Core Computational Type Theory

Robert Constable

1 Introduction

These notes for CS5860 present the basic types of the core Nuprl type theory. They are based on the article The Structure of Nuprl's Type Theory presented in the book *Logic of Computation* edited by M. Broy and H. Schwichtenberg, Springer-Verlag, 1997, pages 123-156. The full paper is available at www.nuprl.org under publications in 1997.

A naive account of Core Type Theory is especially simple, and I think it provides a bridge to understanding the more daunting axiomatization of the Nuprl type theory which is used in all of the on-line libraries. This article presents the Core Theory and relates it to a corresponding part of Nuprl. Comparisons are made to set theory as a way to motivate the concepts.

Another readable account of the core theory appears in my article Naive Computational Type Theory in the book *Proof and System Reliability*, edited by H. Schwichtenberg and R. Steinbruggen, NATO Science Press, 2002, pages 213-259.

1.1 Role of Type Theory

Type theory has emerged as the native language of the most widely used interactive theorem provers and is the default formalism for reasoning about programming languages. The types used in modern implemented type theories such as Agda, Coq, and Nuprl are exerting an influence on the design of modern programming languages because the value of rich type systems has become relevant to more programming languages and formal methods research. Dependent types are especially important, and we discuss them here in a separate section.

1.2 Types and Sets

The informal language of mathematics uses types and sets, a set being on kind of type. However, when mathematicians want to be very rigorous, then tend to rely on pure set theory. That might be changing as proof assistants have a larger impact on mathematical research.

The Core Theory needed for Nuprl involves only six type constructors: product, disjoint union, function space, inductive type, set type, and quotient type. We need some primitive types as well: void, unit, Type, and Prop. I have chosen to add co-inductive types as well although they are not in Nuprl 4.

2 The Core Theory

2.1 Primitive Types

void is a type with no elements
unit is a type with one element, denoted •

There will be other primitive types introduced later. Notice, in set theory we usually have only one primitive set, some infinite set (usually ω). Sometimes the empty set, ϕ, is primitive as well, although it is definable by separation from ω.

Compound Types We build new types using *type constructors*. These tell us how to construct various kinds of objects. (In pure set theory, there is only one kind, *sets*).

The type constructors we choose are motivated both by mathematical and computational considerations. So we will see a tight relationship to the notion of *type* in programming languages. The notes by C.A.R. Hoare, *Notes on Data Structuring* [23], make the point well.

2.2 Cartesian Products

If A and B are types, then so is their product, written $A \times B$. There will be many *formation rules* of this form, so we adopt a simple convention for stating them. We write

$$\frac{A \text{ is a } Type \quad B \text{ is a } Type}{A \times B \ \text{ is a } Type.}$$

The elements of a product are pairs, $\langle a, b\rangle$. Specifically if a belongs to A and b belongs to B, then $\langle a, b\rangle$ belongs to $A \times B$. We abbreviate this by writing

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

In programming languages these types are generalized to n-ary products, say $A_1 \times A_2 \times \ldots \times A_n$. They are the basis for defining *records*.

We say that $\langle a, b\rangle = \langle c, d\rangle$ in $A \times B$ iff $a = c$ in A and $b = d$ in B.

In set theory, equality is uniform and built-in, but in type theory we define equality with each constructor, either built-in (as in Nurpl) or by definition as in this core theory.

There is essentially only one way to decompose pairs. We say things like, "take the first elements of the pair P," symbolically we might say *first*(P) or 1*of*(P). We can also "take the second element of P," *second*(P) or 2*of*(P).

2.3 Function Space

We use the words "function space" as well as "function type" for historical reasons. If A and B are types, then $A \rightarrow B$ is the type of *computable functions* from A to B. These are given by rules which are defined for each a in A and which produce a *unique value.* We summarize by

$$\frac{A \text{ is a } Type \quad B \text{ is a } Type}{A \rightarrow B \;\text{ is a } Type}$$

The function notation we use informally comes from mathematics texts, e.g. Bourbaki's *Algebra.* We write expressions like $x \mapsto b$ or $x \overset{f}{\mapsto} b$; the latter gives a name to the function. For example, $x \mapsto x^2$ is the squaring function on numbers.

If b computes to an element of B when x has value a in A for each a, then we say $(x \mapsto b) \epsilon A \rightarrow B$. We will also use *lambda notation*, $\lambda(x.b)$ for $x \mapsto b$. The informal rule for typing a function $\lambda(x.b)$ is to say that $\lambda(x.b) \in A \rightarrow B$ provided that when x is of type A, b is of type B. We can express these *typing judgments* in the form $x : A \vdash b \in B$. The phrase $x{:}A$ *declares* x to be of type A. The typing rule is then

$$\frac{x : A \vdash b \in B}{\vdash \lambda(x.b) \in A \rightarrow B}$$

If f, g are functions, we define their equality as

$$\boxed{f = g \quad \text{iff } f(x) = g(x) \quad \text{for all} \quad x \text{ in } A.}$$

If f is a function from A to B and $a \epsilon A$, we write $f(a)$ for the value of the function.

2.4 Disjoint Unions (also called Discriminated Unions)

Forming the union of two sets, say $x \cup y$, is a basic operation in set theory. It is basic in type theory as well, *but* for computational purposes, we want to discriminate based on which type an element is in. To accomplish this we put tags on the elements to keep them disjoint. Here we use *inl* and *inr* as the tags.

$$\frac{A \text{ is a } Type \quad B \text{ is a } Type}{A + B \ \text{ is a } Type}$$

The membership rules are

$$\frac{a \;\epsilon\; A}{inl(a) \;\epsilon\; A + B} \qquad \frac{b \;\epsilon\; B}{inr(b) \;\epsilon\; A + B}$$

We say that $inl(a) = inl(a')$ iff $a = a'$ and likewise for $inr(b)$.

We can now use a case statement to detect the tags and use expressions like

$$\begin{array}{lll} \text{if } x = inl(z) & \text{then} \ldots & \text{some expression in } z \ldots \\ \text{if } x = inr(z) & \text{then} \ldots & \text{some expression in } z \ldots \end{array}$$

in defining other objects. The test for $inl(z)$ or $inr(z)$ is computable. There is an operation called *decide* that discriminates on the type tags. The typing rule and syntax for it are given in terms of a typing judgment of the form $E \vdash t \in T$ where is a list of declarations of the form $x_1 : A_1, \ldots, x_n : A_n$ called a *typing environment*. The A_i are types and x_i are variables declared to be of type A_i. The rule is

$$\frac{E \vdash d \in A + B \quad E, u : A \vdash t_1 \in T \quad E, v : B \vdash t_2 \in T}{E \vdash decide(d; u.t_1; v.t_2) \in T}$$

2.5 Subtyping

Intuitively, A is a subtype of B iff every element of A is also an element of B; we write this relation as $A \subseteq B$. Clearly $\phi \subseteq A$ for any A. Notice that A is not a subtype of $A + B$ since the elements of A in $A + B$ have the form $inl(a)$. We have these properties however

$$\frac{A \subseteq A' \qquad B \subseteq B'}{\begin{array}{c} A \times B \subseteq A' \times B' \\ A + B \subseteq A' + B' \\ A' \to B \subseteq A \to B' \end{array}}$$

For $A \subseteq B$ we also require that $a = a'$ in A implies $a = a'$ in B.

In order to use the elements of a co-inductive type, we need some way to force the generator to produce an element. This is done with a form called *out*. It has this property:

$$\boxed{\begin{array}{ll} \text{If} & t \in \nu X.F(X) \\ \text{then} & out(t) \in F(\nu X.F(X)). \end{array}}$$

This form obeys the following *computation rule*.

$$\boxed{\begin{array}{ll} \text{out}(\nu\text{–}ind(d; f, z.b)) & \\ \text{evaluates to} & b[d/z, (y \mapsto \nu\text{–}ind(y; f, z.b))/f]. \end{array}}$$

2.8 Subset Types and Logic

One of the most basic and characteristic types of Nuprl is the so-called *set type* or *subset type*, written $\{x\!:\!A \mid P(x)\}$ and denoting the subtype of A consisting of exactly those elements satisfying condition P. This concept is closely related to the set theory notion written the same way and denoting the "subset" of A satisfying the predicate P. In axiomatic set theory the existence of this set is guaranteed by the *separation axiom*. The idea is that the predicate P separates a subset of A as in the example of say the prime numbers, $\{x\!:\!\mathbb{N} \mid prime(x)\}$.

To understand this type, we need to know something about predicates. In axiomatic set theory the predicates allowed are quite restrictive; they are built from the atomic membership predicate, $x \in y$ using the first order predicate calculus over the universe of sets. In type theory we allow a different class of predicates — those involving predicative higher-order logic in a sense. This topic is discussed in many articles and books on type theory [33, 13, 36, 43, 12, 11] and is beyond the scope of this article, so here we will just assume that the reader is familiar with one account of propositions-as-types or representing logic in type theory.

The Nuprl style is to use the type of propositions, denoted *Prop*. This concept is stratified into $Prop_i$ as in *Principia Mathematica*, and it is related by the propositions-as-types principle to the large types such as *Type*. $Prop_i$ are indeed considered to be a "large types." (See [11] for an extensive discussion of this notion.) For the work we do here we only need the notions of *Type* and *Prop* which we take to be $Type_1$ and $Prop_1$ in the full Nuprl theory.

The point of universes or large types is that they allow us to use expressions like $Type$, $A \rightarrow Type$, $A \times Type$, $Type \rightarrow Type$, etc. as if they were types except that $Type \in Type$ is not allowed. Instead, when we need such a notion we must attend to the level numbers used to stratify the notions of $Type$ and $Prop$. We can say $Type_i \in Type_j$ if $i < j$, and $Type_i \subseteq Type_j$ when $i < j$.

Thus the objects of mathematics we consider include *propositions*. For example, $0 = 0$ in N and $0 < 1$ are true propositions about the type N, so $0 < 1 \in Prop$ and $0 < 0 \in Prop$. We also consider *propositional forms* such as $x =_N y$ or $x < y$. These are sometimes called *predicates*.

Propositional functions on a type A are elements of the type $A \rightarrow Prop$.

We also need types that can be restricted by predicates such as $\{x = N | x = 0$ or $x = 1\}$. This type behaves like the *Booleans*.

The general formation rule is this. If A is a type and $B : A \rightarrow Prop$, then $\{x : A | B(x)\}$ is a *subtype of* A.

The elements of $\{x : A | B(x)\}$ are those a in A for which $B(a)$ is true.

Sometimes we state the formation in terms of predicates, so if P is a proposition for any x in A, then $\{x : A | P\}$ is a subtype of A. We Clearly have $\{x : A | P\} \subseteq A$.

2.9 Dependent types and modules

We will be able to define modules and abstract data types by extending the existing types in a simple but very expressive way — using so-called *dependent types*.

dependent product

Suppose you are writing a business application and you wish to construct a type representing the date:

$$\begin{aligned} Month &= \{1, \ldots, 12\} \\ Day &= \{1, \ldots, 31\} \\ \\ Date &= Month \times Day \end{aligned}$$

We would need a way to check for valid dates. Currently, $\langle 2, 31 \rangle$ is a perfectly legal member of $Date$, although it is not a valid date. One thing we can do is to define

$$\begin{aligned} Day(1) &= \{1, \ldots, 31\} \\ Day(2) &= \{1, \ldots, 29\} \\ &\vdots \\ Day(12) &= \{1, \ldots, 31\} \end{aligned}$$

and we will now write our data type as

$$Date = m : Month \times Day(m).$$

We mean by this that the second element of the pair belongs to the type indexed by the first element. Now, $\langle 2, 20\rangle$ is a legal date since $20 \in Day(2)$, and $\langle 2, 31\rangle$ is illegal because $31 \notin Day(2)$.

Many programming languages implement this or a similar concept in a limited way. An example is Pascal's *variant records*. While Pascal requires the indexing element to be of scalar type, we will allow it to be of any type.

We can see that what we are doing is making a more general product type. It is very similar to $A \times B$. Let us call this type $prod(A, x.B)$. We can display this as $x : A \times B$. The typing rules are:

$$\frac{E \vdash a : A \;\; E \vdash b \in B[a/x]}{E \vdash pair(a, b) : prod(A, x.B)}$$

$$\frac{E \vdash p : prod(A, x.B) \;\; E, u : A, v : B[u/x] \vdash t \in T}{E \vdash spread(p; u, v.t) \in T}$$

Note that we haven't added any elements. We've just added some new typing rules.

dependent functions

If we allow B to be a family in the type $A \to B$, we get a new type, denoted by $fun(A; x.B)$, or $x : A \to B$, which generalizes the type $A \to B$. The rules are:

$$\frac{E, y : A \vdash b[y/x] \in B[y/x]}{E \vdash \lambda(x.b) \in fun(A; x.B)} \; new\ y$$

$$\frac{E \vdash f \in fun(A; x.B) \quad E \vdash a \in A}{E \vdash ap(f; a) \in B[a/x]}$$

Example 2 : Back to our example Dates. We see that $m : Month \to Day[m]$ is just $fun(Month; m.Day)$, where Day is a family of twelve types. And $\lambda(x.maxday[x])$ is a term in it.

3 Equality and Quotient Types

According to Martin-Löf's semantics, a mathematical type is created by specifying notation for its elements, called *canonical names*, and specifying our equality relation on these names. The equality relation can be an equivalence relation. This relation is one feature that distinguishes mere notation from the more abstract idea of an *object*, i.e. from notation with meaning.

For example, sets of ordered pairs of integers, $\langle x, y\rangle$ can be used as constants for rational numbers. In this case, the equality relation should equate $\langle 2, 4\rangle$ and $\langle 1, 2\rangle$, generally $\langle a, b\rangle = \langle c, d\rangle$ iff $a * d = b * c$. Similarly, let $\mathbb{Z}/mod\ n$ denote the congruence integers, i.e. the integers with equality taken $mod\ n$ so that $x = y$ mod n iff n divides $(x - y)$. The only difference between $\mathbb{Z}$ and $\mathbb{Z}/mod\ n$ is the equality relation on the integer constants.

Following Beeson [6] we might speak of the constants without an equality relation as a pre-type. Then a type arises by pairing an equality with a pre-type, say $\langle T, E\rangle$ where E is an equivalence relation on T. But when we think of functions on a type, say $f : \mathbb{Q} \rightarrow \mathbb{Q}$, we do not expect equality information to be included as part of the input to f. The "data" comes from T. This viewpoint is thus similar to Martin-Löf's as long as we provide a way to define new types by changing the equality relation on data and keeping the equality information "hidden" from operations on the type. This is accomplished by the *quotient type*.

To form a quotient type we need a type T and an equivalence relation E on T. The quotient of T by E is denoted $T//E$. (Nuprl uses a slightly more flexible notation as we see in section 3.) The elements of $T//E$ are those of T but equality is defined by E and denoted $x = y$ in $T//E$. For example the rational numbers can be taken to be $(\mathbb{Z} \times \mathbb{N}^+)//E$ where

$$E(\langle x, n\rangle, \langle y, m\rangle) \text{ iff } m * x = n * y.$$

It is noteworthy that because the equality information is "hidden" we cannot in general say

$$x = y \text{ in } T//E \text{ implies } E(x, y)$$

under the propositions-as-types interpretation of implication.

Type theory can express the logical idea that given $x = y$ in $T//E$ we know that $E(x, y)$ is "true" in the sense that there is a proof of $E(x, y)$ but we cannot access it. One way to say this is to replace the predicate $E(x, y)$ by the weaker type $\{\mathbf{1}|E(x, y)\}$. We call this the "squashed type." If $E(x, y)$ is true then this type, call it $sq(E(x, y))$ has $\bullet$ as member according to our rules for the set type. If $E(x, y)$ is not true, then $sq(E(x, y))$ is empty. We can thus say

$$x = y \text{ in } T//E \text{ implies } sq(E(x, y)).$$

4 A Nuprl Type Theory

In this section we look at some features of the Nuprl type theory. The first section is a discussion of the uniform syntax of Nuprl 4 terms. The second section considers Allen's semantics [4, 3] for Nuprl without recursive types. Mendler [35] provides a semantics for recursive types as well, but it is more involved than what we present here.

4.1 Nuprl Term Syntax

Following Frege, Church, and Martin-Löf, we take the basic unit of notation to be a *term*. A formula for a sentence, $\forall x : \mathit{int}.\ x^2 \geq 0$, is a term as is the expression for the square of a number, x^2.

We will distinguish the *concrete syntax* or the *display form* of a term from its abstract structure.

4.1.1 abstract structure of terms

What are the constituent parts of a term?

operator name

In our analysis a term is built from an *operator* and subterms. We choose to name the operator, so there is some means of finding an *operator_name*. This seems to be convenient for computer processing and helps in the conduct of mathematics as well. Informal practice sometimes settles for a glyph or symbol as the operator name, e.g. $\int$.

subterms

Given a term, there must be a finite number of subterms. These could be collected as a list or a multi-set (bag). In many cases, say $\frac{a}{b}$, it is not clear how to order the subterms a and b. But there must be a way to *address* (or locate) uniquely each subterm. We have chosen to *list* the subterms.

binding structure

We know that mathematical notation for sentences can be defined with *combinators* which do not introduce an idea of binding.

We adopt the analysis of notation based on *binding*. So with each operator there is a binding mechanism. Informal mathematics uses many different mechanisms, but we will attempt to analyze all of them as *first-order*. That is, the binding structure can be defined by designating a class of first order variables, i.e. just identifiers, as binding occurrences. The generic form of an operator with binding structure is

$$op_{x_1,\ldots,x_n}(t_1; \ldots; t_m)$$

where x_i are first order variables and t_j are terms. The binding structure is specified by saying which of the x_i is bound in which t_j. This could be done graphically as in

$$op_{x,y,z}(a_{x,y}; b_{y,z}; c_{x,z})$$

In Nuprl we have chosen to use the form

$$op(x, y.\ a_{x,y}; y, z.\ b_{y,z}; x, z.\ c_{x,z}).$$

because it is a simple way to represent the general binding structure described above.

References

[1] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.

[3] Stuart F. Allen. *A non-type-theoretic semantics for type-theoretic language.* PhD thesis, Cornell University, 1987.

[4] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 215–224. IEEE, June 1987.

[5] R. C. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory (part I). *Formal Aspects of Computing*, 1:19–84, 1989.

[6] M. J. Beeson. Formalizing constructive mathematics: Why and how? In F. Richman, editor, *Constructive Mathematics,* Lecture Notes in Mathematics, Vol. 873, pages 146–90. Springer, Berlin, 1981.

[7] M.J. Beeson. *Foundations of Constructive Mathematics.* Springer Berlin, 1985.

[8] U. Berger and H. Schwichtenberg. Program extraction from classical proofs. In Daniel Leivant, editor, *Logic and Computational Complexity*, pages 77–97. Springer, Berlin, 1994.

[9] E. Bishop and D. Bridges. *Constructive Analysis.* Springer, New York, 1985.

[10] S. Buss. The polynomial hierarchy and intuitionistic bounded arithmetic. In *Structure in Complexity Theory,* Lecture Notes in Computer Science. 223, pages 77–103. Springer, Berlin, 1986.

[11] Robert L. Constable. Types in Logic, Mathematics and Programming. In S. Buss, editor, *Handbook of Proof Theory*, North Holland, 1998.

[12] Robert L. Constable. Using reflection to explain and enhance type theory. In Helmut Schwichtenberg, editor, *Proof and Computation*, pages 65–100, Berlin, 1994. NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20-August 1, 1995, NATO Series F, Vol. 139, Springer, Berlin.

[13] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[14] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, London, 1990.

[15] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic,* Lecture Notes in Computer Science, Vol. 417, pages 50–66. Springer, Berlin, 1990.

[16] N. G. deBruijn. A survey of the project Automath. In *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[17] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Proc. of the First Annual Workshop on Logical Frameworks*, pages 280–306, Sophia-Antipolis, France, June 1990. Programming Methodology Group, Chamers University of Technology and University of Göteborg.

[18] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[19] Solomon Feferman. A language and axioms for explicit mathematics. In J. N. Crossley, editor, *Algebra and Logic,* Lecture Notes in Mathematics, Vol. 480, pages 87–139. Springer, Berlin, 1975.

[20] Max B. Forester. Formalizing constructive real analysis. Technical Report TR93-1382, Computer Science Dept., Cornell University, Ithaca, NY, 1993.

[21] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge Tracts in Computer Science, Vol. 7. Cambridge University Press, 1989.

[22] Jason J. Hickey. Objects and theories as very dependent types. In *Proceedings of FOOL 3*, July 1996.

[23] C. A. R. Hoare. Notes on data structuring. In *Structured Programming.* Academic Press, New York, 1972.

[24] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[25] Douglas J. Howe. Equality in lazy computation systems. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 198–203. IEEE Computer Society, June 1989.

[26] Douglas J. Howe. Reasoning about functional programs in Nuprl. *Functional Programming, Concurrency, Simulation and Automated Reasoning,* Lecture Notes in Computer Science, Vol. 693, Springer, Berlin, 1993.

[27] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In *Proceedings of AMAST'96*, 1996. To appear.

[28] Douglas J. Howe and Scott D. Stoller. An operational approach to combining classical set theory and functional programming languages. In and J. C. Mitchell M. Hahiya, editor, Lecture Notes in Computer Science, Vol. 789, pages 36–55, New York, April 1994. International Symposium TACS '94, Springer, Berlin. Theoretical Aspects of Computer Software.

[29] G. Huet. A uniform approach to type theory. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 337–398. Addison-Wesley, 1990.

[30] G. P. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[31] Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, Ithaca, NY, January 1995.

[32] Daniel Leivant. Intrinsic theories and computational complexity. In Daniel Leivant, editor, *Logic and Computational Complexity*, pages 177–194. Springer, Berlin, 1994.

[33] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.

[34] Per Martin-Löf. *Intuitionistic Type Theory, Studies in Proof Theory, Lecture Notes.* Bibliopolis, Napoli, 1984.

[35] P.F. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, Ithaca, NY, 1988.

[36] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

[37] Erik Palmgren. *On Fixed Point Operators, Inductive Definitions and Universes in Martin-Löf's Type Theory.* PhD thesis, Uppsala University, Thunbergsvägen 3, S-752, Uppsala, Sweden, March 1991.

[38] A. M. Pitts. *Operationally-Based Theories of Program Equivalence.* University of Cambridge, Cambridge, UK, 1995.

[39] Helmut Schwichtenberg. *Computational Content of Proofs.* Mathematisches Institut, Universität München, München, Germany, 1995. Working material for Marktoberdorf lecture.

[40] D. Scott. Constructive validity. In D. Lacombe M. Laudelt, editor, *Symposium on Automatic Demonstration,*Lecture Notes in Mathematics, Vol. 5 #3, pages 237–275, New York, 1970. Springer-Verlag.

[41] D. Scott. Data types as lattices. *SIAM J. Comput.*, 5:522–87, 1976.

[42] Anton Setzer. *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe.* PhD thesis, Ludwig-Maximilians-Universität, München, September 1993.

[43] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[44] S.S. Wainer. The hierarchy of terminating recursive programs over N. In Daniel Leivant, editor, *Logic and Computational Complexity,* Lecture Notes in Computer Science, Vol. 959, pages 281–299. Springer, Berlin, 1994.