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 N. If S and T are types, then so is $(S \times T)$; these are the only compound types.

The canonical elements of N are 0 and $suc(n)$ where n is an element of 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 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. $(0 \times \mathbb{N})$, $suc(\mathbb{N})$, $1of(\mathbb{N})$ will *not* be.

Evaluation. Let s and t be terms.

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

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

Remark: $s(\mathbb{N})$ $evals\_to$ $s(\mathbb{N})$, $1of(pair(\mathbb{N};0))$ $evals\_to$ 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

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

The inductive nature of the type 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 N. This tells us that N is a type and $suc(0)$ an element of it. Also $pair(0; suc(0)) = pair(0; suc(0))$ in $(\mathbb{N} \times \mathbb{N})$ which tells us that $(\mathbb{N} \times \mathbb{N})$ is a type with $pair(0; suc(0))$ a member. Also $1of(pair(0;a))$ belongs to 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 N.[30]

$$\frac{\dfrac{0 = 0 \text{ in } \mathbb{N}}{0 = 0 \text{ in } \mathbb{N} suc(0) = suc(o) \text{ in } \mathbb{N}}}{pair(0; suc(0)) = pair(0; suc(0)) \text{ in } \mathbb{N} \times \mathbb{N}} \quad \frac{1of(pair(0; suc(0))) = 1of(pair(0; suc(0))) \text{ in } \mathbb{N} \quad 1of(pair(0; suc(0))) \ evals\_to \ 0}{2of(pair(0; suc(0))) = 2of(pair(0; suc(0))) \text{ in } \mathbb{N} \quad 2of(pair(0; suc(0))) \ evals\_to \ suc(0)}$$

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

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

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 N we are saying that N is a type, that 0 belongs to 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 $0 \in \mathbb{N}$ is to *define* zero, and to say N is a type is to define N. We see this from the rules since there are no separate rules of the form "N is_a type" or 0 *is_a* 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 N is meaningless because although N is a type, *suc* is not a member of it. Similarly, $0 = suc$ in N is meaningless since *suc* is not a member of 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

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

But this "theory" is woefully weak. It cannot

- internally express general statements such as $prd(suc(x)) = x$ in 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 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.

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

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 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

$$\begin{aligned}
Principle_argument &== n = n' \text{ in } \mathrm{N} \\
Aux_argument &== t = t' \text{ in } \mathrm{N} \\
Base_equality &== v = v' \text{ in } A_1 \vdash b = b' \text{ in } B_2 \\
Induction_equality &== u = u' \text{ in } \mathrm{N}, v = v' \text{ in } A_1, i = i' \text{ in } B_2 \vdash h = h' \text{ in } B_2
\end{aligned}$$

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 **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 **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 $list(A)$.