Reprinted from

HANDBOOK OF PROOF THEORY

Edited by

SAMUEL R. BUSS

University of California, San Diego

1998

ELSEVIER

AMSTERDAM • LAUSANNE • NEW YORK • OXFORD • SHANNON • SINGAPORE • TOKYO

we say that the above clauses define the *canonical proofs*, e.g. a canonical proof of $P \,\&\, Q$ is a pair $\langle p, q\rangle$, but $\Rightarrow L(\Rightarrow R(x.\langle x, q\rangle); p)$ is a noncanonical proof of $P \,\&\, Q$ which reduces to $\langle p, q\rangle$ when we "normalize" the proof.

Although this is a suggestive semantics of both proofs and propositions, several questions remain. Given a proposition P, can we be sure that all proofs have the structure suggested by this semantics? Suppose $P \,\&\, Q$ is not proved by proving P and proving Q but instead by a case analysis or by decomposing an implication and then decomposing an existential statement, etc.; so if t proves $P \,\&\, Q$, do we know t is a pair?

If proofs are going to be objects, then what is the right equality relation on them? If t proves $P\&Q$ then is t at least *equal* to a pair $\langle p, q\rangle$? What is the right equality on propositions? If $P = Q$ and p proves P does p prove Q? How can we make sense of Magic as a proof object? It is a proof of $P \vee \neg P$ yet it has no structure of the kind Heyting suggests. We will see that the type theories of the next section provide just the right tools for answering these questions.

3. Type theory

3.1. Introduction

Essential features. In this section I want to give a nontechnical overview of the subject I am calling *type theory*. I will discuss these points:

- It is a *foundational theory* in the sense of providing definitions of the basic notions in logic, mathematics, and computer science in terms of a few primitive concepts.
- It is a *computational theory* in the sense that among the primitive built-in concepts are notions of algorithm, data type, and computation. Moreover these notions are so interwoven into the fabric of the theory that we can discuss the computational aspects of every other idea in the theory. (The theory also provides a foundation for *noncomputational* mathematics, as we explain later.)
- It is *referential* in the sense that the terms denote *mathematical objects*. The referential nature of a term in a type T is determined by the *equality relation* associated with T, written $s = t$ in T. The equality relation is basic to the meaning of the type. All terms of the theory are *functional* over these equalities.
- When properly formalized and *implemented*, the theory provides practical tools for expressing, performing, and reasoning about computation in all areas of mathematics.

A detailed account of these three features will serve to explain the theory. Understanding them is essential to seeing its dynamics. In a sense, the axioms of the theory serve to provide a very abstract account of mathematical data, its transformation by effective procedures, and its assembly into useful knowledge. I summarized my ideas on this topic in Constable [1991].

Language and logic. In a sense, the theory is *logic free*. Unlike our account of typed logic, we do not start with propositions and truth. Instead we begin with more elementary parts of language, in particular, with a theory of computational equality of terms (or expressions). In *Principia* these elementary ideas are considered as part of the meaning of propositions. We separate them more clearly. We examine the mechanism of naming and definition as the most fundamental and later build upon this an account of propositions and truth.

This analysis of language draws on the insights of Frege, Russell, Brouwer, Wittgenstein, Church, Curry, Markov, de Bruijn, Kolmogorov, and Martin-Löf, and it draws on technical advances made by numerous computer scientists and logicians. We can summarize the insights in this way. The notion of *computability* is grounded in rules for processing language (Church [1940], Curry and Feys [1958], Markov [1949]). In particular, they can be organized as rules for a basic (type free) *equality on expressions* closely related to Frege's theory of identity in [1903]. The rules explain when two expressions will have the same reference if they have any reference. (We call these *computation rules*, but they could also be considered simply as general rules of *definitional equality* as in Automath.) De Bruijn showed that to fully understand the definitional rules, we need to understand how expressions are organized into *contexts* in a *tree of knowledge* as we discussed in section 2.12.

Frege not only realized the nature of identity rules, but he explained that the very notion of an *object* (or mathematical object) depends on rules for equality of expressions which are intended to denote objects. The equality rules of a theory serve to define the objects and prepare the ground for a *referential language*, one in which the expressions can be said to denote objects.

Frege also believed that the equality rules were not arbitrary but expressed the primitive truths about *abstract objects* such as numbers and classes. We build on Brouwer's theme that an understanding of the *natural numbers* $\mathbf{N}$ is an especially clear place to begin, and we try to build as much as possible with them. Here the insights of Brouwer [1975] (see van Stigt [1990]) show how to connect intuitions about number to the rules for equality of expressions. Brouwer shows that the idea of natural number and of pairing numbers are meaningful because they arise from mental operations. Moreover, these are the same abilities needed to manipulate the language of expressions (see Chomsky [1988]).[27]

So like Frege and Brouwer (and unlike formalists), we understand type theory to be *referential*, that is, the theory is about mathematical objects, and the meaningful expressions denote them.

Following Russell, we believe that a referential theory is created by classifying expressions into types. Not every expression is meaningful, for example, school children know that $0/0$ is not. We sometimes say that the meaningful expressions are those that refer to mathematical objects, but this seems to presuppose that we

[27] For Brouwer this language is required by an individual only because of the limits and flaws in his or her mental powers. But for our theory, language is essential to the communication among agents (human and artificial or otherwise) needed to establish public knowledge.

know what such objects are. So we prefer to say that the task of type theory is *to provide the means to say when an expression is meaningful.* This is done by classifying expressions into types. Indeed to define a type is to say what *expressions* are of that type. This process also serves to define mathematical objects.[28]

Martin-Löf suggested a particular way of specifying types based on ideas developed by W. W. Tait [1967,1983]. First designate the standard irreducible names for elements of a type, say $t_1, t_2, \ldots$ belong to T. Call these *canonical values.* Then based on the definition of evaluation, extend the membership relation to all t' such that t' evaluates to a canonical value of T; we say that membership is extended by *pre-evaluation.*

Level restrictions. Russell [1908] observed that it is not possible to regard the collection of all types as a type itself. Let Type be this collection of all types. So Type is not an element of Type. Russell suggested schemes for layering or stratifying these "inexhaustible concepts" like Type or Proposition or Set. The idea is to introduce notions of types of various *levels.* In our theory these levels are indicated by *level indexes* such as Type_i. They will be defined later.

Architecture of type theory. What we have said so far lays out a basic structure for the theory. We start with a class of *terms.* This is the linguistic material needed for communication. We use variables and substitution of terms for variables to express relations between terms. Let x, y, z be variables and s, t be terms. We denote the substitution of term s for all free occurrences of variable x in t by $t[s/x]$. The details of specifying this mechanism vary from theory to theory. Our account is conventional and general.

Substitution introduces a primitive *linguistic relationship* among terms which is used to define certain basic *computational equalities* such as $ap(\lambda(x.b); a) = b[a/x]$.

There are other relations expressed on terms which serve to define computation. We write these as evaluation relations

$$t \; evals\_to \; t' \quad \text{also written} \quad t \downarrow t'.$$

Some terms denote types, e. g. $\mathbf{N}$ denotes the type of natural numbers. There are type forming operations that build new types from others, e. g. the Cartesian product $T_1 \times T_2$ of T_1 and T_2. Corresponding to a type constructor like $\times$ there is usually a constructor on elements, e. g. if $t_1 \in T_1$, $t_2 \in T_2$ then $pair(t_1; t_2) \in T_1 \times T_2$. By the Tait pre-evaluation condition above

$$\frac{t' \; evals\_to \; pair(t_1; t_2)}{t' \in T_1 \times T_2}$$

[28] The interplay between expressions and objects has seemed confusing to readers of constructive type theory. In my opinion this arises mainly from the fact that computability considerations cause us to say more about the underlying language than is typical, but the same relationship exists in any formal account of mathematics.

Part of defining a type is defining equality among its numbers. This is written as $s = t$ in T. The idea of defining an equality with a type produces a concept like Bishop's sets (see Bishop [1967], Bishop and Bridges [1985]), that is Bishop [1967,p.63] said "... a *set* is defined by describing what must be done to construct an element of the set, and what must be done to show that two elements are *equal.*"

The basic forms of judgment in this type theory are

- t is a term
  This is a simple context-free condition on strings of symbols that can be checked by a parser. We stress this by calling these *readable expressions.*
- T *is_a type*
  We also write $T \in \mathsf{Type}$ and prefer to write capital letters, S, T, A, B for types. This relationship is not decidable in general and cannot be checked by a parser. There are rules for inferring typehood.
- $t \in T$ (type membership or elementhood)
  This judgement is undecidable in general.
- $s = t$ in T (equality on T)
  This judgement is also undecidable generally.

Inference mechanism. Since Post it has been the accepted practice to define the class of formulas and the notion of proof inductively. Notice our definition of formula in section 2.4, also, for example, a *Hilbert style proof* is a sequence of closed formulas $F_1, \ldots, F_n$ such that F_i is an axiom or follows by a rule of inference from F_j, F_k for $j < i, k < i$. A typical inference rule is expressed in the form of hypotheses above a horizontal line with the conclusion below as in *modus ponens.*

$$\frac{A, \; A \Rightarrow B}{B}$$

This definition of a proof includes a specific presentation of evidence that an element is in the class of all proofs.

The above form of a rule can be used to present any inductive definition. For example, the natural numbers are often defined inductively by one rule with no premise and another rule with one.

$$0 \in \mathbf{N} \qquad \frac{n \in \mathbf{N}}{suc(n) \in \mathbf{N}}$$

This definition of $\mathbf{N}$ is one of the most basic inductive definitions. It is a pattern for all others, and indeed, it is the clarity of this style of definition that recommends it for foundational work.

Inductive definitions are also prominent in set theory. The article of Aczel [1986] "An Introduction to Inductive Definitions" surveys the methods and results. He bases his account on sets Φ of rule instances of the form $\frac{X}{x}$ where X are the *premises* and x the *conclusions.* A set Y is called Φ-closed iff $X \subseteq Y$ implies $x \in Y$. The set inductively defined by Φ is the intersection of all subsets Y of A which are Φ-closed.

3.2. Small fragment — arithmetic

We build a small fragment of a type theory to illustrate the points we have just made. The explanations are all inductive. We let S and T be metavariables for types and let, s, t, s_i, t_i, also s', t', s_i', t_i' denote terms.

We arrange the theory around a *single judgment*, the equality $s = t$ in T. We avoid membership and typehood judgments by "folding them into equality" just to make the fragment more compact. First we look at an informal account of this theory.

The intended meaning of $s = t$ in T is that T is a type and s and t are equal elements of it. Thus a premise such as $s = t$ in T implies that T is a type and that s and t are elements of T (thus subsuming membership judgment).[29]

The only atomic type is $\mathbf{N}$. If S and T are types, then so is $(S \times T)$; these are the only compound types.

The canonical elements of $\mathbf{N}$ are 0 and $suc(n)$ where n is an element of $\mathbf{N}$, canonical or not. The canonical elements of $(S \times T)$ are $pair(s;t)$ where s is of type S and t of type T. The expressions $1of(p)$ and $2of(p)$ are noncanonical. The evaluation of $1of(pair(s;t))$ is s and of $2of(pair(s;t))$ is t.

The inference mechanism must generate the evident judgments of the form $s = t$ in T according to the above semantics. This is easily done as an inductive definition. The rules are all given as clauses in this definition of the usual style (recall Aczel [1977] for example).

We start with *terms* and their evaluation. The only atomic terms are 0 and $\mathbf{N}$. If s and t are terms, then so are $suc(t), (s \times t), pair(s;t), 1of(t), 2of(t)$. Of course, not all terms will be given meaning, e.g. $(0 \times \mathbf{N})$, $suc(\mathbf{N})$, $1of(\mathbf{N})$ will *not* be.

Evaluation. Let s and t be terms.

$$0 \; evals\_to \; 0 \quad \mathbf{N} \; evals\_to \; \mathbf{N} \quad suc(t) \; evals\_to \; suc(t) \quad pair(s;t) \; evals\_to \; pair(s;t)$$

$$1of(pair(s;t)) \; evals\_to \; s \qquad 2of(pair(s;t)) \; evals\_to \; t$$

Remark: $s(\mathbf{N}) \; evals\_to \; s(\mathbf{N})$, $1of(pair(\mathbf{N};0)) \; evals\_to \; \mathbf{N}$. So evaluation applies to meaningless terms. It is a purely formal relation, an *effective calculation*. Thus the base of this theory includes a formal notion of *effective computability* (c.f. Rogers [1967]) compatible with various formalizations of that notion, but not restricted necessarily to them (e.g. Church's thesis is not assumed). Also note that *evals_to* is idempotent; if $t \; evals\_to \; t'$ then $t' \; evals\_to \; t'$ and t' is a value.

general equality

$$\frac{t_1 = t_2 \text{ in } T}{t_2 = t_1 \text{ in } T} \qquad \frac{t_1 = t_2 \text{ in } T \quad t_2 = t_3 \text{ in } T}{t_1 = t_3 \text{ in } T} \qquad \frac{t_1 = t_2 \text{ in } T \quad t_1 \; evals\_to \; t_1'}{t_1' = t_2 \text{ in } T}$$

[29] In the type theory of Martin-Löf [1982], a premise such as $s = t$ in T presupposes that T is a type and that $s \in T, t \in T$. This must be known *before* the judgment makes sense.

typehood and equality

$$0 = 0 \text{ in } \mathbf{N} \qquad \frac{t = t' \text{ in } \mathbf{N}}{suc(t) = suc(t') \text{ in } \mathbf{N}} \qquad \frac{s = s' \text{ in } S \quad t = t' \text{ in } T}{pair(s;t) = pair(s';t') \text{ in } (S \times T)}$$

The inductive nature of the type $\mathbf{N}$ and of the theory in general is apparent from its presentation. That is, from *outside* the theory we can see this structure. We can use induction principles from the informal mathematics (the *metamathematics*) to say, for example, every canonical expression for a number is either 0 or $suc(n)$. But so far there is no construct *inside* the theory which expresses this fact. We will eventually add one in section 3.3.

Examples. Here are examples of true judgments that we can make: $suc(0) = suc(0)$ in $\mathbf{N}$. This tells us that $\mathbf{N}$ is a type and $suc(0)$ an element of it. Also $pair(0; suc(0)) = pair(0; suc(0))$ in $(\mathbf{N} \times \mathbf{N})$ which tells us that $(\mathbf{N} \times \mathbf{N})$ is a type with $pair(0; suc(0))$ a member. Also $1of(pair(0;a))$ belongs to $\mathbf{N}$ and $suc(1of(pair(0;a)))$ does as well for arbitrary a.

Here is a derivation that $suc(1of(pair(0; suc(0)))) = 2of(pair(0; suc(0)))$ in $\mathbf{N}$.[30]

$$\frac{\dfrac{\dfrac{0 = 0 \text{ in } \mathbf{N}}{0 = 0 \text{ in } \mathbf{N} suc(0) \;=\; suc(0) = suc(o) \text{ in } \mathbf{N}}}{pair(0; suc(0)) \;=\; pair(0; suc(0)) \text{ in } \mathbf{N} \times \mathbf{N}}}{1of(pair(0; suc(0))) \;=\; 1of(pair(0; suc(0))) \text{ in } \mathbf{N} \quad 1of(pair(0; suc(0))) \; evals\_to \; 0}$$

$$2of(pair(0; suc(0))) = 2of(pair(0; suc(0))) \text{ in } \mathbf{N} \quad 2of(pair(0; suc(0))) \; evals\_to \; suc(0)$$

$$1of(pair(0; suc(0))) = 0 \text{ in } \mathbf{N} \qquad 2of(pair(0; suc(0))) = suc(0) \text{ in } \mathbf{N}$$

$$suc(1of(pair(0; suc(0)))) = suc(0) \text{ in } \mathbf{N} \qquad suc(0) = 2of(pair(0; suc(0))) \text{ in } \mathbf{N}$$

$$\overline{suc(1of(pair(0; suc(0)))) = 2of(pair(0; suc(0))) \text{ in } \mathbf{N}}$$

Analyzing the fragment. This little fragment illustrates several features of the theory.

First, *evaluation is defined prior to typing.* The *evals_to* relation is purely formal and is grounded in language which is a prerequisite for communicating mathematics. Computation does not take into account the meaning of terms. This definition of computability might be limiting since we can imagine a notion that relies on the information in typehood, and it is possible that a "semantic notion" of computation must be explored in addition, once the types are laid down.[31] Our approach to

[30] In type theory, we will write the derivations in the usual bottom-up style with the conclusion at the bottom, leaves at the top.

[31] In IZF this is precisely the way computation is done, based on the information provided by a membership proof.

computation is compatible with the view taken in computation theory (c.f. Rogers [1967]).

Second, the semantics of even this simple theory fragment shows that the concept of a proposition involves the notion of its meaningfulness (or well-formedness). For example, what appears to be a simple proposition, $t = t$ in T, expresses the judgments that T is a type and that t belongs to this type. These judgments are part of understanding the judgment of truth.

To stress this point, notice that by postulating $0 = 0$ in $\mathbf{N}$ we are saying that $\mathbf{N}$ is a type, that 0 belongs to $\mathbf{N}$ and that it equals itself. The truth judgment is entirely trivial; so the significance of $t = t$ in T lies in the well-formedness judgments implicit in it. These judgments are normally left *implicit* in accounts of logic.

Notice that the well-formedness judgments cannot be *false*. They are a different category of judgment from those about truth. To say that $0 \in \mathbf{N}$ is to *define* zero, and to say $\mathbf{N}$ is a type is to define $\mathbf{N}$. We see this from the rules since there are no separate rules of the form "$\mathbf{N}$ is_a type" or "0 *is_a* $\mathbf{N}$." Note, because $t = t$ whenever t is in a type, *the judgment $t = t$ in T happens to be true exactly when it is well-formed.*

Finally the points about $t = t$ in T might be clarified by contrasting it with $suc = suc$ in 0. This judgment is meaningless in our semantics because 0 is not a type. Likewise $suc = suc$ in $\mathbf{N}$ is meaningless because although $\mathbf{N}$ is a type, *suc* is not a member of it. Similarly, $0 = suc$ in $\mathbf{N}$ is meaningless since *suc* is not a member of $\mathbf{N}$ according to our semantics. None of these expressions, which read like propositions, is false; they are just *senseless*. So we cannot understand, with respect to our semantics, what it would mean for them to be false.

Third, notice that the *semantics of the theory were given inductively* (although informally), and the proof rules were designed to directly express this inductive definition. This feature will be true for the full theory as well, although the basic judgments will involve variables and will be more complex both semantically and proof theoretically.

Fourth, the semantic explanations are *rooted in the use of informal language.* We speak of terms, substitution and evaluation. The use of language is critical to expressing computation. We do not treat terms as mathematical objects nor evaluation as a mathematical relation. To do this would be to conduct metamathematics *about* the system, and that metamathematics would then be based on some prior informal language. When we consider implementing the theory, it is the informal language which we implement, translating it to a programming notation lying necessarily outside of the theory.

Fifth, although the theory is grounded in language, it *refers to abstract objects.* This abstraction is provided by the equality rules. So while $1of(pair(0; suc(0)))$ is not a canonical integer in the term language, we cannot observe this linguistic fact in the theory. This term denotes the number 0. The theory is referential in this sense.

Sixth, the theory is *defined by rules.* Although these rules reflect concepts that we have mastered in language, so are meaningful, and although all of the judgments we assert are evident, it is the rules that define the theory. Since the rules reflect a semantic philosophy, we can see in them answers to basic questions about the objects

of the theory. We can say what a number is, what 0 is, what successor is. Since the fragment is so small, the answers are a bit weak, but we will strengthen it later.

Seventh, the theory is *open-ended.* We expect to extend this theory to formalize ever larger fragments of our intuitions about numbers, types, and propositions. As Gödel showed, this process is never complete. So at any point the theory can be extended. By later specifying how evaluation and typing work, we provide a framework for future extensions and provide the guarantees that extensions will preserve the truths already expressed.

3.3. First extensions

We could extend the theory by adding further forms of computation such as a term, prd, for predecessor along with the evaluation

$$prd(suc(n))\ evals\_to\ n.$$

We can also include a term for addition, $add(s; t)$ along with the evaluation rules

$$add(0; t)\ evals\_to\ t \qquad \frac{add(n; t)\ evals\_to\ s'}{add(suc(n); t)\ evals\_to\ suc(s')}$$

We include, as well, a term for multiplication, $mult(s; t)$ along with the evaluation rule

$$mult(0; t)\ evals\_to\ 0 \qquad \frac{mult(n; t)\ evals\_to\ m \quad add(m; t)\ evals\_to\ a}{mult(suc(n); t)\ evals\_to\ a}$$

These rules enable us to type more terms and assert more equalities. We can easily prove, for instance, that

$$add(suc(0); suc(0)) = mult(suc(0); add(suc(0); suc(0))) \text{ in } \mathbf{N}.$$

But this "theory" is woefully weak. It cannot

- internally express general statements such as $prd(suc(x)) = x$ in $\mathbf{N}$ or $add(suc(x); y) = suc(add(x; y))$ for any x because there is no notion of variable, but these are true in the metalanguage.
- express function definition patterns such as the *primitive recursions* which were used to define add, multiply and for which we know general truths.
- express the inductive nature of $\mathbf{N}$ and its consequences for the uniqueness of functions defined by primitive recursion.

Adding capability to define new functions and state their "functionality" takes us from a concrete theory to an abstract one; from specific equality judgments to *functional judgments.* These functional judgments are the essence of the theory, and they provide the basis for connecting to the propositional functions of typed logic. So we add them next.

The simplest new construct to incorporate is one for constructing any object by following the pattern for the construction of a number. We call it a (*primitive*) *recursion combinator*, R. It captures the pattern of definition of *prd*, *add*, *mult* given above. It will later be used to explain induction as well.

The defining property of R is its rule of computation and its respect for equality. We present the computation rule using substitution.[32] The simplest way to to this as to use the standard mechanism of *bound variables* (as in the lambda calculus or in quantifier notation). To this end we let u, v, w, x, y, z be variables, and given an expression *exp* of the theory, we let $u.exp$ or $u, v.exp$ or $u, v, x.exp$ or generally $u_1, \ldots, u_n.exp$ (also written $\bar{u}.exp$) be a *binding phrase*. We say that the u_i are *binding occurrences* of variables whose *scope* is *exp*. The occurrences of u_i in *exp* are *bound* (by the smallest binding phrase containing them). The unbound variables of *exp* are called *free*, and if x is a free variable of $\bar{u}.exp$, then $\bar{u}.exp[t/x]$ denotes the substitution of t for every free occurrence of x in *exp*. If any of the u_i occur free in t, then as usual $\bar{u}.exp[t/x]$ produces a new binding phrase $\bar{u}'.exp'$ where the binding variables are renamed to prevent *capture* of free variables of t.[33]

$$\frac{b[t/v]\ evals\_to\ c}{R(0; t; v.b; u, v, i.h)\ evals\_to\ c}$$

$$\frac{R(n; t; v.b; u, v, i.h)\ evals\_to\ a \quad h[n/u, t/v, a/i]\ evals\_to\ c}{R(suc(n); t; v.b; u, v, i.h)\ evals\_to\ c}$$

Here is a typical example of R used to define addition in the usual primitive recursive way.

$$R(n; m; v.v; u, v, a.suc(a))$$

We see that

$$R(0; m; --)\ evals\_to\ m\ ,\ i.e.\ 0 + m = m$$
$$R(suc(n); m; --)\ evals\_to\ suc(R(n; m; --)),$$
$$\text{i.e. } suc(n) + m\ evals\_to\ suc(n + m)$$

Once we have introduced binding phrases into terms, the format for equality and consequent typing rules must change. Consider typing R. We want to say that if $v.b$ and $u, v, i.h$ have certain types, then R has a certain type. But the type of b and h will depend on the types of u, v and i. For example, the type of $v.v$ will be T in a context in which the variable v is assumed to have type T. Let us agree to use the judgment $t \in T$ to discuss typing issues, but for this theory fragment (as for Nuprl) this notation is just an abbreviation for $t = t$ in T. We will use it when we intend to focus on typing issues. We might write a rule like

[32] R can also be defined as a combinator without variables. In this case the primitive notion is application rather than substitution.

[33] If u_i is a free variable of t then it is *captured* in $\bar{u}.exp[t/x]$ by the binding occurrence u_i.

$$\frac{n \in \mathbf{N}\ \ t \in A_1 \quad \frac{v \in A_1}{b \in B_2} \quad \frac{u \in \mathbf{N}\ \ v \in A_1\ \ i \in B_2}{h \in B_2}}{R(n; t; v.b; u, v, i.h) \in B_2}$$

The premises

$$\frac{u \in N\ v \in A_1\ i \in B_2}{h \in B_2}$$

reads "h has type B_2 under the assumption that u has type $\mathbf{N}$, u has type A_1 and i has type B_2."

For ease of writing we render this *hypothetical typing judgment* as $u : N,\ v : A_1,\ i : B_2 \vdash h \in B_2$. The syntax $u : N$ is a variant of $u \in N$ which stresses that u is a variable. Now the typing of R can be written

$$\frac{n \in N\ \ t \in N\ \ v{:}A_1 \vdash b \in B_2 \quad u{:}N,\ v{:}A_1,\ i{:}B_2 \vdash h \in B_2}{R(n; t; v.b; u, v, i.\ h) \in B_2}$$

This format tells us that n, t, b and h are possibly compound expressions of the indicated types with v, u, i as variables assumed to be of the indicated types.

Following our practice of subsuming the typing judgment in the equality one, we introduce the following rule.

First let

$$\begin{aligned} Principle\_argument &== n = n' \text{ in } \mathbf{N} \\ Aux\_argument &== t = t' \text{ in } \mathbf{N} \\ Base\_equality &== v = v' \text{ in } A_1 \vdash b = b' \text{ in } B_2 \\ Induction\_equality &== u = u' \text{ in } \mathbf{N}, v = v' \text{ in } A_1, i = i' \text{ in } B_2 \vdash h = h' \text{ in } B_2 \end{aligned}$$

Then the rule is

$$\frac{Principle\_argument \quad Aux\_argument \quad Base\_equality \quad Induction\_equality}{R(n; t; v.\, b; u, v, e.\, h) = R(n'; t'; v'.\, b'; u', v', e'.\, h') \text{ in } B_2}$$

Unit and empty types. We have already seen a need for a type with exactly one element, called a unit type. We take $\mathbf{1}$ as the type name and $\bullet$ as the element, and adopt the rules:

$$\bullet = \bullet \text{ in } \mathbf{1}$$

We adopt the convention that such a rule automatically adds the new terms $\bullet$ and $\mathbf{1}$ to the collection of terms. We also automatically add

$$\bullet\ evals\_to\ \bullet \qquad \mathbf{1}\ evals\_to\ \mathbf{1}$$

to indicate that the new terms are canonical unless we stipulate otherwise with a different evaluation rule.

We will have reasons later for wanting the "dual" of the unit type. This is the empty type, **0**, with no elements. There is no rule for elements, but we postulate **0** *is_a type* from which we have that we **0** as a term and **0** *evals_to* **0**

An interesting point about handling **0** is to decide what we mean by *assuming* $x \in \mathbf{0}$. Does

$$x : \mathbf{0} \vdash x \in \mathbf{0}$$

make sense? Is this a sensible judgment? We seem to be saying that if we assume x belongs to **0** and that **0** is type, then x indeed belongs to **0**. We clearly know functionality vacuously since there are no closed terms t, t' with $t = t'$ in **0**. It is more interesting to ask about such anomalies as

$$x : \mathbf{0} \vdash x \in \mathbf{N} \quad \text{or} \quad x : \mathbf{0} \vdash x \in \mathbf{1}$$

or even the possible nonsense

$$x : \mathbf{0} \vdash \mathbf{N} \in \mathbf{N}.$$

What are we to make of these "boundary conditions" in the design of the theory?

According to our semantics and Martin-Löf's typing judgments, even $x : \mathbf{0} \vdash (suc = \mathbf{N} \text{ in } \mathbf{N})$ is a true judgment because we require that **0** is a type and for t, t' in **0**, if $t = t'$ in **0**, then $suc \in \mathbf{N}, \mathbf{N} \in \mathbf{N}$ and $suc = \mathbf{N}$ in **N**. Since anything is true for all t, t' in **0**, the judgment is true.

This conclusion is somewhat bizarre, but we will see later that there will be other types, of the form $\{x : A \mid P(x)\}$ whose emptiness is unknown. So our recourse is to treat types uniformly and not attempt to make a special judgment in the case of assumptions of the form $x : T$ for which T might be empty.

List types. The list data type is almost as central to computing as the natural numbers. We presented this type in the logic as well, and we follow that example even though we can see lists as a special case of the recursive types to be discussed later (section 4). The rules are more compact and pleasing to examine if we omit the typing context $\mathcal{T}$ and use the typing abbreviation of $t \in T$ for $t = t$ in T. So although we will write a rule like[34]

$$\boxed{\frac{a \in A,\ l \in list(A)}{cons(a; l) \in list(A)}}$$

Without its typing context, we intend the full rule

$$\frac{\mathcal{T} \vdash a = a' \text{ in } A \quad \mathcal{T} \vdash l = l' \text{ in } list(A)}{\mathcal{T} \vdash cons(a; l) = cons(a'; l') \text{ in } list(A).}$$

[34] In this section we use $list(A)$ instead of *A list* to stress that we are developing a different theory than in section 2.

We also introduce a form of primitive recursion on lists, the combinator L whose evaluation rule and typing rules are:

$$\frac{b[t/v]\ evals\_to\ c}{L(nil; p; v.b; h, t, v, i.g)\ evals\_to\ c}$$

$$\frac{L(l, s, v.b, h, t, v, i.g)\ evals\_to\ c_1 \quad g[a/h, l/t, s/v, c_1/i]\ evals\_to\ c_2}{L(cons(a; l); s; v.b; h, t, v, i.g)\ evals\_to\ c_2}$$

Let $L[x; b, g] = L(x; v.\ b; h, t, v, e.\ g)$, and

$H_B == v = v' \text{ in } S \vdash b = b' \in B,$

$H_S == h = h' \text{ in } A,\ t = t' \text{ in } list(A),\ v = v' \text{ in } S,\ i = i' \text{ in } B \vdash g = g' \text{ in } B,$

$C_A == \vdash a = a' \text{ in } A,$

$C_S == \vdash s = s' \text{ in } S,$ and

$C_{Alist} == \vdash l = l' \text{ in } list(A),$ then

$$\frac{H_B \qquad H_S \qquad C_A \qquad C_S \qquad C_{Alist}}{L[cons(a; l), b; g] = L[cons(a'; l'), b', g'] \text{ in } list(A)}$$

$$L(nil; v.b; h, t, v, i.g) = L(nil; v.b'; h', t, v, i.g') \text{ in } list(A)$$

Here are typical generalizations of the functions add, mult, exp to N list to illustrate the use of L. For the list (3, 8, 5, 7, 2) the operations behave as follows. Add $addL$ is $(3 + (8 + (5 + (7 + (2 + 0)))))$, $multL$ is $3 * 8 * 5 * 7 * 2 * 1$, $expL_2$ is $(((((2)^2)^7)^5)^8)^3$.

$$\begin{aligned} addL(l) &== L(l; 0; h, t, a.add(h, a)) \\ multL(l) &== L(l; 1; h, t, m.mult(h, m)) \\ expL(l)_k &== L(l; k; h, t, e.exp(h, e)). \end{aligned}$$

The induction rule for lists is expressed using L as follows. Let $H_S ==$

$$x \in list(A), y \in S,\ v \in S \vdash f[nil/x, v/y] = b \text{ in } B$$

and let $H_{list} ==$

$$x \in list(A), y \in S, h \in A, t \in list(A), v \in S, i \in B \vdash f[cons(h; t)/x, v/y] = g \text{ in } B,$$

then

$$\frac{H_S \qquad H_{list}}{x \in list(A), y \in S \vdash f = L(x; y; v.b; h, t, v, i.g) \text{ in } B}$$

This says that L defines a unique functional expression over $list(A)$ and S because the values as inductively determined by the evaluation rule completely determine functions over $list(A)$.