin{array}{l} A \textit{ set} \\ B(x) \textit{ set} \;\; [x \in A] \\ C(x,y) \textit{ set} \;\; [x \in A,\; y \in B(x)] \\ d(x,y,z) \in A \;\; [x \in A,\; y \in B(x),\; z \in C(x,y)] \\ a \in A \end{array}}{\mathsf{Tree}(A,B,C,d)(a) \textit{ set}}$$

The different parts have the following intuitive meaning:

- A, the name set, is a set of names for the mutually dependent sets.
- $B(x)$, the constructor set, is a set of names for the clauses defining the set x.
- $C(x,y)$, the selector family, is a set of names for selectors of the parts in the clause y in the definition of x.

- $d(x, y, z)$, the component set name, is the name of the set corresponding to the selector z in clause y in the definition of x.
- a determines a particular instance of the family of sets.

Understood as a set of syntax-trees generated by a grammar, the different parts have the following intuitive meaning:

- A is a set of non-terminals.
- $B(x)$ is a set of names for the alternatives defining the non-terminal x.
- $C(x, y)$, is a set of names for positions in the sequence of non-terminals in the clause y in the definition of x.
- $d(x, y, z)$, is the name of the non-terminal corresponding to the position z in clause y in the definition of x.
- a is the start symbol.

In order to reduce the notational complexity, we will write $\mathcal{T}(a)$ instead of $\mathsf{Tree}(A, B, C, d)(a)$ in the rest of this chapter.

The introduction rule for trees has the following form

Tree – introduction

$$\frac{\begin{array}{l} a \in A \\ b \in B(a) \\ c(z) \in \mathcal{T}(d(a, b, z)) \;\; [z \in C(a, b)] \end{array}}{\mathsf{tree}(a, b, c) \in \mathcal{T}(a)}$$

Intuitively:

- a is the name of one of the mutually dependent sets.
- b is one of the constructors of the set a.
- c is a function from $C(a, b)$ to a tree. This function defines the different parts of the element.

The element $\mathsf{tree}(a, b, c)$ in the set $\mathcal{T}(a)$ corresponds to the tree in figure 16.1, where $C(a, b) = \{z_1, \ldots, z_n, \ldots\}$ and $c(z_i) \in \mathcal{T}(d(a, b, z_i))$.

The elimination rule has the form

Tree – elimination

$$\frac{\begin{array}{l} D(x, t) \; set \;\; [x \in A, t \in \mathcal{T}(x)] \\ a \in A \\ t \in \mathcal{T}(a) \\ f(x, y, z, u) \in D(x, \mathsf{tree}(x, y, z)) \\ \qquad [x \in A, \; y \in B(x), \; z(v) \in \mathcal{T}(d(x, y, v)) \, [v \in C(x, y)], \\ \qquad\; u(v) \in D(d(x, y, v), \; z(v)) \, [v \in C(x, y)]] \end{array}}{\mathsf{treerec}(t, f) \in D(a, t)}$$

Its correctness follows from the computation rule for the non-canonical constant $\mathsf{treerec}$ which says that the expression $\mathsf{treerec}(d, e)$ is computed as follows

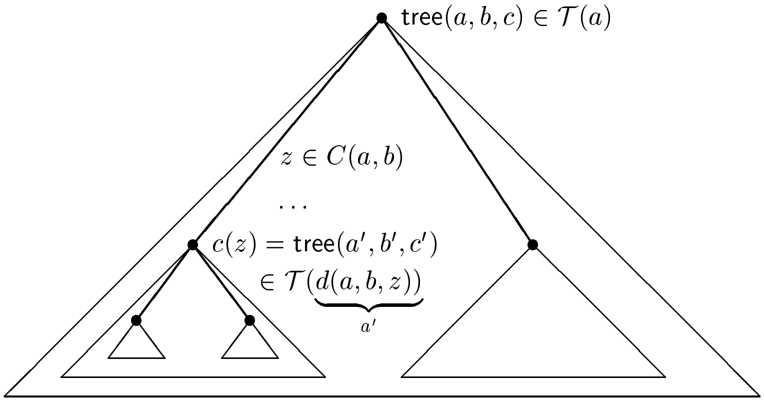

Figure 16.1: An element in the set $\mathsf{Tree}(A,B,C,d)(a)$

1. Evaluate d to canonical form.
2. If the value of d is $\mathsf{tree}(a,b,c)$ then the value of the expression is $e(a,b,c,(x)\mathsf{treerec}(c(x),d))$.

The computation rule is also reflected in the equality rule:

Tree-equality

$$\frac{\begin{array}{l} D(x,t)\ set\ \ [x \in A, t \in \mathcal{T}(x)] \\ a \in A \\ b \in B(a) \\ c(z) \in \mathcal{T}(d(a,b,z))\ \ [z \in C(a,b)] \\ f(x,y,z,u) \in D(x,\mathsf{tree}(x,y,z)) \\ \quad [x \in A, y \in B(x), z(v) \in \mathcal{T}(d(x,y,v))\,[v \in C(x,y)], \\ \quad\ u(v) \in D(d(x,y,v),z(v))\,[v \in C(x,y)]] \end{array}}{\mathsf{treerec}(\mathsf{tree}(a,b,c),f) = f(a,b,c,(x)\mathsf{treerec}(c(x),f)) \in D(a,\mathsf{tree}(a,b,c))}$$

16.2 Relation to the well-order set constructor

A well-order set $\mathsf{W}(B,C)$ can be seen as an instance of a Tree set. We get the well-orders by defining a family of trees on a set with only one element. If we make the definitions:

$$\begin{array}{rcl} \mathsf{W}(B,C) & = & \mathsf{Tree}(\mathsf{T},(x)B,(x,y)C(y),(x,y,z)\mathsf{tt},\mathsf{tt}) \\ \mathsf{sup}(b,c) & = & \mathsf{tree}(\mathsf{tt},b,c) \\ \mathsf{wrec}(t,f) & = & \mathsf{treerec}(t,(x,y,z,u)f(y,z,u)) \end{array}$$

where T is the set consisting of the element tt. Then we can derive the rules for well-orders from the rules for trees as follows:

Formation rule: If we assume that the premises of the well-order formation rule hold, that is, if we assume

$$\begin{array}{l} B\ set \\ C(y)\ set\ \ [y \in B] \end{array}$$

we can infer

$$\begin{array}{l}
\mathsf{T}\ \textit{set} \\
((x)B)(x)\ \textit{set}\ \ [x \in \mathsf{T}] \\
((x,y)C(y))(x,y)\ \textit{set}\ \ [x \in \mathsf{T}, y \in B] \\
((x,y,z)\mathsf{tt})(x,y,z) \in \mathsf{T}\ \ [x \in \mathsf{T}, y \in B, z \in C(y)] \\
\mathsf{tt} \in \mathsf{T}
\end{array}$$

and then, by the Tree-formation rule, get

$$\mathsf{Tree}(\mathsf{T}, (x)B, (x,y)C(y), (x,y,z)\mathsf{tt}, \mathsf{tt})\ \textit{set}$$

which is the same as

$$\mathsf{W}(B,C)\ \textit{set}$$

and also the conclusion of the formation rule. So we have proved that the formation rule holds for the definition of well-orders in terms of trees.

Introduction rule: Assume

$$\begin{array}{l}
b \in B \\
c(z) \in \mathsf{W}(B,C)\ \ [z \in C(b)]
\end{array}$$

From the last assumption we get

$$c(z) \in \mathsf{Tree}(\mathsf{T}, (x)B, (x,y)C(y), (x,y,z)\mathsf{tt}, \mathsf{tt})\ \ [z \in C(b)]$$

It then follows that

$$\begin{array}{l}
\mathsf{tt} \in \mathsf{T} \\
b \in ((x)B)(\mathsf{tt}) \\
c(z) \in \mathsf{Tree}(\mathsf{T}, (x)B, (x,y)C(y), (x,y,z)\mathsf{tt}, ((x,y,z)\mathsf{tt}))(\mathsf{tt}, b, z) \\
\quad [z \in ((x,y)C(y))(b)]
\end{array}$$

and, from the Tree-introduction rule,

$$\mathsf{tree}(\mathsf{tt}, b, c) \in \mathsf{Tree}(\mathsf{T}, (x)B, (x,y)C(y), (x,y,z)\mathsf{tt}, \mathsf{tt})$$

which is the same as

$$\mathsf{sup}(b,c) \in \mathsf{W}(B,C)$$

The elimination and equality rules could be proved in the same way.

16.3 A variant of the tree set constructor

We will in this section introduce a slight variant of the tree set constructor. Instead of having information in the element about what instance of the family a particular element belongs to, we move this information to the recursion operator. We call the new set constructor Tree', the new element constructor tree' and the new recursion operator $\mathsf{treerec}'$. The formation rule for Tree' is exactly the same as for Tree, but the other rules are slightly modified.

Tree'-introduction

$$\frac{\begin{array}{l} a \in A \\ b \in B(a) \\ c(z) \in \mathsf{Tree}'(A, B, C, d, d(a, b, z)) \quad [z \in C(a, b)] \end{array}}{\mathsf{tree}'(b, c) \in \mathsf{Tree}'(A, B, C, d, a)}$$

Tree'-elimination

$$\frac{\begin{array}{l} D(x, t) \textit{ set } \quad [x \in A, t \in \mathsf{Tree}'(A, B, C, d, x)] \\ a \in A \\ t \in \mathsf{Tree}'(A, B, C, d, a) \\ f(x, y, z, u) \in D(x, \mathsf{tree}'(y, z)) \\ \qquad [x \in A, y \in B(x), z(v) \in \mathsf{Tree}'(A, B, C, d, d(x, y, v))\, [v \in C(x, y)], \\ \qquad\; u(v) \in D(d(x, y, v), z(v))\, [v \in C(x, y)]] \end{array}}{\mathsf{treerec}'(d, a, t, f) \in D(a, t)}$$

The formulation of the equality rule is straightforward. Notice that we in the first version of the tree sets can view the constructor tree as a family of constructors, one for each $a \in A$. In this variant we have one constructor for the whole family, but instead we get a family of recursion operators, one for each a in A.

16.4 Examples of different tree sets

16.4.1 Even and odd numbers

Consider the following data type definition in ML:

$$\begin{array}{lll} \textbf{datatype} & \mathit{Odd} & = \mathit{sO}\ \textbf{of}\ \mathit{Even} \\ \textbf{and} & \mathit{Even} & = \mathit{zeroE} \mid \mathit{sE}\ \textbf{of}\ \mathit{Odd}; \end{array}$$

and the corresponding grammar:

$$\begin{array}{lll} \text{<odd>} & ::= & s_O(\text{<even>}) \\ \text{<even>} & ::= & 0_E \ \mid \ s_E(\text{<odd>}) \end{array}$$

If we want to define a set with elements corresponding to the phrases defined by this grammar (and if we consider <odd> as start symbol), we can define

$OddNrs = \mathsf{Tree}(A, B, C, d)(a)$ where:

$$
\begin{aligned}
A &= \{Odd, Even\} \\
\\
a &= Odd \\
\\
B(Odd) &= \{s_O\} \\
B(Even) &= \{zero_E, s_E\} \\
i.e.\ B &= (x)\mathsf{case}_{\{Odd,Even\}}(x, \{s_O\}, \{zero_E, s_E\}) \\
\\
C(Odd, s_O) &= \{pred_O\} \\
C(Even, zero_E) &= \{\} \\
C(Even, s_E) &= \{pred_E\} \\
i.e.\ C &= (x, y)\mathsf{case}_{\{Odd,Even\}}(x, \\
&\qquad\qquad \{pred_O\}, \\
&\qquad\qquad \mathsf{case}_{\{zero_E, s_E\}}(y, \{\}, \{pred_E\})) \\
\\
d(Odd, s_O, pred_O) &= Even \\
d(Even, s_E, pred_E) &= Odd \\
i.e.\ d &= (x, y, z)\mathsf{case}_{\{Odd,Even\}}(x, \\
&\qquad\qquad Even, \\
&\qquad\qquad Odd)
\end{aligned}
$$

The element $s_E(s_O(zero_E))$ is represented by

$$2_E = \mathsf{tree}(Even, s_E, (x)\mathsf{tree}(Odd, s_O, (x)\mathsf{tree}(Even, zero_E, (x)\mathsf{case}_{\{\}}(x)))$$

and $s_O(s_E(s_O(0_E)))$ is represented by

$$3_O = \mathsf{tree}(Odd, s_O, (x)2_E)$$

We get the set of even numbers by just changing the "start symbol"

$$EvenNrs = \mathsf{Tree}(A, B, C, d)(Even)$$

and we can define a mapping from even or odd numbers to ordinary natural numbers by:

$$
\begin{aligned}
tonat(w) = \ & \mathsf{treerec}(w, \\
& \quad (x, y, z, u)\ \mathsf{case}_{\{Odd,Even\}}(x, \\
& \qquad\qquad \mathsf{succ}(u(pred_O)), \\
& \qquad\qquad \mathsf{case}_{\{zero_E, s_E\}}(y, \\
& \qquad\qquad\qquad 0, \\
& \qquad\qquad\qquad \mathsf{succ}(u(pred_E))))))
\end{aligned}
$$

and it is easy to prove that

$$tonat(w) \in \mathsf{N}\ [v \in \{Odd, Even\}, w \in \mathsf{Tree}(A, B, C, d)(v)]$$

16.4.2 An infinite family of sets

In ML, and all other programming languages with some facility to define mutually inductive types, one can only introduce finitely many new data types. A family of sets in type theory, on the other hand, could range over infinite sets and the tree set constructor therefore could introduce families with infinitely many instances. In this section we will give an example where the name set is infinite.

The problem is to define a set $Array(A, n)$, whose elements are lists with exactly n elements from the set A. If we make a generalization of ML's data type construction to dependent types this type could be defined as:

$$\begin{array}{rcl} Array(E,0) & \equiv & empty \\ Array(E,s(n)) & \equiv & add\ \mathbf{of}\ E \times Array(E,n)) \end{array}$$

The corresponding definition with the tree set constructor is:

$$Array(E,n) \quad \equiv \quad \mathsf{Tree}'(\mathsf{N}, B, C, d)(n)$$

where

$$\begin{array}{rcl} B(n) & \equiv & \mathsf{natrec}(n, \{nil\}, (x,y)E) \\ C(n,x) & \equiv & \mathsf{natrec}(n, \{\}, (x,y)\{tail\}) \\ d(n,x,y) & \equiv & \mathsf{natrec}(n, \mathsf{case}_{\{\}}(y), (z,u)z) \end{array}$$

We can then define:

$$\begin{array}{rcl} empty & \equiv & \mathsf{tree}'(nil, \mathsf{case}_{\{\}}) \\ add(e,l) & \equiv & \mathsf{tree}'(e,l) \end{array}$$

as the elements. Notice that we in this example have used the variant of the tree constructor we introduced in section 16.3.

Part II

Subsets

Chapter 17

Subsets in the basic set theory

We will in this section add sets formed by comprehension directly to the basic set theory in a similar way as we have introduced the other primitive sets. As we already have mentioned, we will in this approach not be able to formulate a satisfactory elimination rule.

Let A be a set and B a propositional function (family of sets) defined on the set A, i.e. assume A *set* and $B(x)$ *set* $[x \in A]$. From these assumptions and the explanation of what it means to be a set, it follows that the canonical elements and their equality relation is understood for the set A and for the set $B(a)$ whenever $a \in A$.

The subset of A with respect to B is denoted

$$\{|\}(A, B)$$

where $\{|\}$ is a constant of arity $\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$. Instead of $\{|\}(A, B)$, we shall use the more lucid notation

$$\{x \in A \mid B(x)\}$$

This set forming operation is defined, as all the other sets, by prescribing how to form canonical elements and how to form equal canonical elements: if a is a canonical element in the set A and $B(a)$ is true, i.e. if there exists an element $b \in B(a)$, then a is also a canonical element in the set $\{x \in A \mid B(x)\}$. And if a and c are equal canonical elements in the set A and $B(a)$ is true, then a and c are also equal canonical elements in the set $\{x \in A \mid B(x)\}$. Since every propositional function is extensional in the sense that it yields equal propositions (sets) when it is applied to equal elements, it follows from $a = c \in A$ and $B(x)$ *set* $[x \in A]$ that $B(a)$ and $B(c)$ are equal propositions (sets). And, consequently, from the requirement that $B(a)$ is true, we immediately get that also $B(c)$ is true.

The introduction of the canonical elements makes sense precisely when A is a set and $B(x)$ is a set under the assumption that $x \in A$. Hence, the formation rule for the subset becomes:

Subset – formation

$$\frac{A\ set \qquad B(x)\ set\ \ [x \in A]}{\{x \in A \mid B(x)\}\ set}$$

For many sets, the prescription of how to form canonical elements and equal canonical elements immediately justifies the introduction rules, since the requirements for forming canonical elements can be expressed as premises of the introduction rules. The canonical elements of the subset, however, cannot justify an introduction rule in this way, because the requirement that a should be a canonical element in A cannot be expressed as a premise. So we cannot form the introduction rule according to the general scheme. Instead, the introduction rule introduces expressions both of canonical and noncanonical form. From the explanation of the judgement $a \in A$, we know that a, when evaluated, will yield a canonical element in the set A as result. So if $B(a)$ is true, we know that a will also yield a canonical element in the set $\{x \in A \mid B(x)\}$. The introduction rule becomes:

Subset – introduction 1

$$\frac{a \in A \qquad b \in B(a)}{a \in \{x \in A \mid B(x)\}}$$

And similarly, if $a_1 = a_2 \in A$, the evaluation of a_1 and a_2 will yield equal canonical elements in the set A as result and, therefore, if $B(a_1)$ is true, they will yield equal canonical elements in the set $\{x \in A_2 \mid B_2(x)\}$. Since $b \in B(a_1)$ it follows from $a_1 = a_2 \in A$ and $b \in B(a_1)$ that $b \in B(a_2)$. This justifies the second introduction rule for subsets:

Subset – introduction 2

$$\frac{a_1 = a_2 \in A \qquad b \in B(a_1)}{a_1 = a_2 \in \{x \in A \mid B(x)\}}$$

The subsets are different from all other sets in that the canonical and noncanonical forms of expressions depend only on the parameter set A. So from an element expression alone, it is impossible to determine the form of its set; it may belong to A as well as to a subset of A. But this cannot cause any confusion, since an element is always given *together* with its set.

An elimination rule which captures the way we have introduced elements in a subset is impossible to give in type theory because when we have an element a in a subset $\{x \in A \mid B(x)\}$ we have no explicit construction of the proof element of $B(a)$. The best formulation of an elimination rule we can give is the following:

Subset – elimination 1

$$\frac{c \in \{x \in A \mid B(x)\} \qquad d(x) \in C(x) \;\; [x \in A,\; y \in B(x)]}{d(c) \in C(c)}$$

where y must not occur free in d nor in C

Because of the syntactical restriction on free variables in the subset-elimination rule the strength of this rule is connected with the possibility of having rules in type theory where free variables, other than those discharged by the rule, may disappear in the conclusion. In our basic formulation of Martin-Löf's set theory with the intensional identity Id, there are very few possibilities to get rid of free variables in an essential way.

The strength of adding subsets to set theory with the elimination rule above is discussed in detail in [90] where it is shown that propositions of the form

$$(\forall x \in \{z \in A \mid P(z)\})P(x) \tag{$*$}$$

cannot in general be proved. In the intensional formulation we have of set theory, not even $(\forall x \in \{z \in \mathsf{T} \mid \bot\})\bot$ can be derived. The proof in [90] of this is rather complicated, using a normalization argument.

Propositions of the form $(*)$ are important when modularizing program derivations, using a top-down approach and decomposing the specification into subproblems. When solving the subproblems we may want to use lemmas which have already been proved. The main idea of splitting up a problem into lemmas is, in program derivation as well as in mathematics, that our original problem can be reduced to the lemmas; in particular, there should be no need to look into the proofs of the lemmas. If we have a lemma which talks about subsets we certainly want $(*)$ to be provable since if $a \in \{x \in A \mid P(x)\}$ we want to be able to conclude $P(a)$ without having to investigate the proof of $a \in \{x \in A \mid P(x)\}$.

In set theory with the extensional equality Eq, there are more cases for which $(*)$ can be proved. Let $P(x)$ *set* $[x \in A]$. The predicate $P(x)$ is called *stable* if

$$\neg\neg P(x) \;\rightarrow\; P(x) \; [x \in A]$$

Using strong Eq-elimination together with the universe, it is proved in [90] that $(*)$ holds for all stable predicates, that is

$$(\forall x \in A)(\neg\neg P(x) \rightarrow P(x)) \;\rightarrow\; (\forall x \in \{z \in A \mid P(z)\})P(x)$$

holds in the extensional theory. Extending the basic extensional set theory with subset is discussed in detail in Salvesen [92].

It is also shown in [90] that if we put $P(x)$ equal to

$$(\exists y \in \mathsf{N})T(x,x,y) \vee \neg(\exists y \in \mathsf{N})T(x,x,y)$$

where T is Kleene's T-predicate, and put A equal to N, then $(*)$ cannot be derived in Martin-Löf's set theory extended with the above rules for subsets irrespectively of how we formulate the remaining rules; the only requirements are that the axiom of choice, as formulated in [69, 70], can be proved and that a typable term can be computed by a Turing machine.

So the approach of this chapter to introduce subsets in the same way as the other sets and interpret proposition as sets results in a very weak elimination rule which, at least in the intensional theory, will not work in practice.

Chapter 18

The subset theory

In order to get an elimination rule by which we, for instance, can derive $P(a)$ *true* from $a \in \{x \in A \mid P(x)\}$ we will now, following ideas of Martin-Löf, give a new meaning of the judgement A *set*. We then also have to give new explanations of the other forms of judgement. All judgements will be explained in terms of our previous explanations for set theory. We will call this new theory the *subset theory* and refer to the earlier set theory as the *basic set theory* or just set theory.

The crucial difference between the basic set theory and the subset theory is that propositions will no longer be viewed as sets in the subset theory. However, the semantics of propositions in the subset theory will use propositions as sets in the basic set theory. So we must first extend our language by introducing *primitive* constants for the logical constants: &, $\vee$ and $\supset$ of arity $\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$, $\perp$ of arity $\mathbf{0}$, $\forall$ and $\exists$ of arity $\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$. We also need a primitive constant ID of arity $\mathbf{0}\otimes\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$ for forming the proposition that two elements of a certain set are equal. Instead of $\mathsf{ID}(A, a, b)$ we will often write $a =_A b$.

We will give detailed explanations of the judgements for a subset theory without universes. The intuition behind the semantics is that a set A in the subset theory consists of those elements x in a base set A' in the basic set theory such that $A''(x)$ holds, where A'' is a propositional function on A' in the basic set theory. The situation with universes is somewhat more complicated and will be discussed later in the chapter.

18.1 Judgements without assumptions

As when we explained the meaning of the judgements of the basic set theory, we first explain the judgements not depending on any assumptions.

18.1.1 What does it mean to be a set?

> To know the judgement
>
> $$A \ set$$
>
> in the subset theory is to have a pair (A', A'') where we know that A' is a set in the basic set theory and that A'' is a propositional function on A' in the basic set theory.

So in order to know that A is a set in the subset theory we must have A' and A'' and know the judgements

- $A' \ set$
- $A''(x) \ prop \ \ [x \in A']$

in the way we already have explained in the basic set theory. Note that the judgement $A''(x) \ prop \ \ [x \in A']$ in the basic set theory is just an abbreviation of the judgement $A''(x) \ set \ \ [x \in A']$.

18.1.2 What does it mean for two sets to be equal?

Let A and B be sets in the subset theory. According to the explanation of what it means to be a set in the subset theory, we then have sets A' and B' and propositional functions A'' and B'' on A' and B', respectively. To know the judgement that A and B are equal sets in the subset theory is explained in the following way:

> To know that A and B are equal sets
>
> $$A = B$$
>
> in the sense of the subset theory, is to know that A' and B' are equal sets in the basic set theory and that $A''(x)$ and $B''(x)$ are equivalent propositions on A' in the sense of the basic set theory.

So in order to know that A and B are equal, we must know the judgements

- $A' = B'$
- $A''(x) \Leftrightarrow B''(x) \ true \ \ [x \in A']$

as explained in the basic set theory. Since propositions are interpreted as sets in the basic theory, the judgement $A''(x) \Leftrightarrow B''(x) \ true \ \ [x \in A']$ means that we have an element in $(A''(x) \rightarrow B''(x)) \times (B''(x) \rightarrow A''(x))$ under the assumption $x \in A'$.

18.1.3 What does it mean to be an element in a set?

According to the explanation of the judgement $A \ set$ in the subset theory, A consists of those elements x in A' such that $A''(x)$ holds:

> To know the judgement
>
> $$a \in A$$
>
> where A is a set in the sense of the subset theory, we must know that a is an element in A' and that $A''(a)$ is true.

So in order to know that a is an element in the set A we must know the judgements

- $a \in A'$
- $A''(a)$ *true*

as explained in the basic set theory. Note that $A''(a)$ *true* means that we have an element in the set $A''(a)$.

18.1.4 What does it mean for two elements to be equal in a set?

If $a \in A$ and $b \in A$ then the explanation of equality between a and b is the following.

> To know that a and b are equal elements in a set A
>
> $$a = b \in A$$
>
> in the sense of the subset theory is to know the judgement
>
> $$a = b \in A'$$
>
> in the basic set theory.

So that two elements are equal in a subset means that they must be equal elements in the base set of the subset.

18.1.5 What does it mean to be a proposition?

> To know a proposition P in the subset theory is to know a proposition $P^\star$ in the basic set theory.

Since P may contain quantifiers ranging over subsets, $P^\star$ will depend on the interpretation of subsets. Since propositions are interpreted as sets in the basic set theory, $P^\star$ is nothing but a set in the basic theory.

18.1.6 What does it mean for a proposition to be true?

> To know that the proposition P is true in the subset theory is to know that $P^\star$ is true in set theory.

So a proposition P is true in the subset theory if we have an element in the set $P^\star$ in the basic set theory.

18.2 Hypothetical judgements

The explanation of a judgement depending on assumptions is done, as in the basic set theory, by induction on the number of assumptions. Leaving out higher order assumptions, a member $\mathcal{C}_k$ in an arbitrary context $\mathcal{C}_1, \ldots, \mathcal{C}_n$ in the subset theory is either of the form $x_k \in A_k(x_1, \ldots, x_{k-1})$ where $A_k(x_1, \ldots, x_{k-1})$ is a subset in the context $\mathcal{C}_1, \ldots, \mathcal{C}_{k-1}$ or of the form $P(x_1, \ldots, x_k)$ *true* where $P(x_1, \ldots, x_k)$ is a proposition in the context $\mathcal{C}_1, \ldots, \mathcal{C}_{k-1}$. In order to avoid heavy notation, we will explain hypothetical judgements in the subset theory in a context

$$x \in C,\ P(x)\ \mathit{true},\ y \in D(x)$$

where C is a subset, $P(x)$ a proposition in the context $x \in C$, and $D(x)$ is a subset in the context $x \in C,\ P(x)$ *true*. Given the explanations of the different forms of judgements in this context of length 3, it is straightforward to explain the judgements in an arbitrary context.

18.2.1 What does it mean to be a set under assumptions?

To know the judgement

$$A(x,y)\ \mathit{set}\ [x \in C,\ P(x)\ \mathit{true},\ y \in D(x)]$$

in the subset theory where we already know

$$\begin{array}{l} C\ \mathit{set} \\ P(x)\ \mathit{prop}\ \ [x \in C] \\ D(x)\ \mathit{set}\ \ [x \in C,\ P(x)\ \mathit{true}] \end{array}$$

is to have a pair (A', A'') such that

$$A'(x,y)\ \mathit{set}\ \ [x \in C',\ y \in D'(x)]$$

and

$$A''(x,y,z)\ \mathit{prop}\ \ [x \in C',\ y \in D'(x),\ z \in A'(x,y)]$$

both hold in the basic set theory.

When defining A' and A'' it must be done in such a way that it does not come in conflict with the sets obtained from A by substitution. So we must require the following substitution property:

$$\begin{array}{c} A(a,b)' \text{ is equal to } A'(a,b). \\ A(a,b)'' \text{ is equal to } A''(a,b). \end{array}$$

Note that being a set under assumptions only depends on the base sets of the sets in the assumption list and in particular does not depend on any proposition being true.

18.2.2 What does it mean for two sets to be equal under assumptions?

To know the judgement

$$A(x,y) = B(x,y) \;\; [x \in C,\; P(x) \; true,\; y \in D(x)]$$

in the subset theory, where $A(x,y)$ and $B(x,y)$ are sets in the context $x \in C,\; P(x)\; true,\; y \in D(x)$, is to know the judgements

$$A'(x,y) = B'(x,y) \;\; [x \in C',\; y \in D'(x)]$$

and

$$A''(x,y) \Leftrightarrow B''(x,y) \; true \;\; [x \in C',\; C''(x) \; true,\; P^{\star}(x) \; true,\; y \in D'(x),\; D''(x,y) \; true]$$

in the basic set theory.

So that the base sets of the two equal sets are equal only depends on the base sets of the subsets in the assumption list. The equivalence of the propositional parts of the sets, however, may depend also on the propositional parts of the sets in the assumption list as well as on the truth of propositions.

18.2.3 What does it mean to be an element in a set under assumptions?

To know the judgement

$$a(x,y) \in A(x,y) \;\; [x \in C,\; P(x) \; true,\; y \in D(x)]$$

in the subset theory, where $A(x,y)$ is a set in the context $x \in C,\; P(x)\; true,\; y \in D(x)$, is to know the judgements

$$a(x,y) \in A'(x,y) \;\; [x \in C',\; y \in D'(x)]$$

and

$$A''(x,y,a(x,y)) \; true \;\; [x \in C',\; C''(x) \; true,\; P^{\star}(x) \; true,\; y \in D'(x),\; D''(x,y) \; true]$$

in the basic set theory.

Note that $a(x,y)$ is an element in the base set of $A(x,y)$ only depends on the base sets of the sets in the assumption list and in particular does not depend on any proposition being true.

18.2.4 What does it mean for two elements to be equal in a set under assumptions?

To know the judgement

$$a(x,y) = b(x,y) \in A(x,y) \;\; [x \in C,\; P(x) \; true,\; y \in D(x)]$$

in the subset theory, where $a(x,y) \in A(x,y)$ and $b(x,y) \in A(x,y)$ in the context $x \in C,\ P(x)\ true,\ y \in D(x)$, is to know the judgement

$$a(x,y) = b(x,y) \in A'(x,y) \ \ [x \in A',\ y \in B'(x)]$$

in the basic set theory.

So that two elements are equal in a set under assumptions means that they must be equal already as elements in the base set, only depending on the base sets of the sets in the assumption list.

18.2.5 What does it mean to be a proposition under assumptions?

To know the judgement

$$Q(x,y)\ prop \ \ [x \in C,\ P(x)\ true,\ y \in D(x)]$$

in the subset theory is to know the judgement

$$Q^\star(x,y)\ prop \ \ [x \in C',\ y \in D'(x)]$$

in the basic set theory.

We must also require the substitution property

$$Q(a,b)^\star \text{ is equal to } Q^\star(a,b)$$

18.2.6 What does it mean for a proposition to be true under assumptions?

To know the judgement

$$Q(x,y)\ true \ \ [x \in C,\ P(x)\ true,\ y \in D(x)]$$

in the subset theory, where $Q(x,y)$ is a proposition in the context $x \in C,\ P(x)\ true,\ y \in D(x)$, is to know the judgement

$$\begin{array}{ll} Q^\star(x,y)\ true & [x \in C',\ C''(x)\ true, \\ & \ P^\star(x)\ true,\ y \in D'(x),\ D''(x,y)\ true] \end{array}$$

in the basic set theory.

18.3 General rules in the subset theory

With the exception of the rule Proposition as set, all the general rules of the basic set theory also hold in the subset theory. Let us as an example justify the Set equality rule

$$\frac{a \in A \qquad A = B}{a \in B}$$

By the explanations of judgements of the form $a \in A$ and $A = B$ in the subset theory, we have to show that if the judgements $a \in A'$, $A''(a)$ *true*, $A' = B'$ and $A''(x) \Leftrightarrow B''(x)$ *true* $[x \in A']$ all hold in set theory, then $a \in B'$ and $B''(a)$ *true* also hold in set theory. That $a \in B'$ holds follows from $a \in A'$, $A' = B'$ and the Type equality rule in set theory. From $A''(a)$ *true* and $A''(x) \rightarrow B''(x)$ *true* $[x \in A']$ we get that $B''(a)$ is true by substitution and $\rightarrow$-elimination.

Since a proposition is interpreted as a set in the basic set theory, we did not introduce judgements of the forms P *prop* and P *true* in the formalization of set theory. For instance, an assumption of the form P *true* in set theory can be understood as an assumption $y \in P$ where y is a new variable. In the subset theory, however, we must have judgements of the forms P *prop* and P *true* in the formal system and therefore we have to add general rules involving these two forms of judgement.

Assumption

$$\frac{P \ prop}{P \ true \ \ [P \ true]}$$

By the explanation of what it means to be a proposition in the subset theory, we know that $P^\star$ is a proposition, that is a set, in the basic set theory. Hence, by the assumption rule in set theory, we have $y \in P^\star \ \ [y \in P^\star]$ which is the meaning of P *true* $[P$ *true*$]$.

The judgement

$$C(x) \ prop \ \ [x \in A]$$

means that the judgement $C^\star(x)$ *set* $[x \in A']$ holds in the basic set theory. By the rule Substitution in sets and the substitution property of $C^\star(x)$ we therefore have the rule

Substitution in propositions

$$\frac{C(x) \ prop \ \ [x \in A] \qquad a \in A}{C(a) \ prop}$$

The rule

Cut rule for propositions

$$\frac{Q \ prop \ \ [P \ true] \qquad P \ true}{Q \ prop}$$

is justified in the following way. The judgement Q *prop* $[P$ *true*$]$ in the subset theory means that $Q^\star$ *set* $[y \in P^\star]$ in set theory and the judgement P *true* means that we have an element a in the set $P^\star$. By Substitution in sets we therefore get $Q^\star$ *set*, that is, $Q^\star$ *prop* as desired.

In a similar way, we can justify the rules

Cut rule for equal sets

$$\frac{A = B \ \ [P \ true] \qquad P \ true}{A = B}$$

Cut rule for true propositions

$$\frac{Q \; true \;\; [P \; true] \qquad P \; true}{Q \; true}$$

Cut rule for elements in sets

$$\frac{a \in A \;\; [P \; true] \qquad P \; true}{a \in A}$$

Cut rule for equal elements in sets

$$\frac{a = b \in A \;\; [P \; true] \qquad P \; true}{a = b \in A}$$

18.4 The propositional constants in the subset theory

Without a universe of propositions, which we will introduce later, the propositional constants are the logical constants and the propositional equality.

18.4.1 The logical constants

Let P and Q be propositions in the subset theory. This means that we have propositions, that is sets, $P^\star$ and $Q^\star$ in the basic theory. Propositions built up from P and Q by the sentential connectives are given meaning in the following way:

> $(P\&Q)^\star$ is defined to be the proposition $P^\star \times Q^\star$.
>
> $(P \vee Q)^\star$ is defined to be the proposition $P^\star + Q^\star$.
>
> $(P \supset Q)^\star$ is defined to be the proposition $P^\star \to Q^\star$.

The truth T and absurdity $\perp$ are given meaning in a similar way:

> $\mathsf{T}^\star$ is defined to be the proposition T.
>
> $\perp^\star$ is defined to be the proposition $\emptyset$.

So a sentential constant is given meaning by the use of the same set forming constant as when interpreting proposition as sets. However, the situation is more complicated when we come to the quantifiers.

Let A be a set and P a propositional function on A in the subset theory. We then have, according to the meaning of being a set and a propositional function on a set, a base set A' and propositional functions A'' and $P^\star$ defined on A' in the basic set theory. The propositions obtained from P by quantification on A are given meaning in the following way:

The proposition $((\forall x \in A)P(x))^\star$ is defined to be

$$(\Pi x \in A')(A''(x) \to P^\star(x))$$

The proposition $((\exists x \in A)P(x))^\star$ is defined to be

$$(\Sigma x \in A')(A''(x) \times P^\star(x))$$

It is now easy to justify the rules of first order logic as we have formulated them earlier. As an example, we justify the rules for the universal quantifier.

$\forall$ – formation

$$\frac{A \ prop \qquad P(x) \ prop \ \ [x \in A]}{(\forall x \in A)P(x) \ prop}$$

We must show that $(\Pi x \in A')(A''(x) \to P^\star(x))$ is a proposition, that is a set, in the basic set theory from the assumptions that we already know the judgements $A' \ set$, $A''(x) \ prop \ \ [x \in A']$ and $P^\star(x) \ prop \ \ [x \in A']$. By $\to$-formation we get $A''(x) \to P^\star(x) \ set \ \ [x \in A']$ which gives $(\Pi x \in A')(A''(x) \to P^\star(x)) \ set$ as desired.

$\forall$ – introduction

$$\frac{P(x) \ true \ \ [x \in A]}{(\forall x \in A)P(x) \ true}$$

That we know the judgement $P(x) \ true \ \ [x \in A]$ in the subset theory means that we know the judgement $P^\star(x) \ true \ \ [x \in A', \ A''(x) \ true]$ in the basic set theory. So we have an expression b for which we know the judgement $b(x) \in P^\star(x) \ \ [x \in A', y \in A''(x)]$ in the basic set theory. By $\to$-introduction, we get $\lambda y.b(x) \in A''(x) \to P^\star(x) \ \ [x \in A']$ which, by Π-introduction, gives $\lambda x.\lambda y.b(x) \in (\Pi x \in A')(A''(x) \to P^\star(x))$. Hence, we know the judgement $(\Pi x \in A')(A''(x) \to P^\star(x)) \ true$ as desired.

$\forall$ – elimination 1

$$\frac{(\forall x \in A)P(x) \ true \qquad a \in A}{P(a) \ true}$$

Assume that we have expressions b and c for which we know the judgements $b \in (\Pi x \in A')(A''(x) \to P^\star(x))$, $a \in A$ and $c \in A''(a)$ in the basic set theory. By Π-elimination we get $\mathsf{apply}(b, a) \in A''(a) \to P^\star(a)$ and then, by $\to$-elimination, $\mathsf{apply}(\mathsf{apply}(b, a), c) \in P^\star(a)$. So $P^\star(a)$ is true in set theory as desired.

18.4.2 The propositional equality

Let A be a subset and a and b elements in A. Then the meaning of $a =_A b$ is given by

The proposition $(a =_A b)^\star$ is defined to be $\mathsf{Id}(A', a, b)$.

We have the following rules for the propositional equality:

$=$ – formation

$$\frac{a \in A \qquad b \in A}{a =_A b \ prop}$$

$=$ – introduction

$$\frac{a = b \in A}{a =_A b \; true}$$

$=$ – elimination

$$\frac{C(x) \; prop \;\; [x \in A] \qquad a =_A b \; true \qquad C(a) \; true}{C(b) \; true}$$

We justify the elimination rule. The judgement $a =_A b$ *true* means that we have an element c in the set $\mathsf{Id}(A', a, b)$ and the judgement $C(a)$ *true* means that we have an element d in the set $C^\star(a)$. Using Id-elimination on $c \in \mathsf{Id}(A', a, b)$ and $\lambda u.u \in C^*(x) \to C^*(x) \;\; [x \in A]$ we get $\mathsf{idpeel}(c, (x)\lambda u.u) \in C^*(a) \to C^*(b)$. Since $d \in C^*(a)$ we then obtain, by $\to$-elimination,

$$\mathsf{apply}(\mathsf{idpeel}(c, (x)\lambda u.u), d) \in C^*(b)$$

So, $C^*(b)$ is true in the basic set theory which is the meaning of the judgement $C(b)$ *true* in the subset theory.

18.5 Subsets formed by comprehension

Sets in the subset theory are built up by the set forming operations we already have in the basic set theory and by set comprehension. The semantics of subsets introduced by comprehension is the following:

> $\{x \in A \mid P(x)\}'$ is defined to be the set A' and $\{x \in A \mid P(x)\}''$ is defined to be the propositional function $(z)(A''(z) \times P^\star(z))$ on A'.

The formation rule

Subset – formation

$$\frac{A \; set \qquad P(x) \; prop \;\; [x \in A]}{\{x \in A \mid P(x)\} \; set}$$

is justified in the following way. We assume that we know the interpretations of the premises, that is that we know the judgements A' *set*, $A''(x)$ *prop* $[x \in A']$ and $P^\star(x)$ *prop* $[x \in A']$ as explained in the basic set theory. Since $\{x \in A \mid P(x)\}'$ is defined to be A', we get that $\{x \in A \mid P(x)\}'$ is a set. By $\times$ -introduction we get that $A''(x) \times P^\star(x)$ is a proposition when $x \in A'$.

It is also easy to justify the introduction rule:

Subset – introduction

$$\frac{a \in A \qquad P(a) \; true}{a \in \{x \in A \mid P(x)\}}$$

Now we obtain the desired elimination rules for comprehension.

Subset – elimination for sets

$$\frac{a \in \{x \in A \mid P(x)\} \qquad c(x) \in C(x) \ \ [x \in A,\ P(x)\ true]}{c(a) \in C(a)}$$

This rule is justified as follows. We assume that we already know the judgements

$$\begin{array}{l} a \in A' \\ A''(a) \times P^\star(a)\ true \\ c(x) \in C'(x) \ \ [x \in A'] \\ C''(c(x))\ true \ \ [x \in A',\ A''(x)\ true,\ P^\star(x)\ true] \end{array}$$

in the basic set theory. From the first and third of these judgements we get, by substitution and the substitution property, that $c(a) \in C(a)'$. By $\times$-elimination, substitution and the substitution property we get from the first, second and fourth judgements that $C(a)''(c(a))$ *true* holds. In a similar way we can justify the rule

Subset – elimination for propositions

$$\frac{a \in \{x \in A \mid P(x)\} \qquad Q(x)\ true \ \ [x \in A,\ P(x)\ true]}{Q(a)\ true}$$

By putting $Q(x)$ equal to $P(x)$ in Subset-elimination for proposition we see that now we can derive $P(a)$ *true* from $a \in \{x \in A \mid P(x)\}$ which in general is not possible in the basic theory.

18.6 The individual set formers in the subset theory

For each set A obtained by any of the individual set formers we have to define the set A' and the propositional function A'' on A' in the basic set theory. In general, the formation of a set is made in a context which we will not mention explicitly. In particular, the substitution property must be satisfied when we substitute terms for the variables in the context. Because of the inductive way the set is introduced, it is easy to see that the substitution property holds.

To the rules for the individual sets in the basic theory, we will add rules for proving the truth of propositions by structural induction. These new rules will be called elimination rules for propositions. For the inductively defined sets we will also give equality rules which will reflect their interpretation in the basic set theory.

18.6.1 Enumeration sets

An enumeration set has the same elements in the subset theory as it has in the basic theory:

> $\{i_1, \ldots, i_n\}'$ is defined to be the set $\{i_1, \ldots, i_n\}$ and $\{i_1, \ldots, i_n\}''$ is defined to be the propositional function $(z)\mathsf{T}$ on $\{i_1, \ldots, i_n\}$.

To the rules for enumeration sets in the basic theory we have to add the rule

$\{i_1, \ldots, i_n\}$ – elimination for propositions

$$\frac{\begin{array}{l} a \in \{i_1, \ldots, i_n\} \\ Q(x)\ prop\ \ [x \in \{i_1, \ldots, i_n\}] \\ Q(i_1)\ true \\ \quad\vdots \\ Q(i_n)\ true \end{array}}{Q(a)\ true}$$

This rule is justified in the following way. That the judgement $Q(x)$ *prop* $[x \in \{i_1, \ldots, i_n\}]$ holds in the subset theory means that $Q^\star(x)$ *set* $[x \in \{i_1, \ldots, i_n\}]$ holds in the basic theory since $\{i_1, \ldots, i_n\}'$ is $\{i_1, \ldots, i_n\}$. The judgement $Q(i_k)$ *true* means that we have an element b_k in the set $Q^*(i_k)$. Hence, we can use $\{i_1, \ldots, i_n\}$-elimination 1 to obtain $\mathsf{case}(a, b_1, \ldots, b_n) \in Q^*(a)$. So $Q^*(a)$ is true in the basic set theory and, hence, $Q(a)$ is true in the subset theory as desired.

The other rules for enumeration sets are also straightforward to justify.

18.6.2 Equality sets

The main purpose of the equality sets in the basic set theory is to reflect the judgemental equality to the propositional level. Since propositions are not interpreted as sets in the subset theory, we have introduced equality as a primitive proposition, so there is really no need of equality sets in the subset theory. However, they can be given semantics in the subset theory:

> $\mathsf{Id}(A, a, b)'$ is defined to be the set $\mathsf{Id}(A', a, b)$ and $\mathsf{Id}(A, a, b)''$ is defined to be the propositional function $(z)\mathsf{T}$ on $\mathsf{Id}(A', a, b)$.

18.6.3 Natural numbers

The natural numbers in the subset theory are, of course, the same as the natural numbers in the basic set theory:

> N' is defined to be the set N and N'' is defined to be the propositional function $(z)\mathsf{T}''$ on N.

The rules for N are all easy to justify and as an example we justify the new N-elimination rule.

N – elimination for propositions

$$\frac{\begin{array}{l} Q(x)\ prop\ \ [x \in \mathsf{N}] \\ a \in \mathsf{N} \\ Q(0)\ true \\ Q(\mathsf{succ}(x))\ true\ \ [x \in \mathsf{N},\ Q(x)\ true] \end{array}}{Q(a)\ true}$$

For the justification of the rule, assume that we have expressions d and e and know the judgements $a \in \mathsf{N}$, $d \in Q^\star(0)$ and $e(x, y) \in Q^\star(\mathsf{succ}(x))\ \ [x \in \mathsf{N},\ y \in Q^\star(x)]$ as explained in the basic set theory. By the N-elimination rule in the basic set theory, we get $\mathsf{natrec}(a, d, e) \in Q^\star(a)$. So $Q(a)$ is true in the subset theory as desired.

18.6.4 Cartesian product of a family of sets

An element f in a cartesian product of a family B of sets on a set A in the subset theory is an element in the cartesian product $(\Pi x \in A')B'(x)$ in the basic theory, such that when it is applied on an element a in A' such that $A''(a)$ is true, it gives an element in $B'(a)$ such that $B''(a, \mathsf{apply}(f, a))$ is true:

$((\Pi x \in A)B(x))'$ is defined to be the set $(\Pi x \in A')B(x)'$ and
$((\Pi x \in A)B(x))''$ is defined to be the propositional function

$$(z)((\Pi x \in A')(A''(x) \rightarrow B(x)''(\mathsf{apply}(z, x))))$$

on the set $(\Pi x \in A')B(x)'$.

The rule we have to add is

Π – elimination for propositions

$$\frac{f \in (\Pi x \in A)B(x) \qquad Q(\lambda(y))\ \mathit{true}\ \ [y(x) \in B(x)\ \ [x \in A]]}{Q(f)\ \mathit{true}}$$

In this rule we must use a higher order assumption, which we have not discussed for the subset theory. But we leave out the details of extending our semantics to judgements depending on higher order assumption. Note that the elimination rule for Π involving apply cannot be used to obtain an induction principle for propositions over a Π-type.

We can also justify the equality rule

Π-subset – equality

$$\frac{\begin{array}{l} A\ \mathit{set} \\ B(x)\ \mathit{set}\ \ [x \in A] \\ P(x)\ \mathit{prop}\ \ [x \in A] \\ Q(x, y)\ \mathit{prop}\ \ [x \in A,\ y \in B(x)] \end{array}}{\begin{array}{l} (\Pi x \in \{u \in A \mid P(u)\})\{v \in B(x) \mid Q(x, v)\} \ = \\ \{z \in (\Pi x \in A)B(x) \mid (\forall u \in A)(P(u) \supset Q(u, \mathsf{apply}(z, x)))\} \end{array}}$$

18.6.5 Disjoint union of two sets

The semantics of a disjoint union of two sets is the following:

> $(A+B)'$ is defined to be the set $A'+B'$ and $(A+B)''$ is defined to be the propositional function
>
> $$(z)((\exists x \in A')(A''(x) \ \times \mathsf{Id}(A', z, \mathsf{inl}(x))) + (\exists y \in B')(B''(y) \ \times \ \mathsf{Id}(B', z, \mathsf{inr}(y))))$$
>
> on the set $A' + B'$.

The elimination rule we have to add is

+ – elimination for propositions

$$\frac{c \in A+B \qquad Q(\mathsf{inl}(x)) \ \mathit{true} \ \ [x \in A] \qquad Q(\mathsf{inr}(y)) \ \mathit{true} \ \ [y \in B]}{Q(c) \ \mathit{true}}$$

We also have the equality rule

+-subset – equality

$$\frac{A \ \mathit{set} \qquad P(x) \ \mathit{prop} \ \ [x \in A] \qquad Q(y) \ \mathit{prop} \ \ [y \in B]}{\begin{array}{l}\{x \in A \mid P(x)\} \ + \ \{y \in B \mid Q(y)\} \ = \\ \{z \in A+B \mid (\exists x \in A)(P(x) \ \& \ z =_A \mathsf{inl}(x)) \ \vee \\ (\exists y \in B)(Q(y) \ \& \ z =_B \mathsf{inr}(y))\}\end{array}}$$

18.6.6 Disjoint union of a family of sets

The semantics of a disjoint union of a family of sets is given by:

> $((\Sigma x \in A)B(x))'$ is defined to be $(\Sigma x \in A')B'(x)$ and $((\Sigma x \in A)B(x))''$ is defined to be the propositional function
>
> $$(z)(A''(\mathit{fst}(z)) \times B(\mathit{fst}(z))''(\mathit{snd}(z)))$$
>
> on $(\Sigma x \in A')B'(x)$.

We have to add the rule

Σ – elimination for propositions

$$\frac{c \in \Sigma(A,B) \qquad Q(\langle x,y\rangle) \ \mathit{true} \ \ [x \in A, \ y \in B(x)]}{Q(c) \ \mathit{true}}$$

We can also justify the equality rule

Σ-subset – equality

$$\frac{\begin{array}{l}A \ \mathit{set} \\ B(x) \ \mathit{set} \ \ [x \in A] \\ P(x) \ \mathit{prop} \ \ [x \in A] \\ Q(x,y) \ \mathit{prop} \ \ [x \in A, \ y \in B(x)]\end{array}}{\begin{array}{l}(\Sigma x \in \{u \in A \mid P(u)\})\{v \in B(x) \mid Q(x,v)\} \ = \\ \{z \in (\Sigma x \in A)B(x) \mid P(\mathit{fst}(z)) \times Q(\mathit{fst}(z), \mathit{snd}(z))\}\end{array}}$$

18.6.7 Lists

Let A be a set in the subset theory. The base set $\mathsf{List}(A)'$ is then put equal to the set $\mathsf{List}(A')$ in the basic set theory. The propositional function $\mathsf{List}(A)''$ on $\mathsf{List}(A')$ must satisfy that $\mathsf{List}(A)''(\mathsf{nil})$ is true and, for $a \in A'$ and $b \in \mathsf{List}(A')$, that $\mathsf{List}(A)''(\mathsf{cons}(a,b))$ is true if $A''(a)$ and $\mathsf{List}(A)''(b)$ both are true. So $\mathsf{List}(A)''$ must be defined by a set valued recursion. The only way we can do this is by using the universe U and we then obtain the following semantics for $\mathsf{List}(A)$:

> $\mathsf{List}(A)'$ is defined to be the set $\mathsf{List}(A')$ and $\mathsf{List}(A'')$ is defined to be the propositional function
>
> $$(z)(\mathsf{Set}(\mathsf{listrec}(z, \widehat{\mathsf{T}}, (x,y,u)(\widehat{A''(x)}\ \widehat{\times}\ u))))$$

By the notation $\widehat{C}$ we mean the code for the small set C. The code $\widehat{C}$ can be defined by induction on the formation of the set C.

The use of U when giving semantics to $\mathsf{List}(A)$ is not satisfactory since it cannot be extended to subsets involving a universe for subsets. We will discuss this problem in the section on the universe in the subset theory and suggest other ways of giving semantics to $\mathsf{List}(A)$.

18.6.8 Well-orderings

As for lists, we must use the universe when giving semantics for well-orderings:

> $((\mathsf{W}x \in A)B(x))'$ is defined to be the set $(\mathsf{W}x \in A')B'(x)$ and $((\mathsf{W}x \in A)B(x))''$ is defined to be the propositional function
>
> $$(z)(\mathsf{Set}(\mathsf{wrec}(z, (x,y,u)(\widehat{A''(x)}\ \widehat{\times}\ (\widehat{\Pi} v \in \widehat{B(x)}')(\widehat{B(x)}''(v) \widehat{\to} u(v))))))$$

18.7 Subsets with a universe

We will now introduce a subset U reflecting the subsets introduced so far and a subset P reflecting the propositions we have introduced. We must then extend the syntax by adding constants

$$\widehat{\&},\ \widehat{\vee},\ \widehat{\supset},\ \widehat{\perp}\ \widehat{\exists},\ \widehat{\forall} \text{ and } \widehat{\mathsf{ID}}$$

which code the propositional constants and a constant Prop for the function which decodes an element in P.

We first give the rules and then indicate how an interpretation of the subset theory extended with U and P can be given in the basic set theory, using the universe U of the basic set theory.

P – formation

$$\mathsf{P}\ prop$$

P – introduction 1

$$\frac{P \in \mathsf{P} \qquad Q \in \mathsf{P}}{P \widehat{\&} Q \in \mathsf{P}}$$

Prop – introduction 1

$$\frac{P \in \mathsf{P} \qquad Q \in \mathsf{P}}{\mathsf{Prop}(P \widehat{\&} Q) \Leftrightarrow (\mathsf{Prop}(P) \& \mathsf{Prop}(Q))}$$

P – introduction 2

$$\frac{P \in \mathsf{P} \qquad Q \in \mathsf{P}}{P \widehat{\vee} Q \in \mathsf{P}}$$

Prop – introduction 2

$$\frac{P \in \mathsf{P} \qquad Q \in \mathsf{P}}{\mathsf{Prop}(P \widehat{\vee} Q) \Leftrightarrow (\mathsf{Prop}(P) \vee \mathsf{Prop}(Q))}$$

P – introduction 3

$$\frac{P \in \mathsf{P} \qquad Q \in \mathsf{P}}{P \widehat{\supset} Q \in \mathsf{P}}$$

Prop – introduction 3

$$\frac{P \in \mathsf{P} \qquad Q \in \mathsf{P}}{\mathsf{Prop}(P \widehat{\supset} Q) \Leftrightarrow (\mathsf{Prop}(P) \supset \mathsf{Prop}(Q))}$$

P – introduction 4

$$\widehat{\perp} \in \mathsf{P}$$

Prop – introduction 4

$$\mathsf{Prop}(\widehat{\perp}) \Leftrightarrow \perp$$

P – introduction 5

$$\frac{A \in \mathsf{U} \qquad P(x) \in \mathsf{P}\ \ [x \in \mathsf{Set}(A)]}{\widehat{\forall}(A, P) \in \mathsf{P}}$$

Prop – introduction 5

$$\frac{A \in \mathsf{U} \qquad P(x) \in \mathsf{P}\ \ [x \in \mathsf{Set}(A)]}{\mathsf{Prop}(\widehat{\forall}(A, P)) \Leftrightarrow (\forall x \in \mathsf{Set}(A))\mathsf{Prop}(P(x))}$$

P – introduction 6

$$\frac{A \in \mathsf{U} \qquad P(x) \in \mathsf{P}\ \ [x \in \mathsf{Set}(A)]}{\widehat{\exists}(A, P) \in \mathsf{P}}$$

Prop – introduction 6

$$\frac{A \in \mathsf{U} \qquad P(x) \in \mathsf{P}\ \ [x \in \mathsf{Set}(A)]}{\mathsf{Prop}(\widehat{\exists}(A, P)) \Leftrightarrow (\exists x \in \mathsf{Set}(A))\mathsf{Prop}(P(x))}$$

P – introduction 7

$$\frac{A \in \mathsf{U} \qquad a \in \mathsf{Set}(A) \qquad b \in \mathsf{Set}(A)}{\widehat{\mathsf{ID}}(A, a, b) \in \mathsf{P}}$$

Prop – introduction 7

$$\frac{A \in \mathsf{U} \qquad a \in \mathsf{Set}(A) \qquad b \in \mathsf{Set}(A)}{\widehat{\mathsf{ID}}(A, a, b) \Leftrightarrow \mathsf{ID}(A, a, b)}$$

To the rules for U in the basic set theory, excluding the elimination rule, we must add rules reflecting subsets introduced by comprehension.

U – introduction 9

$$\frac{A \in \mathsf{U} \qquad P(x) \in \mathsf{P} \;\; [x \in \mathsf{Set}(A)]}{\widehat{\{|\}}(A, P) \in \mathsf{U}}$$

Set – introduction 9

$$\frac{A \in \mathsf{U} \qquad P(x) \in \mathsf{P} \;\; [x \in \mathsf{Set}(A)]}{\mathsf{Set}(\widehat{\{|\}}(A, P)) = \{x \in \mathsf{Set}(A) \mid \mathsf{Prop}(P(x))\}}$$

We will now indicate how the subset theory with U and P can be interpreted in the basic set theory. The interpretation of U will then reflect the interpretation we already have given of the subset theory without a universe. This leads to the following definition of U':

$$\mathsf{U}' \quad \equiv \quad (\Sigma x' \in \mathsf{U})(\mathsf{Set}(x') \to \mathsf{U})$$

where U in the definiens is the universe in the basic set theory. U'' is trivially defined by

$$\mathsf{U}'' \quad \equiv \quad (z)\mathsf{T}$$

In the interpretation of the subset theory without a universe, the elements of a set are interpreted by themselves. However, this is no longer possible when having a universe since an element in U' is a pair, reflecting that a set A in the subset theory is interpreted as a set A' in the basic set theory together with a propositional function A'' on A'. So if $a \in \mathsf{U}$ in the subset theory, then we cannot have $a \in \mathsf{U}'$; instead we must also interpret a as a pair, which we will denote by a'.

The interpretation of Set is then given by

$$\begin{aligned} \mathsf{Set}(a)' \quad &\equiv \quad \mathsf{Set}(\mathit{fst}(a')) \\ \mathsf{Set}(a)''(z) \quad &\equiv \quad \mathsf{Set}(\mathsf{apply}(\mathit{snd}(a'), z)) \end{aligned}$$

Since propositions are interpreted as sets, the interpretation of P must reflect this:

$$\begin{aligned} \mathsf{P}' \quad &\equiv \quad \mathsf{U} \\ \mathsf{P}''(z) \quad &\equiv \quad \mathsf{T} \end{aligned}$$

The interpretation of Prop is then given by

$$\begin{array}{rcl} \mathsf{Prop}(a)' & \equiv & \mathsf{Set}(a') \\ \mathsf{Prop}(a)''(z) & \equiv & \mathsf{T} \end{array}$$

We must now also define the mapping $'$ on elements. For codes of sets formed by comprehension, we have

$$\widehat{\{|\}}(a,b)' \quad \equiv \quad \langle \mathit{fst}(a'), \lambda z.(\mathsf{apply}(\mathit{snd}(a'), z)\widehat{\&}b'(z)\rangle$$

The mapping $'$ is defined in a similar way for elements coding sets of the other forms, reflecting the interpretation of the corresponding set. The mapping $'$ will commute with all the constants for elements which are not codes in U. So, for instance, $\mathsf{pair}(a,b)' \equiv \mathsf{pair}(a',b')$.

When defining $'$ on codes for lists and well-orderings there is, however, a problem since the interpretation of these types is using the universe. One way of solving this problem would be to add an infinite sequence

$$\mathsf{U}_1, \ldots, \mathsf{U}_n, \ldots$$

of universes so that when interpreting U_n one could use U_{n+1}. Another way, discussed in [102], would be to extend the basic set theory with the possibility of defining sets directly by recursion, not using the universe. Defining sets by induction on lists, we would have to extend the syntax with a new constant $\mathsf{Listrec}$ of arity $\mathbf{0}{\otimes}\mathbf{0}{\otimes}(\mathbf{0}{\otimes}\mathbf{0}{\otimes}\mathbf{0}{\twoheadrightarrow}\mathbf{0}){\twoheadrightarrow}\mathbf{0}$ and add the rules

Listrec – formation

$$\frac{\begin{array}{l} l \in \mathsf{List}(A) \\ C \ set \\ E(x,y,Z) \ set \ \ [x \in A, \ y \in \mathsf{List}(A), \ Z \ set] \end{array}}{\mathsf{Listrec}(l,C,E) \ set}$$

Listrec – equality 1

$$\frac{C \ set \quad E(x,y,Z) \ set \ \ [x \in A, \ y \in \mathsf{List}(A), \ Z \ set]}{\mathsf{Listrec}(\mathsf{nil},C,E) = C}$$

Listrec – equality 2

$$\frac{\begin{array}{l} l \in \mathsf{List}(A) \\ C \ set \\ E(x,y,Z) \ set \ \ [x \in A, \ y \in \mathsf{List}(A), \ Z \ set] \end{array}}{\mathsf{Listrec}(a.l,C,E) = E(a,l,\mathsf{Listrec}(l,C,E))}$$

We can now give the semantics for lists in the subset theory without using a universe:

> $\mathsf{List}(A)'$ is defined to be the set $\mathsf{List}(A')$ and $\mathsf{List}(A)''$ is defined to be the propositional function $(z)((\mathsf{Listrec}(z,\mathsf{T},(x,y,Z)(A''(x) \ \times \ Z))))$

Part III

Monomorphic sets

Chapter 19

Types

In the previous chapters, we have defined a collection of sets and set forming operations and presented proof rules for these sets. We have introduced the constants for each set and then presented the proof rules in a natural deduction style. Another way of introducing sets is to use the more primitive notion of type. Intuitively, a type is a collection of objects together with an equivalence relation. Examples of types are the type of sets, the type of elements in a set, the type of propositions, the type of set-valued functions over a given set, and the type of predicates over a given set.

In this chapter we will describe a theory of types and show how it can be used to present a theory of sets. We will get possibilities of using variables ranging over sets and higher order objects. The possibility of abstracting over these kind of variables is essential for structuring big programs and proofs. It also gives possibilities to use more elegant formulations of the elimination rules for the Π-set and the well-orderings. The theory of types can also be used as a logical framework [48] in which it is possible to formalize different logics. It can also be used as a theory of expressions where the types replaces the arities; hence, we will in this chapter not rely on the theory of expressions developed in chapter 3.

If one looks in a text book on logic like, for instance, Kleene's Introduction to Metamathematics, one hardly finds any completely formal derivations. In general, the derivations depend on metavariables ranging over formulas. For instance, in the formal derivation

$$\dfrac{\dfrac{x = y \,\&\, x = z}{x = y}}{x = y \,\&\, x = z \supset x = y}$$

we can replace the formulas $x = y$ and $x = z$ by arbitrary formulas A and B respectively thereby obtaining the schematic derivation

$$\dfrac{\dfrac{A \,\&\, B}{A}}{(A \,\&\, B) \supset A}$$

which no longer is a formal derivation in predicate logic.

Most of the derivations in this book are also made under some general assumptions like "Let A be a set and $B(x)$ a family of sets over A". When implementing type theory on a computer these kinds of assumptions have to be made formal. In the Nuprl-system [25] this is made by using universes; for instance the assumption

$$\text{“Let } X \text{ be a set”}$$

is translated into the formal assumption

$$X \in \mathsf{U}$$

However, this does not really capture the assumption that X is an arbitrary set, because U is only the set of small sets which has a fixed inductive definition. What we really want to assume is that X is an arbitrary set, that is, something satisfying the semantical requirements of being a set. In particular, X may in the future be interpreted as some set which we have not yet defined. It may also be interpreted as some set involving U and then it cannot be a small set.

19.1 Types and objects

We will now extend type theory so that assumptions like "X is a set" can be made. We will do that by introducing an even more basic concept than that of a set, namely the notion of type. Intuitively, a type is a collection of objects together with an equivalence relation.

What does it mean that something is a type? To know that A is a type is to know what it means to be an object of the type, as well as what it means for two objects to be the same. The identity between objects must be an equivalence relation and it must be decidable. The requirement of decidability of identity comes from the general requirement of decidability of the new forms of judgements that we are introducing in this chapter. In these judgements everything is there which is needed to be convinced of them: They carry their own proof.

As an example of a type, we will later define the type *Set* whose objects are monomorphic sets by explaining what it means to be a set as well as what it means for two sets to be the same.

We will write

$$A \ \mathit{type}$$

for the judgement that A is a type. That a is an object of type A is written

$$a : A$$

and that a and b are the same object of type A will be written

$$a = b : A$$

and, finally, that two types A and B are identical will be written

$$A = B$$

What does it mean for two types to be the same? Two types are the same if an object of one type is also an object of the other type and identical objects of the one type are identical objects of the other type.

19.2 The types of sets and elements

The type *Set* which contains (monomorphic) sets as objects is explained by explaining what a set is and when two sets are identical. To know a set A is to know how the canonical elements of A are formed and when two canonical elements are identical. Two sets are identical if a canonical element of one set is a canonical element of the other set and if two identical canonical elements in one set also are identical in the other set.

Hence, we have the axiom

Set formation

$$\mathit{Set}\ \mathit{type}$$

Notice that this explanation of what the type *Set* is, is totally open. We have not exhausted the possibilities of defining new sets. This is in contrast with the set U, whose canonical elements are codings of a fixed number of set constructing operations. A set is always an inductive structure, we know that a canonical element in it has been formed according to one of its introduction rules.

If A is a set, then $El(A)$ is a type. It is the type whose objects are the elements of A. We know that a is an object in $El(A)$ if we know that the value of a is a canonical element of A. Two objects in $El(A)$ are identical if their values are identical canonical elements in A. So we have the rules

El-formation

$$\frac{A : \mathit{Set}}{El(A)\ \mathit{type}} \qquad \frac{A = B : \mathit{Set}}{El(A) = El(B)}$$

We will use the abbreviations

$$\begin{array}{rcl} A\ \mathit{set} & \equiv & A : \mathit{Set} \\ a \in A & \equiv & a : El(A) \end{array}$$

in accordance with the earlier used notation.

19.3 Families of types

In much the same way as the notion of set is extended to families of sets, we will now introduce families of types.

A *context* is a sequence

$$x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n$$

such that

- A_1 is a type,
- $A_2[x_1 := a_1]$ is a type for an arbitrary object a_1 of type A_1,

$\vdots$

- $A_n[x_1 := a_1][x_2 := a_2]\cdots[x_{n-1} := a_{n-1}]$ is a type for arbitrary objects $a_1,\ a_2,\ \ldots,a_{n-1}$ of types

  $$A_1,\ A_2[x_1 := a_1],\ \ldots,\ A_{n-1}[x_1 := a_1][x_2 := a_2]\cdots[x_{n-2} := a_{n-2}]$$

  respectively.

That A is a family of types in the context

$$x_1 : A_1,\ x_2 : A_2, \ldots,\ x_n : A_n,$$

which we formally write

$$A\ \mathit{type}\ \ [x_1 : A_1, x_2 : A_2,\ \ldots, x_n : A_n]$$

means that

> $A[x_1 := a_1][x_2 := a_2]\cdots[x_n := a_n]$ is a type for arbitrary objects $a_1,\ a_2,\ \ldots,\ a_{n-1}$ of types $A_1,\ A_2[x_1 := a_1],\ \ldots,$ $A_n[x_1 := a_1][x_2 := a_2]\cdots[x_{n-1} := a_{n-1}]$ respectively.

As for families of sets, we also require that A must be extensional in the context, that is, if

$$\begin{aligned}
a_1 &= b_1 : A_1,\\
a_2 &= b_2 : A_2[x_1 := a_1],\\
&\vdots\\
a_n &= b_n : A_n[x_1 := a_1][x_2 := a_2]\cdots[x_{n-1} := a_{n-1}]
\end{aligned}$$

then it follows from

$$A\ \mathit{type}\ \ [x_1 : A_1,\ \ldots,\ x_n : A_n]$$

that

$$A[x_1 := a_1][x_2 := a_2]\cdots[x_n := a_n] \ = \ A[x_1 := b_1][x_2 := b_2]\cdots[x_n := b_n]$$

As an example, the two rules for *El*-formation express that $\mathit{El}(X)$ is a family of types over *Set*.

The explanation of the remaining three forms of judgements:

$$\begin{gathered} A = B\\ a : A\\ a = b : A \end{gathered}$$

in the context

$$x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n$$

is done in a similar way as the first form

$$A\ \mathit{type}\ \ [x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n]$$

by reducing the explanation to the corresponding form with empty context by substituting appropriate closed expressions for the variables.

19.4 General rules

Since the identity relation on a type is required to be an equivalence relation and since two types are identical if they have the same objects and identical objects of one of the types are also identical objects of the other, we have the following identity rules.

Reflexivity

$$\frac{a : A}{a = a : A} \qquad \frac{A \ type}{A = A}$$

Symmetry

$$\frac{a = b : A}{b = a : A} \qquad \frac{A = B}{B = A}$$

Transitivity

$$\frac{a = b : A \qquad b = c : A}{a = c : A} \qquad \frac{A = B \qquad B = C}{A = C}$$

Type identity

$$\frac{a : A \qquad A = B}{a : B} \qquad \frac{a = b : A \qquad A = B}{a = b : B}$$

The explanations of families of types in a context of the form $x : A$ directly give rules for substitution:

Substitution in types

$$\frac{C \ type \ \ [x : A] \qquad a : A}{C[x := a] \ type} \qquad \frac{C \ type \ \ [x : A] \qquad a = b : A}{C[x := a] = C[x := b]}$$

Substitution in objects

$$\frac{c : C \ \ [x : A] \qquad a : A}{c[x := a] : C[x := a]} \qquad \frac{c : C \ \ [x : A] \qquad a = b : A}{c[x := a] = c[x := b] : C[x := a]}$$

Substitution in identical types

$$\frac{B = C \ \ [x : A] \qquad a : A}{B[x := a] = C[x := a]}$$

Substitution in identical objects

$$\frac{b = c : B \ \ [x : A] \qquad a : A}{b[x := a] = c[x := a] : B[x := a]}$$

These rules can in the same way as in chapter 5 be extended to general contexts of the form $x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n$ where n simultaneous substitutions are made.

19.5 Assumptions

Our main reason for introducing types is that we want the possibility to make assumptions of a more general form than $x \in A$, where A is a set. The assumptions we can now make are of the form

$$x : C$$

where C is a type. To be more formal, we have the rule

Assumption

$$\frac{C\ \mathit{type}}{x : C\ \ [x : C]}$$

The premise C *type* in this rule may depend on a nonempty context, but as usual in natural deduction, we only explicitly show that part of the context which is changed by the rule. By using the axiom that *Set* is a type we can now make the assumption that X is an arbitrary set:

$$\frac{\mathit{Set\ type}}{X : \mathit{Set}\ \ [X : \mathit{Set}]}$$

which, by the definition above, we can also write

$$\frac{\mathit{Set\ type}}{X\ \mathit{set}\ \ [X\ \mathit{set}]}$$

Assumptions in set theory without types are always of the form

$$x \in A$$

where A is a set and they can now be obtained as special cases of assumptions in the theory of types by the following derivation:

$$\frac{\dfrac{A\ \mathit{set}}{\mathit{El}(A)\ \mathit{type}}}{x : \mathit{El}(A)\ \ [x : \mathit{El}(A)]}$$

Using our notational conventions, we can write the conclusion of this derivation

$$x \in A\ \ [x \in A]$$

Note that this derivation is not formal because of the occurrence of the metavariable A, which denotes an arbitrary set. It is now possible to make the derivation completely formal by making an assumption of the form X *set*:

$$\frac{\dfrac{\dfrac{\mathit{Set\ type}}{X : \mathit{Set}\ \ [X : \mathit{Set}]}}{\mathit{El}(X)\ \mathit{type}\ \ [X : \mathit{Set}]}}{x : \mathit{El}(X)\ \ [X : \mathit{Set},\ x : \mathit{El}(X)]}$$

We can also write the conclusion of the derivation more in the style of previous chapters:

$$x \in X\ \ [X\ \mathit{set},\ x \in X]$$

19.6 Function types

We have not yet defined enough types to turn an assumption like

Let A be a set and B a family of sets over A

into a formal assumption. To do this we need function types. If A is a type and B is a family of types for $x : A$ then $(x : A)B$ is the type which contains functions from A to B as objects. All free occurrences of x in B become bound in $(x : A)B$.

Fun formation

$$\frac{A \textit{ type} \qquad B \textit{ type } [x : A]}{(x : A)B \textit{ type}} \qquad\qquad \frac{A_1 = A_2 \qquad B_1 = B_2 \ [x : A_1]}{(x : A_1)B_1 = (x : A_2)B_2}$$

To define the type of functions $(x : A)B$ we must explain what it means to be a function and when two functions are the same. To know that an object c is in the type $(x : A)B$ means that we know that when we apply it to an object a in A we get an object $c(a)$ in $B[x := a]$ and that we get identical objects in $B[x := a_1]$ when we apply it to identical objects a_1 and a_2 in A. Two objects c_1 and c_2 in $(x : A)B$ are identical if $c_1(a) = c_2(a) : B[x := a]$ for an arbitrary a in A. Hence, we have the following two rules

Application

$$\frac{c : (x : A)B \qquad a : A}{c(a) : B[x := a]} \qquad\qquad \frac{c_1 = c_2 : (x : A)B \qquad a = b : A}{c_1(a_1) = c_2(a_2) : B[x := a]}$$

Functions can be formed by abstraction, if $b : B \ [x : A]$ then $(x)b$ is an object in $(x : A)B$. All free occurences of x in b become bound in $(x)b$.

Abstraction

$$\frac{b : B \ [x : A]}{(x)b : (x : A)B}$$

The abstraction is explained by the ordinary β-rule which defines what it means to apply an abstraction to an object in A.

β – rule

$$\frac{a : A \qquad b : B \ [x : A]}{((x)b)(a) = b[x := a] : B[x := a]}$$

It is possible to justify the following rules:

ξ – rule

$$\frac{b_1 = b_2 : B \ [x : A]}{(x)b_1 = (x)b_2 : (x : A)B}$$

α – rule

$$\frac{b : B \ [x : A]}{(x)b = (y)(b[x := y]) : (x : A)B}$$

$$y \text{ must not occur free in } b$$

η – rule

$$\frac{c : (x : A)B}{(x)(c(x)) = c : (x : A)B}$$

$$x \text{ must not occur free in } c$$

In a context we will often write $x \in A$ instead of $x : El(A)$ and $y(x) \in B(x)\ \ [x \in A]$ instead of $y : (x : El(A))El(B(x))$.

Example. Translating between hypothetical judgements and functions

From the judgement

$$a : A\ \ [x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n]$$

we can derive, by repeated abstractions,

$$(x_1, \ldots, x_n)a : (x_1 : A_1)(x_2 : A_2) \cdots (x_n : A_n)\, A$$

We can go in the other direction by repeated applications of the rules Assumption and Application.

Instead of

$$(x : A)(y : B)C$$

we will often write

$$(x : A, y : B)C$$

and, similarly, repeated application will be written $f(a, b)$ instead of $f(a)(b)$ and repeated abstraction will be written $(x, y)e$ instead of $(x)(y)e$. When B does not depend on the variable x, we will use the following definition:

$$(A)B \equiv (x : A)B$$

Example. Looking at a family of sets as an object of a type

We can now formalize an assumption of the form "Let $Y(x)$ be a family of sets over a set X" by the following derivation:

By *Set*-formation we have

$$Set\ type$$

and, hence, we can use Assumption to obtain

$$X : Set\ \ [X : Set]$$

from which we get, by *El*-formation,

$$El(X)\ type\ \ [X : Set]$$

We can now use Assumption to get

$$x : El(X) \;\; [X : Set,\; x : El(X)]$$

By applying Fun formation we get

$$(x : El(X))\; Set\; type \;\; [X : Set]$$

The objects in the type $(x : El(X))\; Set$ are set-valued functions indexed by elements in X. We can now use Assumption to get

$$Y : (x : El(X))Set \;\; [X : Set,\; Y : (x : El(X))Set]$$

Hence, by Assumption and application,

$$Y(x) : Set \;\; [X : Set,\; Y : (x : El(X))Set,\; x : El(X)]$$

Using our notational conventions, this may also be written

$$Y(x)\; set \;\; [X\; set,\; Y(x)\; set[x \in X],\; x \in X]$$

and we may read this

> Assume that $Y(x)$ is a set under the assumptions that X is a set and $x \in X$.

Chapter 20

Defining sets in terms of types

We will in this chapter, very briefly, describe the objects in the type *Set*, thereby illustrating how the theory of types can be used to formulate a theory of sets.

We will introduce the different sets by defining constants of different types and asserting equalities between elements in the sets. The sets we get are different from the one previously presented. The major difference is that they are *monomorphic*, which means that all constants contain explicit information about which sets the rest of the arguments belong to. In the polymorphic set theory presented in the previous chapters, the constant apply, for example, takes two arguments, a function from A to B and an element in A. In the monomorphic version, apply will take four arguments. First the two sets, A and B, then the function in $A \to B$, and finally the element in A. One advantage with a monomorphic version is that all important information about the validity of a judgement is contained in the judgement itself. Given a judgement, it is possible to reconstruct a derivation of the judgement. The disadvantage, of course, is that programs will contain a lot of information which is irrelevant for the computation.

Another difference between the two type theory versions is that all functional constants introduced in this chapter are curried and written in prefix form. The reason is that we did only introduce a function type in the chapter about types. The selectors also take their arguments in a different order.

We may define a stripping function on the expressions in the monomorphic theory which takes away the set information and we would then obtain expressions of the polymorphic theory. A derivation in the monomorphic theory is, after the stripping, a correct derivation in the polymorphic theory; this can easily be shown by induction on the length of a derivation in the monomorphic theory since each rule in the monomorphic theory becomes a rule in the polymorphic theory after stripping. Nevertheless, the polymorphic theory is fundamentally different from the monomorphic theory; in Salvesen [91] it is shown that there are derivable judgements in the polymorphic theory which cannot come from any derivable judgement in the monomorphic theory by stripping.

If we declare the constants for the extensional equality Eq in the theory of types, we will not be able to derive the strong Eq-elimination rule. So this equality does not fit into the monomorphic theory of sets.

20.1 Π sets

The notation $(A)B$ is used instead of $(x : A)B$ whenever B does not contain any free occurrences of x. We will write $(x_1 : A_1, \ldots , x_n : A_n)B$ instead of $(x_1 : A_1) \ldots (x_n : A_n)B$ and $b(a_1, \ldots , a_n)$ instead of $b(a_1) \ldots (a_n)$ in order to increase the readability.

The Π-sets are introduced by introducing the following constants.

$$\begin{aligned}
\Pi \quad &: \quad (X : Set, (El(X))Set)\, Set \\
\lambda \quad &: \quad (X : Set, Y : (El(X))Set, (x : El(X))El(Y(x))) \\
&\qquad El(\Pi(X, Y)) \\
\mathsf{apply} \quad &: \quad (X : Set, Y : (El(X))Set, El(\Pi(X, Y)), x : El(X)) \\
&\qquad El(Y(x))
\end{aligned}$$

and asserting the equality:

$$\mathsf{apply}(A, B, \lambda(A, B, b), a) = b(a) : El(B(a))$$

where

$$\begin{aligned}
A \quad &: \quad Set \\
B \quad &: \quad (El(A))\, Set \\
a \quad &: \quad El(A) \\
b \quad &: \quad (x : El(A))\, El(B(x))
\end{aligned}$$

An alternative notation for the function type is $x : A \twoheadrightarrow B$. The type of the constants for Π is then written as follows:

$$\begin{aligned}
\Pi \quad &: \quad X : Set \twoheadrightarrow (El(X) \twoheadrightarrow Set) \twoheadrightarrow Set \\
\lambda \quad &: \quad X : Set \twoheadrightarrow (Y : El(X) \twoheadrightarrow Set) \twoheadrightarrow \\
&\qquad (x : El(X) \twoheadrightarrow El(Y(x))) \twoheadrightarrow \\
&\qquad El(\Pi(X, Y)) \\
\mathsf{apply} \quad &: \quad X : Set \twoheadrightarrow (Y : El(X) \twoheadrightarrow Set) \twoheadrightarrow \\
&\qquad (El(\Pi(X, Y))) \twoheadrightarrow \\
&\qquad (x : El(X)) \twoheadrightarrow \\
&\qquad El(Y(x))
\end{aligned}$$

We get the ordinary function set by asserting the equality

$$A \to B = \Pi(A, (x)B)) : Set \quad [A : Set, B : Set]$$

In a more conventional formulation the typing of the constants correspond to the following derivable inference rules (compare with the formation, introduction and elimination rules in chapter 7):

$$\frac{X : Set \qquad Y(x) : Set \;\; [x : El(X)]}{\Pi(X, Y) : Set}$$

$$\frac{X : Set \qquad Y(x) : Set \ \ [x : El(X)] \qquad b(x) : El(Y(x)) \ \ [x : El(X)]}{\lambda(X, Y, b) : El(\Pi(X, Y))}$$

$$\frac{X : Set \qquad Y(x) : Set \ \ [x : El(X)] \qquad c : El(\Pi(X, Y)) \qquad a : El(X)}{\mathsf{apply}(X, Y, c, a) : El(Y(a))}$$

and the equality corresponds to the rule (compare with the equality rule)

$$\frac{\begin{array}{l} X : Set \\ Y(x) : Set \ \ [x : El(X)] \\ b(x) : El(Y(x)) \ \ [x : El(X)] \\ a : El(X) \end{array}}{\mathsf{apply}(X, Y, \lambda(X, Y, b), a) = b(a) : El(Y(a))}$$

20.2 Σ sets

We get the Σ sets by declaring the constants:

$$\begin{array}{rcl} \Sigma & : & (X\!:\!Set, (El(X))Set)\,Set \\ \mathsf{pair} & : & (X\!:\!Set, Y\!:\!(El(X))Set, x\!:\!El(X), El(Y(x)))\,El(\Sigma(X, Y)) \\ \mathsf{split} & : & (X\!:\!Set, Y\!:\!(El(X))Set, Z\!:\!(El(\Sigma(X, Y)))Set, \\ & & \quad (x\!:\!El(X), y\!:\!El(Y(x)))El(Z(\mathsf{pair}(X, Y, x, y))), \\ & & \quad\quad w\!:\!El(\Sigma(X, Y))) \\ & & \quad\quad\quad El(Z(w)) \end{array}$$

and asserting the equality:

$$\mathsf{split}(A, B, C, d, \mathsf{pair}(A, B, a, b)) = d(a, b) : El(C(\mathsf{pair}(A, B, a, b)))$$

where

$$\begin{array}{rcl} A & : & Set \\ B & : & (El(A))\,Set \\ C & : & (El(\Sigma(A, B)))\,Set \\ d & : & (x\!:\!El(A), y\!:\!El(B(x)))\,El(C(\mathsf{pair}(A, B, a, b))) \\ a & : & El(A) \\ b & : & El(B(a)) \end{array}$$

The usual cartesian product is defined by

$$A \times B = \Sigma(A, (x)B) : Set \ \ [A : Set, B : Set]$$

20.3 Disjoint union

The disjoint unions are introduced by declaring the constants:

$$\begin{array}{rcl}
+ & : & (\mathit{Set}, \mathit{Set})\,\mathit{Set} \\
\mathsf{inl} & : & (X:\mathit{Set}, Y:\mathit{Set}, \mathit{El}(X))\,+(X,Y) \\
\mathsf{inr} & : & (X:\mathit{Set}, Y:\mathit{Set}, \mathit{El}(Y))\,+(X,Y) \\
\mathsf{when} & : & (X:\mathit{Set}, Y:\mathit{Set}, Z:(\mathit{El}(+(X,Y)))\mathit{Set}, \\
& & \quad (x:\mathit{El}(X))\mathit{El}(Z(\mathsf{inl}(X,Y,x))), \\
& & \quad\quad (y:\mathit{El}(Y))\mathit{El}(Z(\mathsf{inr}(X,Y,y))), \\
& & \quad\quad\quad z:\mathit{El}(+(X,Y))) \\
& & \quad\quad\quad\quad \mathit{El}(Z(z))
\end{array}$$

and the equalities

$$\begin{array}{r}
\mathsf{when}(A,B,C,d,e,\mathsf{inl}(A,B,a)) = d(a) : \mathit{El}(C(\mathsf{inl}(A,B,a))) \\
\mathsf{when}(A,B,C,d,e,\mathsf{inr}(A,B,b)) = e(b) : \mathit{El}(C(\mathsf{inr}(A,B,b)))
\end{array}$$

where

$$\begin{array}{rcl}
A & : & \mathit{Set} \\
B & : & \mathit{Set} \\
C & : & (\mathit{El}(+(A,B)))\,\mathit{Set} \\
d & : & (x:\mathit{El}(A))\,\mathit{El}(C(\mathsf{inl}(A,B,x))) \\
e & : & (y:\mathit{El}(B))\,\mathit{El}(C(\mathsf{inr}(A,B,y))) \\
a & : & \mathit{El}(A) \\
b & : & \mathit{El}(B)
\end{array}$$

20.4 Equality sets

The equality sets are introduced by declaring the constants:

$$\begin{array}{rcl}
\mathsf{Id} & : & (X:\mathit{Set}, \mathit{El}(X), \mathit{El}(X))\,\mathit{Set} \\
\mathsf{id} & : & (X:\mathit{Set}, x:\mathit{El}(X))\,\mathsf{Id}(X,x,x) \\
\mathsf{idpeel} & : & (X:\mathit{Set}, x:\mathit{El}(X), y:\mathit{El}(X), \\
& & \quad Z:(x:\mathit{El}(X), y:\mathit{El}(X), \mathit{El}(\mathsf{Id}(X,x,y)))\,\mathit{Set}, \\
& & \quad\quad (z:\mathit{El}(X))\,\mathit{El}(Z(z,z,\mathsf{id}(X,z))), \\
& & \quad\quad\quad u:\mathit{El}(\mathsf{Id}(X,x,y))) \\
& & \quad\quad\quad\quad \mathit{El}(Z(x,y,u))
\end{array}$$

and the equality

$$\mathsf{idpeel}(A,a,b,C,d,\mathsf{id}(A,a)) = d(a) : \mathit{El}(C(a,a,\mathsf{id}(A,a)))$$

where

$$\begin{array}{rcl}
A & : & \mathit{Set} \\
a & : & \mathit{El}(A) \\
b & : & \mathit{El}(A) \\
C & : & (x:\mathit{El}(A), y:\mathit{El}(A), \mathit{El}(\mathsf{Id}(A,x,y)))\,\mathit{Set} \\
d & : & (x:\mathit{El}(A))\,\mathit{El}(C(x,x,\mathsf{id}(A,x)))
\end{array}$$

20.5 Finite sets

We introduce the empty set and the one element set as examples of finite sets. The empty set is introduced by declaring the constants:

$$\begin{array}{rcl} \{\} & : & Set \\ \mathsf{case}_{\{\}} & : & ((Z:El(\{\}))Set, x:El(\{\}))\, El(Z(x)) \end{array}$$

The one element set is introduced by declaring the constants:

$$\begin{array}{rcl} \mathsf{T} & : & Set \\ \mathsf{tt} & : & El(\mathsf{T})) \\ \mathsf{case}_{\mathsf{T}} & : & (Z:(El(\mathsf{T}))Set, El(Z(\mathsf{tt}))\,, x:El(\mathsf{T}))\, El(Z(x)) \end{array}$$

and the equality

$$\mathsf{case}_{\mathsf{T}}(C, b, \mathsf{tt}) = b(\mathsf{tt}) : El(C(\mathsf{tt}))$$

where $C : (El(\mathsf{T}))Set$ and $b : El(C(\mathsf{tt}))$.

20.6 Natural numbers

The set of natural numbers is introduced by declaring the constants:

$$\begin{array}{rcl} \mathsf{N} & : & Set \\ \mathsf{0} & : & El(\mathsf{N}) \\ \mathsf{succ} & : & (El(\mathsf{N}))\, El(\mathsf{N}) \\ \mathsf{natrec} & : & (Z:(El(\mathsf{N}))\, Set, \\ & & \quad El(Z(\mathsf{0}))\,, \\ & & \quad\quad (x:El(\mathsf{N})\,, El(Z(x)))\, El(Z(\mathsf{succ}(x)))\,, \\ & & \quad\quad\quad n:El(\mathsf{N})) \\ & & \quad\quad\quad\quad El(Z(n)) \end{array}$$

and the equalities

$$\begin{array}{l} \mathsf{natrec}(C, d, e, \mathsf{0}) = d : El(C(\mathsf{0})) \\ \mathsf{natrec}(C, d, e, \mathsf{succ}(a)) = e(a, \mathsf{natrec}(C, d, e, a)) : El(C(\mathsf{succ}(a))) \end{array}$$

where

$$\begin{array}{rcl} C & : & (x:El(\mathsf{N}))\, Set \\ d & : & El(C(\mathsf{0})) \\ e & : & (x:El(\mathsf{N})\,, El(C(x)))\, El(C(\mathsf{succ}(x))) \\ a & : & El(\mathsf{N}) \end{array}$$

20.7 Lists

Lists are introduced by declaring the constants:

$$
\begin{array}{rcl}
\mathsf{List} & : & (Set)\,Set \\
\mathsf{nil} & : & (X : Set)\,El(\mathsf{List}(X)) \\
\mathsf{cons} & : & (X : Set, El(X), El(\mathsf{List}(X)))\,El(\mathsf{List}(X)) \\
\mathsf{listrec} & : & (X : Set, Z : (El(\mathsf{List}(X)))\,Set, \\
& & \quad El(Z(\mathsf{nil}(X))), \\
& & \quad\quad (x : El(X), y : El(\mathsf{List}(X)), El(Z(x)))\,El(Z(\mathsf{cons}(X, x, y))), \\
& & \quad\quad\quad u : El(\mathsf{List}(X))) \\
& & \quad\quad\quad\quad El(Z(u))
\end{array}
$$

and the equalities

$$
\begin{array}{l}
\mathsf{listrec}(A, C, d, e, \mathsf{nil}(A)) = d : El(C(\mathsf{nil}(A))) \\
\quad \mathsf{listrec}(A, C, d, e, \mathsf{cons}(A, a, b)) = e(a, b, \mathsf{listrec}(A, C, d, e, b)) \\
\quad : El(C(\mathsf{cons}(A, a, b)))
\end{array}
$$

where

$$
\begin{array}{rcl}
A & : & Set \\
C & : & (x : El(\mathsf{List}(A)))\,Set \\
d & : & El(C(\mathsf{nil}(A))) \\
e & : & (x : El(X), y : El(\mathsf{List}(A))\;\; El(C(y)))\,El(C(\mathsf{cons}(A, x, y))) \\
a & : & El(X) \\
b & : & El(\mathsf{List}(X))
\end{array}
$$

Part IV

Examples

Chapter 21

Some small examples

21.1 Division by 2

In this example we give a derivation of the proposition

$$(\exists y \in \mathsf{N})(x =_{\mathsf{N}} y * 2) \vee (x =_{\mathsf{N}} y * 2 \oplus 1) \quad [x \in N] \tag{21.1}$$

and then by interpreting propositions as sets show how to obtain a program which for each natural number n computes the integral part of $n/2$. In this chapter we are using the infix symbol $\oplus$ for addition between natural numbers.

We prove (21.1) by induction on x.

Base: By definition of $*$ we have

$$0 =_{\mathsf{N}} 0 * 2$$

from which we get, by $\vee$-introduction and $\exists$-introduction,

$$(\exists y \in \mathsf{N})((0 =_{\mathsf{N}} y * 2) \vee (0 =_{\mathsf{N}} y * 2 \oplus 1))$$

Induction step: We want to prove

$$(\exists y \in \mathsf{N})((x \oplus 1 =_{\mathsf{N}} y * 2) \vee (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1))$$

from the assumptions

$$x \in \mathsf{N}, \quad (\exists y \in \mathsf{N})((x =_{\mathsf{N}} y * 2) \vee (x =_{\mathsf{N}} y * 2 \oplus 1)) \tag{21.2}$$

We will use $\exists$-elimination on (21.2) and therefore assume

$$y \in \mathsf{N}, \quad x =_{\mathsf{N}} y * 2 \vee x =_{\mathsf{N}} y * 2 \oplus 1 \tag{21.3}$$

There are two cases corresponding to the two disjuncts in (21.3):

(i) Assume

$$x =_{\mathsf{N}} y * 2 \tag{21.4}$$

By substitution we get

$$x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1$$

and by $\vee$-introduction we then get

$$(x \oplus 1 =_{\mathsf{N}} y * 2) \vee (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)$$

Hence, by $\exists$-introduction,

$$(\exists y \in \mathsf{N})((x \oplus 1 =_{\mathsf{N}} y * 2) \vee (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)) \quad (21.5)$$

(ii) Assume

$$x =_{\mathsf{N}} y * 2 \oplus 1 \quad (21.6)$$

By elementary arithmetic we get

$$x \oplus 1 =_{\mathsf{N}} (y \oplus 1) * 2$$

and by $\vee$-introduction we then get

$$(x \oplus 1 =_{\mathsf{N}} (y \oplus 1) * 2) \vee (x \oplus 1 =_{\mathsf{N}} (y \oplus 1) * 2 \oplus 1)$$

Hence, by $\exists$-introduction,

$$(\exists y \in \mathsf{N})((x \oplus 1 =_{\mathsf{N}} y * 2) \vee (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)) \quad (21.7)$$

Since we have derived (21.5) from (21.4) and (21.7) from (21.6) we can use $\vee$-elimination to obtain

$$(\exists y \in \mathsf{N})(x \oplus 1 =_{\mathsf{N}} y * 2) \vee (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1) \quad (21.8)$$

thereby discharging the assumptions (21.4) and (21.6). The proposition (21.8) depends on the assumption list (21.3) which we discharge by using $\exists$-elimination and thereby (21.1) is proved.

We will now translate this derivation using the interpretation of propositions as sets. Viewed as a set, the truth of the proposition

$$(\exists y \in \mathsf{N})((x =_{\mathsf{N}} y * 2) \vee (x =_{\mathsf{N}} y * 2 \oplus 1)) \; [x \in \mathsf{N}]$$

means that we know how to construct an element in the corresponding set; that is, we know how to construct an expression such that when we substitute a natural number n for x we get a natural number m such that

$$(n =_{\mathsf{N}} m * 2) \vee (n =_{\mathsf{N}} m * 2 \oplus 1)$$

So, the constructed element will give us a method for computing the integral part of $n/2$.

There are two possibilities when interpreting the existential quantifier in type theory: either to use the Σ set or to use a subset. Since we are interested in the program that computes the integral part of $n/2$ and not in the proof element of

$$(n =_{\mathsf{N}} m * 2) \vee (n =_{\mathsf{N}} m * 2 \oplus 1)$$

it is natural to use a subset, that is to interpret the proposition by the set

$$\{y \in \mathsf{N} \mid (x =_{\mathsf{N}} y * 2) \vee (x =_{\mathsf{N}} y * 2 \oplus 1)\} \; [x \in N] \quad (21.9)$$

However, using the subset it is not possible to directly translate the proof above to type theory because subset-elimination is not strong enough to interpret ∃-elimination. So we will instead use the Σ set when translating the proof. We will then get an element in the set

$$(\Sigma y \in \mathsf{N})((x =_{\mathsf{N}} y * 2) + (x =_{\mathsf{N}} y * 2 \oplus 1)) \ [x \in \mathsf{N}]$$

and by applying the projection *fst* on this element we will get a program satisfying (21.9).

Our proof of

$$(\exists y \in \mathsf{N})((x =_{\mathsf{N}} y * 2) \vee (x =_{\mathsf{N}} y * 2 \oplus 1)) \ [x \in \mathsf{N}]$$

was by induction, so we will construct an element of the set

$$(\Sigma y \in \mathsf{N})((x =_{\mathsf{N}} y * 2) + (x =_{\mathsf{N}} y * 2 \oplus 1)) \ [x \in \mathsf{N}] \tag{21.10}$$

by N-elimination, remembering that induction corresponds to N-elimination in type theory.

Base: By N-equality and Id-introduction we have

$$\mathsf{id}(0) \in (0 =_{\mathsf{N}} 0 * 2)$$

So, by +-introduction and Σ-introduction, we get

$$\langle 0, \mathsf{inl}(\mathsf{id}(0))\rangle \in (\Sigma y \in \mathsf{N})((0 =_{\mathsf{N}} y * 2) + (0 =_{\mathsf{N}} y * 2 \oplus 1))$$

Recursion step: We want to construct an element in the set

$$(\Sigma y \in \mathsf{N})((x \oplus 1 =_{\mathsf{N}} y * 2) + (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1))$$

from the assumptions

$$x \in \mathsf{N}, z_1 \in (\Sigma y \in \mathsf{N})((x =_{\mathsf{N}} y * 2) + (x =_{\mathsf{N}} y * 2 \oplus 1)) \tag{21.11}$$

We will use Σ-elimination on (21.11) and therefore assume

$$y \in \mathsf{N}, z_2 \in ((x =_{\mathsf{N}} y * 2) + (x =_{\mathsf{N}} y * 2 \oplus 1)) \tag{21.12}$$

There are two cases:

(i) Assume

$$z_3 \in (x =_{\mathsf{N}} y * 2) \tag{21.13}$$

Substitution in the propositional function $\mathsf{Id}(\mathsf{N}, x \oplus 1, z \oplus 1) \ [z \in \mathsf{N}]$ gives

$$\mathit{subst}(z_3, \mathsf{id}(x \oplus 1)) \in (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)$$

and by +-introduction we then get

$$\mathsf{inr}(\mathit{subst}(z_3, \mathsf{id}(x \oplus 1))) \in (x \oplus 1 =_{\mathsf{N}} y * 2) + (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)$$

Hence, by Σ-introduction,

$$\begin{aligned} &\langle y, \mathsf{inr}(\mathit{subst}(z_3, \mathsf{id}(x \oplus 1)))\rangle \\ &\in (\Sigma y \in \mathsf{N})((x \oplus 1 =_{\mathsf{N}} y * 2) + (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)) \end{aligned} \tag{21.14}$$

(ii) Assume

$$z_4 \in (x =_{\mathsf{N}} y * 2 \oplus 1) \tag{21.15}$$

By elementary arithmetic we get a construction

$$c(x, y, z_4) \in (x \oplus 1 =_{\mathsf{N}} (y \oplus 1) * 2)$$

and by $+$-introduction we then get

$$\mathsf{inl}(c(x, y, z_4)) \in (x \oplus 1 =_{\mathsf{N}} (y \oplus 1) * 2 + x \oplus 1 =_{\mathsf{N}} (y \oplus 1) * 2 \oplus 1)$$

Hence, by Σ-introduction,

$$\begin{array}{l} \langle y \oplus 1, \mathsf{inl}(c(x, y, z_4))\rangle \\ \in (\Sigma y \in \mathsf{N})((x \oplus 1 =_{\mathsf{N}} y * 2) + (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)) \end{array} \tag{21.16}$$

Since we have a derived (21.14) from (21.13) and (21.16) from (21.15) we can use $+$-elimination to obtain

$$\begin{array}{l} \mathsf{when}(z_2, \\ \quad (z_3)\langle y, \mathsf{inr}(\mathit{subst}(z_3, \mathsf{id}(x \oplus 1)))\rangle, \\ \quad (z_4)\langle y \oplus 1, \mathsf{inl}(c(x, y, z_4))\rangle))) \\ \in (\Sigma y \in \mathsf{N})((x \oplus 1 =_{\mathsf{N}} y * 2) + (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)) \end{array} \tag{21.17}$$

thereby discharging assumptions (21.13) and (21.15). (21.16) depends on the assumption (21.12) which we can discharge by using Σ-elimination:

$$\begin{array}{l} \mathsf{split}(z_1, \\ \quad (y, z_2)\mathsf{when}(z_2, \\ \qquad (z_3)\langle y, \mathsf{inr}(\mathit{subst}(z_3, \mathsf{id}(x \oplus 1)))\rangle, \\ \qquad (z_4)\langle y \oplus 1, \mathsf{inl}(c(x, y, z_4))\rangle)))) \\ \in (\Sigma y \in \mathsf{N})((x \oplus 1 =_{\mathsf{N}} y * 2) + (x \oplus 1 =_{\mathsf{N}} y * 2 \oplus 1)) \end{array}$$

Now we can use N-elimination to obtain

$$\begin{array}{l} \mathsf{natrec}(x, \\ \quad \langle 0, \mathsf{inl}(\mathsf{id}(0))\rangle, \\ \quad (x, z_1)\mathsf{split}(z_1, \\ \qquad (y, z_2)\mathsf{when}(z_2, \\ \qquad\quad (z_3)\langle y, \mathsf{inr}(\mathit{subst}(z_3, \mathsf{id}(x \oplus 1)))\rangle, \\ \qquad\quad (z_4)\langle y \oplus 1, \mathsf{inl}(c(x, y, z_4))\rangle))))) \\ \in (\Sigma y \in \mathsf{N})((x =_{\mathsf{N}} y * 2) + (x =_{\mathsf{N}} y * 2 \oplus 1)) \\ [x \in \mathsf{N}] \end{array} \tag{21.18}$$

Defining *half_proof* by

$$\begin{array}{l} \mathit{half\_proof} \equiv \\ \lambda x.\mathsf{natrec}(x, \\ \quad \langle 0, \mathsf{inl}(\mathsf{id}(0))\rangle, \\ \quad (x, z_1)\mathsf{split}(z_1, \\ \qquad (y, z_2)\mathsf{when}(z_2, \\ \qquad\quad (z_3)\langle y, \mathsf{inr}(\mathit{subst}(z_3, \mathsf{id}(x \oplus 1)))\rangle, \\ \qquad\quad (z_4)\langle y \oplus 1, \mathsf{inl}(c(x, y, z_4))\rangle)))) \end{array}$$

and *half* by

$$half(x) \equiv fst(half\_proof \cdot x)$$

we get, by applying Σ-elimination twice and then using subset introduction,

$$half(x) \in \{y \in \mathsf{N} | (x =_{\mathsf{N}} y * 2) + (x =_{\mathsf{N}} y * 2 \oplus 1)\} \; [x \in N]$$

Note that we in this type theory derivation not only have constructed the program *half* but also simultaneously have given an almost formal proof that the program satisfies the specification, that is that $half(n)$ computes the integral part of $n/2$ for each natural number n.

In the proof there was a proof of a trivial arithmetic equation which we did not carry out. Note, however, that this proof element is never used in the computation of the program *half*.

Since the program was constructed from a derivation using logic, there occur parts in the program which one normally would not use when constructing the program in a traditional way. For instance, the when-part of the program comes from an application, in the induction step, of $\vee$-elimination where one is using the induction hypothesis which tells you that a number is either even or odd. Thinking operationally, one would here probably have used some construction involving if then else .

21.2 Even or odd

By using the previous example and a proof of

$$((\exists x \in A)(P(x) \vee Q(x))) \supset ((\exists x \in A)P(x) \vee (\exists x \in A)Q(x)) \tag{21.19}$$

we will derive a program $even(n)$ in the set Bool which has value true if the natural number n is even and false if n is odd.

This can be proved in the following bottom-up way: By $\exists$-introduction and $\vee$-introduction we get

$$(\exists x \in A)P(x) \vee (\exists x \in A)Q(x) \;\; [x \in A, \; P(x)]$$

and

$$(\exists x \in A)P(x) \vee (\exists x \in A)Q(x) \;\; [x \in A, \; Q(x)]$$

Now we can use $\vee$-elimination to get

$$(\exists x \in A)P(x) \vee (\exists x \in A)Q(x) \;\; [x \in A, \; P(x) \vee Q(x)]$$

Finally, by $\exists$-elimination and $\supset$-introduction we obtain (21.19).

Translating this proof, using propositions as sets, gives the following derivation. By Σ -introduction and $+$ -introduction we get

$$\mathsf{inl}(\langle x, u\rangle) \in (\Sigma x \in A)P(x) + (\Sigma x \in A)Q(x) \;\; [x \in A, \; u \in P(x)]$$

and

$$\mathsf{inr}(\langle x, v\rangle) \in (\Sigma x \in A)Q(x) + (\Sigma x \in A)Q(x) \;\; [x \in A, \; v \in Q(x)]$$

We can now use + -elimination to get

$$\begin{aligned}&\mathsf{when}(y,(u)\mathsf{inl}(\langle x,u\rangle),(v)\mathsf{inr}(\langle x,v\rangle)) \in\\&(\Sigma x \in A)P(x)+(\Sigma x \in A)Q(x)\ [x \in A,\ y \in P(x)+Q(x)]\end{aligned}$$

By Σ -elimination we obtain

$$\begin{aligned}&\mathsf{split}(z,(x,y)\mathsf{when}(y,(u)\mathsf{inl}(\langle x,u\rangle),(v)\mathsf{inr}(\langle x,v\rangle)))\\&\in (\Sigma x \in A)P(x)+(\Sigma x \in A)Q(x)\end{aligned}$$

under the assumption that $z \in (\Sigma x \in A)(P(x)+Q(x))$. We can now use $\rightarrow$-introduction to obtain

$$\mathit{distr} \in (\Sigma x \in A)(P(x)+Q(x)) \rightarrow (\Sigma x \in A)P(x)+(\Sigma x \in A)Q(x)$$

where

$$\mathit{distr} \ \equiv\ \mathsf{split}(z,(x,y)\mathsf{when}(y,(u)\mathsf{inl}(\langle x,u\rangle),(v)\mathsf{inr}(\langle x,v\rangle)))$$

In the previous example we have derived a program *half_proof* in the set

$$(\Pi x \in \mathsf{N})(\Sigma y \in \mathsf{N})((x =_{\mathsf{N}} y*2)+(x =_{\mathsf{N}} y*2\oplus 1))$$

Hence, by putting

$$\begin{aligned}P(y) &\equiv (x =_{\mathsf{N}} y*2)\\Q(y) &\equiv (x =_{\mathsf{N}} y*2\oplus 1)\\\mathit{Even}(x) &\equiv (\Sigma y \in \mathsf{N})(x =_{\mathsf{N}} y*2)\\\mathit{Odd}(x) &\equiv (\Sigma y \in \mathsf{N})(x =_{\mathsf{N}} y*2\oplus 1)\end{aligned}$$

we get, by $\rightarrow$-elimination,

$$\mathit{distr}\cdot(\mathit{half\_proof}\cdot x) \in \mathit{Even}(x)+\mathit{Odd}(x)$$

Defining *even_or_odd* by

$$\mathit{even\_or\_odd}(n) \equiv \mathit{distr}\cdot(\mathit{half\_proof}\cdot x)$$

we have that $\mathit{even\_or\_odd}(n)$ has a value whose outermost form is inl if and only if n is even. So we can now define *even* by

$$\mathit{even}(n) \equiv \mathsf{when}(\mathit{even\_or\_odd}(n),(u)\mathsf{true},(v)\mathsf{false})$$

and by + -elimination we have

$$\mathit{even}(n) \in \mathsf{Bool}\ \ [n \in \mathsf{N}]$$

Clearly, $\mathit{even}(n)$ has value true if n is even and value false if n is odd.

21.3 Bool has only the elements true and false

We prove the proposition

$$((\exists b \in \mathsf{Bool})P(b)) \supset (P(\mathsf{true}) \vee P(\mathsf{false}))$$

by showing that the set

$$((\Sigma b \in \mathsf{Bool})P(b)) \to (P(\mathsf{true}) + P(\mathsf{false}))$$

is inhabited.

We start the derivation by assuming

$$w \in (\Sigma b \in \mathsf{Bool})P(b) \tag{21.1}$$

and then look for an element in the set

$$P(\mathsf{true}) + P(\mathsf{false})$$

We continue by making two more assumptions

$$w_1 \in \mathsf{Bool} \tag{21.2}$$
$$w_2 \in P(w_1) \tag{21.3}$$

Unfortunately, there is now not a straightforward way to get an element in the set $P(\mathsf{true}) + P(\mathsf{false})$ from the assumptions we have introduced. Instead we must first derive an element in the set

$$P(w_1) \to (P(\mathsf{true}) + P(\mathsf{false}))$$

by case analysis on w_1 and then apply this element on w_2 to get an element in the set

$$P(\mathsf{true}) + P(\mathsf{false})$$

We use +-introduction and →-introduction on the assumption

$$q \in P(\mathsf{true})$$

to get

$$\lambda(\mathsf{inl}) \in P(\mathsf{true}) \supset (P(\mathsf{true})+P(\mathsf{false})) \tag{21.4}$$

In the same way we also get

$$\lambda(\mathsf{inr}) \in P(\mathsf{false}) \supset (P(\mathsf{true})+P(\mathsf{false})) \tag{21.5}$$

By applying Bool-elimination on (21.2), (21.4) and (21.5), we get

$$\begin{array}{l} \mathsf{if}\ w_1\ \mathsf{then}\ \lambda(\mathsf{inl})\ \mathsf{else}\ \lambda(\mathsf{inr}) \\ \in P(w_1) \to (P(\mathsf{true})+P(\mathsf{false})) \end{array} \tag{21.6}$$

Then →-elimination, applied on (21.3) and (21.6), gives

$$\begin{array}{l} \mathsf{apply}(\mathsf{if}\ w_1\ \mathsf{then}\ \lambda(\mathsf{inl})\ \mathsf{else}\ \lambda(\mathsf{inr}), w_2) \\ \in P(\mathsf{true})+P(\mathsf{false}) \end{array} \tag{21.7}$$

Now we can apply the ∃-elimination rule on (21.1) and (21.7) and thereby discharging assumption (21.2) and (21.3):

$$\begin{aligned}&\mathsf{split}(w,\\&\qquad (w_1,w_2)\mathsf{apply}(\mathsf{if}\ w_1\ \mathsf{then}\ \lambda(\mathsf{inl})\ \mathsf{else}\ \lambda(\mathsf{inr}),\\&\qquad\qquad w_2))\\&\in P(\mathsf{true}){+}P(\mathsf{false})\end{aligned} \tag{21.8}$$

Finally, by →-introduction, we discharge (21.1) and get

$$\begin{aligned}&\lambda w.\mathsf{split}(w,\\&\qquad (w_1,w_2)\mathsf{apply}(\mathsf{if}\ w_1\ \mathsf{then}\ \lambda(\mathsf{inl})\ \mathsf{else}\ \lambda(\mathsf{inr}),\\&\qquad\qquad w_2))\\&\in P(\mathsf{true}){+}P(\mathsf{false})\end{aligned} \tag{21.9}$$

In essentially the same way we can prove the propositions:

$$\begin{aligned}&(\exists x \in \mathsf{N})P(x) \supset (P(\mathsf{0}) \vee (\exists y \in \mathsf{N})P(\mathsf{succ}(y)))\\&(\exists x \in \mathsf{List}(A))P(x) \supset (P(\mathsf{nil}) \vee (\exists y \in A)(\exists z \in \mathsf{List}(A))P(\mathsf{cons}(x,y)))\\&(\exists x \in A+B)P(x) \supset ((\exists y \in A)P(\mathsf{inl}(y)) \vee (\exists z \in B)P(\mathsf{inr}(z)))\\&(\exists x \in A\times B)P(x) \supset (\exists y \in A)(\exists z \in B)P(\langle y,z\rangle)\end{aligned}$$

21.4 Decidable predicates

The disjoint union can be used to express that a predicate (propositional function) is *decidable*. Consider the set $B(x)$ *set* $[x \in A]$. To say that B is decidable means that there is a mechanical procedure which for an arbitrary element $a \in A$ decides if $B(a)$ is true or if it is false. In order to formally express that a predicate B is decidable for elements from A, one can use the disjoint union. If the set

$$\mathit{Decidable}(A,B) \;\equiv\; (\Pi x \in A)\, B(x) \vee \neg B(x)$$

is nonempty, then B is decidable and an element in the set is a decision procedure for the predicate.

As an example of a decidable predicate and a decision procedure, we will show that there is an element in the set $\mathit{Decidable}(\mathsf{N},(n)\mathsf{Id}(\mathsf{N},\mathsf{0},n))$, thereby getting a decision procedure that decides if a natural number is equal to zero. We start the derivation by assuming

$$n \in \mathsf{N}$$

We then proceed to find an element in the set

$$\mathsf{Id}(\mathsf{N},\mathsf{0},n) \vee \neg\mathsf{Id}(\mathsf{N},\mathsf{0},n)$$

by induction on n.

The base case: By N-introduction, Id-introduction and ∨-introduction, we get

$$\mathsf{inl}(\mathsf{id}(\mathsf{0})) \in \mathsf{Id}(\mathsf{N},\mathsf{0},\mathsf{0}) \vee \neg\mathsf{Id}(\mathsf{N},\mathsf{0},\mathsf{0})$$

The induction step: We first introduce the induction assumptions

$$x \in \mathsf{N}$$

$$y \in \mathsf{Id}(\mathsf{N}, 0, x) \vee \neg\mathsf{Id}(\mathsf{N}, 0, x)$$

and then continue with the assumption

$$z \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(x))$$

By the proof of Peano's fourth axiom, we have

$$\mathit{peano4} \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(n)) \to \{\}\ [n \in \mathsf{N}]$$

By $\{\}$-elimination and $\to$-introduction, we get

$$\lambda((z)\mathsf{case}(\mathit{peano4} \cdot z)) \in \neg\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(x))$$

We can then use $\vee$-introduction to get

$$\mathsf{inr}(\lambda((z)\mathsf{case}(\mathit{peano4} \cdot z)) \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(x)) \vee \neg\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(x))$$

and the N-elimination rule therefore gives us

$$\begin{array}{r} \mathsf{natrec}(n, \mathsf{inl}(\mathsf{id}(0)), (x, y)\mathsf{inr}(\lambda((z)\mathsf{case}(\mathit{peano4} \cdot z)))) \in \\ \mathsf{Id}(\mathsf{N}, 0, n) \vee \neg\mathsf{Id}(\mathsf{N}, 0, n) \end{array}$$

Finally, by $\to$-introduction,

$$\begin{array}{l} \lambda((n)\mathsf{natrec}(n, \mathsf{inl}(\mathsf{id}(0)), (x, y)\mathsf{inr}(\lambda((z)\mathsf{case}(\mathit{peano4} \cdot z))))) \\ \in \mathit{Decidable}(\mathsf{N}, (n)\mathsf{Id}(\mathsf{N}, 0, n)) \end{array}$$

So, we have derived a decision procedure for the predicate $(n)\mathsf{Id}(\mathsf{N}, 0, n)$.

21.5 Stronger elimination rules

It is possible to formulate stronger versions of the elimination rules, for instance, the rule of strong Σ-elimination:

Strong Σ – elimination

$$\frac{\begin{array}{l} c \in \Sigma(A, B) \\ C(v)\ \mathit{set}\ \ [v \in \Sigma(A, B)] \\ d(x, y) \in C(\langle x, y\rangle)\ \ [x \in A,\ y \in B(x),\ \langle x, y\rangle =_{\Sigma(A,B)} c\ \mathit{true}] \end{array}}{\mathsf{split}'(c, d) \in C(c)}$$

The third premise is weaker than the corresponding premise in the ordinary rule for Σ-elimination in that the assumption $\langle x, y\rangle =_{\Sigma(A,B)} c\ \mathit{true}$ is added. The constant split has been replaced by the defined constant split'. This rule can be seen as a derived rule in the following way:

Let

$$\begin{array}{l} c \in \Sigma(A, B) \\ C(v)\ \mathit{set}\ \ [v \in \Sigma(A, B)] \\ d(x', y') \in C(\langle x', y'\rangle)\ \ [x' \in A,\ y' \in B(x'),\ \langle x', y'\rangle =_{\Sigma(A,B)} c\ \mathit{true}] \end{array}$$

We are going to use the ordinary Σ-elimination rule on c and the family

$$C'(u) \equiv (u =_{\Sigma(A,B)} c) \to C(u)$$

So, assume $x \in A$ and $y \in B(x)$ and we want to find an element in

$$C'(\langle x, y\rangle) \equiv (\langle x, y\rangle =_{\Sigma(A,B)} c) \to C(\langle x, y\rangle)$$

Assume therefore that $z \in (\langle x, y\rangle =_{\Sigma(A,B)} c)$. But then

$$d(x, y) \in C(\langle x, y\rangle)$$

and $\to$-introduction gives that

$$\lambda z.d(x, y) \in (\langle x, y\rangle =_{\Sigma(A,B)} c) \to C(\langle x, y\rangle)$$

thereby discharging the last assumption. Σ-elimination gives

$$\mathsf{split}(c, (x, y)\lambda z.d(x, y)) \in (c =_{\Sigma(A,B)} c) \to C(c)$$

thereby discharging the remaining two assumptions. Since we know that $\mathsf{id}(c) \in (c =_{\Sigma(A,B)} c)$ we can use $\to$-elimination to finally conclude that

$$\mathsf{split}'(c, d) \in C(c)$$

where

$$\mathsf{split}'(c, d) \equiv \mathsf{apply}(\mathsf{split}(c, (x, y)\lambda z.d(x, y)), \mathsf{id}(c)).$$

Notice, that if the premises of the strong elimination rule hold then the value of $\mathsf{split}'(c, d)$ is equal to the value of $\mathsf{split}(c, d)$ which can be seen from the following computation steps:

$$\cfrac{\cfrac{c \Rightarrow \langle a, b\rangle \qquad \lambda z.d(a, b) \Rightarrow \lambda z.d(a, b)}{\mathsf{split}(c, (x, y)\lambda z.d(x, y)) \Rightarrow \lambda z.d(a, b)} \qquad d(a, b) \Rightarrow q}{\mathsf{apply}(\mathsf{split}(c, (x, y)\lambda z.d(x, y)), \mathsf{id}(c)) \Rightarrow q}$$

We can strengthen the elimination-rules for Π, $+$, and the enumeration sets in an analogous way:

Strong Π–elimination

$$\frac{\begin{array}{l} c \in \Pi(A, B) \\ C(v)\ set\ \ [v \in \Pi(A, B)] \\ d(y) \in C(\lambda(y))\ \ [y(x) \in B(x)\ [x \in A],\ c =_{\Pi(A,B)} \lambda(y)\ true] \end{array}}{\mathsf{funsplit}'(c, d) \in C(c)}$$

where

$$\mathsf{funsplit}'(c, d) \equiv \mathsf{apply}(\mathsf{funsplit}(c, (y)\lambda z.d(y)), \mathsf{id}(c))$$

Strong +–elimination

$$\frac{\begin{array}{l} c \in A + B \\ C(v)\ set\ \ [v \in A + B] \\ d(x) \in C(\mathsf{inl}(x))\ \ [x \in A,\ c =_{A+B} \mathsf{inl}(x)\ true] \\ e(y) \in C(\mathsf{inr}(y))\ \ [y \in B,\ c =_{A+B} \mathsf{inr}(y)\ true] \end{array}}{\mathsf{when}'(c, d, e) \in C(c)}$$

where

$$\mathsf{when}'(c, d, e) \equiv \mathsf{apply}(\mathsf{when}(c,\ (x)\lambda z.d(x),\ (y)\lambda z.e(y)),\ \mathsf{id}(c))$$

Strong Bool–elimination

$$\frac{\begin{array}{l} b \in \mathsf{Bool} \\ C(v)\ set\ \ [v \in \mathsf{Bool}] \\ c \in C(\mathsf{true})\ \ [b =_{\mathsf{Bool}} \mathsf{true}\ true] \\ d \in C(\mathsf{false})\ \ [b =_{\mathsf{Bool}} \mathsf{false}\ true] \end{array}}{\mathsf{if}'(b, c, d) \in C(b)}$$

where

$$\mathsf{if}'(b, c, d) \equiv \mathsf{apply}(\mathsf{if}(b, \lambda z.c, \lambda z.d), \mathsf{id}(b))$$

Chapter 22

Program derivation

One of the main reasons for using type theory for programming is that it can be seen as a theory both for writing specifications and constructing programs. In type theory a specification is expressed as a set and an element of that set is a program that satisfies the specification.

Programming in type theory corresponds to theorem proving in mathematics: the specification plays the rôle of the proposition to be proved and the program is obtained from the proof. We will in this chapter formulate the rules of type theory as tactics, corresponding to constructing programs top down. The idea of synthesising programs from constructive proofs has been used e.g. by Manna and Waldinger [62] Takasu [106] and Constable and his coworkers at Cornell University [25].

22.1 The program derivation method

As already has been mentioned, programming in type theory is like theorem proving in mathematics. However, since parts of the proofs are used in the actual construction of programs, the proofs have to be more detailed and formal than they usually are in mathematics. In this respect, derivations of programs in type theory are similar to proofs of mathematical theorems in a formal system. Being formal is also a necessity when dealing with complex problems since one then certainly need computer support. For the examples in this chapter the solutions are so simple that there are no problems in doing the derivations informally. But already in the solution of Dijkstra's problem of the Dutch national flag using arrays [87], there are so many steps and so much book-keeping that it is appropriate to make the derivation in such a way that it could be checked by a computer. So, in order to illustrate the method, our example is carried out in such a detail that it should be straightforward to obtain a completely formal derivation. Differences between proofs in traditional mathematics and program derivations as well as the rôle of formalization are discussed by Scherlis and Scott [94].

The usual way of presenting a formal derivation, e.g. in text books on logic, is to go from axioms and assumptions to the conclusion. When deriving programs in type theory this would mean that you first start constructing the smaller parts of the program and then build up the program from these parts.

This is not a good programming methodology. Instead we want to use the top-down approach from structured programming [32]. So, instead of starting the derivation from axioms and assumptions, we will proceed in the opposite direction. We will start with the specification, split it into subspecifications and then compose solutions to these subproblems to a solution of the original problem. In the LCF-system [44] there is a goal directed technique for finding proofs in this style.

Corresponding to the judgement

$$a \in A$$

we have the goal A which is achieved by an element a if we have a proof of $a \in A$. Corresponding to each of the other forms of judgement, we have a goal which has the same form as the judgement and which is achieved if we have a proof of it. For instance the goal $a = b \in A$ is achieved if we have a proof of $a = b \in A$. Notice that in general goals may depend on assumptions. The different methods that can be used to split a goal into subgoals are called *tactics*.

22.1.1 Basic tactics

The basic tactics come from reading the rules of type theory bottom-up. For example, the introduction rule for conjunction

$$\frac{A\ \mathit{true} \qquad B\ \mathit{true}}{A \,\&\, B\ \mathit{true}}$$

becomes, when viewed as a tactic:

> The goal
>
> $$A \,\&\, B\ \mathit{true}$$
>
> may be split into the subgoals
>
> $$A\ \mathit{true}$$
>
> and
>
> $$B\ \mathit{true}$$

We can describe the tactic in the following way:

> $\lceil A \,\&\, B\ \mathit{true}$ by &-introduction
> $\quad[A\ \mathit{true}$ by ...
> $\quad[B\ \mathit{true}$ by ...
> $\lfloor$

Similarly, the introduction rule for the cartesian product

$$\frac{a \in A \qquad b \in B}{\langle a, b\rangle \in A \,\&\, B}$$

can be read as a tactic:

> The problem of finding a program that achieves the goal
>
> $$A \times B$$
>
> can be split into the problem of finding a program a that achieves the goal
>
> $$A$$
>
> and the problem of finding a program b that achieves the goal
>
> $$B$$
>
> The goal $A \times B$ is then achieved by the program $\langle a, b\rangle$.

When deriving a program from a specification, applying a tactic will give a part of the program one is in the process of constructing. In the case of the $\times$-introduction tactic, one gets a part on pair-form. The $\times$-introduction tactic can also be described in the following way:

$\lceil A \times B$ by $\times$-introduction
    $\lceil A$ by ...
    $\lfloor \ni a$
    $\lceil B$ by ...
    $\lfloor \ni b$
$\lfloor \ni \langle a, b\rangle$

This schematical way of describing a tactic can be extended to full derivations. It can also be used when a derivation is not yet complete and then give the structure of the derivation made so far as well as the structure of the program obtained at that stage.

Another example is the rule for $\times$-elimination:

$$\frac{p \in A \times B \qquad e(x,y) \in C(\langle x, y\rangle \;\; [x \in A, y \in B]}{\mathsf{split}(p, e) \in C(p)}$$

we get the following $\times$-elimination tactic in type theory:

> The problem of finding a program that achieves the goal
>
> $$C(p)$$
>
> can be replaced by proving that $p \in A \times B$ and the problem of finding a program $e(x, y)$ that achieves the goal
>
> $$C(\langle x, y\rangle)$$
>
> under the assumptions that $x \in A$ and $y \in B$.
>
> The goal $C(p)$ is then achieved by the program $\mathsf{split}(p, e)$.

In our notation:

$\lceil C(p)$ by $\times$-elimination
  $\lceil A \times B$ by ...
  $\lfloor \ni p$
  $[x \in A, y \in B]$
    $\lceil C(\langle x, y\rangle)$ by ...
    $\lfloor \ni e(x, y)$
$\lfloor \ni \mathsf{split}(p, e)$

In this way all rules of type theory may be formulated as tactics. This is also the approach taken in the system for type theory developed at Cornell University [25]. We give two more examples of translating rules into tactics by formulating the Π-introduction rule and the List-elimination rule as tactics. Both tactics will be used in the derivation of a program for the problem of the Dutch flag.

Corresponding to the Π-introduction rule

$$\frac{b(x) \in B(x)\ [x \in A]}{\lambda(b) \in (\Pi x \in A)B(x)}$$

we have the tactic:

$\lceil (\Pi x \in A)\, B(x)$ by Π-introduction
  $[x \in A]$
    $\lceil B(x)$ by ...
    $\lfloor \ni b(x)$
$\lfloor \ni \lambda(b)$

The List-elimination rule,

$$\frac{\begin{array}{l} l \in \mathsf{List}(A) \\ a \in C(\mathsf{nil}) \\ b(x, y, z) \in C(\mathsf{cons}(x, y))\ \ [x \in A,\ y \in \mathsf{List}(A),\ z \in C(y)] \end{array}}{\mathsf{listrec}(l, a, b) \in C(l)}$$

becomes, when formulated as a tactic:

$\lceil C(l)$ by List-elimination
  $\lceil \mathsf{List}(A)$ by ...
  $\lfloor \ni l$
  $\lceil C(\mathsf{nil})$ by ...
  $\lfloor \ni a$
  $[x \in A,\ y \in \mathsf{List}(A),\ z \in C(y)]$
    $\lceil C(\mathsf{cons}(x, y))$ by ...
    $\lfloor \ni b(x, y, z)$
$\lfloor \ni \mathsf{listrec}(l, a, b)$

22.1.2 Derived tactics

If we have a proof of a judgement then we also have a derived tactic corresponding to the judgement. We can look at a tactic as another way of reading a hypothetical judgement. For instance, if we have a proof of the hypothetical judgement

$$c(x, y) \in C(x, y)\ \ [x \in A,\ y \in B(x)] \qquad (J1)$$

then we can use the following tactic:

$$
\begin{array}{l}
\lceil C(x,y) \text{ by } J1 \\
\quad \lceil A \text{ by } \ldots \\
\quad \lfloor \ni x \\
\quad \lceil B(x) \text{ by } \ldots \\
\quad \lfloor \ni y \\
\lfloor \ni c(x,y)
\end{array}
$$

As a simple example, after having made the derivation

$$
\begin{array}{l}
[p \in A \times B] \\
\quad \lceil A \text{ by } \times\text{-elimination} \\
\quad\quad \lceil A \times B \text{ by assumption} \\
\quad\quad \lfloor \ni p \\
\quad\quad [x \in A,\ y \in B] \\
\quad\quad\quad \lceil A \text{ by assumption} \\
\quad\quad\quad \lfloor \ni x \\
\quad \lfloor \ni \mathsf{split}(p, (x,y)x) \ \equiv\ \mathit{fst}(p)
\end{array}
$$

which is a proof of the judgement

$$\mathit{fst}(p) \in A \quad [p \in A \times B] \qquad (\times - elim1)$$

$$
\begin{array}{l}
[p \in A \times B] \\
\quad \lceil A \text{ by } \times\text{-elim1} \\
\quad \lfloor \ni \mathit{fst}(p)
\end{array}
$$

If we had a mechanical proof checker, it would not be necessary to check the correctness of a derived tactic more than once. In an application of it, there is no need to go through each step in the proof since by the construction of a derived tactic (that it comes from a judgement) we know that if we apply it to proofs of the subgoals it always yield a proof of the goal.

22.2 A partitioning problem

In this section, we will derive a program for Dijkstra's *problem of the Dutch national flag* [32]: Construct a program, that given a sequence of objects, each having one of the colours red, white and blue, rearranges the objects so that they appear in the order of the Dutch national flag. In type theory, the natural way of formulating this partitioning problem is to use lists. Our solution will then, we think, result in the simplest possible program for the problem; the program one would write in a functional language like ML. However, the program will not satisfy Dijkstra's requirements concerning space efficiency, which is one of the main points of his solution. In [87] a similar problem is solved, using arrays instead of lists and following Dijkstra's more sophisticated method.

We will use the following general assumptions about the problem: We assume that A is a set and each element in A has a colour, i.e. there is a function $colour(x) \in Colour$, where $Colour$ is the enumeration set $\{\mathsf{red}, \mathsf{white}, \mathsf{blue}\}$. We will also assume that A has a decidable equality. So we introduce the following assumptions:

$$
\begin{array}{l}
A \ set \\
colour(x) \in Colour \ \ [x \in A] \\
eqd(A,x,y) \in \{z \in \mathsf{Bool} \mid z =_{\mathsf{Bool}} \mathsf{true} \ \Leftrightarrow\ x =_A y\} \ \ [x \in A,\ y \in A]
\end{array}
$$

We start by introducing the following definitions:

$$
\begin{array}{rcl}
Colouredlist(s) & \equiv & \mathsf{List}(\{x \in A \mid colour(x) =_{Colour} s\}) \\
Reds & \equiv & Colouredlist(\mathsf{red}) \\
Whites & \equiv & Colouredlist(\mathsf{white}) \\
Blues & \equiv & Colouredlist(\mathsf{blue}) \\
append(l_1, l_2) & \equiv & \mathsf{listrec}(l_1, l_2, (x, y, z)\,\mathsf{cons}(x, z)) \\
l_1 \approx_P l_2 & \equiv & (\forall x \in A)\,\mathsf{Id}(\mathsf{N}, occin(x, l_1), occin(x, l_2)) \\
occin(x, l) & \equiv & \mathsf{listrec}(l, 0, (u, v, w)\,\mathsf{if}\ eqd(A, x, u)\ \ \mathsf{then}\ \ \mathsf{succ}(w)\ \ \mathsf{else}\ \ w) \\
l_1 @ l_2 & \equiv & append(l_1, l_2)
\end{array}
$$

We have here used a definition of permutation which requires the equality relation on A to be decidable. This restriction can be removed, but the definition will then be more complicated.

The specification can now be given by the set

$$S \equiv (\Pi\, l \in \mathsf{List}(A))\ Flag(l)$$

where

$$Flag(l) \ \equiv \ \{\langle l', l'', l'''\rangle \in Reds \times Whites \times Blues \mid l \approx_P l' @ l'' @ l'''\}$$

using the notation $\{\langle x, y, z\rangle \in A \times B \times C \mid P(x, y, z)\}$ for the subset

$$\{u \in A \times (B \times C) \mid P(fst(u), fst(snd(u)), trd(u))\}$$

where trd is defined by

$$trd \ \equiv \ (u)snd(snd(u))$$

Note that a program that satisfies this specification will give a triple of lists as output. To get a solution to Dijkstra's formulation of the problem, these three lists should be concatenated.

Deriving a program that satisfies the specification is nothing but finding a program which is a member of the set expressing the specification, or, if we think of the specification as a goal, to find a program that achieves the goal.

The intuitive idea behind the proof is the following: If l is a list of red, white and blue objects then the problem of finding an element in $Flag(l)$ will be solved by induction on l. The base case, i.e. when l is equal to nil, is solved by the partition $\langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil}\rangle$. For the induction step, assume that l is $\mathsf{cons}(x, y)$ and that we have a partitioning z of y and then separate the problem into three cases:

1. x is red. Then $\langle \mathsf{cons}(x, fst(z)), snd(z), trd(z)\rangle$ is a partitioning of the list $\mathsf{cons}(x, y)$.

2. x is white. Then $\langle fst(z), \mathsf{cons}(x, snd(z)), trd(z)\rangle$ is a partitioning of the list $\mathsf{cons}(x, y)$.

3. x is blue. Then $\langle fst(z), snd(z), \mathsf{cons}(x, trd(z))\rangle$ is a partitioning of the list $\mathsf{cons}(x, y)$.

From this intuitive idea, it would not be much work to get, by informal reasoning, a program in type theory which satisfies the specification. We want, however, to do a derivation which easily could be transformed to a completely formal derivation. In the derivation we will assume a few elementary properties about permutations and these properties will be explicitly stated as lemmas.

We begin the derivation by assuming $l \in \mathsf{List}(A)$ and then try to find a program which is an element of the set $\mathit{Flag}(l)$. In other words, we apply the Π-introduction tactic to the specification S, getting the subgoal

$$\mathit{Flag}(l)\ [l \in \mathsf{List}(A)]$$

From this problem we proceed by list induction on l, i.e., we split the goal into three subgoals, corresponding to the three premises in the List-elimination rule. Schematically, the derivation we have made so far is:

$$\begin{array}{ll}
\multicolumn{2}{l}{\lceil(\Pi\, l \in \mathsf{List}(A))\ \mathit{Flag}(l) \text{ by } \Pi\text{-intro}} \\
\multicolumn{2}{l}{\quad [l \in \mathsf{List}(A)]} \\
\text{G1:} & \lceil \mathit{Flag}(l) \text{ by } \mathsf{List}\text{-elim} \\
& \quad \lceil \mathsf{List}(A) \text{ by assumption} \\
& \quad \lfloor \ni l \\
\text{Base:} & \quad \lceil \mathit{Flag}(\mathsf{nil}) \text{ by } \ldots \\
& \quad [x \in A, y \in \mathsf{List}(A), z \in \mathit{Flag}(y)] \\
\text{Ind. step:} & \qquad \lceil \mathit{Flag}(\mathsf{cons}(x,y)) \text{ by } \ldots
\end{array}$$

So if we succeed to solve the two subgoals finding an element a which achieves the base case and finding an element $b(x, y, z)$ which achieves the induction step then we can complete the derivation:

$$\begin{array}{ll}
\multicolumn{2}{l}{\lceil(\Pi\, l \in \mathsf{List}(A))\ \mathit{Flag}(l) \text{ by } \Pi\text{-intro}} \\
\multicolumn{2}{l}{\quad [l \in \mathsf{List}(A)]} \\
\text{G1:} & \lceil \mathit{Flag}(l) \text{ by } \mathsf{List}\text{-elim} \\
& \quad \lceil \mathsf{List}(A) \text{ by assumption} \\
& \quad \lfloor \ni l \\
\text{Base:} & \quad \lceil \mathit{Flag}(\mathsf{nil}) \text{ by } \ldots \\
& \quad \lfloor \ni a \\
& \quad [x \in A,\ y \in \mathsf{List}(A),\ z \in \mathit{Flag}(y)] \\
\text{Ind. step:} & \qquad \lceil \mathit{Flag}(\mathsf{cons}(x,y)) \text{ by } \ldots \\
& \qquad \lfloor \ni b(x,y,z) \\
& \lfloor \ni \mathsf{listrec}(l, a, b) \\
\multicolumn{2}{l}{\lfloor \ni \lambda((l)\ \mathsf{listrec}(l, a, b)}
\end{array}$$

Let us start with the base case in the induction. We have the goal

$$\mathit{Flag}(\mathsf{nil}) \equiv \{\langle l', l'', l'''\rangle \in \mathit{Reds} \times \mathit{Whites} \times \mathit{Blues} \mid \mathsf{nil} \approx_P l' @ l'' @ l'''\}$$

Following the intuitive idea for the proof, this goal is achieved by $\langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil}\rangle$. Formally, then, we have to show that $\langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil}\rangle \in \mathit{Flag}(\mathsf{nil})$. In order to do this, we apply the Subset/Triple introduction tactic which is the tactic corresponding to the following judgement:

$$\langle a, b, c\rangle \in \{\langle l', l'', l'''\rangle \in A \times B \times C \mid P(l', l'', l''')\}$$
$$[a \in A,\ b \in B,\ c \in C,\ P(a, b, c)\ \mathit{true}]$$

We leave out the derivation of this judgement. By List-introduction we know that nil satisfies the three subgoals *Reds*, *Whites*, *Blues* and then we have to verify the subgoal

$$\mathsf{nil} \approx_P \mathsf{nil@nil@nil}$$

Lemma 1 $\mathsf{nil} \approx_P \mathsf{nil@nil@nil}$

Proof: The lemma follows from the fact that nil is an identity for $\mathsf{@}$ and that permutation is reflexive:

$$\begin{array}{ll}
 & \mathsf{nil@nil@nil} \\
= & \{\ \mathsf{nil@nil} = \mathsf{nil}\ \} \\
 & \mathsf{nil} \\
\approx_P & \{\ l \approx_P l \quad [l \in \mathsf{List}(A)]\ \} \\
 & \mathsf{nil}
\end{array}$$

□

So

$$\langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle \in \mathit{Flag}(\mathsf{nil})$$

and we have solved the base-step. We can summarize the derivation made so far:

$$\begin{array}{ll}
\multicolumn{2}{l}{\lceil (\Pi\, l \in \mathsf{List}(A))\ \mathit{Flag}(l) \text{ by } \Pi\text{-intro}} \\
\multicolumn{2}{l}{\quad [l \in \mathsf{List}(A)]} \\
\text{G1:} & \lceil \mathit{Flag}(l) \text{ by } \mathsf{List}\text{-elim} \\
 & \quad \lceil \mathsf{List}(A) \text{ by assumption} \\
 & \quad \lfloor \ni l \\
\text{Base:} & \quad \lceil \mathit{Flag}(\mathsf{nil}) \text{ by Lemma 2} \\
 & \quad \lfloor \ni \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle \\
 & \quad [x \in A, y \in \mathsf{List}(A), z \in \mathit{Flag}(y)] \\
\text{Ind. step:} & \qquad \lceil \mathit{Flag}(\mathsf{cons}(x, y)) \text{ by } \ldots \\
 & \qquad \lfloor \ni b(x, y, z) \\
 & \lfloor \ni \mathsf{listrec}(l, \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle, b) \\
\multicolumn{2}{l}{\lfloor \ni \lambda((l)\ \mathsf{listrec}(l, \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle, b)}
\end{array}$$

where Lemma 2 is the following derived tactic:

Lemma 2 $\langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle \in \mathit{Flag}(\mathsf{nil})$

Proof: This is a formal derivation of the lemma:

$$\begin{array}{l}
\lceil \mathit{Flag}(\mathsf{nil}) \text{ by Subset/Triple-introduction} \\
\quad \lceil \mathit{Reds} \text{ by } \mathsf{List}\text{-intro} \\
\quad \lfloor \ni \mathsf{nil} \\
\quad \lceil \mathit{Whites} \text{ by } \mathsf{List}\text{-intro} \\
\quad \lfloor \ni \mathsf{nil} \\
\quad \lceil \mathit{Blues} \text{ by } \mathsf{List}\text{-intro} \\
\quad \lfloor \ni \mathsf{nil} \\
\quad \mathsf{nil} \approx_P \mathsf{nil@nil@nil} \ \ \mathit{true} \text{ by Lemma 1} \\
\lfloor \ni \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle
\end{array}$$

□

It now remains to achieve the induction step:

$$\begin{array}{l} \mathit{Flag}(\mathsf{cons}(x,y)) \equiv \{\langle l', l'', l''' \rangle \in \mathit{Reds} \times \mathit{Whites} \times \mathit{Blues} \mid \\ \qquad \mathsf{cons}(x,y) \approx_P l' @ l'' @ l''' \} \end{array}$$

under the assumptions

$$l \in \mathsf{List}(A),\ x \in A,\ y \in \mathsf{List}(A),\ z \in \mathit{Flag}(y)$$

We apply the Subset/Triple elimination tactic, which is the derived tactic (we leave out the derivation):

$$\begin{array}{l}
\lceil C(p) \text{ by Subset/Triple-elim} \\
\quad \lceil \{\langle l', l'', l''' \rangle \in A \times B \times C \mid P(l', l'', l''')\} \text{ by } \ldots \\
\quad \lfloor \ni p \\
\quad [z' \in A,\ z'' \in B,\ z''' \in C,\ P(z', z'', z''')\ \mathit{true}] \\
\qquad \lceil C(\langle z', z'', z''' \rangle) \text{ by } \ldots \\
\qquad \lfloor \ni e(z', z'', z''') \\
\lfloor \ni \mathsf{split}_3(p, e)
\end{array}$$

We then get the two subgoals

1. $\mathit{Flag}(y) \equiv \{\langle l', l'', l''' \rangle \in \mathit{Reds} \times \mathit{Whites} \times \mathit{Blues} \mid y \approx_P l' @ l'' @ l''' \}$
2. $[z' \in \mathit{Reds}, z'' \in \mathit{Whites}, z''' \in \mathit{Blues}, y \approx_P z' @ z'' @ z'''\ \mathit{true}]$
   $\mathit{Flag}(\mathsf{cons}(x,y))$

The first subgoal is achieved by z and the second subgoal says that the problem is to find a program which is an element of $\mathit{Flag}(\mathsf{cons}(x,y))$ under the extra assumptions about z', z'', z'''. Following the intuitive proof idea, we divide the remaining subgoal into three cases: when the element x is red, when it is white and when it is blue. From one of the assumptions done earlier we know that

$$\mathit{colour}(x) \in \mathit{Colour}\ \ [x \in A]$$

so it is appropriate to apply the *Colour*-elimination tactic:

$$\begin{array}{l}
\lceil C(p) \text{ by } \mathit{Colour}\text{-elimination} \\
\quad \lceil \mathit{Colour} \text{ by } \ldots \\
\quad \lfloor \ni p \\
\quad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{red}] \\
\qquad \lceil C(p) \text{ by } \ldots \\
\qquad \lfloor \ni a \\
\quad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{white}] \\
\qquad \lceil C(p) \text{ by } \ldots \\
\qquad \lfloor \ni b \\
\quad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{blue}] \\
\qquad \lceil C(p) \text{ by } \ldots \\
\qquad \lfloor \ni c \\
\lfloor \ni \mathsf{case}_{\mathit{Colour}}(p, a, b, c)
\end{array}$$

We then get the following derivation:

$\lceil Flag(\mathsf{cons}(x,y))$ by *Colour*-elimination
$\quad\lceil Colour$ by assumption
$\quad\lfloor \ni colour(x)$
$\quad[colour(x) =_{Colour} \mathsf{red}]$
$\qquad\lceil Flag(\mathsf{cons}(x,y)$ by ...
$\quad[colour(x) =_{Colour} \mathsf{white}]$
$\qquad\lceil Flag(\mathsf{cons}(x,y)$ by ...
$\quad[colour(x) =_{Colour} \mathsf{blue}]$
$\qquad\lceil Flag(\mathsf{cons}(x,y)$ by ...

That the program

$$\langle \mathsf{cons}(x,z'), z'', z'''\rangle$$

achieves the red case is seen by the following derivation, which we call A1.

$\lceil Flag(\mathsf{cons}(x,y)) \equiv$
$\{\langle l', l'', l'''\rangle \in Reds \times Whites \times Blues \mid \mathsf{cons}(x,y) \approx_P l' @ l'' @ l'''\}$
by Subset/Triple-intro
$\quad\lceil Reds \equiv \mathsf{List}(\{x \in A \mid colour(x) = \mathsf{red}\})$ by List-intro
$\qquad\lceil\{x \in A \mid colour(x) = \mathsf{red}\}$ by subset-intro
$\qquad\quad\lceil A$ by assumption
$\qquad\quad\lfloor \ni x$
$\qquad\quad\lceil colour(x) = \mathsf{red}\ true$ by assumption
$\qquad\quad\lfloor$
$\qquad\lfloor \ni x$
$\qquad\lceil\{x \in A \mid colour(x) = \mathsf{red}\}$ by assumption
$\qquad\lfloor \ni z'$
$\quad\lfloor \ni \mathsf{cons}(x,z')$
$\quad\lceil Whites$ by assumption
$\quad\lfloor \ni z''$
$\quad\lceil Blues$ by assumption
$\quad\lfloor \ni z'''$
$\quad\lceil \mathsf{cons}(x,z') @ z'' @ z''' \approx_P \mathsf{cons}(x,y)\ true$ by Lemma 3
$\quad\lfloor$
$\lfloor \ni \langle \mathsf{cons}(x,z'), z'', z'''\rangle$

The following lemma has been used in the derivation.

Lemma 3 *If A is a set, $x \in A$, $y \in \mathsf{List}(A)$, $z' \in \mathsf{List}(A)$, $z'' \in \mathsf{List}(A)$, $z''' \in \mathsf{List}(A)$ and $y \approx_P z' @ z'' @ z'''$ true, then*

$$\mathsf{cons}(x,z') @ z'' @ z''' \approx_P \mathsf{cons}(x,y)\ true$$

Proof:

$$\begin{array}{ll} & \mathsf{cons}(x,z') @ z'' @ z''' \\ = & \{\ List - equality\ \} \\ & \mathsf{cons}(x, z' @ z'' @ z''') \\ \approx_P & \{\ z' @ z'' @ z''' \approx_P y, \mathsf{cons}(x,z) \approx_P \mathsf{cons}(x,y)\quad [x \in A, z \approx_P y\ true]\ \} \\ & \mathsf{cons}(x,y) \end{array}$$

□

We can achieve the remaining subgoals in a similar way, letting A2 and A3 correspond to A1 in the white and blue cases, respectively:

$$
\begin{array}{l}
\lceil \mathit{Flag}(\mathsf{cons}(x,y)) \text{ by } \mathit{Colour}\text{-elimination}\\
\quad \lceil \mathit{Colour} \text{ by assumption}\\
\quad \lfloor \ni \mathit{colour}(x)\\
\quad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{red}]\\
\qquad\quad \lceil \mathit{Flag}(\mathsf{cons}(x,y)) \text{ by A1}\\
\qquad\quad \lfloor \ni \langle \mathsf{cons}(x,z'), z'', z''' \rangle\\
\quad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{white}]\\
\qquad\quad \lceil \mathit{Flag}(\mathsf{cons}(x,y) \text{ by A2}\\
\qquad\quad \lfloor \ni \langle z', \mathsf{cons}(x,z''), z''' \rangle\\
\quad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{blue}]\\
\qquad\quad \lceil \mathit{Flag}(\mathsf{cons}(x,y) \text{ by A3}\\
\qquad\quad \lfloor \ni \langle z', z'', \mathsf{cons}(x,z''') \rangle
\end{array}
$$

Combining the solutions of the last three subproblems gives us that the goal is achieved by

$$
\begin{array}{l}
\mathsf{case}_{\mathit{Colour}}(\mathit{colour}(x),\\
\qquad\qquad \langle \mathsf{cons}(x,z'), z'', z''' \rangle,\\
\qquad\qquad \langle z', \mathsf{cons}(x,z''), z''' \rangle,\\
\qquad\qquad \langle z', z'', \mathsf{cons}(x,z''') \rangle)
\end{array}
$$

We can now form a program that achieves the induction step:

$$
\begin{array}{l}
\mathsf{split}_3(z,\\
\qquad (z', z'', z''')\mathsf{case}_{\mathit{Colour}}(\mathit{colour}(x),\\
\qquad\qquad\qquad\qquad \langle \mathsf{cons}(x,z'), z'', z''' \rangle,\\
\qquad\qquad\qquad\qquad \langle z', \mathsf{cons}(x,z''), z''' \rangle,\\
\qquad\qquad\qquad\qquad \langle z', z'', \mathsf{cons}(x,z''') \rangle))
\end{array}
$$

G1 is then achieved by

$$
\begin{array}{l}
\mathsf{listrec}(l, \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle\\
\qquad (x,y,z)\mathsf{split}_3(z,\\
\qquad\qquad (z', z'', z''')\mathsf{case}_{\mathit{Colour}}(\mathit{colour}(x),\\
\qquad\qquad\qquad\qquad\qquad \langle \mathsf{cons}(x,z'), z'', z''' \rangle,\\
\qquad\qquad\qquad\qquad\qquad \langle z', \mathsf{cons}(x,z''), z''' \rangle,\\
\qquad\qquad\qquad\qquad\qquad \langle z', z'', \mathsf{cons}(x,z''') \rangle)))
\end{array}
$$

And, finally

$$
\begin{array}{l}
\lambda((l)\ \mathsf{listrec}(l, \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle\\
\qquad\qquad (x,y,z)\mathsf{split}_3(z,\\
\qquad\qquad\qquad (z', z'', z''')\mathsf{case}_{\mathit{Colour}}(\mathit{colour}(x),\\
\qquad\qquad\qquad\qquad\qquad\qquad \langle \mathsf{cons}(x,z'), z'', z''' \rangle,\\
\qquad\qquad\qquad\qquad\qquad\qquad \langle z', \mathsf{cons}(x,z''), z''' \rangle,\\
\qquad\qquad\qquad\qquad\qquad\qquad \langle z', z'', \mathsf{cons}(x,z''') \rangle))))
\end{array}
$$

is a program that achieves our original problem and consequently also a program that satisfies the specification.

The whole derivation is described in figure 22.1.

$$
\begin{array}{l}
\lceil (\Pi\, l \in \mathsf{List}(A))\ \mathit{Flag}(l) \text{ by } \Pi\text{-intro} \\
\quad [l \in \mathsf{List}(A)] \\
\text{G1:} \quad \lceil \mathit{Flag}(l) \text{ by } \mathsf{List}\text{-elim} \\
\qquad\quad \lceil \mathsf{List}(A) \text{ by assumption} \\
\qquad\quad \lfloor \ni l \\
\text{Base:} \quad\ \lceil \mathit{Flag}(\mathsf{nil}) \text{ by Lemma 2} \\
\qquad\quad \lfloor \ni \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle \\
\qquad\quad [x \in A, y \in \mathsf{List}(A), z \in \mathit{Flag}(y)] \\
\text{Ind. step:} \quad \lceil \mathit{Flag}(\mathsf{cons}(x,y)) \text{ by Subset/Triple-elim} \\
\qquad\qquad \lceil \mathit{Flag}(y) \equiv \\
\qquad\qquad \{\langle l', l'', l''' \rangle \in \mathit{Reds} \times \mathit{Whites} \times \mathit{Blues} \mid y \approx_P l' @ l'' @ l''' \} \text{ by ass.} \\
\qquad\qquad \lfloor \ni z \\
\qquad\qquad [z' \in \mathit{Reds}, z'' \in \mathit{Whites}, z''' \in \mathit{Blues}, y \approx_P z' @ z'' @ z'''\ \mathit{true}] \\
\qquad\qquad\quad \lceil \mathit{Flag}(\mathsf{cons}(x,y)) \text{ by } \mathit{Colour}\text{-elimination} \\
\qquad\qquad\qquad \lceil \mathit{Colour} \text{ by assumption} \\
\qquad\qquad\qquad \lfloor \ni \mathit{colour}(x) \\
\qquad\qquad\qquad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{red}] \\
\qquad\qquad\qquad\qquad \lceil \mathit{Flag}(\mathsf{cons}(x,y)) \text{ by A1} \\
\qquad\qquad\qquad\qquad \lfloor \ni \langle \mathsf{cons}(x,z'), z'', z''' \rangle \\
\qquad\qquad\qquad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{white}] \\
\qquad\qquad\qquad\qquad \lceil \mathit{Flag}(\mathsf{cons}(x,y) \text{ by A2} \\
\qquad\qquad\qquad\qquad \lfloor \ni \langle z', \mathsf{cons}(x,z''), z''' \rangle \\
\qquad\qquad\qquad [\mathit{colour}(x) =_{\mathit{Colour}} \mathsf{blue}] \\
\qquad\qquad\qquad\qquad \lceil \mathit{Flag}(\mathsf{cons}(x,y) \text{ by A3} \\
\qquad\qquad\qquad\qquad \lfloor \ni \langle z', z'', \mathsf{cons}(x,z''') \rangle \\
\qquad\qquad\quad \lfloor \ni \mathsf{case}_{\mathit{Colour}}(\mathit{colour}(x), \\
\qquad\qquad\qquad\qquad\qquad \langle \mathsf{cons}(x,z'), z'', z''' \rangle, \\
\qquad\qquad\qquad\qquad\qquad \langle z', \mathsf{cons}(x,z''), z''' \rangle, \\
\qquad\qquad\qquad\qquad\qquad \langle z', z'', \mathsf{cons}(x,z''') \rangle) \\
\qquad\quad \lfloor \ni \mathsf{split}_3(z, \\
\qquad\qquad\qquad (z', z'', z''')\mathsf{case}_{\mathit{Colour}}(\mathit{colour}(x),\ \langle \mathsf{cons}(x,z'), z'', z''' \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle z', \mathsf{cons}(x,z''), z''' \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle z', z'', \mathsf{cons}(x,z''') \rangle)) \\
\quad \lfloor \ni \mathsf{listrec}(l, \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle, \\
\qquad\qquad (x,y,z)\mathsf{split}_3(z, \\
\qquad\qquad\qquad (z', z'', z''')\mathsf{case}_{\mathit{Colour}}(\mathit{colour}(x),\ \langle \mathsf{cons}(x,z'), z'', z''' \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle z', \mathsf{cons}(x,z''), z''' \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle z', z'', \mathsf{cons}(x,z''') \rangle))) \\
\lfloor \ni \lambda((l)\ \mathsf{listrec}(l, \langle \mathsf{nil}, \mathsf{nil}, \mathsf{nil} \rangle, \\
\qquad\qquad (x,y,z)\mathsf{split}_3(z, \\
\qquad\qquad\qquad (z', z'', z''')\mathsf{case}_{\mathit{Colour}}(\mathit{colour}(x),\ \langle \mathsf{cons}(x,z'), z'', z''' \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle z', \mathsf{cons}(x,z''), z''' \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle z', z'', \mathsf{cons}(x,z''') \rangle))))
\end{array}
$$

Figure 22.1: Derivation of a program for the Dutch flag

Chapter 23

Specification of abstract data types

During the last 10 years, programmers have become increasingly aware of the practical importance of what Guttag [47] and others have called abstract data type specifications. A module is a generalization of an abstract data type. It is a tuple

$$\langle A_1, A_2, \ldots, A_n \rangle$$

where some A_i are sets and some are functions and constants defined on these sets. It is a dependent tuple in the sense that the set that a component belongs to in general can depend on previous components in the tuple. The classical programming example of a module is a stack which is a set together with some operations defined on the set. An example from mathematics is a group

$$\langle G, *, inv, u \rangle$$

where G is a set, $* \in G \times G \to G$, $inv \in G \to G$, $u \in G$ and certain relationships hold between the components.

In this section, we will show how to *completely* specify modules in type theory using the set of small sets and the dependent sets. We will have a fifth reading of the judgement A *set* :

A is a module specification

and also a fifth reading of $a \in A$:

a is an implementation of the module specification A

By an abuse of notation, we will not distinguish between sets and their codings in a universe. We will therefore write A instead of $\widehat{A}$ and not use the function Set explicitly. It is always obvious from the context if an expression refers to a set or its corresponding element in U.

A simple example is the specification of a stack which in type theory is expressed by the following set:

$$
\begin{array}{l}
(\Sigma StackN \in \mathsf{U}) \\
\quad (\Sigma empty \in StackN) \\
\quad (\Sigma push \in \mathsf{N} \times StackN \to StackN) \\
\quad (\Sigma pop \in StackN \to StackN) \\
\quad (\Sigma top \in StackN \to \mathsf{N}) \\
\qquad (\Pi t \in StackN)(\Pi n \in \mathsf{N}) \\
\qquad\quad ([pop \cdot empty \ =_{StackN} \ empty] \times \\
\qquad\quad\ [pop \cdot (push \cdot \langle n, t\rangle) \ =_{StackN} \ t] \times \\
\qquad\quad\ [top \cdot empty \ =_{StackN} \ 0] \times \\
\qquad\quad\ [top \cdot (push \cdot \langle n, t\rangle) \ =_{\mathsf{N}} \ n])
\end{array}
$$

Using the logical notation for some of the sets, the specification can be reformulated to something that resembles an algebraic specification [47] but with a completely different semantic explanation:

$$
\begin{array}{l}
(\exists StackN \in \mathsf{U}) \\
\quad (\exists empty \in StackN) \\
\quad (\exists push \in \mathsf{N} \times StackN \to StackN) \\
\quad (\exists pop \in StackN \to StackN) \\
\quad (\exists top \in StackN \to \mathsf{N}) \\
\qquad (\forall t \in StackN)(\forall n \in \mathsf{N}) \\
\qquad\quad ([pop \cdot empty \ =_{StackN} \ empty] \ \& \\
\qquad\quad\ [pop \cdot (push \cdot \langle n, t\rangle) \ =_{StackN} \ t] \ \& \\
\qquad\quad\ [top \cdot empty \ =_{StackN} \ 0] \ \& \\
\qquad\quad\ [top \cdot (push \cdot \langle n, t\rangle) \ =_{\mathsf{N}} \ n])
\end{array}
$$

The semantic explanation of this set is an instance of the general schema for explaining the meaning of a set in terms of canonical expressions and their equality relation. The canonical expressions of the set $(\Sigma StackN \in \mathsf{U})\, B_1$ are ordered pairs $\langle st, b_1\rangle$, where $st \in U$ and $b_1 \in B_1[StackN := st]$. Since B_1 is also a Σ-set, the canonical objects of B_1 must also be ordered pairs $\langle es, b_2\rangle$, where $es \in \mathsf{Set}(st)$ and $b_2 \in B_2$, and so on. If each part of the set is analyzed with respect to its semantic explanation, one can see that each member of the set must be equal to a tuple:

$$\langle st, es, pu, po, to, p\rangle$$

where

$$\langle a, ..., b, c\rangle \ \equiv \ \langle a, \langle ..., \langle b, c\rangle\rangle\rangle$$

and

$$
\begin{array}{l}
st \in \mathsf{U} \\
es \in \mathsf{Set}(st) \\
pu \in \mathsf{N} \times \mathsf{Set}(st) \to \mathsf{Set}(st) \\
po \in \mathsf{Set}(st) \to \mathsf{Set}(st) \\
to \in \mathsf{Set}(st) \to \mathsf{N} \\
p \in (\forall t \in \mathsf{Set}(st))(\forall n \in \mathsf{N}) \ [po \cdot es \ =_{\mathsf{Type}(st)} \ es] \times [\ldots] \times [\ldots] \times [\ldots]
\end{array}
$$

Notice that the first component is an element in the set of small sets. This is of course a limitation, we would like to allow an arbitrary set. This could be done, but then we must use something like a Σ-type-forming operation on the level of types. The last judgement expresses that st, es, pu and to have the properties required for the stack operations. So the semantics of the specification is given in terms of the canonical expressions of the set, or, in other words, in terms of the *correct (canonical) implementations* of the specification. The specification expresses requirements on implementations of the specification and it is, of course, possible to have requirements which cannot be satisfied. In type theory, a specification with such requirements does not cause any harm; the result is just that it is impossible to find an implementation for it. It is sometimes even possible to show that a specification never can be satisfied by proving it equivalent to the empty set.

In the stack specification given above, we specified modules which are equal to objects:

$$\langle st, es, pu, po, to, p\rangle$$

where the last component

$$p \in (\forall s \in \mathsf{Set}(st))(\forall n \in \mathsf{N})[po \cdot es =_{\mathsf{Set}(st)} es] \times [\ldots] \times [\ldots] \times [\ldots]$$

only contains information obtained from the proof that the previous components of the tuple have the properties required for a stack. This component is computationally uninteresting, and if we use a subset instead of a Σ-set we have a specification of a stack without the irrelevant last component:

$$\begin{array}{l}
(\Sigma\, StackN \in U) \\
\quad (\Sigma empty \in StackN) \\
\quad (\Sigma push \in \mathsf{N} \times StackN \to StackN) \\
\quad (\Sigma pop \in StackN \to StackN) \\
\quad \{top \in StackN \to \mathsf{N} \mid \\
\qquad (\forall t \in StackN)(\forall n \in \mathsf{N}) \\
\qquad\quad ([pop \cdot empty =_{StackN} empty]\ \& \\
\qquad\quad\ \ [pop \cdot (push \cdot \langle n, t\rangle) =_{StackN} t]\ \& \\
\qquad\quad\ \ [top \cdot empty =_{StackN} 0]\ \& \\
\qquad\quad\ \ [top \cdot (push \cdot \langle n, t\rangle) =_{\mathsf{N}} n])\}
\end{array}$$

As expected, this is a specification of a module which is equal to a 5-tuple:

$$\langle st, es, pu, po, to\rangle$$

whose components have the properties we require for a stack.

A small problem with this approach is that the equality we get between stacks is the equality of the implementation of the stack. At the same time as we specify a stack we would like to have the possibility to express that two stacks are considered equal when they are observationally equal, i.e. when they cannot be distinguished by any operation defined on stacks. This needs something like a quotient set forming operation, which redefines the equality on a set. This would be a major change in the set theory and we will not explore it further here.

23.1 Parameterized modules

Specifications of parameterized modules, such as a stack of A elements, for an arbitrary set A, are neatly handled in type theory. The parameterized module is specified by means of the Π-set former. The specification is the set

$$
\begin{array}{ll}
\mathit{STACK} \equiv & \\
\quad (\Pi\, A \in \mathsf{U}) & \text{in logical notation: } (\forall A \in \mathsf{U}) \\
\quad\quad (\Sigma \mathit{Stack} \in U) & \\
\quad\quad\quad (\Sigma \mathit{empty} \in \mathit{Stack}) & \\
\quad\quad\quad (\Sigma \mathit{push} \in \mathsf{Set}(A) \times \mathit{Stack} \to \mathit{Stack}) & \\
\quad\quad\quad (\Sigma \mathit{pop} \in \mathit{Stack} \to \mathit{Stack}) & \\
\quad\quad\quad\quad \vdots &
\end{array}
$$

The canonical expressions of a set $(\Pi\, A \in \mathsf{U})\, B$ are functions $\lambda x.s$, such that whenever they are applied to an object $C \in \mathsf{U}$, they will yield an object in the set $B[A := C]$. This means that an implementation of the specification *STACK* is a function, which when applied to an element A of the set U returns an implementation of a stack of A elements. So, if $st \in \mathit{STACK}$, then $st \cdot \widehat{\mathsf{N}}$ is a module of stacks of natural numbers and $st \cdot \widehat{\mathsf{N}} \widehat{\times} \widehat{\mathsf{N}}$ is a module of stacks of pairs of natural numbers. These modules can then be decomposed in the same way as earlier to get their components.

23.2 A module for sets with a computable equality

The module

$$\langle X, e \rangle$$

is a computable equality if X is (a coding of) a set and e is a boolean function computing the equality defined on X, i.e.

$$e \cdot \langle x, y \rangle \ =_{\mathsf{Bool}}\ \mathit{true}, \text{ if and only if } x =_X y$$

This can be specified by the set

$$
\begin{array}{l}
\mathit{CompEq} \equiv \\
\quad (\Sigma X \in \mathsf{U}) \\
\quad\quad \{e \in X \times X \to \mathsf{Bool} \mid \\
\quad\quad\quad (\forall y, z \in X)([e \cdot \langle y, z \rangle \ =_{\mathsf{Bool}}\ \mathit{true}\,] \Leftrightarrow\ [y\ =_X\ z])\}
\end{array}
$$

Notice that the specification expresses exactly the requirements on the function e, an arbitrary boolean valued function will not do!

We can now use this module specification to define a module *FSET* for finite sets:

$$
\begin{array}{l}
FSET \quad \equiv \\
\quad (\Pi A \in CompEq) \\
\qquad (\Sigma FSet \in \mathsf{U}) \\
\qquad\quad (\Sigma eset \in FSet) \\
\qquad\quad (\Sigma add \in A_1 \times FSet \to FSet) \\
\qquad\quad \{mem \in A_1 \times FSet \to Bool \mid \\
\qquad\qquad (\forall t \in FSet)(\forall a \in A_1)(\forall b \in A_1) \\
\qquad\qquad\quad ([mem \cdot \langle a, eset \rangle \ =_{\mathsf{Bool}} \ false] \ \& \\
\qquad\qquad\quad\ [mem \cdot \langle a, add \cdot \langle b, t \rangle \rangle \ =_{\mathsf{Bool}} \\
\qquad\qquad\qquad if \ A_2 \cdot \langle a, b \rangle \ then \ \mathsf{true} \ else \ mem \cdot \langle a, t \rangle]\}
\end{array}
$$

An object of this set is a function which when applied to an object $\langle A, e\rangle$ in *CompEq* yields an implementation of $FSET$ for the particular arguments chosen. Note how the Π set-former is used for specifying a dependent function set, in which the elements are functions for which the value of the first arguments determines which set the second argument should be a member of.

Bibliography

[1] Peter Aczel. The Type Theoretic Interpretation of Constructive Set Theory. In *Logic Colloquium '77*, pages 55–66, Amsterdam, 1978. North-Holland Publishing Company.

[2] Peter Aczel. The Type Theoretic Interpretation of Constructive Set Theory: Choice Principles. In *The L. E. J. Brouwer Centenary Symposium*, pages 1–40. North-Holland Publishing Company, 1982.

[3] Peter Aczel. The Type Theoretic Interpretation of Constructive Set Theory: Inductive Definitions. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishing B.V., 1986.

[4] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127 – 141, 1989.

[5] Roland Backhouse. Algorithm Development in Martin-Löf's Type Theory. Technical report, University of Essex, 1985.

[6] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[7] Roland C. Backhouse. *Program Construction and Verification*. Prentice-Hall, 1986.

[8] Joseph L. Bates and Robert L. Constable. Proofs as Programs. *ACM Trans. Prog. Lang. Sys.*, 7(1):113–136, 1985.

[9] M. J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, New York, 1985.

[10] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.

[11] Errett Bishop. Mathematics as a numerical language. In Myhill, Kino, and Vesley, editors, *Intuitionism and Proof Theory*, pages 53–71, Amsterdam, 1970. North Holland.

[12] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer-Verlag, New York, 1985.

[13] Bror Bjerner. Verifying some Efficient Sorting Strategies in Type Theory. PMG Memo 26, Chalmers University of Technology, S–412 96 Göteborg, January 1985.

[14] Bror Bjerner. *Time Complexity of Programs in Type Theory.* PhD thesis, Dept. of Computer Science, University of Göteborg, Göteborg, Sweden, January 1989.

[15] R. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[16] L. E. J. Brouwer. *Collected Works*, volume 1. North-Holland Publishing Company, Amsterdam, 1975. Ed. A. Heyting.

[17] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley Publishing Company, Reading, Mass., 1975.

[18] R. M. Burstall, D. B. McQueen, and D. T. Sannella. Hope: An Experimental Applicative Language. In *Proceedings of the 1980 ACM Symposium on Lisp and Functional Programming*, pages 136–143, Stanford, CA, August 1980.

[19] Rod Burstall. Proving Properties of Programs by Structural Induction. *Computer Journal*, 12(1):41–48, 1969.

[20] P. Chisholm. Derivation of a Parsing Algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, 8:1–42, 1987.

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

[22] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[23] R. L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of IFIP Congress*, pages 229–233, Ljubljana, 1971. North-Holland.

[24] R. L. Constable and M. J. O'Donnell. *A Programming Logic.* Winthrop Publishing Inc., Cambridge, Massachusetts, 1978.

[25] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[26] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

[27] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

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

[29] O-J Dahl, E. W. Dijkstra, and C. A. R. Hoare. Structured Programming. *Academic Press*, 1972.

[30] N. G. de Bruijn. The Mathematical Language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61, Versailles, France, 1968. IRIA, Springer-Verlag.

[31] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, New York, 1980. Academic Press.

[32] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[33] M. Dummett. *Elements of intuitionism.* Clarendon Press, Oxford, 1977.

[34] P. Dybjer, B. Nordström, K. Petersson, and J. Smith (eds.). Proceedings of the Workshop of Specification and Derivation of Programs. Technical Report PMG-18, Programming Methodology Group, Chalmers University of Technology, Göteborg, June 1985.

[35] P. Dybjer, B. Nordström, K. Petersson, and J. Smith (eds.). Proceedings of the Workshop on Programming Logics. Technical Report PMG-37, Programming Methodology Group, Chalmers University of Technology, Göteborg, June 1987.

[36] Peter Dybjer. Inductively Defined Sets in Martin-Löf's Type Theory. In *Proceedings of the Workshop on General Logic, Edinburgh, February 1987*, number ECS-LFCS-88-52 in LFCS Report Series, 1988.

[37] Roy Dyckhoff. Category Theory as an Extension of Martin-Löf's type theory. Technical Report CS/85/3, University of St. Andrews, 1985.

[38] J. E. Fenstad, editor. *Proceedings of the Second Scandinavian Logic Symposium.* North-Holland Publishing Company, 1971.

[39] G. Frege. Function and concept. In P. Geach and M. Black, editors, *Translations from the Philosophical Writings of Gottlob Frege.* Blackwell, Oxford, 1967.

[40] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen.* North-Holland Publishing Company, Amsterdam, 1969. Ed. E.Szabo.

[41] C. Goad. *Computational Uses of the Manipulation of Formal Proofs.* PhD thesis, Computer Science Department, Stanford University, August 1980.

[42] C. Goad. Proofs as Descriptions of Computation. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 39–52. Les Arcs, France, Springer-Verlag, 1980.

[43] Kurt Gödel. Über eine bisher noch nicht benutze erweitrung des finiten standpunktes. *Dialectica*, 12, 1958.

[44] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[45] S. Goto. Program synthesis from natural deduction proofs. In *Proceedings of IJCAI*, Tokyo, 1979.

[46] David Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[47] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, Department of Computer Science, University of Toronto, 1975.

[48] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. In *Proceedings of the Symposium on Logic in Computer Science*, pages 194–204, Ithaca, New York, June 1987.

[49] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic*. Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1988.

[50] Arend Heyting. *Intuitionism: An Introduction*. North-Holland, Amsterdam, 1956.

[51] C. A. R. Hoare. Recursive Data Structures. *International Journal of Computer and Information Sciences*, 4(2):105–132, 1975.

[52] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[53] Gérard Huet. Formal Structures for Computation and Deduction. Lecture Notes for International Summer School on Logic Programming and Calculi of Discrete Design, Marktoberdorf, Germany, May 1986.

[54] Gérard Huet. Induction Principles Formalized in the Calculus of Constructions. In *Proceedings of TAPSOFT 87*, pages 276–286. Springer-Verlag, LNCS 249, March 1987.

[55] Gérard Huet. A Uniform Approach to Type Theory. Technical report, INRIA, 1988.

[56] Gérard Huet. The Constructive Engine. Technical report, INRIA, 1989.

[57] R. J. M. Hughes. Why Functional Programming Matters. PMG Report 16, Department of Computer Sciences, Chalmers University of Technology, S–412 96 Göteborg, November 1984.

[58] L. S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system*, volume 83 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1979.

[59] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

[60] A. N. Kolmogorov. Zur Deutung der intuitionistischen Logik. *Matematische Zeitschrift*, 35:58–65, 1932.

[61] H. Lauchli. An abstract notion of realizability for which intuitionistic predicate logic is complete. In Myhill, Kino, and Vesley, editors, *Intuitionism and Proof Theory*. North Holland, Amsterdam, 1970.

[62] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Trans. Prog. Lang. Sys.*, 2:90–121, 1980.

[63] Zohar Manna and Richard Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[64] Per Martin-Löf. A Theory of Types. Technical Report 71–3, University of Stockholm, 1971.

[65] Per Martin-Löf. Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions. In J. E. Fenstad, editor, *In Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland Publishing Company, 1971.

[66] Per Martin-Löf. An Intuitionistic Theory of Types. Technical report, University of Stockholm, 1972.

[67] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *In Proceedings of the Third Scandinavian Logic Symposium*, pages 81–109. North-Holland Publishing Company, 1975.

[68] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118, Amsterdam, 1975. North-Holland Publishing Company.

[69] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[70] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[71] Per Martin-Löf. Truth of a Proposition, Evidence of a Judgement, Validity of a Proof. Transcript of a talk at the workshop Theories of Meaning,Centro Fiorentino di Storia e Filosofia della Scienza, Villa di Mondeggi, Florence, June 1985.

[72] R. Milner. Standard ML Core Language. Technical Report CSR-168, University of Edinburgh, Internal report, 1984.

[73] John Mitchell and Gordon Plotkin. Abstract types have existential type. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, pages 37–51, New York, 1985.

[74] Christine Mohring. Algorithm Delvelopment in the Calculus of Constructions. In *Proceedings Symposium on Logic in Computer Science*, pages 84–91, Cambridge, Mass., 1986.

[75] Bengt Nordström. Programming in Constructive Set Theory: Some examples. In *Proceedings 1981 Conference on Functional Languages and Computer Architecture.* ACM, October 1981.

[76] Bengt Nordström. Multilevel Functions in Type Theory. In *Proceedings of a Workshop on Programs as Data Objects*, volume 217, pages 206–221, Copenhagen, October 1985. Springer-Verlag, Lecture Notes in Computer Science.

[77] Bengt Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

[78] Bengt Nordström and Kent Petersson. Types and Specifications. In R. E. A. Mason, editor, *Proceedings of IFIP 83*, pages 915–920, Amsterdam, October 1983. Elsevier Science Publishers.

[79] Bengt Nordström and Jan Smith. Propositions, Types and Specifications in Martin-Löf's Type Theory. *BIT*, 24(3):288–301, October 1984.

[80] Christine Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions*. PhD thesis, L'Universite Paris VII, 1989.

[81] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[82] Lawrence Paulson. Verifying the unification algorithmn in LCF. *Science of Computer Programming*, 5:143–169, 1985.

[83] Lawrence C. Paulson. Natural Deduction Proof as Higher-Order Resolution. Technical report 82, Universtiy of Cambridge Computer Laboratory, Cambridge, 1985.

[84] Lawrence C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[85] Lawrence C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.

[86] Kent Petersson. A Programming System for Type Theory. PMG report 9, Chalmers University of Technology, S–412 96 Göteborg, 1982, 1984.

[87] Kent Petersson and Jan Smith. Program Derivation in Type Theory: A Partitioning Problem. *Computer Languages*, 11(3/4):161–172, 1986.

[88] Kent Petersson and Dan Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Proceedings of the 1989 Conference on Category Theory and Computer Science, Manchester, U. K.*, volume 389. Lecture Notes in Computer Science, Springer-Verlag, 1989.

[89] J. A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *ACM*, 12:23–41, 1965.

[90] A. Salvesen and J. M. Smith. The Strength of the Subset Type in Martin-Löf's type theory. In *Proceedings of LICS '88*, Edinburgh, 1988. IEEE.

[91] Anne Salvesen. Polymorphism and Monomorphism in Martin-Löf's Type Theory. Technical report, Norwegian Computing Center, P.b. 114, Blindern, 0316 Oslo 3, Norway, December 1988.

[92] Anne Salvesen. *On Information Discharging and Retrieval in Martin-Löf's Type Theory*. PhD thesis, Institute of Informatics, University of Oslo, 1989.

[93] M. Sato. Towards a mathematical theory of program synthesis. In *Proceedings of IJCAI*, Tokyo, 1979.

[94] W. L. Scherlis and D. Scott. First Steps Toward Inferential Programming. In *Proceedings IFIP Congress*, Paris, 1983.

[95] David Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[96] Peter Schroeder-Heister. Generalized Rules for Operators and the Completeness of the Intuitionistic Operators &, $\vee$, $\supset$, $\perp$, $\forall$, $\exists$. In Richter et al, editor, *Computation and Proof Theory*, volume 1104 of *Lecture Notes in Mathematics*. Springer-Verlag, 1984.

[97] Dana Scott. Constructive validity. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, Berlin, 1970.

[98] J. M. Smith. On a Nonconstructive Type Theory and Program Derivation. In *The Proceedings of Conference on Logic and its Applications, Bulgaria*. Plenum Press, 1986.

[99] Jan M. Smith. The Identification of Propositions and Types in Martin-Löf's Type Theory. In *Foundations of Computation Theory, Proceedings of the Conference*, pages 445–456, 1983.

[100] Jan M. Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.

[101] Jan M. Smith. The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes. *Journal of Symbolic Logic*, 53(3), 1988.

[102] Jan M. Smith. Propositional Functions and Families of Types. *Notre Dame Journal of Formal Logic*, 30(3), 1989.

[103] Sören Stenlund. *Combinators, λ-terms, and Proof Theory*. D. Reidel, Dordrecht, The Netherlands, 1972.

[104] Göran Sundholm. Constructions, proofs and the meaning of the logical constants. *The Journal of Philosophical Logic*, 12:151–172, 1983.

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

[106] S. Takasu. Proofs and Programs. *Proceedings of the 3rd IBM Symposium on Mathematical Foundations of Computer Science*, 1978.

[107] A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, New York, 1973.

[108] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics. An Introduction*, volume I. North-Holland, 1988.

[109] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics. An Introduction*, volume II. North-Holland, 1988.

[110] D. A. Turner. SASL Language Manual. Technical report, University of St. Andrews, 1976.

[111] Å. Wikström. *Functional Programming Using Standard ML*. Prentice-Hall, London, 1987.

Index

Appendix A

Constants and their arities

A.1 Primitive constants in the set theory

Name	Arity	Can/Noncan	Type
0	$\mathbf{0}$	canonical	N
succ	$\mathbf{0}\twoheadrightarrow\mathbf{0}$	canonical	N
natrec	$\mathbf{0}\otimes\mathbf{0}\otimes(\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	noncanonical	N
nil	$\mathbf{0}$	canonical	List(A)
cons	$\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$	canonical	List(A)
listrec	$\mathbf{0}\otimes\mathbf{0}\otimes(\mathbf{0}\otimes\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	noncanonical	List(A)
λ	$(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	canonical	$A \to B,\ \Pi(A,B)$
apply	$\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$	noncanonical	$A \to B,\ \Pi(A,B)$
funsplit	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	noncanonical	$A \to B,\ \Pi(A,B)$
$\langle\rangle$	$\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$	canonical	$A \times B, \Sigma(A,B)$
split	$\mathbf{0}\otimes(\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	noncanonical	$A \times B, \Sigma(A,B)$
inl	$\mathbf{0}\twoheadrightarrow\mathbf{0}$	canonical	$A+B$
inr	$\mathbf{0}\twoheadrightarrow\mathbf{0}$	canonical	$A+B$
when	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	noncanonical	$A+B$
sup	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	canonical	W(A,B)
wrec	$\mathbf{0}\otimes(\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	noncanonical	W(A,B)
tree	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	canonical	Tree(A,B,C,d)
treerec	$\mathbf{0}\otimes(\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	noncanonical	Tree(A,B,C,d)

Name	Arity	Can/Noncan	Type
id	$\mathbf{0}$	canonical	$\mathsf{Id}(A, a, b)$
idpeel	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	noncanonical	$\mathsf{Id}(A, a, b)$
$\widehat{\{i_1, \ldots, i_n\}}$	$\mathbf{0}$	canonical	U
$\widehat{\mathsf{N}}$	$\mathbf{0}$	canonical	U
$\widehat{\mathsf{List}}$	$\mathbf{0}\twoheadrightarrow\mathbf{0}$	canonical	U
$\widehat{\mathsf{Id}}$	$\mathbf{0}\otimes\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$	canonical	U
$\widehat{+}$	$\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$	canonical	U
$\widehat{\Pi}$	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	canonical	U
$\widehat{\Sigma}$	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	canonical	U
$\widehat{\mathsf{W}}$	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$	canonical	U
urec	(se page 93)	noncanonical	U

A.2 Set constants

Name	Arity
$\{i_1, \ldots, i_n\}$	$\mathbf{0}$
N	$\mathbf{0}$
List	$\mathbf{0}\twoheadrightarrow\mathbf{0}$
Π	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$
$\rightarrow$	$\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$
Σ	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$
$\times$	$\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$
$+$	$\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$
Id	$\mathbf{0}\otimes\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0}$
W	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$
Tree	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0})\otimes(\mathbf{0}\otimes\mathbf{0}\otimes\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}\twoheadrightarrow\mathbf{0}$
U	$\mathbf{0}$
$\{\mid\}$	$\mathbf{0}\otimes(\mathbf{0}\twoheadrightarrow\mathbf{0})\twoheadrightarrow\mathbf{0}$

Appendix B

Operational semantics

The following is a formal description of the operational semantics of the polymorphic set theory. We use the notation $a \Rightarrow b$ to mean that the program a computes to the value b. We start with programs on constructor form, which already are evaluated, then we continue with programs on selector form.

$$i_1 \Rightarrow i_1 \qquad \ldots \qquad i_n \Rightarrow i_n$$

$$0 \Rightarrow 0 \qquad \mathsf{succ}(d) \Rightarrow \mathsf{succ}(d) \qquad \mathsf{nil} \Rightarrow \mathsf{nil}$$

$$\mathsf{cons}(d,e) \Rightarrow \mathsf{cons}(d,e) \qquad \lambda(c) \Rightarrow \lambda(c) \qquad \mathsf{inl}(d) \Rightarrow \mathsf{inl}(d)$$

$$\mathsf{inr}(e) \Rightarrow \mathsf{inr}(e) \qquad \langle c,d\rangle \Rightarrow \langle c,d\rangle \qquad \mathsf{sup}(c,d) \Rightarrow \mathsf{sup}(c,d)$$

$$\frac{a \Rightarrow i_1 \qquad b_1 \Rightarrow q}{\mathsf{case}_n(a, b_1, \ldots, b_n) \Rightarrow q} \qquad \frac{a \Rightarrow i_n \qquad b_n \Rightarrow q}{\mathsf{case}_n(a, b_1, \ldots, b_n) \Rightarrow q}$$

$$\frac{a \Rightarrow 0 \qquad b \Rightarrow q}{\mathsf{natrec}(a,b,c) \Rightarrow q} \qquad \frac{a \Rightarrow \mathsf{succ}(d) \qquad c(d, \mathsf{natrec}(d,b,c)) \Rightarrow q}{\mathsf{natrec}(a,b,c) \Rightarrow q}$$

$$\frac{a \Rightarrow \mathsf{nil} \qquad b \Rightarrow q}{\mathsf{listrec}(a,b,c) \Rightarrow q} \qquad \frac{a \Rightarrow \mathsf{cons}(d,e) \qquad c(d,e,\mathsf{listrec}(e,b,c)) \Rightarrow q}{\mathsf{listrec}(a,b,c) \Rightarrow q}$$

$$\frac{a \Rightarrow \lambda(c) \qquad c(b) \Rightarrow q}{\mathsf{apply}(a,b) \Rightarrow q} \qquad \frac{a \Rightarrow \lambda(c) \qquad b(c) \Rightarrow q}{\mathsf{funsplit}(a,b) \Rightarrow q}$$

$$\frac{a \Rightarrow \mathsf{inl}(d) \qquad b(d) \Rightarrow q}{\mathsf{when}(a,b,c) \Rightarrow q} \qquad \frac{a \Rightarrow \mathsf{inr}(e) \qquad c(e) \Rightarrow q}{\mathsf{when}(a,b,c) \Rightarrow q}$$

$$\frac{a \Rightarrow \langle c,d\rangle \qquad b(c,d) \Rightarrow q}{\mathsf{split}(a,b) \Rightarrow q} \qquad \frac{a \Rightarrow \mathsf{sup}(c,d) \qquad b(c,d,(x)\mathsf{wrec}(d(x),b)) \Rightarrow q}{\mathsf{wrec}(a,b) \Rightarrow q}$$

B.1 Evaluation rules for noncanonical constants

The following is an informal description of the operational semantics of type theory. Only the rules for the selectors are given, since each expression on constructor form is already evaluated. Let x be a variable and a, b, c, d and e expressions of suitable arity.

Expression		Computation rule
$\mathsf{case}_n(a, b_1, \ldots, b_n)$	1.	Evaluate a to canonical form
where $a \in \{i_1, \ldots, i_n\}$	2a.	If the value is of the form i_1
		then continue with b_1
	2b.	If the value is of the form i_2
		then continue with b_2
	...	
	2u.	If the value is of the form i_n
		then continue with b_n
$\mathsf{natrec}(a, b, c)$	1.	Evaluate a to canonical form
	2a.	If the value is of the form $\mathsf{0}$
		then continue with b
	2b.	If the value is of the form $\mathsf{succ}(d)$
		then continue with $c(d, \mathsf{natrec}(d, b, c))$
$\mathsf{listrec}(a, b, c)$	1.	Evaluate a to canonical form
	2a.	If the value is of the form nil
		then continue with b
	2b.	If the value is of the form $\mathsf{cons}(d, e)$
		then continue with $c(d, e, \mathsf{listrec}(e, b, c))$
$\mathsf{apply}(a, b)$	1.	Evaluate a to canonical form
	2.	If the value is of the form $\lambda(c)$
		then continue with $c(b)$
$\mathsf{funsplit}(a, b)$	1.	Evaluate a to canonical form
	2.	If the value is of the form $\lambda(c)$
		then continue with $b(c)$.
$\mathsf{split}(a, b)$	1.	Evaluate a to canonical form
	2.	If the value is of the form $\langle c, d \rangle$
		then continue with $b(c, d)$

Expression		Computation rule
$\mathsf{when}(a, b, c)$	1.	Evaluate a to canonical form
	2a.	If the value is of the form $\mathsf{inl}(d)$ then continue with $b(d)$
	2b.	If the value is of the form $\mathsf{inr}(e)$ then continue with $c(e)$
$\mathsf{wrec}(a, b)$	1.	Evaluate a to canonical form
	2.	If the value is of the form $\mathsf{sup}(c, d)$ then continue with $b(c, d, (x)\mathsf{wrec}(d(x), b))$