Type Constructors with Quotients

Roland Backhouse (RUG)
Paul Chisholm (HW)
Erik Saaman (RUG)

The purpose of this note is to record discussions held at Edinburgh University and at the Rijksuniversiteit Groningen about the introduction of new type constructors in Martin-Löf's theory of types when equalities occur within either the premises or the conclusions of the introduction rules. The work extends the work in [1] where it was claimed that the elimination rule and the computation rules for a given type constructor Θ can always be inferred from the formation rule and introduction rules for Θ, but where a justification for the claim was given only for extremal type constructors. The effect of adding equalities to the introduction rules is equivalent to defining a congruence relation on the objects of the extremal type.

The method of inferring the elimination rule from the introduction rules is described by way of examples rather than formally, although a formal method does indeed underlie our descriptions and should be evident. We also omit the derivation of the computation rules; this should be obvious to readers familiar with [1].

The examples given here are of finite sets and binary numerals. Finite sets are discussed in more detail in [3]; here they are introduced via lists, the latter also being used to briefly review the techniques discussed in [1].

1. Lists

The list type constructor is familiar [2]. The formation rule and two introduction rules are as follows.

$$\frac{A\ type}{List(A)\ type}$$ List formation

$$\frac{A\ type}{[\,] \in List(A)}$$ [ ]-introduction

$$\frac{\begin{array}{l}A\ type\\ a \in A\\ l \in List(A)\end{array}}{a : l \in List(A)}$$:-introduction

It is normal to omit the premises of the formation rule from the premises of the introduction rules. Thus the premise "*A type*" would normally be omitted from the [ ]- and :-introduction rules above. We shall follow the same practice in the remainder of this note.

The (single) elimination rule for a given type constructor performs two functions: it says how to reason about objects of the type and it says how to define functions over objects of the type. (Because proofs are interpreted constructively these amount to the same thing.) The first premise (excluding the premises of the formation rule) of the elimination rule for type constructor Θ is therefore the statement that C, say, is a family of types indexed by objects of Θ. In other words C is postulated to be a property of objects of type Θ. The introduction rules represent the only way that canonical objects of the type may be constructed; so, in order to show that property C holds of an arbitrary object of type Θ, it suffices to show that it holds of each of the different sorts of canonical objects. There is thus one premise in the elimination rule for each of the introduction rules. Moreover the premises of an introduction rule become assumptions in the corresponding premise of the elimination rule.

In the case of lists there are just two sorts of canonical element, the empty list and composite lists consisting of a head element and a tail list. In order to prove that a property C is true of an arbitrary list we thus have to show that it is true of the empty list and of composite lists. Equally, to define a function over lists it suffices to define its value on the empty list and its value when applied to a composite list. The elimination rule is therefore as follows.

$$
\begin{array}{l}
[\!| \;\; w \in \mathit{List}(A) \\
\triangleright \;\; C(w)\ \mathit{type} \\
|\!] \\
x \in \mathit{List}(A) \\
y \in C([\,]) \\
[\!| \;\; a \in A; l \in \mathit{List}(A); h \in C(l) \\
\triangleright \;\; z(a,l,h) \in C(a:l) \\
|\!] \\
\hline
\mathit{List\text{-}elim}(x,y,z) \in C(x)
\end{array}
\qquad \text{List-elimination}
$$

In this rule the third premise is the one corresponding to [ ]-introduction; it is not hypothetical since apart from the premises of List formation there are no premises in the [ ]-introduction rule. The fourth premise corresponds to the :-introduction rule; it is hypothetical since the :-introduction rule has two premises in addition to the premises of List formation. To emphasise the way in which the premises of the introduction rule become assumptions of the corresponding premise in the elimination rule we have used the same symbols, a and l in the :-introduction rule and in the elimination rule.

Note that there is an additional assumption ("$h \in C(l)$") in the elimination rule arising from the fact that l is a recursive introduction variable.

2. Finite Sets

Suppose we wish to define a type constructor $\Im$ such that $\Im(A)$ is the type of finite subsets of A. Any such subset can be constructed by listing its elements. Conversely any list of elements of A may be regarded as a finite subset of A provided that we disregard the order of the elements and repeated occurrences of the same element. $\Im(A)$ is thus the quotient of $List(A)$ with respect to the equivalence relation that defines two lists as equal if they have the same elements independent of order and number of repeated occurrences.

We define the type constructor $\Im$ by adding to the introduction rules for List two additional rules defining the above equivalence. In full the rules are as follows.

$$\frac{A\ type}{\Im(A)\ type}\qquad \Im\text{-formation}$$

$$\frac{}{\phi \in \Im(A)}\qquad \phi\text{-introduction}$$

$$\frac{a \in A \qquad s \in \Im(A)}{a; s \in \Im(A)}\qquad ;\text{-introduction}$$

$$\frac{a \in A \qquad s \in \Im(A)}{a; a; s = a; s \in \Im(A)}\qquad \text{repetition}$$

$$\frac{a \in A \qquad b \in A \qquad s \in \Im(A)}{a; b; s = b; a; s \in \Im(A)}\qquad \text{order}$$

How should we construct the elimination rule for $\Im$? The best way to begin is to view the rule as a method of defining a function over objects of the type. If a function is to be truly a function then it must give equal values when applied to equal objects. Looking at it from the point of view of proofs, a proof that an object has some property must be independent of the way the object was constructed. Thus the $\Im$-elimination rule is constructed like the List-elimination rule but with two additional premises, one corresponding to the repetition rule and the other corresponding to the order rule.

$$
\frac{\begin{array}{l}
[\![\ w \in \Im(A) \\
\triangleright\ C(w)\ type \\
]\!] \\
x \in \Im(A) \\
y \in C(\phi) \\
[\![\ a \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ z(a,s,h) \in C(a;s) \\
]\!] \\
[\![\ a \ \Xi\ A; s \in \Im(A); h \in C(s) \\
\triangleright\ z(a,\ a;s,\ z(a,s,h)) = z(a,s,h) \in C(a;s) \\
]\!] \\
[\![\ a \in A; b \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ z(a,\ b;s,\ z(b,s,h)) = z(b,\ a;s,\ z(a,s,h)) \in C(a;b;s) \\
]\!]
\end{array}}{\Im\text{-}elim(x,y,z) \in C(x)}\ \Im\text{-elimination}
$$

The premise corresponding to the repetition rule

$$
\begin{array}{l}
[\![\ a \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ z(a,\ a;s,\ z(a,s,h)) = z(a,s,h) \in C(a;s) \\
]\!]
\end{array}
$$

is constructed as follows. The assumptions are derived from the premises of the repetition rule as in our discussion of lists. The judgement asserts that the proof object of $C(a;s)$ is the same whether we choose to evaluate it from $a;s$ or $a;a;s$. In the former case we evaluate $z(a,s,h)$ and in the latter case we evaluate $z(a,\ a;s,\ z(a,s,h))$.

The premise corresponding to the order rule

$$
\begin{array}{l}
[\![\ a,b \in A; s \in \Im(A); h \in C(s) \\
\triangleright\ z(a,\ b;s,\ z(b,s,h)) = z(b,\ a;s,\ z(a,s,h)) \in C(a;b;s) \\
]\!]
\end{array}
$$

is constructed similarly.

An example—the size of a set

The size of a set, s, is computed as follows. If s is ϕ then its size is 0. If it is $a;t$ and a is not an element of t then it is one more than the size of t; otherwise it is the size of t. Thus the object defining the size of s is

(1) $$\Im\text{-}elim(s, 0, (a, t, l)\textbf{if } mem(a, t) \textbf{ then } l \textbf{ else } l + 1 \textbf{ fi})$$

where $mem(a, t)$ is an object of $Bool = \{T, F\}$ and expresses membership of a in t. It is also computed using $\Im$-elim. Assuming that $.eq.$ is a Boolean function expressing equality on the base type of the set s, the value of $mem(a, s)$ equals the value of

(2) $$\Im\text{-}elim(s, F, (b, t, m)\textbf{if } a.eq.b \textbf{ then } T \textbf{ else } m \textbf{ fi}).$$

To verify that (1) is indeed a natural number we have to show that it respects the repetition and order rules of $\Im$-introduction. That is, we have to establish the 5th and 6th premises of the $\Im$-elimination rule. Of course, this also involves showing that (2) is an object of $Bool$ and to do so we have to show that it respects the repetition and order rules of $\Im$-introduction. Specifically, therefore, we have to demonstrate four properties which are formally written as follows:

$|[\ a \in A$
$\triangleright\ |[\ b \in A; t \in \Im(A); m \in Bool$
$\quad \triangleright$ if $a.eq.b$ then T else if $a.eq.b$ then T else m fi fi
$\quad\ =$
if $a.eq.b$ then T else m fi $\in Bool$
$\quad]|$
$]|$

(mem—repetition rule)

$|[\ a \in A$
$\triangleright\ |[\ b, c \in A; t \in \Im(A); m \in Bool$
$\quad \triangleright$ if $a.eq.b$ then T else if $a.eq.c$ then T else m fi fi
$\quad\ =$
if $a.eq.c$ then T else if $a.eq.b$ then T else m fi fi $\in Bool$
$\quad]|$
$]|$

(mem—order rule)

$|[\ a \in A; t \in \Im(A); l \in \mathbb{N}$
$\triangleright$ if $mem(a, a; t)$
then if $mem(a, t)$ then l else $l + 1$ fi
else if $mem(a, t)$ then l else $l + 1$ fi $+ 1$
$\quad =$ fi
if $mem(a, t)$ then l else $l + 1$ fi $\in \mathbb{N}$
$]|$

(size—repetition rule)

and

$$
\begin{array}{ll}
|[& a, b \in A; t \in \Im(A); l \in \mathbb{N} \\
\triangleright & \quad \textbf{if } mem(a, b; t) \\
& \quad \textbf{then if } mem(b, t) \textbf{ then } l \textbf{ else } l + 1 \textbf{ fi} \\
& \quad \textbf{else if } mem(b, t) \textbf{ then } l \textbf{ else } l + 1 \textbf{ fi } + 1 \\
= & \quad \textbf{fi} \\
& \quad \textbf{if } mem(b, a; t) \\
& \quad \textbf{then if } mem(a, t) \textbf{ then } l \textbf{ else } l + 1 \textbf{ fi} \\
& \quad \textbf{else if } mem(a, t) \textbf{ then } l \textbf{ else } l + 1 \textbf{ fi } + 1 \\
& \quad \textbf{fi} \\
& \quad \in \mathbb{N} \\
]| &
\end{array}
$$

(size—order rule)

3. Polynomials over $\{0,1\}$

Consider now the representation of numbers in binary form. A binary numeral is a list of 1's and 0's in which leading 0's are insignificant. Thus $11 = 011 = 0011$ etc. A binary numeral is, however, one particular interpretation of such a list. More generally we may regard such a list as denoting a polynomial; thus, 11 denotes $1 \times x + 1$. Using Λ to denote the empty list we can define a type, called P say, of lists of 0's and 1's in which leading 0's are insignificant as follows.

$$\frac{}{\Lambda \in P} \quad \Lambda\text{-introduction}$$

$$\frac{p \in P}{p0 \in P} \quad 0\text{-introduction}$$

$$\frac{p \in P}{p1 \in P} \quad 1\text{-introduction}$$

$$\frac{}{\Lambda 0 = \Lambda \in P} \quad \text{leading zeroes}$$

Given these four introduction rules the elimination rule for P has four premises. The four premises state that to define a function over P it is necessary to consider three cases — the case where the argument is Λ, the case where it is of the form $p0$ and the case where it is of the form $p1$ — and furthermore it is necessary to show that the insignificance of leading zeroes is respected. Specifically, we have the following rule.

$$\frac{\begin{array}{l}[\![\ w \in P \\ \triangleright\ C(w)\ type \\ ]\!] \\ x \in P \\ y_1 \in C(\Lambda) \\ [\![\ p \in P; h \in C(p) \\ \triangleright\ y_2(p,h) \in C(p0) \\ ]\!] \\ [\![\ p \in P; h \in C(p) \\ \triangleright\ y_3(p,h) \in C(p1) \\ ]\!] \\ y_2(\Lambda, y_1) = y_1 \in C(\Lambda)\end{array}}{P\text{-}elim(x, y_1, y_2, y_3) \in C(x)} \quad P\text{-elimination}$$

3.1 Remainder Computation in Integer Arithmetic

The type P can be used to model some of the tasks that a hardware designer faces. Suppose that we regard objects of P as binary representations of natural numbers; the task is to construct functions that represent the common arithmetic operations, addition, subtraction and so on. All the functions must be implemented within the type; we cannot assume, for instance, that an inequality test on natural numbers is available to us. Thus the addition function we require has type $P \times P \longrightarrow P$. Here we shall describe a package of operations on P leading up to the construction of a remainder function.

To verify that the constructed operations on objects of P do indeed represent operations on numbers it is necessary to relate the two. Thus we shall define an operation denoted by a postfixed $'$ that maps an object p of P to an object p' of $\mathbb{N}$. We also define an operation $add1 \in P \longrightarrow P$ and prove that $add1(p)' = succ(p')$. Similar commutativity properties are discussed for addition and remainder computation.

To begin with let us consider the function of type $P \longrightarrow \mathbb{N}$ that, given $p \in P$, determines the number represented by p. This is $abs(p)$ where $abs(p) \equiv P\text{-}elim(p, 0, (y, n)2 \times n, (y, n)2 \times n + 1)$. Note that to verify the well-definedness of abs we have to verify the following.

(a) $0 \in \mathbb{N}$

(b) $|[\ y \in P; n \in \mathbb{N}$
$\triangleright\ 2 \times n \in \mathbb{N}$
$]|$

(c) $|[\ y \in P; n \in \mathbb{N}$
$\triangleright\ 2 \times n + 1 \in \mathbb{N}$
$]|$

(d) $0 = 2 \times 0 \in \mathbb{N}$

Clause (d) is of course the appropriate instance of the leading coefficient rule. For brevity we denote $abs(p)$ by p'.

Consider now the function $add1 \in P \longrightarrow P$ which represents the operation of adding one to a number. This is defined as

$$add1(p) \equiv P\text{-}elim(p, \Lambda 1, (q, h)q1, (q, h)h0)$$

Essentially this states that adding 1 to p' is represented by $\Lambda 1$ if p is Λ (i.e. p' is zero), by $q1$ if p takes the form $q0$, and by $h0$ if p takes the form $q1$ and the result of adding 1 to q is h.

Formally we can verify $add1$ by establishing the judgement

$$|[\ p \in P$$
$$\triangleright\ add1(p) \in \{q \in P |\ q' = succ(p')\}$$
$$]|$$

This is done as follows.

0.0 |[$p \in P$

▷ % The family of types C in the P-elim rule is taken to be $(w)\ \{h \in P |\ h' = succ(w')\}$

% The four premises of the P-elim. rule are verified in turn.

%

% 1st premise

%

{Λ-intro, 1-intro}

0.1 $\Lambda 1 \in P$

0.2 $succ(\Lambda')$

$=_{\mathbb{N}}$ { P-comp, subst. }

$succ(0)$

$=_{\mathbb{N}}$ { P-comp, $\mathbb{N}$-comp }

$(\Lambda 1)'$

% 0.1,0.2, subtype-intro %

0.3 $\Lambda 1 \in \{h \in P |\ h' = succ(\Lambda')\}$

%

% 2nd premise

%

0.4.0 |[$q \in P; h \in \{h \in P |\ h' = succ(q')\}$

▷ % 1-intro %

0.4.1 $q1 \in P$

0.4.1 $succ((q0)')$

$=_{\mathbb{N}}$ { P-comp,subst }

$succ(2 \times q')$

$=_{\mathbb{N}}$ { P-comp, $\mathbb{N}$-comp }

$(q1)'$

% 4.1, 4.2, subtype intro. %

0.4.1 $q1 \in \{h \in P |\ h' = succ((q0)')\}$

]|

%

% 3rd premise

%

0.4.0 |[$q \in P; h \in \{h \in P |\ h' = succ(q')\}$

▷ % 0-intro %

0.4.1 $h0 \in P$

0.4.2 $succ((q1)')$

$=_{\mathbb{N}}$ { P-comp, subst. }

$succ(2 \times q' + 1)$

$=_{\mathbb{N}}$ { $\mathbb{N}$-comp }

$2 \times succ(q')$

$=_{\mathbb{N}}$ { 0.5.0, subst }

$2 \times h'$

$=_{\mathbb{N}}$ { P-comp }

$(h0)'$

% 0.5.1,0.5.2, subtype-intro %

0.4.3 $h0 \in \{h \in P |\ h' = succ((q1)')\}$

]|

%

% 4th premise

%

0.4 $\Lambda 1$

$=_P$ { refl. }

$\Lambda 1$

]|

Although apparently long-winded the above argument does indeed reflect the steps taken to construct the function $add1$. In later examples we supply the function and possibly some of the steps in its construction and leave the reader to verify it against its specification.

Given the function $add1$ we can define a representation function that computes a binary numeral from a given natural number n.

$$rep(n) \equiv \mathbb{N}\text{-}elim(n, \Lambda, (m, x)add1(x))$$

It is easy to see that rep and abs are inverses of each other, i.e.

$$abs(rep(n)) =_{\mathbb{N}} n$$

and

$$rep(abs(p)) =_P p.$$

Complementary to $add1$ is the function $sub1$ that subtracts 1 from a binary numeral p.

$$sub1 \in (\forall p \in \{p \in P |\ p \neq \Lambda\})\ \{q \in P |\ add1(q) = p\}$$
$$sub1(p) \equiv P\text{-}elim(p, z, (q, r)r1, (q, r)q0)$$

It is interesting to note that $sub1$ does not always compute the shortest representation of a number. For example,

$$\begin{array}{ccc} & sub1(\Lambda 1) & \\ = & \{\ P\text{-comp}\ \} & \\ & \Lambda 0. & \end{array}$$

The addition function is harder to develop. The argument we use is that the result of adding Λ to any binary numeral is always that numeral; otherwise we consider the four different combinations of the forms $r0$, $r1$ for p and $t0$ and $t1$ for q. Adding $r0$ to $t0$ is calculated by appending 0 to the result of adding r to t. Adding $r0$ to $t1$ is calculated by appending 1 to the result of adding r to t. Similarly for adding $r1$ to $t0$. Adding $r1$ to $t1$ is calculated by appending 0 to the result of adding one (using $add1$) to the result of adding r and t. The realization of this argument using P-elimination is the difficult part. For binary numeral p we take as inductive hypothesis

$$C(p) \equiv (\forall q \in P)\ \{s \in P |\ s' = p' + q'\}$$

An object of $C(p)$ is thus a function which when applied to an object q of P yields $add(p, q)$. Specifically, $add(p, q) \equiv g(q)$ where

$$\begin{array}{l} g \equiv P\text{-}elim(p \\ \qquad ,\lambda q.q \\ \qquad ,\{p = r0; f \in (\forall q \in P)\ \{s \in P |\ s' = r' + q'\}\} \\ \qquad (r, f)\lambda q.P\text{-}elim(q \\ \qquad\qquad ,r0 \\ \qquad\qquad ,\{q = t0\} \\ \qquad\qquad (t, s)(ft)0 \end{array}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad ,\{q = t1\} \\
\qquad\qquad\qquad\qquad\quad (t,s)(ft)1 \\
\qquad\qquad\qquad\qquad\quad) \\
,\{p = r1; f \in (\forall q \in P)\ \{s \in P |\ s' = r' + q'\}\} \\
(r,f)\lambda q.P\text{-}elim(q \\
\qquad\qquad\qquad\qquad\quad ,r1 \\
\qquad\qquad\qquad\qquad\quad ,\{q = t0\} \\
\qquad\qquad\qquad\qquad\quad (t,s)(ft)1 \\
\qquad\qquad\qquad\qquad\quad ,\{q = t1\} \\
\qquad\qquad\qquad\qquad\quad (t,s)(add1(ft))0 \\
) \\
)
\end{array}
$$

We supply below some of the details of verifying the third and fourth premises of the P-elimination rule.

0.0 |[$p \in P$

▷ % Use P-elim with $C(p) \equiv (\forall q \in P)\ \{s \in P |\ s' = p' + q'\}$ %

% Case $p = r1$

0.1.0 |[$r \in P; f \in (\forall q \in P)\ \{s \in P |\ s' = r' + q'\}$

▷ % Preparatory to using $\forall$-intro. %

0.1.1.0 |[$q \in P$

▷ % Use P-elim on q with $C(q) \equiv \{s \in P |\ s' = (r1)' + q'\}$ %

0.1.1.1 $r1 \in \{s \in P |\ s' = (r1)' + \Lambda'\}$

0.1.1.2.0 |[$t \in P; s \in \{s \in P |\ s' = (r1)' + t'\}$

▷ % 0.1.0, 0.1.1.2.0, $\forall$-elim %

0.1.1.2.1 $ft \in \{s \in P |\ s' = r' + t'\}$

% 0.1.1.2.1, P-comp., $\mathbb{N}$-comp. %

0.1.1.2.2 $(ft)1 \in \{s \in P |\ s' = (r1)' + (t0)'\}$

]|

0.1.1.2.0 |[$t \in P; s \in \{s \in P |\ s' = (r1)' + t'\}$

▷ % 0.1.0, 0.1.1.3.0, $\forall$-elim %

0.1.1.2.1 $ft \in \{s \in P |\ s' = r' + t'\}$

% 0.1.1.3.1, P-comp., $\mathbb{N}$-comp. %

0.1.1.2.2 $(add1(ft))0 \in \{s \in P |\ s' = (r1)' + (t1)'\}$

]|

% We now have to show that $r1 = (f\Lambda)1$

% in order to verify the final premise of P-elim

{ property of $+$}

0.1.1.2 $f\Lambda \in \{s \in P |\ s' = r' + \Lambda'\} \equiv f\Lambda \in \{s \in P |\ s' = r'\}$

% 0.1.1.4, monotonicity %

0.1.1.3 $(f\Lambda)1 = r1 \in P$

% 0.1.1.1, 0.1.1.2, 0.1.1.3, 0.1.1.4, P-elim. %

0.1.1.4 $P\text{-}elim(q, r1, (t,s)(ft)1, (t,s)(add1(ft))0) \in \{s \in P |\ s' = (r1)' + q'\}$

]|

% $\forall$-intro %

0.1.1 $\lambda q.P\text{-}elim(q, r1, (t,s)(ft)1, (t,s)(add1(ft))0) \in (\forall q \in P)\ \{s \in P |\ s' = (r1)' + q'\}$

%

% To verify the final premise of the P-elimination rule we have to show that

%$\lambda q.q = \lambda q.P\text{-}elim(q, \Lambda 0, (t,s)((\lambda q.q)t)0, (t,s)((\lambda q.q)t)1)$

{ leading zeroes}

0.1.2 $\Lambda 0 = \Lambda \in P$

0.1.3.0 |[$t \in P$

0.1.3.1 ▷ $(\lambda q.q)t = t \in P$

]|

% 0.1.3, 0.1.4, monotonicity %

0.1.3.0 |[$q \in P$

0.1.3.1 ▷ $P\text{-}elim(q, \Lambda 0, (t,s)((\lambda q.q)t)0, (t,s)((\lambda q.q)t)1) = P\text{-}elim(q, \Lambda, (t,s)t0, (t,s)t1) \in P$

]|

% 1.5, closure properties of P, extensionality %

0.1.3 $\lambda q.P\text{-}elim(q, \Lambda 0, (t,s)((\lambda q.q)t)0, (t,s)((\lambda q.q)t)1) = \lambda q.q \in (\forall q \in P)\ \{s \in P |\ s' = \Lambda' + q'\}$

% Finally, P-elim may be used %

0.1.4 ...

]|

]|

The same idea that was used to construct the addition function can be used to compare binary numerals p and q. Let us denote by $p.\ <=>\ .q$ the type

$$\{z \in P |\ (z \neq \Lambda) \wedge add(p,z) = q\} \vee (p = q) \vee \{z \in P |\ (z \neq \Lambda) \wedge add(q,z) = p\}.$$

We also use constants l,m and r to denote respectively the injection of a value into the left, middle or right component of this disjunction, and $\omega_{\mathbf{lmr}}(x,s,t,u)$ to denote the corresponding elimination construct. To construct an object of $p.\ <=>\ .q$.we prove by induction on p that the type

$$C(p) \equiv (\forall q \in P)(p.\ <=>\ .q)$$

is non-empty. The case $p = \Lambda$ is not as straightforward as might be expected since we have to take account of leading zeroes. Using induction on q, if $q = \Lambda$ then p and q are equal, if $q = t1$ for some t then p is certainly less than q, if $q = t0$ for some t then p is less than q if p is less than t and by the amount $t0$, but if $p = t$ then $t = t0 = \Lambda$. Thus the object constructed in the base case of $C(p)$ is

$\lambda q.P\text{-}elim(q$
$\quad ,\{q = \Lambda\}\mathbf{m}(e)$
$\quad ,\{q = t0; s \in \Lambda.\ <=>\ .t\}$
$\quad (t,s)\omega_{\mathbf{lmr}}(s,(d)\mathbf{l}(t0),(d)\mathbf{m}(\mathbf{e}),\mathbf{z})$
$\quad ,\{q = t1; s \in \Lambda.\ <=>\ .t\}(t,s)\mathbf{l}(t1)$
$\quad)$

The case $p = r0$ is handled like addition. We use induction on q. The complication just considered reoccurs in the case that q is Λ. When q is $t0$ then r and t are compared. The sign (l, m or r) of $r0.\ <=>\ .t0$ is the same as the sign of $r.\ <=>\ .t$ and, in the case that r and t are different, 0 is appended to the difference of r and t. Finally, when q is $t1$ we use the fact that $2 \times t' + 1 - 2 \times r' = 2 \times (t' - r') + 1 = -(2 \times (t' - r') - 1)$. Thus, if the sign of $r.\ <=>\ .t$ is l or m ($r' < t'$) then p' is less than q by the amount $(s1)'$ where s is the difference between r and t. Otherwise p' is greater than q' by the amount $sub1(s)0'$ where, again, s is the (positive) difference between r and t. Summarising, the object constructed in the case that $p = r0$ takes the following form.

$(r,f)\{p = r0; f \in (\forall q \in P)(r.\ <=>\ .q)\}$
$\quad \lambda q.P\text{-}elim(q$
$\quad\quad ,\{q = \Lambda\}\omega_{\mathbf{lmr}}(f\Lambda,\mathbf{z},(d)\mathbf{m}(e),(d)\mathbf{r}(r0))$
$\quad\quad ,(t,s)\{q = t0; s \text{ is insignificant}\}$
$\quad\quad \omega_{\mathbf{lmr}}(ft,(d)\mathbf{l}(d0),(d)\mathbf{m}(\mathbf{e}),(d)\mathbf{r}(d0))$
$\quad\quad ,(t,s)\{q = t1; s \text{ is insignificant}\}$
$\quad\quad \omega_{\mathbf{lmr}}(ft,(d)\mathbf{l}(d1),(d)\mathbf{l}(\Lambda 1),(d)\mathbf{r}(sub1(d)1))$
$\quad\quad)$

A similar argument leads to the following object for the case that $p = r1$.

$(r,f)\{p = r1; f \in (\forall q \in P)(r.\ <=>\ .q)\}$
$\quad \lambda q.P\text{-}elim(q$
$\quad\quad ,\{q = \Lambda\}\mathbf{r}(r1)$

$$
\begin{array}{l}
,(t,s)\{q = t0; s \text{ is insignificant}\} \\
\omega_{\mathbf{lmr}}(ft, (d)\mathbf{l}(sub1(d0)), (d)\mathbf{r}(\Lambda 1), (d)\mathbf{r}(d1)) \\
,(t,s)\{q = t1; s \text{ is insignificant}\} \\
\omega_{\mathbf{lmr}}(ft, (d)\mathbf{l}(d0), (d)\mathbf{m}(\mathbf{e}), (d)\mathbf{r}(d0)) \\
)
\end{array}
$$

The last step before we can exhibit the algorithm for remainder computation is to present a function red that evaluates the expression "**if** $p > q$ **then** $p - q$ **else** p **fi**." This is easy to compute using . <=> . and takes the following form

$$
\begin{array}{rl}
red(p,q) \equiv & \omega_{\mathbf{lmr}}(p. <=> .q \\
& ,\{p < q\}(d)p \\
& ,\{p = q\}(d)\Lambda \\
& ,\{p > q\}(d)d \\
&) \\
\in & \{s \in P |\ s' = \textbf{if } p' \geq q' \textbf{ then } p' - q' \textbf{ else } p' \}
\end{array}
$$

We now have all the ingredients for remainder computation. The value of $rem(p,q)$, the remainder after dividing p by q, is calculated by P-elimination on p. If p is Λ then $rem(p,q)$ is also Λ. Otherwise we use the fact that

$$(2 \times k + b) \bmod m = (2 \times (k \bmod m) + b) \bmod m$$

where k, b and m are natural numbers. Moreover, when $0 \leq b \leq 1$, we have

$$2 \times (k \bmod m) + b \leq 2 \times (m-1) + 1 = 2 \times m - 1.$$

Finally, for a number $n \leq 2 \times q - 1$

$$n \bmod m = \textbf{if } n > m \textbf{ then } n - m \textbf{ else } n \textbf{ fi}.$$

Translating the above argument into binary numerals we obtain the following code.

$$
\begin{array}{rl}
rem(p,q) \equiv & P\text{-}elim(p \\
& ,\{p = \Lambda\}\Lambda \\
& ,\{p = r0; s = rem(r,q)\}(r,s)red(s0,q) \\
& ,\{p = r1; s = rem(r,q)\}(r,s)red(s1,q) \\
&) \\
\in & \{s \in P |\ (\exists d \in \mathbb{N})(p' = d \times q' + s') \wedge 0 \leq s' < q' \}
\end{array}
$$

Exercise.

Construct the function $nosigbits \in (\forall p \in P)\ \{s \in P |\ s'$ is the smallest number such that $2^{s'} > p' \}$. (In other words $nosigbits(p)$ is the (binary representation of) the number of significant bits in p.)

Exercise.

Construct the function $.eq. : P \times P \longrightarrow P$ such that $p.eq.q \in (p = q) \vee \neg(p = q)$.

References

[1] R.C. Backhouse ,"On the meaning and construction of the rules in Martin-Löf's theory of types"Rijksuniversiteit Groningen, Department of Mathematics and Computing Science, Report No.

[2] B. Nordström, J. Smith and K. Petersson "An introduction to Martin-Löf's Theory of Types", Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology, Sweden, Midsummer 1986.

[3] P.Chisholm "A Theory of Sets in ~~Martin-Löf's~~ Constructive Type Theory" Heriot-Watt University Technical Report (1987)