Pattern Matching with Dependent Types

Thierry Coquand*
Chalmers University

Preliminary version, June 1992

Introduction

This note deals with notation in type theory. The definition of a function by pattern matching is by now common, and quite important in practice, in functional programming languages (see for instance [1]). We try here to introduce such definitions by pattern matching in Martin-Löf's logical framework.

1 Statement of the Problem

1.1 A Short Presentation of Martin-Löf's Logical Framework

For a more complete presentation of Martin-Löf's logical framework, which is implemented in ALF, we refer to the book "Programming in Martin-Löf's Type Theory" [16], chapter 19 and 20. We recall that each type T is of the form $(x_1 : A_1, \ldots, x_n : A_n)A$ where A is Set or of the form $El(a)$. If A is of the form $El(a)$, we say that T is an **small type**, and it is a **large type** otherwise if A is Set. If a type of a term is of the form $(x_1 : A_1, \ldots, x_n : A_n)A$, we say that n is the **arity** of this term. An **instance** of a term u of arity n is a term definitionally equal to a term of the form $u(v_1, \ldots, v_n)$.

A **context** is a list of type declaration $\Gamma = x_1 : A_1, \ldots, x_n : A_n$. As in [19], we relativize all judgements of type theory with respect to a context. An **interpretation** or **contextual mapping** between two contexts $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $\Delta = y_1 : B_1, \ldots, y_m : B_m$ is a simulateneous substitution $S = \{y_1 := v_1, \ldots, y_m := v_m\}$ such that

$$v_1 : B_1\ (\Gamma),\ v_2 : B_2[v_1]\ (\Gamma), \ldots, v_m : B_m[v_1, \ldots, v_{m\ 1}]\ (\Gamma)^1.$$

We write in this case $S : \Gamma \to \Delta$. If M is an open expression in Δ, we write by simple juxtaposition MS the result of the substitution S to M. Notice that if A is a type in Δ then AS is a type in Γ, and if $a : A\ (\Delta)$, then $aS : AS\ (\Gamma)$.

If $T_1 : \Gamma_1 \to \Gamma$ and $T_2 : \Gamma_2 \to \Gamma_1$ we write $T_2; T_1 : \Gamma_2 \to \Gamma$ the composition of T_1 and T_2.

Martin-Löf's logical framework is an open framework: the user can add new constants and new computation rules.

*coquand@cs.chalmers.se

[1] If v_i is definitionally equal to y_i, we omit $y_i := v_i$ in the writing of the interpretation; thus, $\{x := 0\}$ is a contextual mapping from $y : \mathsf{N}$ to $x : \mathsf{N}, y : \mathsf{N}$ meaning $\{x := 0, y := y\}$.

For instance, we get the Σ sets by declaring the constants [2]:

$$\begin{array}{rcl}
\Sigma & : & (X : \mathsf{Set}, (X)\mathsf{Set})\,\mathsf{Set} \\
\mathsf{pair} & : & (X : \mathsf{Set}, Y : (X)\mathsf{Set})\,(x : X, Y(x))\,\Sigma(X, Y) \\
\mathsf{split} & : & (X : \mathsf{Set}, Y : (X)\mathsf{Set})\,(Z : (\Sigma(X, Y))\mathsf{Set}) \\
 & & \quad ((x : X, y : Y(x))Z(\mathsf{pair}(X, Y, x, y))) \\
 & & \quad\quad (w : \Sigma(X, Y)) \\
 & & \quad\quad\quad Z(w)
\end{array}$$

and asserting the equality (which can be read as a computation rule):

$$\mathsf{split}(A, B, Z, z, \mathsf{pair}(A, B, a, b)) = z(a, b) : Z(\mathsf{pair}(A, B, x, y))$$

where

$$\begin{array}{rcl}
A & : & \mathsf{Set}, \\
B & : & (A)\,\mathsf{Set}, \\
Z & : & (\Sigma(A, B))\,\mathsf{Set} \\
a & : & A, \\
b & : & B(a) \\
z & : & (x : A, y : B(x))\,Z(\mathsf{pair}(A, B, a, b))
\end{array}$$

The usual cartesian product is defined by

$$A \times B = \Sigma(A, (x)B) : \mathsf{Set} \quad [A : \mathsf{Set}, B : \mathsf{Set}]$$

The set of natural numbers is introduced by declaring the constants:

$$\begin{array}{rcl}
\mathsf{N} & : & \mathsf{Set} \\
\mathsf{0} & : & \mathsf{N} \\
\mathsf{succ} & : & (\mathsf{N})\,\mathsf{N} \\
\mathsf{natrec} & : & (C : (x : \mathsf{N})\,\mathsf{Set}, C(\mathsf{0}), (x : \mathsf{N}, y : C(x)\,C(\mathsf{succ}(x)), n : \mathsf{N})C(n)
\end{array}$$

and the equalities (which can be read as computation rules):

$$\begin{array}{l}
\mathsf{natrec}(C, x, z, \mathsf{0}) = x : C(\mathsf{0}) \\
\mathsf{natrec}(C, x, z, \mathsf{succ}(a)) = z(a, \mathsf{natrec}(C, x, z, a))
\end{array}$$

where

$$\begin{array}{rcl}
C & : & (x : \mathsf{N})\,\mathsf{Set} \\
x & : & C(\mathsf{0}) \\
z & : & (x : \mathsf{N}, y : C(x))\,C(\mathsf{succ}(x))
\end{array}$$

The computation rules generate the definitional equality between terms.

[2] We allow ourselves to write in general A instead of $El(A)$.

Quite important is the distinction between **canonical** and **non-canonical constants**. In the examples above, Σ, pair N, 0 and succ are canonical constants, but split, natrec and $\times$ are non canonical.

If the type of a canonical constant C is of the form $(x_1 : A_1, \ldots, x_n : A_n)\mathsf{Set}$, we say that C is a **connective**. The meaning of a connective C is given by a set of canonical constants of types of the form $(y_1 : B_1, \ldots, y_m : B_m)El(C(a_1, \ldots, a_n))$, that are called **constructors** of C. (In the case of mutual inductive definitions, we can have a set of connectives that are simultaneously defined by a set of canonical constants.) By extension, we consider also that connectives are constructors of the type Set.

A canonical constant whose type is a small type is considered to be a primitive notion, that is self-justifying. In the example above, 0 and succ are considered to be primitive notions, and the canonical set N is defined by its set of constructors 0 and succ.

We say that a term is in **constructor form** iff it is definitionally equal to a term of the form $c(u_1, \ldots, u_n)$ where c is a constructor of arity n. The constructor c is then uniquely determined. We say that a term t is **directly structurally smaller** than a term u iff

- both u and v are of small types and of arity 0,
- u is of constructor form $c(a_1, \ldots, a_n)$ and t is definitionally equal to one a_j of arity 0 or one instance of one a_j of arity > 0.

Being **structurally smaller** is defined by taking the transitive closure of this relation.

We use in an essential way the "no confusion" property of constructors. This covers two properties. The first is that a definitional equality between two terms of the form $a(u_1, \ldots, u_n)$ and $b(v_1, \ldots, v_m)$ if a and b are two distinct constructors, cannot hold. The second is that, if c is a constructor of type $(x_1 : A_1, \ldots, x_n : A_n)A$, then the equality $c(u_1, \ldots, u_n) = c(v_1, \ldots, v_n) : A[u_1, \ldots, u_n]$ implies

$$u_1 = v_1 : A_1, \ldots, u_n = v_n : A_n[u_1, \ldots, u_{n\ 1}].$$

The non canonical constant $\times$ is **explicitely defined** in term of split.

The definitions of split and natrec are not explicit, and we refer to these constants as **implicitely defined** constants. The meaning of implicitely defined constants is given by their computation rules.

It is an important problem to give some criteria that ensure the correctness of the addition of new constants and computation rules. We try here to analyse this problem using the pattern matching notation introduced in functional languages (see for instance [1]).

1.2 Inductively defined connectives

We shall consider only connectives that are inductively defined The relation of being structurally smaller is then expected to be well-founded. We shall take this well-foundness property as a fundamental assumption on the constructors, without trying to analyse it further here. We simply mention that the constructors presented in [7, 8] satisfy this well-foundness property.

Here are two counter-examples.

The set

$$V : \mathsf{Set},$$

with one constructor

$$\Lambda : ((A : \mathsf{Set})(A)A)V.$$

The polymorphic identity $(A, x)x$ is of type $(A : \mathsf{Set})(A)A$, and hence the term $\Lambda((A, x)x)$ is of type V. This term is structurally smaller than itself. It follows that the relation of being structurally smaller is not well-founded.

Likewise, the set

$$U : \mathsf{Set},$$

with one constructor

$$c : (\mathsf{Set})U,$$

has to be rejected. The reason is however more subtle than for the previous counter-example. We notice first that, were this set accepted, so would be $T : (U)\mathsf{Set}$ by $T(c(X)) = X : \mathsf{Set}$. We could then introduce a set $W : \mathsf{Set}$ with only one constructor $\mathsf{sup} : (x : U)((T(x))W)W$ (which is inductively defined given U, T). But then $\mathsf{sup}(c(W), (x)x)$ is structurally smaller than itself.

The first example, suggested by a remark of Per Martin-Löf, shows that the well-foundness requirement on the relation of being structurally smaller is a stronger requirement than mere normalisation. Indeed the set V is defined by a second-order quantification, and it can be shown, by the usual reducibility method, that its addition to inductively defined sets preserves the normalisation property.

1.3 Some difficulties with the usual elimination schemas

It is known how to associate to any inductively defined connective an elimination constant, together with its computation rules. This is described for instance in [6]. One can check that all the examples of implicitely defined constants and computations rules described in [16] are of this form. A first criteria for ensuring the correctness of the addition of new constants and computation rules is to allow only the addition of such elimination constants. Experiments with restricting the addition of implicitely defined constants to be elimination constants have shown some drawbacks of this approach.

One first drawback is that we do not quite get the expected computational behaviour. If we define for instance $\mathsf{add} : (\mathsf{N}; \mathsf{N})\mathsf{N}$ by $\mathsf{add}(x, y) = \mathsf{natrec}(y, x, (u, v)\mathsf{succ}(v))$, then $\mathsf{add}(x, \mathsf{succ}(y))$ reduces to $\mathsf{succ}(\mathsf{natrec}(y, x, (u, v)\mathsf{succ}(v))$ and one needs to fold back this expression to get the expected $\mathsf{succ}(\mathsf{add}(x, y))$.

One second drawback is readability. For instance, we want to consider an object such as $\mathsf{half} : (\mathsf{N})\mathsf{N}$ defined by

$$\mathsf{half}(0) = 0,\ \mathsf{half}(\mathsf{succ}(0)) = 0,\ \mathsf{half}(\mathsf{succ}(\mathsf{succ}(x))) = \mathsf{succ}(\mathsf{half}(x)),$$

as given directly by these equations, rather than being given by an explicit definition which is a "coding" of this object in term of natrec.

The second drawback is in practice quite important. The pattern matching notation is essential in functional programming languages[3].

The next anomaly is the necessity to consider higher "sets" for defining naturally a function such as $\mathsf{inf} : (\mathsf{N}; \mathsf{N})\mathsf{N}$. It is quite surprising that, in order to justify the equations

$$\mathsf{inf}(\mathsf{0}, y) = \mathsf{0},\ \mathsf{inf}(\mathsf{succ}(x), \mathsf{0}) = \mathsf{0},\ \mathsf{inf}(\mathsf{succ}(x), \mathsf{succ}(y)) = \mathsf{succ}(\mathsf{inf}(x, y)),$$

one needs to introduce the set of numerical functions.

Another problem appeared for inductively defined families. Given a connective with an arity > 1, there are several possible elimination constants depending on what arguments are considered to be parameters. For instance, there are two different elimination constants for the connective $\mathsf{Id} : (A : \mathsf{Set}, x, y : A)\mathsf{Set}$ of unique constructor $\mathsf{refl} : (A : \mathsf{Set}; x : A)\mathsf{Id}(A, x, x)$. In this case, it is yet unknown if these two elimination constants are equivalent.

The first and, in particular, second drawbacks are strong motivations for allowing the introduction of implicitely defined constants defined by computation rules that are pattern matching equations. This seems to solve in general the third anomaly. In an unexpected way, this seems to have some bearing on the fourth problem, as we will try to explain below.

2 A General Presentation of Pattern Matching

There are two independent requirements for the correctness of the introduction of one implicitely defined constant together with its computation rules. These requirements are only sufficient in ensuring that the constant does define a total function on the underlying datatype.

The first is the requirement that all definitions, that may be recursive, are well-founded.

The second is that the equations cover all possible cases of the arguments and do not introduce ambiguities in the computation. We ensure this by imposing the definitions to be exhaustive and mutually disjoint.

2.1 Well-founded Definitions

A simple condition ensures the fact that all definitions are well-founded, and seem furthermore sufficient in practice. Let n be the arity of the implicitely defined constant f to be defined. The condition is that there exists an index $i \leq n$ such that, for all equations $f(u_1, \ldots, u_n) = e$, and all recursive call $f(v_1, \ldots, v_n)$ of f in e, the constant f does not occur in $v_1, \ldots, v_n$ and the term v_i is structurally smaller than the term u_i.

It would be possible to give a less restrictive condition, by considering instead a lexicographic extension of the structural ordering. However, this restriction suffices to recover the usual elimination schemas. It is also quite simple to ensure that this condition holds.

Notice that this condition provides more general equations than the ones provided by the usual primitive recursive schema. In the usual primitive recursive schema indeed, the parameters cannot vary in recursive calls. This is not required here.

[3]The earliest, to our knowledge, mention of this notation appears in [2]. A proposal of extending functional language with an "inductive" case expression, which hence ensures termination, is presented in [3].

For instance, this will justify directly the following kind of definitions of a a function $f : (\mathsf{N}, \mathsf{N})\mathsf{N}$.

$$f(\mathsf{0}, m) = g(m),\ \ f(\mathsf{succ}(n), m) = h(n, m, f(n, k(n, m))),$$

where $g : (\mathsf{N})\mathsf{N}$, $h : (\mathsf{N}, \mathsf{N}, \mathsf{N})\mathsf{N}$ and $k : (\mathsf{N}, \mathsf{N})\mathsf{N}$ are previously defined functions. Notice that the parameter m changes to $k(n, m)$ in the recursive call of f. This can be done only using the set of numerical function if we restrict ourselves the usual schema of primitive recursion (see [4]).

2.2 Covering

To analyse further the condition that the definitions are exhaustive and mutually disjoint, we introduce one notion reminiscent of a notion used in Per Martin-Löf's representation of choice sequences in type theory.

Let us motivate briefly what follows. We want to add a new implicitely defined constant f of type $(x_1 : A_1, \ldots, x_n : A_n)A$, together with a set of computation rules. Let Δ be the context $x_1 : A_1, \ldots, x_n : A_n$ of arguments of f. We only consider computation rules for f of the form

$$f(a_1, \ldots, a_n) = e : A[a_1, \ldots, a_n]\ (\Gamma),$$

with $a_1 : A_1, \ldots, a_n : A_n[a_1, \ldots, a_{n\ 1}]$. We can think of $a_1, \ldots, a_n$ as defining a contextual mapping $S : \Gamma \to \Delta$, and this suggests to introduce the notation $f(S) = e : AS\ (\Gamma)$ for such a computation rule.

With this notation, the conditions on a system of computation rules $f(S_j) = e_j : AS_j\ (\Gamma_j)$ will be expressed as conditions on a system of contextual mappings $S_j : \Gamma_j \to \Delta$. We want to express that such a system defines a "partition of the space defined by Δ."

We are going to analyse this problem in the same way that pattern matching in ordinary functional languages is reduced to a succession of case expressions over a variable (see [1]).

We say first that a system of contextual mapping $S_1 : \Gamma_1 \to \Delta, \ldots, S_m : \Gamma_m \to \Delta$ over a common context $\Delta = x_1 : A_1, \ldots, x_n : A_n$ is an **elementary covering** of Δ iff there exists an index $i \leq n$ such that

- all terms $x_i S_j : A_i S_j\ (\Gamma_j)$, for $j \leq m$, are in constructor form,
- if $S : \Gamma \to \Delta$ is a contextual mapping such that $x_i S$ is in constructor form, then there exists one and only one $j \leq m$ and $T : \Gamma \to \Gamma_j$ such that $S = T; S_j$.

This definition may look complicated but it is a possible way of specifying what is a case expression over the ith argument. In the case of a context with only non dependent types, we recover the usual notion of case expression as in [1]. In the general case however, we cannot keep the same notion of patterns of [1] (as the examples below will show, we need for instance to consider non linear patterns), and our abstract definition seems necessary.

An instance is the elementary covering defined by $x = \mathsf{0}$ and $x = \mathsf{succ}(y)$ $(y : \mathsf{N})$ of the context $x : \mathsf{N}$.

A second example is the empty set of contextual maps over the context

$$\Delta = p : \mathsf{Id}(\mathsf{N}, \mathsf{0}, \mathsf{succ}(\mathsf{0})).$$

This is an elementary covering. Indeed, the only constructor of the connective Id is refl, and a term of the form $\mathsf{refl}(A, u)$ cannot be of type $\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0))$. Otherwise, we would have

$$\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)) = \mathsf{Id}(A, u, u),$$

and hence, because Id is a constructor, $0 = u : \mathsf{N}$ and $\mathsf{succ}(0) = u : \mathsf{N}$. But this implies $0 = \mathsf{succ}(0) : \mathsf{N}$, which does not hold, because 0 and succ are different constructors.

A more elaborate example is for the context

$$\Delta = x, y : \mathsf{N}; p, q : \mathsf{Id}(\mathsf{N}, x, y).$$

It can be checked that, if we define

$$\Gamma = x : \mathsf{N}, p : \mathsf{Id}(\mathsf{N}, x, x),$$

then the unique contextual mapping

$$\{y := x, q := \mathsf{refl}(\mathsf{N}, x)\} : \Gamma \to \Delta,$$

defines an elementary covering of Δ. Indeed, this follows from the fact that refl is the only constructor of Id and that if $\mathsf{refl}(\mathsf{N}, u)$ is of type $\mathsf{Id}(\mathsf{N}, v, w)$, we have

$$\mathsf{Id}(\mathsf{N}, u, u) = \mathsf{Id}(\mathsf{N}, v, w) : \mathsf{Set},$$

and hence, since Id is a constructor, we have $u = v : \mathsf{N}$ and $u = w : \mathsf{N}$.

We define now what it means for a system of contextual mapping $S_i : \Delta_i \to \Delta$ into a common context Δ to be a **covering** of Δ :

- the identity interpretation $\Delta \to \Delta$ is a covering of Δ,
- if $S_i : \Delta_i \to \Delta$, for $i \leq p$ is an elementary covering of Δ and $T_{ij} : \Delta_{ij} \to \Delta_i$, for $j \leq q_i$, is a covering of Δ_i, then $T_{ij}; S_i : \Delta_{ij} \to \Delta$ is a covering of Δ.

For instance $x = 0$, together with $x = \mathsf{succ}(0)$ and $x = \mathsf{succ}(\mathsf{succ}(y))$ $(y : N)$ define a covering of $x : N$.

An example of covering of the context $\Delta = x : \mathsf{N}, y : \mathsf{N}$ is given by

- $\{x := 0\} : (y : \mathsf{N}) \to \Delta$,
- $\{x := \mathsf{succ}(x_1), y := 0\} : (x_1 : \mathsf{N}) \to \Delta$ and
- $\{x := \mathsf{succ}(x_1), y := \mathsf{succ}(y_1)\} : (x_1 : \mathsf{N}, y_1 : \mathsf{N}) \to \Delta$.

If we take again our last example of an elementary covering, it can be checked that the unique contextual mapping

$$\{p := \mathsf{refl}(\mathsf{N}, x)\} : (x : \mathsf{N}) \to \Gamma,$$

is an elementary covering of Γ. Hence, the unique contextual mapping

$$\{y := x, p := \mathsf{refl}(\mathsf{N}, x), q := \mathsf{refl}(\mathsf{N}, x)\} : (x : \mathsf{N}) \to \Delta,$$

is a covering of $\Delta = x, y : \mathsf{N}; p, q : \mathsf{Id}(\mathsf{N}, x, y)$.

Following Per Martin-Löf's terminology, we call **neighbourhood** of a context any contextual map that is part of a covering of this context. The collection of neighbourhoods of a covering of a context can be thought of as defining a partition of the "space" defined by this context. This notion of neighbourhood corresponds to the notion of patterns used in functional programming languages: in the case of a context with only non dependent types, we recover exactly the notion of pattern matching described in [3, 1].

2.3 Sufficient Conditions For Correctness

The sufficient conditions ensuring the correctness of the addition of a new implicitely defined constant f of type $(x_1 : A_1, \ldots, x_n : A_n)A$, of argument context $\Delta = x_1 : A_1, \ldots, x_n : A_n$ and of computation rules of the form $f(S_j) = e_j : AS_j\ (\Gamma_j)$ are that:

- there is no nested occurence of f in e_j, and all recursive call of f are done on structurally smaller arguments than the lefthandside arguments (which can be ensured as described above),
- the system of contextual maps $S_j : \Gamma_j \to \Delta$ is a covering of Δ.

2.4 Some comments on this method

The method followed here can be described as follows. When justifying a rule

$$f : (x_1 : A_1, \ldots, x_n : A_n)A,$$

we analyse exhaustively the possible forms S of the arguments of f, and in each possible case S, we build a term e_S of type AS, using constructors and already defined constants.

We allow recursive calls of the constant we are defining, provided these calls are on structurally smaller arguments.

Naturally associated to this justification of an implicitely defined constant

$$f : (x_1 : A_1, \ldots, x_n : A_n)$$

is the following computation rule for f. If a given argument $(a_1, \ldots, a_n)$ is an instance of the case S, then the value of $f(a_1, \ldots, a_n)$ is the value of the corresponding instance of e_S. Otherwise, the argument list of f is not "instantiated enough", and $f(a_1, \ldots, a_n)$ cannot be head reduced.

2.5 Some Examples

The function $\mathsf{inf} : (\mathsf{N}, \mathsf{N})\mathsf{N}$ which is defined implicitely by:

$$\mathsf{inf}(\mathsf{0}, y) = \mathsf{0},\ \mathsf{inf}(\mathsf{succ}(x), \mathsf{0}) = \mathsf{0},\ \mathsf{inf}(\mathsf{succ}(x), \mathsf{succ}(y)) = \mathsf{succ}(\mathsf{inf}(x, y)).$$

The recursive call is justified by the fact that it is structurally smaller on the first (or the second) argument.

It is standard how to reduce such a definition to the usual elimination rules over the type N, by using the set of numerical functions.

By contrast, it is not clear how to represent the following computation rule in term of the usual elimination rules[4]. We have seen that the unique contextual mapping

$$\{y := x, p := \mathsf{refl}(\mathsf{N}, x), q := \mathsf{refl}(\mathsf{N}, x)\} : (x : \mathsf{N}) \to \Delta,$$

is a covering of $\Delta = x, y : \mathsf{N}; p, q : \mathsf{Id}(\mathsf{N}, x, y)$. It follows that it is correct to add a new constant $f : (x, y : \mathsf{N}; p, q : \mathsf{Id}(\mathsf{N}, x, y))\mathsf{Id}(\mathsf{Id}(\mathsf{N}, x, y), p, q)$ together with the computation rule

$$f(x, x, \mathsf{refl}(\mathsf{N}, x), \mathsf{refl}(\mathsf{N}, x)) = \mathsf{refl}(\mathsf{Id}(\mathsf{N}, x, x), \mathsf{refl}(\mathsf{N}, x)) \ (x : \mathsf{N})$$

The next example still concerns the connective Id. As we said before, there are two possible elimination rules over this connective, depending on what arguments are considered to be paramaters.

The first one, with the first argument is a parameter, is

$$\begin{array}{rcl} F & : & (A : \mathsf{Set}; C : (x, y : A; \mathsf{Id}(A, x, y))\mathsf{Set}; \\ & & d : (x : A)C(x, x, \mathsf{refl}(A, x)); a, b : A; c : \mathsf{Id}(A, a, b)) \\ & & C(a, b, c) \end{array}$$

of computation rule

$$F(A, C, d, a, a, \mathsf{refl}(A, a)) = d(a) : C(a, a, \mathsf{refl}(A, a)),$$

where

$$A : \mathsf{Set},\ C : (x, y : A; \mathsf{Id}(A, x, y))\mathsf{Set},\ d : (x : A)C(x, x, \mathsf{refl}(A, x)),\ a : A.$$

The second one, with the first two arguments are parameters, is

$$\begin{array}{rcl} G & : & (A : \mathsf{Set}; a : A; C : (y : A; \mathsf{Id}(A, a, y))\mathsf{Set}; \\ & & d : C(a, \mathsf{refl}(A, a)); b : A; c : \mathsf{Id}(A, a, b)) \\ & & C(b, c) \end{array}$$

of computation rule

$$G(A, a, C, d, a, \mathsf{refl}(A, a)) = d : C(a, \mathsf{refl}(A, a)),$$

where

$$A : \mathsf{Set},\ a : A,\ C : (y : A; \mathsf{Id}(A, a, y))\mathsf{Set},\ d : C(a, \mathsf{refl}(A, a)).$$

It can be checked that both constants satisfy the sufficient conditions for correctness given above. Only the covering condition has to be checked, because there is no recursive call.

The last example is the well-founded set connective:

$$\mathsf{W} : (A : \mathsf{Set}, B : (A)\mathsf{Set})\mathsf{Set},$$

of unique constructor

$$\mathsf{sup} : (A : \mathsf{Set}, B : (A)\mathsf{Set}, a : A, u : (B(a))\mathsf{W}(A, B))\mathsf{W}(A, B).$$

[4]This problem has been independently suggested by Thomas Streicher.

We can introduce the implicitely defined constant

$$\begin{array}{lll} \mathsf{wrec} & : & (A : \mathsf{Set}, B : (A)\mathsf{Set}, C : (\mathsf{W}(A,B))\mathsf{Set}, \\ & & f : (a : A, u : (B(a))\mathsf{W}(A,B), (x : B(a))C(u(x)))C(\mathsf{sup}(A,B,u)), t : \mathsf{W}(A,B))C(t) \end{array}$$

with the computation rule

$$\mathsf{wrec}(A,B,C,f,\mathsf{sup}(A,B,a,u)) = f(a,u,(x)\mathsf{wrec}(A,B,C,f,u(x))),$$

where

$$A : \mathsf{Set}, B : (A)\mathsf{Set}, C : (\mathsf{W}(A,B))\mathsf{Set}, f : (a : A, u : (B(a))\mathsf{W}(A,B), (x : B(a))C(u(x))).$$

This is justified since $u(x)$ is structurally smaller than $\mathsf{sup}(A,B,a,u)$.

3 How to build coverings

3.1 Unification Problem

If Δ is a context, A a type in Δ, and u, v two terms in Δ of type A, we define a solution of the unification problem

$$u = v : A \ (\Delta)$$

to be a finite system of contextual mappings $S_j : \Gamma_j \to \Delta$ such that

- for all j, we have $uS_j = vS_j : AS_j \ (\Gamma_j)$, and
- if $S : \Gamma \to \Delta$ is a contextual mapping such that $uS = vS : AS \ (\Gamma)$, then there exists one and only one j and $T : \Gamma \to \Gamma_j$ such that $T; S_j = S$.

For a description of the unification problem with dependent types, see [18] and [9]. Since this problem contains already the similar problem for simply typed lambda-calculus, described in [13], we cannot expect to have a general algorithm to solve it. It is however possible to describe a simple algorithm[5], that has three possible outputs

- the system with no contextual mapping (this ensures that the unification problem has no solution),
- a system with exactly one contextual mapping (this ensures that the unification problem has a most general solution),
- the algorithm fails (which corresponds to a difficult unification problem).

[5]This algorithm is similar to the first order unification algorithm, using the fundamental fact that constructors are one to one function.

3.2 Splitting Contexts

We give first a way to build elementary coverings, as it is implemented in ALF. We cannot ensure that this generates all possible elementary coverings, but it is not clear yet how to extend this algorithm, and whether such an extension is needed or not in practice.

Given a context

$$\Delta = x_1 : A_1, \ldots, x_n : A_n,$$

and an index $i \leq n$ such that A_i is a small type, we describe an operation called **splitting the context Δ along i**. This is an algorithm that tries to produce an elementary covering of Δ :

- if A_i is of arity > 0, or if A_i is not in constructor form, then the algorithm fails to produce any covering,
- otherwise, A_i is of the form $El(C(u_1, \ldots, u_n))$ and we can list all the constructors of the connectives C. For each such constructor c of type $(y_1 : B_1, \ldots, y_m : B_m)El(C(v_1, \ldots, v_m))$, we apply the previous unification algorithm for the equation

  $$C(u_1, \ldots, u_n) = C(v_1, \ldots, v_n) : \mathsf{Set}\ (x_1 : A_1, \ldots, x_{i\ 1} : A_{i\ 1}, y_1 : B_1, \ldots, y_m : B_m),$$

  and we collect all the solutions.

Given the fundamental "no confusion" property of constructor, this produces in case of success an elementary covering of Δ.

3.3 General Coverings

General coverings can now be built interactively. Given a context Δ, the user chooses an index i and tries to split Δ along i. If the system answers by giving an elementary covering, the user can then choose to split some of the new produced contexts, and so on, until the user stops eventually producing by composition a covering of Δ.

This interactive way of building coverings has been implemented in ALF, and seems in practice to be quite convenient for the user in ensuring that no cases have been forgotten during the definition of a function by pattern matching. This is in contrast with the usual presentation in functional languages, where one should write the possible cases, and the compiler warns the user that some cases have been forgotten.

The following is a semi-algorithm that checks whether or not a system of contextual mappings $S_j : \Gamma_j \to \Delta$ is a covering[6] (thanks to G. Huet).

First, the system with only the identity mapping is a covering. Otherwise, choose an index i such that all $x_i S_j$ are in constructor form. Then, if possible, split Δ along i. If the answer is an elementary covering $T_i : \Delta_i \to \Delta$ of Δ, this induces a partition of the original system $S_j : \Gamma_j \to \Delta$ into a system of mappings $\Gamma_j \to \Delta_i$. We then recursively check that each of these systems is a covering.

[6] If we think of a covering as a collection of disjoint "pieces" that form a partition of a space, this semi algorithm solves a typical "puzzle" problem. We are given some "pieces" of a space (contextual mapping), and we try to see whether or not they form a partition of this space.

4 Addition of Subsets

Kent Petersson, and independently A. Salvesen, suggested the following notion of subsets which seems to fit nicely with the present notion of implicitely defined constants. We limit here ourselves to the description of a simple example.

The meaning of a connective, such as $\mathsf{N} : \mathsf{Set}$, is given by the set of its constructors

$$0 : \mathsf{N},\ \mathsf{succ} : (\mathsf{N})\mathsf{N}.$$

It is quite natural to allow the introduction of (direct) **subsets** of N, that we get simply by selecting a subset of this set of constructors. For instance, we can introduce the subset $\mathsf{ISZERO} : \mathsf{Set}$ with the only constructor 0, and the subset $\mathsf{POS} : \mathsf{Set}$ with the only constructor succ.

This notion of subsets fits well with the present way of defining a function by pattern matching, where one important step is to list the constructors of a given connective.

For instance, the unique computation rule defines then correctly an implicitely defined function $p : (\mathsf{POS})\mathsf{N}$.

$$p(\mathsf{succ}(x_1)) = x_1\ (x_1 : \mathsf{N}),$$

because the context $x : \mathsf{POS}$ is covered by the contextual mapping $x = \mathsf{succ}(x_1) : \mathsf{N}\ (x_1 : \mathsf{N})$.

We can dually allow the introduction of (direct) **supersets** of N, that we get by adding new constructors. Typically, the set of ordinals $\mathsf{Ord} : \mathsf{Set}$ extends the set N by the addition of one constructor

$$\mathsf{lim} : ((\mathsf{N})\mathsf{Ord})\mathsf{Ord}.$$

The following computation rules define then correctly an implicitely defined function $g : (\mathsf{Ord})\mathsf{N}$.

$$g(0) = 0,\ g(\mathsf{succ}(x)) = \mathsf{succ}(x),\ g(\mathsf{lim}(u)) = g(u(0)).$$

This definition is justified since $u(0)$ is structurally smaller than $\mathsf{lim}(u)$.

We can then define a general inclusion relation between connectives, by taking the transitive closure of the direct inclusion relation defined by the introduction of subsets and supersets. This is a decidable relation.

As the last example shows, the addition of subsets and supersets introduces some overloading facilities. These do not however compromise the decidability of the following problems:

- is the expression A a correct type in the context Γ?
- given a type A in the context Γ, is the expression a a correct term of type A in the context Γ?

as we can convince ourselves by noting that the usual algorithm for these problems apply almost without changes (using the decidability of the inclusion relation between connectives).

It is hoped that, with these new operations, one can represent rather faithfully the example presented in [15].

A more elaborate notion of subtypings appears in [11].

Conclusion

As an experiment of using pattern matching, we have done in ALF the Gilbreath Trick, presented by G. Huet last year [17], which is a non trivial inductive proof. This example shows well the gain in readibility that brings the pattern matching notation. While doing other experiments, it appeared that a quite useful extension of the system would be the introduction of case expressions for proofs, where the case is over a term that may not be in variable form. More generally, the goal seems to be to develop nice enough notations that will hopefully help the analysis of inductive arguments.

The method we follow here has some similarities with Lars Hallnäs notion of partial inductive definitions (see [12, 10]), and with the way proofs are represented in Elf [17]. What we do seems to correspond to a suggestion of [12] to use this notion as a "basis for a logical framework". These connections have to be precised.

In the present analysis of pattern matching, a crucial rôle is played by the "no confusion" property of constructors. In "Language and Philosophical Problems," [20], p. 163 - 167, S. Stenlund emphasizes from a philosophical perspective the importance of this property.

From a proof-theoretic viewpoint, our treatment can be characterized as fixing the meanings of the logical constants by the introduction rules. This possibility is discussed in [5], and constrasted to the dual possibility, which is to fix the meanings of logical constants by the elimination rules.

References

[1] Augustsson, L. "Compiling Pattern Matching." In Compiling Lazy Functional Languages Part II, Ph. D. Thesis, Chalmers, 1987.

[2] Burstall, R.M. "Proving properties of programs by structural induction." Computer Journal 12(1), p. 41 48, 1969.

[3] Burstall, R.M. "Inductively Defined Functions in Functional Programming Languages." Journal of Computer and System Sciences, vol. 34, p. 409 421, 1987.

[4] Colson, L. "About Primitive Recursive Algorithms." LNCS 372, p. 194 206, 1989.

[5] Dummett, M. (1991) *The Logical Basis of Metaphysics.* Duckworth ed.

[6] Dybjer, P. "Inductive Sets and Families in Martin-Löf's Type Theory" Chalmers Report 62, also p. 280-306 in Logical Frameworks, eds. G. Huet and G. Plotkin, Cambridge University Press, 1991.

[7] Dybjer, P. "An inversion principle for Martin-Löf 's type theory." Proceedings of the Workshop on Programming Logic in Bastad, May 1989, Programming Methodology Group Report 54, p. 177-190.

[8] Dybjer, P. "Universes and a General Notion of Simultaneous Inductive-Recursive Definition in Type Theory." in these proceedings.

[9] Elliott, C. M. "Higher-Order Unification with Dependent Function Types." p. 121 136, Proc. Rewriting Techniques and Applications

[10] Eriksson, L.H. "A Finitary Version of the Calculus of Partial Inductive Definitions." SICS research report, also to be published in LNCS, Proceedings of the Second Workshop on Extensions of Logic Programming.

[11] Freeman, T. and Pfenning, F. "Refinement Types for ML." to appear in ACM SIGPLAN 1991, Conference on Programming Language Design and Implementation.

[12] Hallnäs, L. "Partial Inductive Definitions." Theoretical Computer Science 87, 1991, p. 115 - 142.

[13] Huet, G. "A unification algorithm for typed λ-calculus." Theoretical Computer Science, p. 27 57, 1975.

[14] Huet, G. "The Gilbreath Trick: A Case Study in Axiomatization and Proof Development in the COQ Proof Assistant." Technical Report 1511, INRIA, September, 1991.

[15] Kahn, G. "Natural Semantics." INRIA Technical report, 601, 1987.

[16] Nordström B., Petersson K., Smith. J. M. (1990), *Programming in Martin-Löf Type Theory.* Oxford Science Publications, Clarendon Press, Oxford.

[17] Pfenning, F. "Logic Programming in the LF logical framework" in G.Huet and G. Plotkin, Logical Frameworks, Cambridge University Press.

[18] Pym, D. *Proofs, Search and Computation in General Logic.* Thesis, University of Edinburgh, November 1990.

[19] Ranta. A. (1988), "Constructing possible worlds," Mimeographed, University of Stockholm, to appear in Theoria.

[20] Stenlund, S. (1991), *Language and Philosophical Problems.* Routledge ed.