Formal Aspects of Computing (1991) 3: 218–252
© 1991 BCS

Formal Aspects of Computing

Normalising the Associative Law: An Experiment with Martin-Löf's Type Theory

Michael Hedberg

Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden

Keywords: Functional programming; Type theory; Inductive types; Associative law

Abstract. Martin-Löf's type theory contains a logic, a specification language and a programming language, so it is a tool with different uses. Although it is traditionally used as an *integrated* programming logic, it may well be used as an *external* logic, which is necessary if one wants to use the formalism of type theory to verify the correctness of an external program. Different tools, such as well founded recursion, measure functions, or the separation of correctness into termination and partial correctness, may be used to obtain a correct type theory program. Type theory is viewed as an *open* system with respect to *inductively defined* types and predicates, which makes it easy to represent an external program as a *graph*. Formal proofs have been edited using Larry Paulson's ISABELLE.[1]

1. Introduction

1.1. Conditional Expressions

In Boyer and Moore [BoM79], a tautology checker for classical propositional logic is presented. The logic is based on the ternary if-connective and names of

Correspondence and offprint requests to: Michael Hedberg, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden.

[1] The paper is a slightly revised version of a Thesis for the Licentiate Degree in Computer Science at University of Göteborg with the same title. The Thesis lists some ISABELLE proof sessions in its appendix.

the form at(. . .) for atomic propositions.[2] The expressions denoting propositions are also called *conditional* expressions. The difficult part of the tautology checker is to eliminate tested ifs from an expression. This subproblem may well be described as normalisation with respect to the equation

$$\mathsf{if}(\mathsf{if}(x, y, z), u, v) = \mathsf{if}(x, \mathsf{if}(y, u, v), \mathsf{if}(z, u, v))$$

This leads to a *derivation* problem, that of constructing a normaliser. Boyer and Moore also give a general recursive program,

$$\begin{cases} \mathsf{norm_cond}(\mathsf{at}(a)) = \mathsf{at}(a) \\ \mathsf{norm_cond}(\mathsf{if}(\mathsf{at}(a), u, v)) = \mathsf{if}(\mathsf{at}(a), \mathsf{norm_cond}(u), \mathsf{norm_cond}(v)) \\ \mathsf{norm_cond}(\mathsf{if}(x, y, z), u, v)) = \mathsf{norm_cond}(\mathsf{if}(x, \mathsf{if}(y, u, v), \mathsf{if}(z, u, v))) \end{cases}$$

which leads to the *verification* problem of proving that this program is indeed a normaliser. The difficult thing here is termination.

1.2. Programming Logics

In order to reason about programs, one has to have a programming logic in which to express proofs of termination or correctness. What logic to use depends on the programming language. Hoare logic, for instance, is suitable for imperative programs, Hennessy–Milner logic talks about the parallel and nondeterministic agents of CCS, and LCF is about functional programs. All these are what Girard [Gir86] calls *external* logics, because programs and proofs live in different worlds – the correctness proof is thought of as a comment, external to the program. Martin-Löf's type theory, on the other hand, is an *integrated* logic. The concept of integrated logic is made possible through the Curry–Howard analogy between proofs and functional expressions, so the dichotomy between external and integrated logic concerns primarily functional programming. The Curry–Howard analogy is the main idea behind type theory as a programming logic and it is a useful conceptual tool that may play a central rôle in the implementation of functional programming logics. When enhanced to a doctrine, however, it becomes problematic, because it leads to the unnatural contamination of values with proof garbage. If you ask your computer to print an even natural number of its liking, and it chooses the number 6, then you want it to print this number, without justifying its answer by explaining *why* 6 is an even number. Although silly, the example illustrates the essence of the contamination problem, which has inspired to the formulation of type theories with a relaxed Curry–Howard doctrine.

Other respects in which programming logics may differ are the following:

Is the *logic* constructive or classical?
Are *objects* total or partial?
Is *recursion* primitive or general?

[2] Properly speaking, there are names for the truth values as well, but they will have no bearing on the subproblem addressed.

In this classification, Martin-Löf's type theory has *integrated constructive* logic, *total* objects and *primitive* recursion, while LCF has *external classical* logic, *partial* objects and *general* recursion. Paulson [Pau86] compares type theory and LCF as representatives of constructive respective classical logic, and Dybjer [Dyb91] extends this comparison by also regarding logics which lie between these extremes and he emphasises the external versus integrated dichotomy. To illustrate their points, they let logics come to work at the normaliser for conditional expressions.

1.3. Our Concern

The purpose of this work is fourfold:

To see how *different* proof techniques, such as measure functions and well-founded recursion, can be expressed in *one* logic, Martin-Löf's type theory

To pay more attention to *correctness* than Paulson, whose main concern is termination

To perform computer checked derivations, using ISABELLE; see Paulson [Pau89]

To illustrate the use of inductively defined types and predicates in connection with Martin-Löf's type theory, which gives a more direct problem representation than encodings using well-orderings

We will, however, work with a simplified problem which will be recognised as normalising semigroup expressions with respect to the associative law.

1.4. A Simpler Problem

The associative law arises from the conditional expressions by restricting the conditional expressions to be in *multiplicative form*,[3] which means that each if has identical second and third arguments. Such expressions can be viewed as being generated from atoms by a binary operator,

$$\mathsf{mul}(x, y) = \mathsf{if}(x, y, y)$$

Forcing the two sides of the normalising equation for conditional expressions into multiplicative form, one arrives at the following normalising equation for multiplicative form expressions

$$\mathsf{mul}(\mathsf{mul}(x, y), u) = \mathsf{mul}(x, \mathsf{mul}(y, u))$$

which is the associative law for mul. Similarly, the general recursive program takes the form

$$\begin{cases} \mathsf{norm_mul}(\mathsf{at}(a)) & = \mathsf{at}(a) \\ \mathsf{norm_mul}(\mathsf{mul}(\mathsf{at}(a), z)) & = \mathsf{mul}(\mathsf{at}(a), \mathsf{norm_mul}(z)) \\ \mathsf{norm_mul}(\mathsf{mul}(\mathsf{mul}(x, y), z)) & = \mathsf{norm_mul}(\mathsf{mul}(x, \mathsf{mul}(y, z))) \end{cases}$$

[3] We will later introduce *multiplicative expressions* as a datatype of its own and the concept *multiplicative form expression* will be used only in this discussion as a tie between *conditional* and *multiplicative* expressions.

We shall see that, by following the advice of Polya to look at a simpler problem, we gain some insight into the original one. A primitive recursive algorithm for the associative law generalises nicely to the case of conditional expressions. On the other hand, a program using fixed point approximations to simulate the general recursive program, behaves quite well in the associative case, since an exact measure can be given and the correctness can be proved without using any arithmetic at all. For the conditional expressions, no exact measure can be given and the correctness proof must use some arithmetic.

1.5. What is Primitive Recursion?

Traditionally primitive recursion refers to a scheme for defining functions from Nat^n to Nat. In type theory this notion is generalised to comprise any recursive function definition according to the elimination rule for any type. Barring transfinite recursion, which is of no concern to us in this paper, the generalisation may be performed in two steps, which are related to the nature of the types occurring as inputs and output to the function being defined by primitive recursion:

Structural recursion. The type of natural numbers may be replaced by arbitrary elementary types. An elementary type is essentially the term language of a one sorted first order signature, so it is the initial solution to a set equation of the form

$$D \cong D^{n_1} + \cdots + D^{n_m}$$

where "+" stands for disjoint union. A function defined by recursion along the definition of an elementary type is said to be defined by primitive recursion. The phrase "structural recursion" is perhaps more suitable when the elementary type is something else than the natural numbers.

Higher type recursion. The values returned need not come from an elementary type, but from an arbitrary type. The way, for instance, to define Ackermann's function in type theory is first to define the curried version by primitive recursion of higher type and then to apply the result to the second argument. Such functions, defined by primitive recursion of higher type, are also called *primitive recursive functionals.*

This terminology is worth mentioning since the type theory usage of "primitive recursion" is much wider than the traditional usage.

1.6. Overview of the Paper

In Section 2, multiplicative expressions are introduced and the associative law is briefly analysed as a rewriting system. Section 3 gives a summary of Martin-Löf's type theory and the inductive types and predicates needed for specifying the normalisation problem. Section 4 shows how the general recursive program, norm_mul, can be represented in type theory, not as a type theory program but as an inductively defined binary relation, the *graph* of the program, and how correctness is expressed in terms of this graph. In Section 5, an operator, assrec, based on a particular termination proof, is introduced and applied to get an adaptation, norm_ho, to the associative law

of Paulson's "obviously total" normaliser and to get an exact measure, to be used in fixed point approximations. Section 6 contains primitive recursive normalisers, norm_pr and norm_cond_pr, both for multiplicative and conditional expressions and a comment on Taylor [Tay88] who has also studied the associative law. Section 7 contains some concluding remarks.

2. Multiplicative Expressions and Normalising the Associative Law

The associative law is the basic axiom of semigroups. In fact, a semigroup is by definition a set together with an associative binary operator acting on it.

A *multiplicative* expression is an expression intended to denote an element of a semigroup. To be more precise, we say that multiplicative expressions are generated by the grammar

$$\langle \mathsf{Mul}(Id) \rangle ::= \mathsf{at}(\langle Id \rangle) \mid (\langle \mathsf{Mul}(Id) \rangle \bullet \langle \mathsf{Mul}(Id) \rangle)$$

where the set *Id* is used to label the atomic parts of the expression. Throughout this paper, the set of atom labels, *Id*, will be fixed to Nat, the set of natural numbers, and we write Mul rather than Mul(Nat). An *interpretation* of the language Mul(*Id*) is given by laying down what associative operator $\bullet$ stands for and how to map the atoms to semigroup elements. Two expressions are *equivalent* when they are mapped to the same semigroup element under every interpretation. An equivalent, but more manageable, definition is that two expressions are equivalent if they can be *proved* to be equivalent using only standard equational reasoning and the associative law. To be explicit, we have the following rules for establishing equivalence between multiplicative expressions:

$x \sim x$
$y \sim x$, if $x \sim y$
$x \sim z$, if $x \sim y$ and $y \sim z$
$x \bullet y \sim x' \bullet y'$, if $x \sim x'$ and $y \sim y'$
$(x \bullet y) \bullet z \sim x \bullet (y \bullet z)$

Among these rules, all but the last one express standard equational reasoning. What makes this syntactical notion of equivalence interesting is that it is equivalent to the semantical notion mentioned above. The equivalence of the semantical and the syntactical notions is a consequence of the standard initial algebra construction, which depends on forming the quotient set Mul/$\sim$, where by definition

$$[x]_\sim = [y]_\sim \in \mathsf{Mul}/\sim \quad \text{if and only if } x \sim y$$

The difference between the set and the quotient set is the use of the equality sign. The equality in the set may be simple, in our case it is just syntactical identity, whereas the equality in the quotient set may be quite complicated. The quotient set "construction" does not really construct anything but reflects only the knowledge that a certain relation has several formal properties in common with the equality sign. From a computer science point of view, it is

natural to pay special attention to sets that are equipped with decidable equality relations. If we want to be able to use the quotient set as a set with decidable equality, we have to supply a procedure for refuting or accepting proposed equalities in the quotient set.

One method, or rather class of methods, is to reduce $\sim$-equivalence to ordinary equality in Mul. This involves defining a class of expressions, the class of *normal* expressions, on which the quotient projection acts as a bijection. This means that the class is small enough to guarantee that members of the class are equivalent only when they are equal, but also big enough so that every expression has an equivalent in the class.

A special case of this method is when the associative law is used as a basis for a rewriting system. Then the associative law generates a (one step) *rewriting* relation, which is defined like the $\sim$-relation, but with the clauses for reflexivity, symmetry and transitivity excluded, and the congruence clause split into two clauses. In this context an expression is in normal form when it cannot be rewritten, that is, it does not match the left-hand side of the associative law and nor do any of its subexpressions. If the rewriting system is *confluent* and *noetherian,* then one has a relative[4] decision procedure for $\sim$-equivalence, namely that of repeatedly rewriting expressions until they are in normal form and then checking equality. This procedure always terminates provided the rewriting system is noetherian. The method, of course, can never erroneously answer "yes", and when the rewriting system is confluent, it can never give an erroneous "no"-answer.

This rewriting system for the associative law is noetherian. This is proved by exhibiting a function $\mu : \mathsf{Mul} \to \mathsf{Nat}$, which is properly decreasing with rewriting. Such a function is defined by primitive recursion:

$$\begin{cases} \mu(\mathsf{at}(a)) = 0 \\ \mu(x \bullet y) \;\; = 1 + 2\mu(x) + \mu(y) \end{cases}$$

Note how the measure gives more weight to the left subexpression. Now

$$\mu((x \bullet y) \bullet z) = 3 + 4\mu(x) + 2\mu(y) + \mu(z)$$

and

$$\mu(x \bullet (y \bullet z)) = 2 + 2\mu(x) + 2\mu(y) + \mu(z)$$

so the left-hand side of the associative law does indeed have a greater μ than the right-hand side. This observation alone does not prove that μ decreases with rewriting in general. One has also to consider rewriting in a context, that is, one has to prove that

$$\mu(x \bullet y) > \mu(x' \bullet y) \quad \text{if } \mu(x) > \mu(x')$$

and

$$\mu(x \bullet y) > \mu(x \bullet y') \quad \text{if } \mu(y) > \mu(y')$$

which of course follows immediately from the definition of μ and elementary arithmetic.

The rewriting system is confluent since it is noetherian and *locally* confluent. Local confluence is proved by *critical pair* analysis.

[4] It is rather a method for reducing equivalence to equality as, for instance, equality in $\mathsf{Mul}(A)$ in an obvious way is reduced to equality in A.

Note that the rewriting process, as described above, is nondeterministic. It does not prescribe which of several possible rewritings should be chosen in a particular step. The programming problem we will consider is that of giving a *deterministic* algorithm for finding the normal form of an expression. We suppose that "normal" is given by the *inductive* definition

> An expression is normal, if it is either atomic or the product of an atomic and a normal expression.

and we state the problem:

> Find an algorithm which, given any multiplicative expression, produces a $\sim$-equivalent and normal one.

3. Martin-Löf Type Theory

The reader is assumed to be familiar with Martin-Löf's type theory as presented in for instance Martin-Löf [Mar84] or Nordström et al. [NPS90]. The purpose of this exposition is to present the syntax and to introduce the type formers which are essential for a simple statement of the normalisation problem.

3.1. Introduction

The present theory, which we call Martin-Löf Type Theory, can be summarised as *intensional, polymorphic do-it-yourself*[5] type theory without well-orderings and universes. It has four forms of judgement.[6]

Judgement form	*Intuitive reading*
A **Type**	A is a type
$A = B$	A and B are equal types
$a : A$	a is an element in the type A
$a = b : A$	a and b are equal elements of the type A

Judgements are really made under assumptions, which are lists of variable declarations

$$[x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n]$$

also called *contexts*. The most important feature of these contexts is that one must know that something is a type before one declares a variable of that type. The "propositions as types" principle used in type theory means that a proposition is identified with the type of its proofs. The Curry–Howard analogy shows how, under this identification, the standard connectives correspond to familiar type forming operators. The set of proofs of the conjunction of two propositions, for instance, is the cartesian product of their respective proof sets. In type theory, therefore, there do not exist any

[5] Cf. Backhouse et al. [BCM89].
[6] We will not make a distinction between *set* and *type* and use the words as synonyms.

propositions, but only their corresponding proof sets. For readability, however, we will use the familiar logical symbols when a type is used as a proposition. Here are some of the standard type forming constants and, in relevant places, their logical synonyms.

$\prod \equiv \forall$, for the cartesian product of a family of types

$$(\lambda x)\,\mathsf{succ}(x) : (\prod x : \mathsf{Nat})\mathsf{Nat}$$

$$(\lambda x, y) f(x, y) = (\lambda x)((\lambda y) f(x, y)) = \lambda((x)\lambda((y) f(x, y)))$$

Application is written as infix ` and associates to the left.

$$\frac{f(x) : B(x)[x : A] \quad a : A}{((\lambda x) f(x))`a = f(a) : B(a)}$$

$$f`x`y = (f`x)`y$$

$\rightarrow \equiv \Rightarrow$, for the type of functions between two types

$$A \rightarrow B = (\prod x : A)B, \text{ (where } B \text{ does not depend on } x)$$

$\sum \equiv \exists$, for the sum (disjoint union) of a family of types

$$\langle a, p \rangle : (\sum x : A)B(x)[a : A, p : B(a)]$$

$\times \equiv \bigwedge$, for binary cartesian product of types

$$A \times B = (\sum x : A)B, \text{ (where } B \text{ does not depend on } x)$$

$$\mathsf{split}(\langle a, b \rangle, (x, y)x) = a : \cdots$$

$+ \equiv \bigvee$, for binary sum of types
$\mathsf{Eq}(A, x, y) \equiv x =_A y$, for the equality type
Zero, the empty type
One, the one element type
Nat, the type of natural numbers

Note the difference between *judgemental* equality, $m = n : \mathsf{Nat}$, and *propositional* equality, $m =_{\mathsf{Nat}} n$, the former being a judgement and the latter a proposition (= type). A judgemental equality cannot be proved by induction or case analysis. You prove a judgement by applying the rules of type theory. You prove a proposition, P, by proving a judgement of the form $p : P$ for some p. The judgemental equality between types has no propositional counterpart!

Note also the distinction between *conversion* and *reduction*. Reduction is the reflexive, transitive and congruent closure of the equalities stated by elimination rules, while conversion adds symmetry. Given a type, A, and elements, $a, b : A$, we say that a *converts* to b, if we can prove the judgemental equality $a = b : A$, and if we can do it *without* any appeal to the symmetry of equality, a *reduces* to b. These concepts apply to types as well. Given two types, one may reduce to the other or they may be convertible.

A third distinction is that between judgemental equality and metadefinitions which are written as in

$$\mathsf{add} \equiv (x, y)\, \mathsf{natrec}(x, y, (z, u)\, \mathsf{succ}(u))$$

or, equivalently,

$$\mathsf{add}(x, y) \equiv \cdots$$

We will, however, use the ordinary equality sign (=) for metadefinitions where it cannot lead to confusion.

This enumeration of type formers does not mention well-orderings and universes[7] partly because they will not appear in the proofs, but also because they are more difficult to understand. Some of the logical power lost this way is regained by the use of user introduced types and predicates. Such do-it-yourself type theory has been explored by Backhouse et al. [BCM89] (types) and Coquand and Paulin [CoP89] and Dybjer [Dyb89] (predicates).

The types we need to describe our programming problem are

Mul, the type of multiplicative expressions
Norm, the class of normalised expressions
$\sim$, the relation of equivalent expressions

In the subsequent presentation of the new type formers, some complex elimination and equality rules are omitted which are irrelevant to the proofs to come.

Some abuse of notation is illustrated by the following example, where we define the addition function for natural numbers.

The function $\mathsf{add}(x, y) : \mathsf{Nat}[x, y : \mathsf{Nat}]$ is defined by recursion on $x : \mathsf{Nat}$

$$\begin{cases} \mathsf{add}(0, y) & = y \\ \mathsf{add}(\mathsf{succ}(x), y) & = \mathsf{succ}(\mathsf{add}(x, y)) \end{cases}$$

We introduce the symbol add by a typing judgement and some definitional equalities. This means that we omit the explicit definition in terms of natrec and go straight to the typing and equalities that the defeniendum inherits from the definiens in virtue of the elimination rule for (in this case) the natural numbers. We sometimes omit the "colon type" part of equality judgements when this is obvious from the context, and particularly in function definitions like the above, we also omit the variable declarations.

3.2. Multiplicative Expressions

The *multiplicative expressions* are those generated by the grammar

$$\begin{aligned} \langle \mathsf{Mul} \rangle ::= {} & \mathsf{at}(\langle \mathsf{Nat} \rangle) \\ & \mid (\langle \mathsf{Mul} \rangle \bullet \langle \mathsf{Mul} \rangle) \end{aligned}$$

Informally, we will call expressions of the form $\mathsf{at}(x)$ *atomic* and refer to them as *atoms*.

[7] One use, among others, of universes is to prove propositions to the effect that certain things are not equal, as the Peano axiom saying that no successor number equals zero. In Paulsons's ISABELLE implementation, this Peano axiom was explicitly stated as a primitive rule.

Formation

$$\frac{}{\mathsf{Mul}\ \mathbf{Type}}$$

Introduction

$$\frac{a:\mathsf{Nat}}{\mathsf{at}(a):\mathsf{Mul}}$$

$$\frac{x:\mathsf{Mul}\quad y:\mathsf{Mul}}{(x \bullet y):\mathsf{Mul}}$$

Elimination

$$\frac{e:\mathsf{Mul}\quad f(m):C(\mathsf{at}(m))\qquad [m:\mathsf{Nat}]\qquad [x:\mathsf{Mul}, y:\mathsf{Mul}, p:C(x), q:C(y)]\quad g(x,y,p,q):C(x \bullet y)}{\mathsf{mulrec}(e,f,g):C(e)}$$

Equality

$$\frac{n:\mathsf{Nat}\quad \begin{matrix}[\cdots]\\ f\cdots\end{matrix}\quad \begin{matrix}[\cdots]\\ g\cdots\end{matrix}}{\mathsf{mulrec}(\mathsf{at}(n),f,g)=f(n):C(\mathsf{at}(n))}$$

$$\frac{x:\mathsf{Mul}\quad y:\mathsf{Mul}\quad \begin{matrix}[\cdots]\\ f\cdots\end{matrix}\quad \begin{matrix}[\cdots]\\ g\cdots\end{matrix}}{\mathsf{mulrec}((x \bullet y),f,g)=g(x,y,\mathsf{mulrec}(x,f,g),\mathsf{mulrec}(y,f,g)):C(x \bullet y)}$$

The dots indicate the same premises about f and g as in the elimination rule. Note that some premises of a rule may be implicit. The elimination rule, for instance, does not make sense if one does not know already that C really is a family of types over Mul.

Note also that this is *polymorphic* type theory, where essential information about the proofs of judgements are missing in the syntax. A *monomorphic* type theory would demand C as an argument to the recursion operator, mulrec. The point deserves mention since Salvesen [Sal89] has showed that the two versions are not equivalent. It is not possible, to see the polymorphic judgements in general as mere abbreviations of some corresponding monomorphic ones. We do not believe, however, that our proofs depend on polymorphic tricks that could not be performed in the monomorphic theory.

3.3. Normalised Expressions

The concept of a *normalised expression* can be defined either explicitly as a multiplicative expression that cannot be transformed with the (directed) associative law or, equivalently, in an inductive manner by saying that a multiplicative expression is normalised if it is either atomic or the product of an atom (as left factor) and normalised expression.

The two definitions are of course equivalent, but really proving the equivalence or working with the first one, which is perhaps the more natural in the context of normalisation, would involve introducing concepts such as *subexpression* and *context.* We will therefore use the inductive definition,

saying that Norm is the sublanguage of Mul defined by the grammar

$$\langle \mathsf{Norm} \rangle ::= \mathsf{at}(\langle \mathsf{Nat} \rangle) \mid (\mathsf{at}(\langle \mathsf{Nat} \rangle) \bullet \langle \mathsf{Norm} \rangle)$$

Type theory proper has no subtype concept, so to model a subset in type theory we observe that a subset corresponds in an obvious way to a predicate over its base set, that a predicate is a propositional function, and that in type theory propositions are modelled as types. In this way our subset has been transformed into a family of types.

Formation

$$\frac{x : \mathsf{Mul}}{\mathsf{Norm}(x)\ \mathbf{Type}}$$

Introduction

$$\frac{m : \mathsf{Nat}}{\mathsf{norm_a}(m) : \mathsf{Norm}(\mathsf{at}(m))}$$

$$\frac{m : \mathsf{Nat} \quad x : \mathsf{Mul} \quad p : \mathsf{Norm}(x)}{\mathsf{norm_ma}(m, x, p) : \mathsf{Norm}(\mathsf{at}(m) \bullet x)}$$

Elimination

$$\frac{\begin{array}{l} x : \mathsf{Mul} \\ p : \mathsf{Norm}(x) \\ f(m) : C(\mathsf{at}(m), \mathsf{norm_a}(m))[m : \mathsf{Nat}] \\ g(m, y, q, r) : C(\mathsf{at}(m) \bullet y), \mathsf{norm_ma}(m, y, q)) \\ \qquad\qquad [m : \mathsf{Nat}, y : \mathsf{Mul}, q : \mathsf{Norm}(y), r : C(y, q)] \end{array}}{\mathsf{normrec}(p, f, g) : C(x, p)}$$

When C does not depend on its second argument, an alternative elimination rule can be derived. Here it is displayed with suppressed proof objects.

$$\frac{x : \mathsf{Mul} \quad \mathsf{Norm}(x) \quad C(\mathsf{at}(a)) \quad [a : \mathsf{Nat}] \quad [a : \mathsf{Nat}, y : \mathsf{Mul}, \mathsf{Norm}(y), C(y)] \quad C(\mathsf{at}(a) \bullet y)}{C(x)}$$

The grammar describing the set of normalised expressions suggests a definition of normalised expressions as a type of its own. This view is in fact the clue to the primitive recursive program, normalise, given in Section 6.1, but the specification relation becomes more natural the way we do it. On the other hand, a type of normalised expressions would display Norm-elimination as induction along an elementary ground type, which is a simpler concept than induction along an inductively defined predicate.

3.4. Equivalent Expressions

The relation of *equivalent* expressions ($_ \sim _$) is the smallest equivalence relation, compatible with $\bullet$, which makes $\bullet$ associative. Informally, two multiplicative expressions are equivalent if they can be proved to be equal

using only the associative law and standard equational reasoning. Equivalently, they are equivalent if one is obtained from the other after a sequence of associative transformations. An associative transformation consists of replacing a subexpression matching one side of the associative law with the other side. The first formulation is in a natural way modelled as an inductively defined family of types, $(x \sim y)[x, y : \mathsf{Mul}]$.

Formation

$$\frac{x : \mathsf{Mul} \quad y : \mathsf{Mul}}{(x \sim y)\ \mathbf{Type}}$$

Introduction

$$\frac{x : \mathsf{Mul}}{\mathsf{refl}(x) : (x \sim x)}$$

$$\frac{x, y : \mathsf{Mul} \quad p : (x \sim y)}{\mathsf{symm}(x, y, p) : (y \sim x)}$$

$$\frac{x, y, z : \mathsf{Mul} \quad p : (x \sim y) \quad q : (y \sim z)}{\mathsf{trans}(x, y, z, p, q) : (x \sim z)}$$

$$\frac{x, y, z, t : \mathsf{Mul} \quad p : (x \sim y) \quad q : (z \sim t)}{\mathsf{cong}(x, y, z, t, p, q) : ((x \bullet z) \sim (y \bullet t))}$$

$$\frac{x, y, z : \mathsf{Mul}}{\mathsf{ass}(x, y, z) : (((x \bullet y) \bullet z) \sim (x \bullet (y \bullet z)))}$$

3.5. Problem Statement

Now we have the tools to define the specification relation, that is the input/output relation which our program has to conform to. What we want to do is to transform an arbitrary multiplicative expression into a normalised and equivalent one. This leads to the definition

$$(x \overset{\mathsf{Spec}}{\longmapsto} y) = \mathsf{Norm}(y) \wedge (x \sim y)$$

Now, we can associate different problems with this relation. In type theory, the formal counterpart of a problem is a type, in the sense that finding an element in the type corresponds to solving the problem. So the type

$$(\forall x : \mathsf{Mul})(\exists y : \mathsf{Mul})(x \overset{\mathsf{Spec}}{\longmapsto} y)$$

corresponds to the problem of proving that for any $x : \mathsf{Mul}$ you can find a corresponding y, while

$$(\exists f : \mathsf{Mul} \to \mathsf{Mul})(\forall x : \mathsf{Mul})(x \overset{\mathsf{Spec}}{\longmapsto} (f\,`x))$$

is the problem of finding *one* method which, when applied to an arbitrary x, gives a correct y.

A normal solution to the first problem gives you a function which when applied to $x:\mathsf{Mul}$, returns a pair, $\langle y, p\rangle$, where $y:\mathsf{Mul}$ is a value, and $p:(x \overset{\mathsf{Spec}}{\longmapsto} y)$ is a proof witness, while a solution to the second gives a pair $\langle f, q\rangle$, where $f:\mathsf{Mul}\to\mathsf{Mul}$, is a function returning values,

$$f\text{`}x:\mathsf{Mul}[x:\mathsf{Mul}]$$

and $q:(\forall x:\mathsf{Mul})(x \overset{\mathsf{Spec}}{\longmapsto} f\text{`}x)[x:\mathsf{Mul}]$ is a function returning proof witnesses,

$$q\text{`}x:x \overset{\mathsf{Spec}}{\longmapsto} f\text{`}x[x:\mathsf{Mul}]$$

The two problems, or types, are logically equivalent in type theory in the sense that you can, in a canonical way, define a function from one type to the other and vice versa. The two functions are each other's inverses only up to *extensional* equality of functions, they are not inverses with respect to propositional equality. Another illustration of the difference between the problems is that in the classical reading of the quantifiers, you have to postulate the axiom of choice in order to get one direction of the equivalence. In type theory, the axiom of choice is *derivable.* The *values* you get by applying the axiom of choice to a solution of the first problem, may in general be computed only in mutual recursion with the corresponding *proof witnesses.*

The problem you want to solve is therefore really the second, but in the informal phase you can get a better understanding by working with the first one which has a simpler logical form. In this paper, we focus on the second problem.

4. The General Recursive Program as External to Type Theory

The general recursive program

$$\begin{cases} \mathsf{norm_mul}(\mathsf{at}(m)) & = \mathsf{at}(m) \\ \mathsf{norm_mul}(\mathsf{at}(m) \bullet z) & = \mathsf{at}(m) \bullet \mathsf{norm_mul}(z) \\ \mathsf{norm_mul}((x \bullet y) \bullet z) & = \mathsf{norm_mul}(x \bullet (y \bullet z)) \end{cases}$$

describes the process of normal order rewriting of the associative law.

> If you start with an expression of the form $\mathsf{norm_mul}(e)$ where e is a multiplicative expression, then rewriting according to norm_mul gives you expressions with at most one occurrence of "norm_mul". The sequence of multiplicative expressions you get by deleting "norm_mul" from a computation of norm_mul is a rewriting sequence according to the associative law, which is in normal order in the sense that topmost redexes are preferred to other possible redexes. The occurrence of "norm_mul" could be seen as a tag marking out the topmost associative redex!

It is, however, not in primitive recursive form because of the third clause, which does not describe $\mathsf{norm_mul}((x \bullet y) \bullet z)$ in terms of x, y, z, $\mathsf{norm_mul}(x)$, $\mathsf{norm_mul}(y)$ and $\mathsf{norm_mul}(z)$. In order to talk about it in type theory, we must give some object of type theory, which somehow captures the essence of it. One possibility is to describe some abstract machine computing

it. This could involve the introduction of some type of states and predicates for transitions and final states. In doing this, there is a choice of how much detail to reveal in the description of computation [Plo81]. A suitable choice here would be an inductive definition of the *graph* of the program. The clauses in the definition is based on a "strict" evaluation strategy, since we want a computation to result in a pure multiplicative expression with no unevaluated parts. The "strictness" comes from the second clause, which forces evaluation under a constructor.

$$\begin{cases} \dfrac{}{\mathsf{at}(m) \overset{\mathsf{norm}}{\longmapsto} \mathsf{at}(m)} \\[2ex] \dfrac{x \overset{\mathsf{norm}}{\longmapsto} u}{\mathsf{at}(m) \bullet x \overset{\mathsf{norm}}{\longmapsto} \mathsf{at}(m) \bullet u} \\[2ex] \dfrac{x \bullet (y \bullet z) \overset{\mathsf{norm}}{\longmapsto} u}{(x \bullet y) \bullet z \overset{\mathsf{norm}}{\longmapsto} u} \end{cases}$$

Note that apart from trivial syntactical abuses, this inductive definition is the PROLOG program corresponding to norm_mul. The query

$$?(e \overset{\mathsf{norm}}{\longmapsto} \mathsf{Y})$$

will succeed with $\mathsf{Y} = f$, provided e is a multiplicative expression and f is its normal form. With this description of norm_mul, we can separate correctness into

Partial correctness, which means that the program never computes an erroneous value, or equivalently, $\overset{\mathsf{norm}}{\longmapsto} \subseteq \overset{\mathsf{Spec}}{\longmapsto}$:

$$(\forall x, y : \mathsf{Mul}) x \overset{\mathsf{norm}}{\longmapsto} y \Rightarrow x \overset{\mathsf{Spec}}{\longmapsto} y$$

Termination, which means that something is always returned, that is, the domain of $\overset{\mathsf{norm}}{\longmapsto}$ is the whole of Mul:

$$(\forall x : \mathsf{Mul})(\exists y : \mathsf{Mul}) x \overset{\mathsf{norm}}{\longmapsto} y$$

Patial correctness is easy to prove by $\overset{\mathsf{norm}}{\longmapsto}$-elimination, since all the introduction rules for $\overset{\mathsf{norm}}{\longmapsto}$ are valid as *derived* rules of $\overset{\mathsf{Spec}}{\longmapsto}$.

1. Clearly, an atom, $\mathsf{at}(m)$, is in normal form and $\sim$-equivalent to itself.
2. If $x \sim u$ then, by reflexivity and congruence, $\mathsf{at}(m) \bullet x \sim \mathsf{at}(m) \bullet u$ and if u is in normal form, then so is $\mathsf{at}(m) \bullet u$.
3. If $x \bullet (y \bullet z) \sim u$ and u is in normal form, then by associativity and transitivity, $(x \bullet y) \bullet z \sim u$, and u is in normal form by assumption.

Termination can be proved by proving the following derived rules for the termination predicate,

$$\frac{}{(\exists\, u : \mathsf{Mul})\ \mathsf{at}(m) \overset{\mathsf{norm}}{\longmapsto} u}$$

$$\frac{(\exists\, u : \mathsf{Mul}) z \overset{\mathsf{norm}}{\longmapsto} u}{(\exists\, u : \mathsf{Mul})\ \mathsf{at}(m) \bullet z \overset{\mathsf{norm}}{\longmapsto} u}$$

$$\frac{(\exists\, u : \mathsf{Mul}) x \bullet (y \bullet z) \overset{\mathsf{norm}}{\longmapsto} u}{(\exists\, u : \mathsf{Mul})(x \bullet y) \bullet z \overset{\mathsf{norm}}{\longmapsto} u}$$

and then use these as lemmas in the termination proof, which is a nested induction of higher type. Intuitively these rules are justified by the syntactical structure of the definition of norm_mul – they express that norm_mul applied to the Muls in the conclusion of a rule is defined in terms of norm_mul applied to those appearing in the premises.

To make this modularisation of the termination proof explicit, we use these rules to *define* a predicate,

$$\frac{}{\mathsf{Term}(\mathsf{at}(m))}\,(\mathrm{t1})$$

$$\frac{\mathsf{Term}(z)}{\mathsf{Term}(\mathsf{at}(m) \bullet z)}\,(\mathrm{t2})$$

$$\frac{\mathsf{Term}(x \bullet (y \bullet z))}{\mathsf{Term}((x \bullet y) \bullet z)}\,(\mathrm{t3})$$

and then divide the termination proof into two steps:

$$(\forall\, x : \mathsf{Mul})\ \mathsf{Term}(x)$$

$$(\forall\, x : \mathsf{Mul})\ \mathsf{Term}(x) \Rightarrow (\exists\, u : \mathsf{Mul}\ x \overset{\mathsf{norm}}{\longmapsto} u$$

where the second-step now becomes a trivial Term-elimination, based on the three derived rules stated above for the termination predicate.

The Term predicate captures the essence of termination of norm_mul. Let $\prec$ denote the recursion relation defined informally as $y \prec x$, if and only if (the definition of) norm_mul tells you to compute norm_mul(y) in order to get a value of norm_mul(x). In terms of this relation, we can say that the computation of norm_mul(x) terminates if x belongs to the *well founded part* of $\prec$; see Aczel [Acz77]. The well founded part is in the general case given by an inductive definition parametrised by the $\prec$-relation. Our Term-predicate is an optimised version of this well founded part $\prec$!

The proof that all multiplicative expressions have the Term property, that is, that they belong to the well founded part of $\prec$, combined with the elimination rule for Term, gives a recursion operator, assrec, which expresses a principle of recursion along $\prec$. This principle should be compared with the general scheme described in Nordström [Nor88]. The principle we derive is based on the *elementary* definition of *one* predicate, tailored to *one*

termination problem and is suitable only for solving problems that are rather similar to that one. What we do, we do in type theory *instantiated* with some suitable data types and predicates, while Nordström proposes an *extension* of type theory. In this extended theory, the general recursive program can actually be constructed – a term norm_mul can be defined and will satisfy the equalities of the definition of norm_mul in the sense of valid *reductions*. We can also define a term norm_mul, but we will get the equalities as *conversions* only.

Here is a sketch of the particular termination proof implicit in assrec.

1. Prove Term(e), for arbitrary e : Mul by induction on e. This gives two cases, (1.1) and (1.2).

1.1. Assume m : Nat, and prove Term(at(m)). Immediate by (t1).

1.2. Assume x, y : Mul, Term(x), Term(y) and prove Term($x \bullet y$). We do this by proving the stronger assertion, $P(x)$, that $(\forall z : \mathsf{Mul})\ \mathsf{Term}(z) \Rightarrow \mathsf{Term}(x \bullet z)$. Clearly, a proof of $P(x)$ would apply to the assumptions y : Mul and Term(y), to give the desired conclusion. The proof of $P(x)$ is again by induction, giving the cases (1.2.1) and (1.2.2).

1.2.1. Assume m : Nat and prove P(at(m)), that is

$$(\forall z : \mathsf{Mul})\ \mathsf{Term}(z) \Rightarrow \mathsf{Term}(\mathsf{at}(m) \bullet z)$$

but this follows from (t2).

1.2.2. Assume u, v : Mul, $P(u)$, $P(v)$ and prove $P(u \bullet v)$, that is,

$$(\forall z : \mathsf{Mul})\ \mathsf{Term}(z) \Rightarrow \mathsf{Term}((u \bullet v) \bullet z)$$

so we add

$$\begin{array}{l} z : \mathsf{Mul} \\ \mathsf{Term}(z) \end{array}$$

to our assumptions and prove Term($(u \bullet v) \bullet z$). Here is a picture of how the assumptions are combined to give the desired conclusion.

$$\dfrac{\dfrac{\dfrac{\overline{\mathsf{Term}(z)}}{\mathsf{Term}(v \bullet z)}\ (P(v), z : \mathsf{Mul})}{\mathsf{Term}(u \bullet (v \bullet z))}\ (P(u), u \bullet v : \mathsf{Mul})}{\mathsf{Term}((u \bullet v) \bullet z)}\ (\mathrm{t3})$$

Although rather detailed, this sketch is far from being formal.

5. Higher Order Programs

5.1. The assrec Operator . . .

The termination proof for the general recursive program, norm_mul, inspires the formulation of a derived rule of type theory in terms of a defined constant,

assrec,

assrec-typing

$$\frac{\begin{array}{l} f(m):C(\mathsf{at}(m))[m:\mathsf{Nat}] \\ g(m,z,p):C(\mathsf{at}(m)\bullet z)[m:\mathsf{Nat},z:\mathsf{Mul},p:C(z)] \\ h(x,y,z,p):C((x\bullet y)\bullet z)[p:C(x\bullet(y\bullet z))] \\ e:\mathsf{Mul} \end{array}}{\mathsf{assrec}(f,g,h,e):C(e)}$$

assrec-equalities

$$\begin{array}{l} f:\cdots \\ g:\cdots \\ h:\cdots \end{array}$$

$$\begin{cases} F(\mathsf{at}(m)) & =f(m) \\ F(\mathsf{at}(m)\bullet z) & =g(m,z,F(z)) \\ F((x\bullet y)\bullet z) & =h(x,y,z,F(x\bullet(y\bullet z))) \end{cases}$$

where we have written $F(e)$ instead of $\mathsf{assrec}(f,g,h,e)$. These are derived from the definition

$$\begin{array}{ll} F(x) & :C(x)[x:\mathsf{Mul}] \\ F(\mathsf{at}(m)) & =f(m) \\ F(x\bullet y) & =G(x)\,`y\,`F(y) \\ G(x) & :(\prod y:\mathsf{Mul})C(y)\to C(x\bullet y)[x:\mathsf{Mul}] \\ G(\mathsf{at}(m)) & =(\lambda z)(\lambda p)g(m,z,p) \\ G(x\bullet y) & =(\lambda z)(\lambda p)h(x,y,z,G(x)\,`(y\bullet z)\,`(G(y)\,`z\,`p)) \end{array}$$

Recall our abuse of syntax in function definitions (Section 3.1), so what is indicated here is really a metadefinition

$\mathsf{assrec} \equiv (f,g,h,x)$ terrible expression beginning with mulrec

Note that, when deriving the assrec-equalities, we must use the G-equalities in *both* directions, so the assrec-equalities express *conversion* and not *reduction.*

From an abstract point of view, termination of norm_mul is equivalent to the well-foundedness of the recursion relation $<$, and assrec should be thought of as the composition of the elimination rule for Term with a particular termination proof (cf. Section 4).

5.2. ... and Its Relation to Our Problem

The partial correctness proof we indicated for norm_mul depends on establishing the following derived rules for the specification relation.

$$\frac{m : \mathsf{Nat}}{\mathsf{spec_a}(m) : \mathsf{at}(m) \overset{\mathsf{Spec}}{\longmapsto} \mathsf{at}(m)}$$

$$\frac{m : \mathsf{Nat} \quad x, u : \mathsf{Mul} \quad p : x \overset{\mathsf{Spec}}{\longmapsto} u}{\mathsf{spec_ma}(m, x, u, p) : \mathsf{at}(m) \bullet x \overset{\mathsf{Spec}}{\longmapsto} \mathsf{at}(m) \bullet u}$$

$$\frac{x, y, z : \mathsf{Mul} \quad p : x \bullet (y \bullet z) \overset{\mathsf{Spec}}{\longmapsto} u}{\mathsf{spec_ass}(x, y, z, u, p) : (x \bullet y) \bullet z \overset{\mathsf{Spec}}{\longmapsto} u}$$

Now, suppose that we have a function, $\varphi(x) : \mathsf{Mul}[x : \mathsf{Mul}]$, for which we can prove the *judgemental* equalities defining norm_mul, that is,

$$\begin{cases} \varphi(\mathsf{at}(m)) = \mathsf{at}(m) & [m : \mathsf{Nat} \\ \varphi(\mathsf{at}(m) \bullet z) = \mathsf{at}(m) \bullet \varphi(z) & [m : \mathsf{Nat}, z : \mathsf{Mul}] \\ \varphi((x \bullet y) \bullet z) = \varphi(x \bullet (y \bullet z)) & [x, y, z, u : \mathsf{Mul}] \end{cases}$$

then we can prove that φ is a correct normaliser of multiplicative expressions by applying the assrec typing rule to the family

$$\mathsf{C}(x) = x \overset{\mathsf{Spec}}{\longmapsto} \varphi(x)[x : \mathsf{Mul}]$$

The result is the term

$$\begin{aligned} p(e) = \mathsf{assrec}(&(m)\ \mathsf{spec_a}(m), \\ &(m, x, p)\ \mathsf{spec_ma}(m, x, \varphi(x), p), \\ &(x, y, z, p)\ \mathsf{spec_ass}(x, y, z, \varphi(x \bullet (y \bullet z)), p) \\ ): \quad &(e \overset{\mathsf{Spec}}{\longmapsto} \varphi(e))[e : \mathsf{Mul}] \end{aligned}$$

This leads to a derived rule, which we state informally:

Lemma. Any function $\varphi(x) : \mathsf{Mul}[x : \mathsf{Mul}]$, which satisfies the equalities in the definition of norm_mul as *judgemental* equalities is a correct normaliser of multiplicative expressions.

We could state and prove a stronger version of this lemma where only propositional equalities were assumed, but the proof would then be longer due to equality eliminations. An important ingredient in the proof of the weak version of the lemma, is type conversion, which is the rule that says that equal types have the same members. In the stronger version, these type conversions would correspond to equality eliminations in the form of Leibniz's rule,

$$\frac{q : a =_A b \quad p : P(a)}{\cdots(a, b, q, p) : P(b)}$$

which is derivable in type theory.

5.3. Applying the assrec Construction to Get a Higher Order Normaliser

Here we will use the assrec-operator:

1. To get a function, $\mathsf{norm_ho}(x):\mathsf{Mul}[x:\mathsf{Mul}]$, satisfying the judgemental equalities of the general recursive program, norm_ho.
2. To prove the correctness of the program, by an appeal to the lemma of the previous section.

A suitable function, $\mathsf{norm_ho}(x):\mathsf{Mul}[x:\mathsf{Mul}]$ is obtained as

$$\mathsf{norm_ho}(e) = \mathsf{assrec}((m)\mathsf{at}(m),\ (m, x, u)\mathsf{at}(m) \bullet u,\ (x, y, z, u)u,\ e)$$

Using the equalities of assrec, norm_ho is easily seen to satisfy the judgemental equalities of norm_mul, so we have a case where we can apply the lemma relating assrec to $\overset{\mathsf{Spec}}{\longmapsto}$.

Note that the assrec tactic may well be used on the problem

$$(\exists\, y:\mathsf{Mul})\, x \overset{\mathsf{Spec}}{\longmapsto} y[x:\mathsf{Mul}]$$

to give a program, which computed the value of y in mutual recursion with the *proof* that y stands in the correct relation to x, but the y computed in *this* way would certainly not satisfy the judgemental equalities of norm_mul.

Instead of using assrec, norm_ho can be defined in two steps,

$$\begin{cases} \mathsf{norm_ho}(\mathsf{at}(a)) = \mathsf{at}(a):\mathsf{Mul} \\ \mathsf{norm_ho}(x \bullet y) = \mathsf{lamaux}(x)\text{`}(\mathsf{norm_ho}(y)):\mathsf{Mul} \\ \mathsf{lamaux}(\mathsf{at}(a)) = (\lambda z)\,\mathsf{at}(a) \bullet z:\mathsf{Mul}\to\mathsf{Mul} \\ \mathsf{lamaux}(x \bullet y) = (\lambda z)\,\mathsf{lamaux}(x)\text{`}(\mathsf{lamaux}(y)\text{`}z):\mathsf{Mul}\to\mathsf{Mul} \end{cases}$$

or, hiding the application symbols, writing $\mathsf{aux_ho}(x, y) = \mathsf{lamaux}(x)\text{`}y$,

$$\begin{cases} \mathsf{norm_ho}(\mathsf{at}(a)) = \mathsf{at}(a):\mathsf{Mul} \\ \mathsf{norm_ho}(x \bullet y) = \mathsf{aux_ho}(x, \mathsf{norm_ho}(y)):\mathsf{Mul} \\ \mathsf{aux_ho}(\mathsf{at}(a), z) = \mathsf{at}(a) \bullet z:\mathsf{Mul} \\ \mathsf{aux_ho}(x \bullet y, z) = \mathsf{aux_ho}(x, \mathsf{aux_ho}(y, z)):\mathsf{Mul} \end{cases}$$

Also the correctness of norm_ho can be established without explicit mention of assrec, by observing that the functions $\mathsf{lamaux}(x):\mathsf{Mul}\to\mathsf{Mul}[x:\mathsf{Mul}]$ all map normal expressions into normal expressions and that $x \bullet y \sim \mathsf{lamaux}(x)\text{`}y$.

It will appear that a small change will give the primitive recursive program, norm_pr, described in Section 6.1.

5.4. The General Recursive Program as the Limit of Fixed Point Approximations

One way to look at the general recursive program, norm_mul, is as a *fixed point* of the functional,

$$\mathsf{step}(f) = (\lambda x)\,\mathsf{apstep}(f, x):\mathsf{Mul}\to\mathsf{Mul}[f:\mathsf{Mul}\to\mathsf{Mul}]$$

where

$$\begin{cases} \mathsf{apstep}(f, \mathsf{at}(m)) & = \mathsf{at}(m) \\ \mathsf{apstep}(f, \mathsf{at}(m) \bullet z) & = \mathsf{at}(m) \bullet (f\,\text{`}z) \\ \mathsf{apstep}(f, (x \bullet y) \bullet z) & = f\,\text{`}(x \bullet (y \bullet z)) \end{cases}$$

What do we mean by a fixed point in the context of type theory? Here are some possible readings of the phrase "f is a fixed point of step" based on different interpretations of the equality sign and different degrees of extensionality. (We assume that both f and step are derived in an empty context):

(J1) *Judgemental equality.*

$$f = \mathsf{step}(f) : \mathsf{Mul} \to \mathsf{Mul}$$

(J2) *Applied judgemental equality,*

$$f\,\text{`}x = \mathsf{step}(f)\,\text{`}x : \mathsf{Mul}[x : \mathsf{Mul}]$$

(J3) *Applied judgemental equality by cases.* As (J2), but split into three statements of judgemental equality, corresponding to the clauses of norm_mul

$$\begin{aligned} f\,\text{`}(\mathsf{at}(m)) &= \mathsf{step}(f)\,\text{`}(\mathsf{at}(m))[m : \mathsf{Nat}] \\ f\,\text{`}(\mathsf{at}(m) \bullet x) &= \mathsf{step}(f)\,\text{`}(\mathsf{at}(m) \bullet x)[m : \mathsf{Nat}, x : \mathsf{Mul}] \\ f\,\text{`}((x \bullet y) \bullet z) &= \mathsf{step}(f)\,\text{`}((x \bullet y) \bullet z)[x, y, z : \mathsf{Mul}] \end{aligned}$$

(P1) *Propositional equality,*

$$\cdots : f =_{\mathsf{Mul} \to \mathsf{Mul}} \mathsf{step}(f)$$

(P2) *Applied propositional equality,*

$$\cdots : (\forall\, x : \mathsf{Mul})\, f\,\text{`}x =_{\mathsf{Mul}} \mathsf{step}(f)\,\text{`}x$$

This is also expressed by saying that f and $\mathsf{step}(f)$ are *extensionally equal*

(P3) *Applied propositional equality by cases.* This would be equivalent to (P2), since you can prove a proposition by case analysis.

(R1) *Reduction,*

f reduces to $\mathsf{step}(f)$

(R2) *Applied reduction,*

$f\,\text{`}x$ reduces to $\mathsf{step}(f)\,\text{`}x[x : \mathsf{Mul}]$

(R3) *Applied reduction by cases,*

$$\begin{aligned} f\,\text{`}(\mathsf{at}(m)) &\text{ reduces to } \mathsf{at}(m)[m : \mathsf{Nat}] \\ f\,\text{`}(\mathsf{at}(m) \bullet x) &\text{ reduces to } \mathsf{at}(m) \bullet (f\,\text{`}x)[m : \mathsf{Nat}, x : \mathsf{Mul}] \\ f\,\text{`}((x \bullet y) \bullet z) &\text{ reduces to } f\,\text{`}(x \bullet (y \bullet z))[x, y, z : \mathsf{Mul}] \end{aligned}$$

Note that the three last readings are not official type theory judgements, but type theory may well be extended with a new judgement form expressing reduction. This would make explicit the relation between type theory and computation.

The unapplied readings, (J1), (P1) and (R1), seem to be too strong to be of interest. Note that (P1) is in fact equivalent to (J1), since if we have an actual proof of the propositional equality between two closed terms in a type, then the terms are convertible. (J2) is equivalent to (J1) when both f and step are on λ-form, due to the ξ-rule for conversion. (P2) and (P3) are clearly equivalent since you can prove propositions by case analysis, they are the ones most readily established and they come close to norm_mul interpreted as a *proposition* about a function, but not as the *definition* of a function. Summing up we have:

(J1), (J2) and (P1) are equivalent and difficult to establish, and do not capture the notion that the fixed point is something being *defined* by being a fixed point.

(P2) and (P3) are equivalent and easy to establish, but do not capture the notion that the fixed point is something being *defined* by being a fixed point.

(R1) and (R2) are clearly related to the notion of a fixed point equation as a definition, but seem to be difficult to establish. One reason why (R2) and (J2) are difficult to establish is, in our case, the apstep(f, x) does not reduce until x appears as one of the cases in norm_mul.

Although (J3) does not capture the definitional interpretation of fixed point equations, it is a step on the way, and in our case it seems to be the best we can establish. Once we have done that, proofs become shorter than corresponding proofs based on (P3) since we can replace equality eliminations with type conversion.

(R3) is the reading which best captures the notion of a fixed point equation as a definition of functional procedure. This is the preferred notion of fixed point. This concept of applied reduction by cases provides a reasonable explanation of what it means for a general recursive program to be definable in type theory.

> To find a solution, f, of the fixed point equation in the sense of (R3) is the same as defining the general recursive program in type theory. The symbol norm_mul appearing in the general recursive program would then correspond to the pure abstraction (expression with a hole) $(x)(f`x)$.

In fact, our way of presenting functions in this paper is by giving the type of the function and a set of equations in the form of a general recursive program, with the understanding that the reader could insert the metadefinition ($\equiv$) that would yield the equations. The implicit metadefinition then provides a solution of the equations given just in the sense of (R3).

Note that all readings not ending with 3 are explained in terms of an arbitrary step (of the right type), but writing down (J3) amd (P3) depends on knowing the case analysis made by step (the left-hand sides in the definition of norm_mul) and (R3) also depends on what is being returned in the different cases (the right-hand sides). It would be tempting to fold the definition of step in (R3) and write

$$(\text{R3}')\begin{cases} f`(\text{at}(m)) & \text{reduces to } \text{step}(f)`(\text{at}(m)) \\ f`(\text{at}(m) \bullet z) & \text{reduces to } \text{step}(f)`(\text{at}(m) \bullet z) \\ f`((x \bullet y) \bullet z) & \text{reduces to } \text{step}(f)`((x \bullet y) \bullet z) \end{cases}$$

but since reduction is not symmetrical and we have to use the definition of **step** backwards to get from (R3) to (R3'), (R3') does not mean the same as (R3). The symmetry of conversion implies that the folded and unfolded versions of (J3) are indeed equivalent, the unfolded version looking exactly like the general recursive program, but with a different meaning (of course) of the equality sign.

There are several approaches to fixed point equations. In most of them continuity, in some suitable sense, of the functional **step** plays a central rôle. The most natural way to look at this functional is as sending *partial* functions into partial functions and to regard the general recursive program as the limit of the sequence $\mathbf{step}^n(\varnothing)$ as $n \to \infty$, where $\varnothing$ denotes the empty (totally undefined) function. In this setting, partial correctness of the general recursive program may conveniently be expressed in terms of the functional, namely that it sends partially correct partial functions into partially correct partial functions and we have a nice separation of partial correctness and termination.

Relevant to fixed point approximations is the iterated functional

$$\mathsf{itstep}(m, f, x) = \mathsf{step}^m(f)`x : \mathsf{Mul}[m : \mathsf{Nat}, f : \mathsf{Mul} \to \mathsf{Mul}, x : \mathsf{Mul}]$$

where iteration $F^m(f)$ is defined by recursion on $m : \mathsf{Nat}$,

$$\frac{A\ \mathbf{Type} \quad F(x) : A\ [x : A]}{F^m(a) = \mathsf{natrec}(m, a, (n, an)F(an)) : A\ [m : \mathsf{Nat}, a : A]}$$

with $A = \mathsf{Mul} \to \mathsf{Mul}$ and $F(f) = \mathsf{step}(f)[f : \mathsf{Mul} \to \mathsf{Mul}]$.

In type theory all functions are total, so instead of using the totally undefined function as a start approximation, we supply a dummy function, the identity function, $\mathsf{id} = (\lambda u)u : \mathsf{Mul} \to \mathsf{Mul}$, for instance. For any measure function, $\mu(e) : \mathsf{Nat}[e : \mathsf{Mul}]$, we can define

$$\mathsf{norm_fixp}_\mu(x) = \mathsf{itstep}(\mu(x), \mathsf{id}, x)$$

but the definition is relevant to our problem of course, only when we can prove that it defines a correct normaliser. In Dybjer [Dyb91] this approach is applied to the case of conditional expressions and here this means that correctness is proved by proving the stronger proposition,

$$(\forall x : \mathsf{Mul})\, n \geq \mu(x) \Rightarrow x \overset{\mathsf{Spec}}{\longmapsto} (\mathsf{apstep}^n(\mathsf{id}))`x$$

by induction on $n : \mathsf{Nat}$. What properties of μ are essential for an induction in the style of Dybjer [Dyb91] to go through? In terms of the recursion relation, $<$, the parts of the proof relating to μ depends on μ satisfying

$$(\forall x, y : \mathsf{Mul})\, y < x \Rightarrow \mu(x) \geq 1 + \mu(y)$$

or using standard mathematical notation,

$$\mu(x) \geq 1 + \sup\{\mu(y) \mid y < x\}[\forall x \in \mathsf{Mul}]$$

where the supremum is taken over the natural numbers, so that the supremum of the empty set is the natural number 0. The latter formulation gives an "explicit" lower bound for the measure of $x : \mathsf{Mul}$ in terms of the measures of the $<$-predecessors of x.

Any such measure gives a proof of $\forall x : \mathsf{Mul}\ \mathsf{Term}(x)$ using induction on the propositional function

$$(\forall x : \mathsf{Mul})\, n \geq \mu(x) \Rightarrow \mathsf{Term}(x)[n : \mathsf{Nat}]$$

as well as a correctness proof of $\mathsf{norm_fixp}_\mu$ as in Dybjer [Dyb91].

The requirement on μ is also sufficient for $\mathsf{norm_fixp}_\mu$ to be a fixed point of step, in the sense of (P2). This could be shown by proving

$$(\forall x : \mathsf{Mul})(\forall f, g : \mathsf{Mul} \to \mathsf{Mul}) n \geq \mu(x) \Rightarrow \mathsf{itstep}(n, f, x) =_{\mathsf{Mul}} \mathsf{itstep}(\mu(x), g, x)$$

using induction on $n : \mathsf{Nat}$. Proofs of this kind must use some arithmetic. The base case relies on absurdity elimination and Peano's axiom that no successor equals zero, and the induction step uses inverse monotonicity of the successor function,

$$\mathsf{succ}(x) \geq \mathsf{succ}(y) \Rightarrow x \geq y$$

The requirement on μ has a smallest solution, provided it has a solution at all, where the inequalities are replaced by equalities. These equalities can be expressed as recurrence relations on μ giving a general recursive program for computing μ.

$$\begin{cases} \mu(\mathsf{at}(m)) & = \mathsf{succ}(0) \\ \mu(\mathsf{at}(m) \bullet z) & = \mathsf{succ}(\mu(z)) \\ \mu((x \bullet y) \bullet z)) & = \mathsf{succ}(\mu(x \bullet (y \bullet z))) \end{cases}$$

Note how the supremum operation has disappeared. This general recursive program has precisely the same $<$-relation as the original general recursive program, norm_mul, for the normaliser! Therefore the measure must be defined in some other way. In our case, the definition

$$\mu(x) = |x| - \rho(x)$$

where $|*|$ counts the total number of $\bullet$s and ats in a multiplicative expression,

$$\begin{cases} |\mathsf{at}(m)| & = 1 \\ |x \bullet y| & = 1 + |x| + |y| \end{cases}$$

and $\rho(x)$ counts the length of x interpreted as a left consed list of multiplicative expressions,

$$\begin{cases} \rho(\mathsf{at}(m)) & = 0 \\ \rho(x \bullet y) & = 1 + \rho(y) \end{cases}$$

would solve the μ-requirement, actually up to propositional, but not judgemental equality.

In our case, we can use the assrec operator to solve the recurrence relations for μ up to *conversion*. The reason for this is that the general recursive program corresponding to these recurrence relations have the same $<$-relation as norm_mul and thus the same Term-predicate! Recall that assrec embodies a termination proof for norm_mul, where termination was defined as the well founded part of the $<$-relation associated with norm_mul.

With a measure μ defined with assrec to obtain the measure requirement with *judgemental equalities* instead of inequalities, it is straightforward to prove

that $\mathsf{norm_fixp}_\mu$ satisfies the judgemental equalities of the general recursive program, or equivalently, it is a fixed point of **step** up to conversion. Therefore its correctness may be asserted according to the lemma relating **assrec** to our problem.

Note though, that this depends heavily on $\{y \in \mathsf{Mul} \mid y \prec x\}$ always having at most one member. In the case of conditional expressions, this property breaks down, and we would have great difficulty in defining a measure, μ, such that the corresponding $\mathsf{norm_fixp}_\mu$ would be a fixed point up to *conversion*. The difficulty with exact measures in this case is that the two branches of a conditional expression may be in different stages of normalisation.

The second clause for **apstep** for conditional expressions is

$$\mathsf{apstep}(f, \mathsf{if}(\mathsf{at}(m), x, y)) = \mathsf{if}(\mathsf{at}(m), f`x, f`y))$$

which gives a corresponding clause in the requirement for the measure, μ,

$$\mu(\mathsf{if}(\mathsf{at}(m), x, y) \geq \mathsf{succ}(\mathsf{max}(\mu(x), \mu(y)))$$

Suppose now that we have succeeded in defining μ so that we have this requirement with judgemental equality instead of inequality (the **assrec** operator generalises to the conditional expressions!) and that we try to prove that $\mathsf{norm_fixp}_\mu$ is a fixed point in the sense of (J3). Then we have to consider the second case for $\mathsf{norm_fixp}_\mu$,

$$\begin{aligned}
&\mathit{norm_fixp}_\mu(\mathsf{if}(\mathsf{at}(m), x, y)) \\
&\quad = \mathsf{itstep}(\mu(\mathsf{if}(\mathsf{at}(m), x, y), \mathsf{id}, \mathsf{if}(\mathsf{at}(m), x, y)) \\
&\quad = \mathsf{itstep}(\mathsf{succ}(\mathsf{max}(\mu(x), \mu(y))), \mathsf{id}, \mathsf{if}(\mathsf{at}(m), x, y))) \\
&\quad = \mathsf{if}(\mathsf{at}(m), \mathsf{itstep}(\mathsf{max}(\mu(x), \mu(y)), \mathsf{id}, x), \\
&\qquad\qquad \mathsf{itstep}(\mathsf{max}(\mu(x), \mu(y)), \mathsf{id}, y))
\end{aligned}$$

and to obtain (J3) we would have to know that the last expression equals

$$\mathsf{if}(\mathsf{at}(m), \mathsf{norm_fixp}_\mu(x), \mathsf{norm_fixp}_\mu(y))$$

which would mean that we knew

$$\mathsf{itstep}(\mathsf{max}(\mu(x), \mu(y)), \mathsf{id}, x) = \mathsf{itstep}(\mu(x), \mathsf{id}, x)$$

and

$$\mathsf{itstep}(\mathsf{max}(\mu(x), \mu(y)), \mathsf{id}, y) = \mathsf{itstep}(\mu(y), \mathsf{id}, y)$$

and these are clearly impossible to derive for arbitrary (= unspecified) $x, y : \mathsf{Cond}$.[8] For the multiplicative expressions this problem will not arise. Instead of $\mathsf{max}(\mu(x), \mu(y))$ we have just $\mu(x)$ and no y at all.

6. A Proper Primitive Recursive Normaliser

In type theory all recursion is primitive recursion, so we use the word "proper" to qualify that recursion is not of higher type.

[8] Of course, μ cannot be a constant function!

6.1. Primitive Recursion with Norm Elimination

Let us look at the higher order normaliser again and see what makes it higher order. It is an instance of the scheme

$$\begin{cases} \mathsf{mknorm}(F, \mathsf{at}(m)) = \mathsf{at}(m) \\ \mathsf{mknorm}(F, x \bullet y) \quad = F(x, y, \mathsf{mknorm}(F, x), \mathsf{mknorm}(F, y)) \end{cases}$$

which gives a correct normaliser if we can prove that

$$x \bullet y \overset{\mathsf{Spec}}{\longmapsto} F(x, y, nx, ny)\,[x, y, nx, ny : \mathsf{Mul}, x \overset{\mathsf{Spec}}{\longmapsto} nx, y \overset{\mathsf{Spec}}{\longmapsto} ny]$$

In fact, mknorm is a variant of the recursion operator for multiplicative expressions with a fixed base case,

$$\mathsf{mknorm}(F, e) \equiv \mathsf{mulrec}(e, (m)\mathsf{at}(m), (x, y, fx, fy)\, F(x, y, fx, fy))$$

In the higher order normaliser, F was called aux_ho and depended only on x and ny,

$$\mathsf{norm_ho}(e) \equiv \mathsf{mknorm}((x, y, nx, ny)\, \mathsf{aux_ho}(x, ny), e)$$

so what is required from aux_ho is that we can prove that

$$x \bullet y \overset{\mathsf{Spec}}{\longmapsto} \mathsf{aux_ho}(x, ny)\,[x, y, ny : \mathsf{Mul}, y \overset{\mathsf{Spec}}{\longmapsto} ny]$$

where we have deleted nx from the context since it does not appear in the conclusion and then the information about it is not likely to contribute to the proof. In order to prove this, aux_ho was defined in such a way as to obey certain equations by defining $\mathsf{aux_ho}(x, ny)$ as some *function* $\mathsf{lamaux}(x)$ depending on x, *applied* to ny. It is quite natural that the join operator F becomes more complex, when it is not allowed to use all data available from the induction hypothesis.

We get primitive recursion by making F depend on nx instead of x:

$$\begin{cases} \mathsf{norm_pr}(\mathsf{at}(m)) \;= \mathsf{at}(m) \\ \mathsf{norm_pr}(x \bullet y) \;= \mathsf{aux_pr}(\mathsf{norm_pr}(x), \mathsf{norm_pr}(y)) \\ \mathsf{aux_pr}(\mathsf{at}(m), z) = \mathsf{at}(m) \bullet z \\ \mathsf{aux_pr}(x \bullet y, z) \;= x \bullet \mathsf{aux_pr}(y, z) \end{cases}$$

The auxiliary, aux_pr, can be seen as a concatenation function. There is a natural bijection between multiplicative expressions and *lists* of such ending with atoms – you can always write an $e : \mathsf{Mul}$ in the form

$$e = e_1 \bullet (e_2 \bullet \cdots (e_m \bullet \mathsf{at}(n)) \cdots)$$

corresponding to the list $[e_1, e_2, \ldots, e_m, \mathsf{at}(n)]$. So aux_pr corresponds to concatenation of these lists. Using this correspondence, normality of e corresponds to all the e_i being atomic and it is clear that aux_pr maps a pair of normal expressions into a normal expression. In fact, this program comes from a formal manipulation of the program, normalise, of Section 6.2 using the type of normal expressions.

The correctness of norm_pr is most conveniently proved by treating equivalence and normality separately.

Equivalence. An easy induction on $x : \mathsf{Mul}$ shows that

$$x \bullet y \sim \mathsf{aux_pr}(x, y) \quad (1)$$

This is used in the induction proof that $x \sim \mathsf{norm_pr}(x)$. The induction hypothesis in the induction step is

$$x \sim \mathsf{norm_pr}(x)$$
$$y \sim \mathsf{norm_pr}(y)$$

and we use the abbreviation

$$nx = \mathsf{norm_pr}(x)$$
$$ny = \mathsf{norm_pr}(y)$$

in a sketch of the induction step:

$$\dfrac{\dfrac{\dfrac{\text{ind hyps}}{x \bullet y \sim nx \bullet ny}(cong) \quad \dfrac{}{nx \bullet ny \sim \mathsf{aux_pr}(nxm\ ny)}(1)}{x \bullet y \sim \mathsf{aux_pr}(nx, ny)}(\text{trans})}{x \bullet y \sim \mathsf{norm_pr}(x \bullet y)}(\mathsf{norm_pr}(x \bullet y) = \mathsf{aux_pr}(nx, ny))$$

Note that in this informal exposition of (a part of) the proof, no proof objects are mentioned. They are implicit in the structure of the proof. By the rule for equal types, the last two proof objects are in fact identical.

Normality. The property we want from aux_pr, to be used in the induction proof that all values returned by norm_pr are normal, is that we can prove

$$\mathsf{Norm}(\mathsf{aux_pr}(nx, ny))$$

from the assumptions

$$nx, ny : \mathsf{Mul}\ \mathsf{Norm}(nx), \mathsf{Norm}(ny).$$

It is in the proof of this that we use Norm-elimination (see Section 3.3). We use the simpler version, where C does not depend on the proof object, applied to the assumptions $nx : \mathsf{Mul}$, $\mathsf{Norm}(nx)$ and the predicate

$$C(z) = \mathsf{Norm}(\mathsf{aux_pr}(z, ny))[z : \mathsf{Mul}]$$

This gives us two things to prove: first the base case,

$$\mathsf{Norm}(\mathsf{aux_pr}(\mathsf{at}(m), ny))[m : \mathsf{Nat}]$$

and then the induction step

$$\mathsf{Norm}(\mathsf{aux_pr}(\mathsf{at}(m) \bullet z, ny)[m : \mathsf{Nat}, z : \mathsf{Mul}, \mathsf{Norm}(\mathsf{aux_pr}(z, ny))]$$

which are both straightforward.

6.2. The Type of Normal Expressions

A naive method for normalising a multiplicative expression is to first remove all parentheses from the expression, and then reinsert them treating • as a right associative operator. What is then a multiplicative expression without

parentheses? It is a nonempty list of atoms,

$$\frac{}{\mathsf{List}^+ \ \mathbf{Type}}$$

$$\frac{m : \mathsf{Nat}}{\mathsf{unit}(m) : \mathsf{List}^+}$$

$$\frac{m : \mathsf{Nat} \quad x : \mathsf{List}^+}{\mathsf{cons}(m, x) : \mathsf{List}^+}$$

This is the datatype of normal expressions as suggested by the grammar given in Section 3.3.

The naive algorithm then consists of the functions

$$\mathsf{remove_par}(x) : \mathsf{List}^+[x : \mathsf{Mul}]$$
$$\mathsf{insert_par}(y) : \mathsf{Mul}[y : \mathsf{List}^+]$$
$$\mathsf{normalise}(x) = \mathsf{insert_par}(\mathsf{remove_par}(x))[x : \mathsf{Mul}]$$

The definition of insert_par is obvious,

$$\begin{cases} \mathsf{insert_par}(\mathsf{unit}(m)) & = \mathsf{at}(m) \\ \mathsf{insert_par}(\mathsf{cons}(m, ms)) & = (\mathsf{at}(m) \bullet \mathsf{insert_par}(ms)) \end{cases}$$

while the definition of remove_par is in two steps,

$$\begin{cases} \mathsf{remove_par}(\mathsf{at}(m)) & = \mathsf{unit}(m) \\ \mathsf{remove_par}(x \bullet y) & = \mathsf{cat}(\mathsf{remove_par}(x), \mathsf{remove_par}(y)) \\ \mathsf{cat}(\mathsf{unit}(m), y) & = \mathsf{cons}(m, y) \\ \mathsf{cat}(\mathsf{cons}(m, x), y)) & = \mathsf{cons}(m, \mathsf{cat}(x, y)) \end{cases}$$

For the correctness of normalise, we treat normality and equivalence separately. It is clear that insert_par only returns normal expressions, and then this holds for normalise as well. An important property of insert_par is that it translates cat to $\bullet$ up to $\sim$,

$$\mathsf{insert_par}(\mathsf{cat}(x, y)) \sim \mathsf{insert_par}(x) \bullet \mathsf{insert_par}(y)$$

which is proved by induction on $x : \mathsf{List}^+$.

This property of insert_par is the main ingredient in the induction proof that the input of normalise is equivalent to its output. The base case follows as in the previous section from the definition of normalise and reflexivity and the induction step combines the computation

$$\begin{array}{ll} \mathsf{normalise}(x) \bullet \mathsf{normalise}(y) & = \\ \mathsf{insert_par}(\mathsf{remove_par}(x)) \bullet \mathsf{insert_par}(\mathsf{remove_par}(y)) & \sim \\ \mathsf{insert_par}(\mathsf{cat}(\mathsf{remove_par}(x), \mathsf{remove_par}(y)) & = \\ \mathsf{insert_par}(\mathsf{remove_par}(x \bullet y)) & = \\ \mathsf{normalise}(x \bullet y) & \end{array}$$

with, as in the previous section, induction hypothesis, congruence and transitivity to get the result, $x \bullet y \sim \mathsf{normalise}(x, y)$.

As already mentioned, norm_pr of the previous section can be seen as the result of a formal manipulation of normalise – we get norm_pr by literally identifying List^+ with a subset of Mul, thus $\mathsf{unit} = \mathsf{at}$, $\mathsf{cons}(m, y) = \mathsf{at}(m) \bullet y$

and $\mathsf{insert_par}(x) = x$. At this stage cat is not defined in all cases but that is taken care by $\mathsf{aux_pr}$.

6.3. A Proper Primitive Recursive Normaliser for the Conditional Expressions

The type, Cond, of conditional expressions is inductively defined by its constructors

$$\mathsf{at}(m) : \mathsf{Cond}[m : \mathsf{Nat}]$$
$$\mathsf{if}(x, y, z) : \mathsf{Cond}[x, y, z : \mathsf{Cond}]$$

Recall the normalising equation for conditional expressions

$$\mathsf{if}(\mathsf{if}(x, y, z), u, v) = \mathsf{if}(x, \mathsf{if}(y, u, v), \mathsf{if}(z, u, v))$$

which generates the congruence relation, $\sim$, of *equivalent* conditional expressions. A conditional expression is *normal* when it does not match the left-hand side of the normalising equation and nor do any of its subexpressions, so an expression is normal if and only if, it is either atomic or the if of normal expressions, the first being atomic. The predicate Norm, of normal expressions, have the introduction rules

$$\frac{m : \mathsf{Nat}}{\mathsf{Norm}(\mathsf{at}(m))} \qquad \frac{m : \mathsf{Nat}\ x, y : \mathsf{Cond} \quad \mathsf{Norm}(x) \quad \mathsf{Norm}(y)}{\mathsf{Norm}(\mathsf{if}(\mathsf{at}(m), x, y))}$$

We define the normaliser, $\mathsf{norm_cond_pr} \equiv \mathsf{norm_pr}$, as

$$\begin{cases} \mathsf{norm_pr}(\mathsf{at}(m)) & = \mathsf{at}(m) \\ \mathsf{norm_pr}(\mathsf{if}(x, y, z)) & = \mathsf{aux_pr}(\mathsf{norm_pr}(x), \mathsf{norm_pr}(y), \mathsf{norm_pr}(z)) \\ \mathsf{aux_pr}(\mathsf{at}(m), u, v)) & = \mathsf{if}(\mathsf{at}(m), u, v)) \\ \mathsf{aux_pr}(\mathsf{if}(x, y, z), u, v)) & = \mathsf{if}(x, \mathsf{aux_pr}(y, u, v), \mathsf{aux_pr}(z, u, v)) \end{cases}$$

The correctness of $\mathsf{norm_cond_pr}$ depends on $\mathsf{aux_pr}$ having properties related to normality and equivalence.

$\mathsf{aux_pr}$ maps normal expressions into normal expressions,

$$\mathsf{Norm}(\mathsf{aux_pr}(nx, ny, nz)) \qquad [nx, nym\, nz : \mathsf{Cond}, \mathsf{Norm}(nx), \mathsf{Norm}(ny), \mathsf{Norm}(nz)] \quad (1)$$

$\mathsf{aux_pr}$ is $\sim$-equivalent to if,

$$\mathsf{if}(x, y, z) \sim \mathsf{aux_pr}(x, y, z) \quad (2)$$

Before showing (1) and (2), let us see how they are used to prove the correctness of $\mathsf{norm_cond_pr}$.

Normality. $\mathsf{Norm}(\mathsf{norm_cond_pr}(x))[x : \mathsf{Cond}]$. $\mathsf{Norm}(\mathsf{norm_cond_pr}(x))$ is proved by induction on $x : \mathsf{Cond}$. The base case reduces by the definition of $\mathsf{norm_cond_pr}$ to the first introduction rule for Norm, and the induction step follows directly from (1).

Equivalence. $x \sim \mathsf{norm_cond_pr}(x)[x : \mathsf{Cond}]$. The base case reduces by the definition of $\mathsf{norm_cond_pr}$ to reflexivity of $\sim$. In the induction step, we

assume that x, y and z are mapped (by norm_cond_pr) to equivalent expressions nx, ny and nz,

$$\text{ind hyps}\begin{cases} x \sim nx \\ y \sim ny \\ z \sim nz \end{cases} \text{ where } \begin{cases} nx = \mathsf{norm_cond_pr}(x) \\ ny = \mathsf{norm_cond_pr}(y) \\ nz = \mathsf{norm_cond_pr}(z) \end{cases}$$

and we want to show that $\mathsf{if}(x, y, z)$ is equivalent to its image,

$$\mathsf{if}(x, y, z) \sim \mathsf{norm_cond_pr}(\mathsf{if}(x, y, z)) \tag{3}$$

By the definition of norm_cond_pr and using the abbreviations, this reduces to

$$\mathsf{if}(x, y, z) \sim \mathsf{aux_pr}(nx, ny, nz) \tag{4}$$

Now, induction hypothesises and congruence give

$$\mathsf{if}(x, y, z) \sim \mathsf{if}(nx, ny, nz) \tag{5}$$

and property (2) of aux_pr gives

$$\mathsf{if}(nx, ny, nz) \sim \mathsf{aux_pr}(nx, ny, nz) \tag{6}$$

so that (4) follows by transitivity from (5) and (6).
The correctness of norm_cond_pr has thus been reduced to verifying the two properties (1) and (2) of aux_pr. The first one, concerning normality is proved by Norm elimination on the assumption $\mathsf{Norm}(nx)$ and the second is proved by Cond elimination on $x : \mathsf{Cond}$, where the induction step is to prove

$$\mathsf{if}(\mathsf{if}(x, y, z), u, v) \sim \mathsf{aux_pr}(\mathsf{if}(x, y, z), u, v)$$

from the assumptions

$$\mathsf{if}(x, u, v) \sim \mathsf{aux_pr}(x, u, v)$$
$$\mathsf{if}(y, u, v) \sim \mathsf{aux_pr}(y, u, v)$$
$$\mathsf{if}(z, u, v) \sim \mathsf{aux_pr}(z, u, v)$$

By the definition of aux_pr, our problem reduces to

$$\mathsf{if}(\mathsf{if}(x, y, z), u, v) \sim \mathsf{if}(x, \mathsf{aux_pr}(y, u, v), \mathsf{aux_pr}(z, u, v)) \tag{*}$$

Assumptions and congruence give

$$\mathsf{if}(x, \mathsf{if}(y, u, v), \mathsf{if}(z, u, v)) \sim \mathsf{if}(x, \mathsf{aux_pr}(y, u, v), \mathsf{aux_pr}(z, u, v))$$

and the normalising equation gives

$$\mathsf{if}(\mathsf{if}(x, y, z), u, v)) \sim \mathsf{if}(x, \mathsf{if}(y, u, v), \mathsf{if}(z, u, v))$$

so combining these with transitivity gives the result (*)

6.4. A Comment on Taylor's Work

The problem of normalising the associative law has been addressed by Taylor [Tay88], but in quite a different setting. He formulates and proves a "Canonical Form Theorem" for semigroups. Although he is working in the Calculus of Constructions [CoH88], we will sketch his approach using type

theory. Such a translation cannot be faithful in every respect. The main difference between Constructions and Martin-Löf Type Theory concerns the treatment of inductively defined types and predicates.

Martin-Löf Type Theory is an *open* system. We introduce new types and predicates by adding axioms to our formal system expressing introduction and elimination rules. Although it is well understood[9] what these extensions may look like, at least for ordinary (elementary) definitions, the control of their correctness is *outside* the formal system.

The impredicativity of Constructions allows you to introduce inductive types and predicates by *explicit* definitions of constants, and then you *prove* introduction and elimination rules *within* the system. There is a price to pay however – to get an inductive set in Constructions you have defined both a type (prop) and a predicate over that type describing the *inductive elements* of the type.

Notice also that the elimination rules of Martin-Löf Type Theory are computationally more powerful than those of Constructions – it is difficult, if not impossible, to construct a predecessor function with constant time complexity. First some list theory (cf. Section 6.2).

Lists. For an arbitrary type, A, we introduce the type, $\mathsf{List}^+(A)$, of nonempty lists of elements in A, as inductively defined by the introduction rules

$$\frac{a : A}{\mathsf{unit}(a) : \mathsf{List}^+(A)} \qquad \frac{a : A \quad as : \mathsf{List}^+(A)}{\mathsf{cons}(a, as) : \mathsf{List}^+(A)}$$

Informally we will write

$$[a_1, a_2, \ldots, a_m]$$

as a synonym for

$$\mathsf{cons}(a_1, \mathsf{cons}(a_2, \ldots, a_m) \cdots))$$

The concatenation function,

$$\mathsf{cat}(as, bs) : \mathsf{List}(A)[as, bs : \mathsf{List}^+(A)]$$

is defined by list recursion on its first argument,

$$\begin{cases} \mathsf{cat}(\mathsf{unit}(a), bs) & = \mathsf{cons}(a, bs) \\ \mathsf{cat}(\mathsf{cons}(a, as), bs) & = \mathsf{cons}(a, \mathsf{cat}(as, bs)) \end{cases}$$

A hypothetical semigroup. A semigroup consists of a set and a binary associative operation on the set. In order to talk about a hypothetical semigroup, we make the declarations

$$\mathsf{U}\ \mathbf{Type}$$
$$(u \cdot v) : \mathsf{U}[u, v : \mathsf{U}]$$
$$(e \simeq v)\ \mathbf{Type}[u, v : \mathsf{U}]$$

and axioms corresponding to our introduction rules for $\sim$. Note that Taylor's $\simeq$ is *equality* between *semigroup elements,* while our $\sim$ is an *equivalence* relation between *expressions.* Note also that U is not inductively defined, so it has no elimination rule.

[9] Cf. Backhouse [BCM89], Coquand and Paulin [CoP89] and Dybjer [Dyb89].

Built from. Taylor introduces a binary predicate,

$$\mathsf{Built_from}(us, u)\ \mathbf{Type}[us : \mathsf{List}^+(\mathsf{U}), u : \mathsf{U}]$$

inductively defined by the introduction rules

$$\frac{}{\mathsf{Built_from}(\mathsf{unit}(u), u)}\,(\mathsf{bltfrm1})$$

$$\frac{\mathsf{Built_from}(us, u) \quad \mathsf{Built_from}(vs, v)}{\mathsf{Built_from}(\mathsf{cat}(us, vs), u \cdot v)}\,(\mathsf{bltfrm2})$$

or with variable declarations and full syntax for the proof objects

$$\frac{u : \mathsf{U}}{\mathsf{bltfrm1}(u) : \mathsf{Built_from}(\mathsf{unit}(u), u)}$$

$$\frac{u, v : \mathsf{U} \quad us, vs : \mathsf{List}^+(\mathsf{U}) \quad p : \mathsf{Built_from}(us, u) \quad q : \mathsf{Built_from}(vs, v)}{\mathsf{bltfrm2}(us, u, p, vs, v, q) : \mathsf{Built_from}(\mathsf{cat}(us, vs), u \cdot v)}$$

Informally, $\mathsf{Built_from}(us, u)$ is true if $u : \mathsf{U}$ can be obtained by inserting dots, $\cdot$, in the list $us : \mathsf{List}^+(\mathsf{U})$ and then grouping in some arbitrary way. If $[u, v, w] : \mathsf{List}^+(\mathsf{U})$ is given, then we can find proofs of

$$\mathsf{Built_from}([u, v, w], (u \cdot v) \cdot w)$$

and

$$\mathsf{Built_from}([u, v, w], u \cdot (v \cdot w)$$

Normal form. The function

$$\mathsf{combine_dot}(us) : \mathsf{U}[us : \mathsf{List}^+(\mathsf{U})]$$

is used by Taylor to "... specify the normal form ...",

$$\begin{cases} \mathsf{combine_dot}(\mathsf{unit}(u)) & = u \\ \mathsf{combine_dot}(\mathsf{cons}(u, us)) & = u \cdot \mathsf{combine_dot}(us) \end{cases}$$

This function is very much like our insert_par of Section 6.2.

Canonical form theorem. This is the formalisation of the normal form theorem,

$$(\forall\, us : \mathsf{List}^+(\mathsf{U}))(\forall\, u : \mathsf{U})\ \mathsf{Built_from}(us, u) \Rightarrow (\mathsf{combine_dot}(us) \simeq u)$$

The theorem is proved by Built_from-elimination. The base case follows from the definition of combine_dot and $\simeq$ axioms. The induction step is reduced (modulo $\simeq$ axioms and the definition of combine_dot) to the lemma,

$$\mathsf{combine_dot}(us) \cdot \mathsf{combine_dot}(vs)) \simeq \mathsf{combine_dot}(\mathsf{cat}(us, vs)$$
$$[us, vs : \mathsf{List}^+(\mathsf{U}), u, v : \mathsf{U}]$$

The proof of the lemma is a straightforward List^+-induction, using the definitions of cat and combine_dot. Note that we have the analogue of this lemma in Section 6.2.

The canonical form theorem, interpreted as a type, is a strange type for a normalising function. We would expect the output to be of the same type as the input, perhaps augmented with a proof object witnessing to its correctness.

What, then, is the computational contents of the canonical form theorem? A proof of it is a function, proof, such that

$$\mathsf{proof}\text{`}us\text{`}u\text{`}p : \mathsf{combine_dot}(us) \simeq u[us : \mathsf{List}^{+}(\mathsf{U}), u : \mathsf{U}, p : \mathsf{combine_dot}(us) \simeq u]$$

Taylor's idea is to apply the theorem in an *extended context,* with variables declared in U, so assume $u, v, w : \mathsf{U}$ are given. Then we can construct a proof of

$$\mathsf{Built_from}(\mathsf{cat}(\mathsf{cat}([u], [v]), [w]), (u \cdot v) \cdot w)$$

and this is given as input to proof ` · · · ` · · · ` to obtain a proof of

$$\mathsf{combine_dot}(\mathsf{cat}(\mathsf{cat}([u], [v]), [w])) \simeq (u \cdot v) \cdot w$$

and after normalising, we get a proof of

$$u \cdot (v \cdot w) \simeq (u \cdot v) \cdot w$$

indicating that the left-hand side of $\simeq$ is the normal form of the right-hand side, the left-hand side being normal because it is computed by combine_dot.

Since there is no elimination rule for U, we cannot define any interesting functions taking Us as input and therefore we cannot give $(u \cdot v) \cdot w$ as input. It seems therefore that multiplicative expressions, in Taylor's approach, are represented as

Proof trees, $p : \mathsf{Built_from}(us, u)$ when they appear as *input* to normalisation
Semigroup elements, $u : \mathsf{U}$, when they are *output*

Let us finally remark that Taylor's proof has essential structural similarity with the correctness proof we give for normalise in Section 6.2 using the type of normal expressions.

7. Conclusions

7.1. Usage of Type Theory

A relaxed Curry–Howard doctrine finds its expression in our *usage* of type theory and particular in our view on specification – we split the programming problem into a *computational* part and a *logical* part and suggest that, when possible, the computational part be treated first and formally independently of the logical part. In terms of the specification relation

$$\mathsf{Spec}(x, y)\ \mathbf{Type}\ [x, y : \mathsf{Mul}]$$

this means that instead of trying to prove

$$(\forall x : \mathsf{Mul})(\exists y : \mathsf{Mul})\ \mathsf{Spec}(x, y)$$

we try to prove

$$(\exists u : \mathsf{Mul} \to \mathsf{Mul})(\forall x : \mathsf{Mul})\ \mathsf{Spec}(x, u\text{`}x)$$

the canonical proofs of which are of the form

$$\langle (\lambda x) f(x), (\lambda x) p(x) \rangle : (\exists u : \mathsf{Mul} \to \mathsf{Mul})(\forall x : \mathsf{Mul})\ \mathsf{Spec}(x, u\text{`}x)$$

where

$$f(x):\mathsf{Mul}[x:\mathsf{Mul}]$$

is the computational part and

$$p(x):\mathsf{Spec}(x, f(x))[x:\mathsf{Mul}]$$

is the logical part asserting the correctness of the computational part. In situations where on informal grounds you can find a simple function f, you have the advantage of the more specific problem to find a p to go with a known f. ISABELLE has a powerful unification mechanism which helps in filling in the holes in your proof, guided by the tactics you serve. For this feature to be useful, however, you must have some prior knowledge of the problem you want to solve, so that you can guide your proof assistant in the right direction. The synthesising mechanism of ISABELLE does not solve your problem, but helps in keeping track of the formal details. The (proper) primitive recursive normaliser of Section 6.1 is an example where f is simpler than p. In the ISABELLE proof session, we did f, and ISABELLE did p, guided by us.

7.2. The General Recursive Program as a Graph

When representing the general recursive program as an inductively defined input/output relation, its *graph,* we can split correctness in a natural way into *termination* and *partial correctness.* Proofs of termination and partial correctness of the external program can then be combined to obtain a correct *internal* program that satisfies the clauses of the general recursive program as conversions. Note that partial correctness does not enter the computational part, which is done with **assrec**, which essentially embodies a particular termination proof. Partial correctness enters first in the logical part. Note also that the correctness of the internal program was done with a lemma stating that any function satisfying the general recursive clauses (as conversions) is in fact a correct normaliser and that the proof of the lemma was another application of **assrec** and thus termination was involved once again in the correctness proof.

Termination of the *external* program gives the *computational* part of the *internal* program.
Partial correctness *and termination* of the *external* program give the *logical* (correctness) part of the *internal* program.

Since all functions in type theory are total we cannot make sense of a distinction between correctness and partial correctness for internal programs.

7.3. The General Recursive Program as a Fixed Point

The fixed point approach makes best sense in a context where you can talk about partial functions. When you try to describe the general recursive program in terms of a *type theory* functional of which the program is thought of as a fixed point, you lose information about the behaviour of the functional on

partial functions. Neither monotonicity nor continuity of the functional can be expressed in terms of its action on total functions. The absence of partial functions in type theory leads to the question of what notion of fixed point to use, since we cannot talk about *least* fixed points. When searching for a satisfactory notion of fixed point we found that the information lost when restricting to total functions had to be regained by knowledge of the *syntax* of the functional. The explanation of our preferred notion, *applied reduction by cases,* was not in terms of a general functional but in terms of the left- and right-hand sides of the general recursive program.

The fact that, using this approach, we succeeded in producing a type theory imitation of the general recursive program up to conversion (*applied conversion by cases*) is due to the special property of our general recursive program that it never calls itself more than once in each clause, a property that breaks down for the conditional expressions. Note that the graph analysis technique described above generalises well to the conditional expressions.

We think that the functional (our step) is a bad type theory representation of the general recursive program. In type theory we must have compensation for the loss of information caused by the functional being applicable only to total functions. An attempt in that direction would be to analyse the general recursive program in terms of its *splitting* strategy and its *join* operation, but it seems difficult to perform this program in a natural way unless *extensional* type theory is assumed.

Acknowledgements

This research has been carried out as a part of ESPRIT Basic Research Action "Logical Frameworks". It has been supported by STU (Swedish National Board for Technical Development).

References

[Acz77] Aczel, P.: An Introduction to Inductive Definitions. In: *Handbook of Mathematical Logic,* J. Barwise (ed.), pp. 739–782, North-Holland, 1977.

[BCM89] Backhouse, R., Chisholm, P., Malcolm, G. and Saaman, E.: Do-it-Yourself Type Theory (Part 1). *Formal Aspects of Computing,* 1, 19–84 (1989).

[BoM79] Boyer, R. and Moore, J.: *A Computational Logic,* Academic Press, New York, 1979.

[CoH88] Coquand, T. and Huet, G.: The Calculus of Constructions. *Information and Computation,* 76, 95–120 (1988).

[CoP89] Coquand, T. and Paulin, C.: Inductively Defined Types, *Proc. Workshop on Programming Logic, Båstad,* May 1989.

[Dyb91] Dybjer, P.: Comparing Integrated and External Logics of Functional Programs. *Science of Computer Programming,* 14, 57–79 (1990).

[Dyb89] Dybjer, P.: An Inversion Principle for Martin-Löf Type Theory, *Proc. Workshop on Programming Logic, Bqstad,* May 1989.

[Gir86] Girard, J. Y.: Linear Logic and Parallelism. In: *Mathematical Models for the Semantics of Parallelism,* M. Venturini Zilli (ed.), pp. 166–182, LNCS 280, Springer-Verlag, 1986.

[Mar84] Martin-Löf, P.: *Intuitionistic Type Theory,* Bibliopolis, 1984.

[Nor88] Nordström, B.: Terminating General Recursion. *BIT,* 28, 605–619, 1988.

[NPS90] Nordström, B., Petersson, K. and Smith, J.: *Programming in Martin-Löf's Type Theory: an Introduction,* Oxford University Press, 1990.

[Pau86] Paulson, C.: Proving Termination of Normalisation Functions for Conditional Expressions. *Journal of Automated Reasoning*, 2, 63–74 (1986).

[Plo81] Plotkin, G. D.: A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

[Sal89] Salvesen, A.: On Information Discharging and Retrieval in Martin-Löf's Type Theory, PhD thesis, Institute of Informatics, University of Oslo, March 1989.

[Tay88] Taylor, P.: Using Constructions as a Metalanguage, Technical Report ECS-LFCS-88-70, Laboratory for Foundations of Computer Science, University of Edinburgh, December 1988.

Received July 1990
Accepted November 1990 *by R. C. Backhouse*