Computer Science at Kent

A Type Theory with Partially Defined Functions

Yong Luo

Technical Report No. 10 - 05
October 2005

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

A Type Theory with Partially Defined Functions

Yong Luo

Computing Laboratory, University of Kent, Canterbury, UK
Email: Y.Luo@kent.ac.uk

Abstract Only can totally defined functions be introduced in conventional dependently typed systems and such functions are normally defined by eliminators. Because of the limitation of the elimination rules, many (mathematical) functions cannot be defined in these systems. This paper argues that the restriction of totality is unnecessary, and proposes a type theory that allows partially defined functions. In this type theory, functions can be introduced by means of pattern matching. It is in general undecidable in dependently typed systems whether patterns cover all the canonical objects of a type, and it is one of the big problems for implementation. Without the restriction of totality, we don't have such problem of totality checking, and hence we have more flexibility to introduce functions than we do in conventional type systems.

1 Introduction

In conventional type theories, functions are normally defined by eliminators of inductive data types. Although some systems such as Coq [B$^+$00] and Epigram [MM04] use techniques to make the definitions easier and more readable, the underline theory is still very similar to the elimination principles and the computation rules. There are tempts to define function in dependently typed systems by means of pattern matching [Coq92], but full covering is a big problem. It is in general undecidable in dependently typed systems whether patterns cover all the canonical objects of a type. One philosophical reason to require full covering (or totality) is about the understanding of an inductive data type, that is, for example, under the empty context, if x is of boolean type $Bool$, then x is either $True$ or $False$. Of course, this understanding (called adequacy) is very nature and the author has no objection to it. In the system proposed in this paper, the properties such as

$$\forall x : Bool.x =_{Bool} True \vee x =_{Bool} False$$

will always hold under the empty context, where $=_{Bool}$ is the Leibniz equality and $\vee$ is the usual logic operator "or". However, such understanding causes many philosophers to go too far and consequently restrict to totally defined functions only. The normal way to guarantee totality is to define every function by eliminator or similar methods, but in this way, lots of interesting functions cannot be defined (examples will be given later). In conventional type theories, the following function not_1 is acceptable, but the function not_2 is not because it is not totally defined.

$$\begin{aligned} not_1 &: Bool \to Bool \\ not_1(Ture) &= False \\ not_1(False) &= True \end{aligned}$$

$$\begin{aligned} not_2 &: Bool \to Bool \\ not_2(True) &= False \end{aligned}$$

A question often asked is: what is $not_2(False)$ under the empty context? In the system proposed in this paper, the answer is the following.

> $not_2(False)$ is of the type $Bool$. It cannot be computed any further ***at this moment*** and the normal form is itself. It is not a canonical object of the type $Bool$, and the value is unknown ***at this moment***, but we may know its value ***later***. It is ***always*** provable that
>
> $$not_2(False) =_{Bool} True \vee not_2(False) =_{Bool} False$$

Time is employed to explain the informal meaning of the system, in particular, for partially defined functions. Before going to the formal presentation, let's first discuss another question, what can we gain from allowing partially defined functions?

We shall soon give examples to demonstrate what we can gain, but let's examine our purpose further. The reason we allow partially defined functions is not that we really want functions which have to be partially defined. The real reason is because the full covering of patterns is undecidable in dependently typed systems as mentioned before. Without the requirement of totality (or allowing partially defined functions), we simply avoid an undecidable problem. Another important reason is because eliminators cannot define many mathematical functions, even if they are totally defined. The following examples illustrate the limitation of eliminators.

Example 1. The following functions can easily be defined by means of pattern matching, but very difficult to be done by eliminators if possible.

1. Majority function:

$$maj : Bool \to Bool \to Bool \to Bool$$

$$\begin{aligned} maj(True, True, True) &= True \\ maj(True, False, b) &= b \\ maj(False, b, True) &= b \\ maj(b, True, False) &= b \\ maj(False, False, False) &= False \end{aligned}$$

This function cannot be defined in conventional type theories unless we change it to a different function maj' which has eight cases (details are omitted here). For maj, the terms $maj(True, False, b)$ and b are computationally (or intensionally) equal to one another. However, for maj', $maj'(True, False, b)$ is only extensionally equal to b.

2. Ackermann function:

$$ack : Nat \to Nat \to Nat$$

$$\begin{aligned} ack(zero, y) &= succ(y) \\ ack(succ(x), zero) &= ack(x, succ(zero)) \\ ack(succ(x), succ(y)) &= ack(x, ack(succ(x), y)) \end{aligned}$$

where Nat is the type of natural numbers.
This is a nested function, the function symbol ack appears in its arguments in the last equation. This sort of functions cannot be defined in conventional type theories.

3. $even$ and odd number:

$$\begin{aligned} even &: Nat \to Bool \\ odd &: Nat \to Bool \end{aligned}$$

$$\begin{aligned} even(zero) &= True \\ even(succ(x)) &= odd(x) \\ odd(zero) &= False \\ odd(succ(x)) &= even(x) \end{aligned}$$

These are two mutually defined functions. The eliminator of Nat cannot handle such functions.

4. Predecessor:

$$pred : Nat \to Nat$$

$$pred(succ(x)) = x$$

This is a partially defined function. In conventional type theories, we have to give an arbitrary value to $pred(zero)$ even it doesn't make any sense. In the type theory proposed in this paper, we don't have to give a value to $pred(zero)$, but we have the following understanding of it.
$pred(zero)$ is of the type Nat, and we can prove that it is either (extensionally) equal to $zero$ or $succ(x)$ for some x. $pred(zero)$ cannot be computed any further ***at this moment*** and the normal form is itself. It is not a canonical object of the type Nat, and the value is unknown ***at this moment***, but we may know its value ***later***, or we may never know.

The above examples illustrate the limitation of the elimination principle for inductive data types. From the viewpoint of programming, the limitation of eliminator means less flexibility of programming. For example, how do we define the Ackermann function? We may design a new function which can be defined by eliminators and is (extensionally) equivalent to the Ackermann function. But this would loss all the beauty of the function and most programmers would feel very uncomfortable.

The above examples also show that more functions can be introduced by pattern matching. One can also avoid an undecidable problem, totality checking of patterns, by allowing partially defined functions. In other words, functions don't have to be totally defined. Or, we can say, we don't care whether they are totally defined or not. However, we are not trying to include all the functions. There are principles to obey and restrictions such as decidability to take into consideration, when we introduce functions.

Principle 1. As a type theory, it must be consistent.

When one introduces a function symbol and give its type (or kind), one must make sure that it will not potentially cause the system inconsistent. For example, the following two functions are not allowed.

$$emp : Nat \rightarrow Empty$$

$$head : (A : Type) \rightarrow List(A) \rightarrow A$$

where *Empty* is the inductive data type with no canonical object. If these two functions are allowed, then $emp(zero)$ and $head(Empty, nil(Empty))$ are objects of the type *Empty* (where $nil(Empty)$ represents the empty list of the type $List(Empty)$), and hence the system is inconsistent. However, the following function $head_{Bool}$ should be fine.

$$head_{Bool} : List(Bool) \rightarrow Bool$$

In conventional type theories, functions are defined by the eliminators of inductive data types. So, the principle of consistency is guaranteed by the elimination principles. In the system proposed in this paper, this is also the case if a function is defined in such a way. However, for the functions are not (or cannot be) defined by the eliminators, the principle of consistency has to be guaranteed by other means.

It is well-known that the inhabitation of dependent types is undecidable in general. But we do have sub-sets of types in which the inhabitation is decidable (examples will be given in Section 4). How large such a sub-set can be depends on the complexity of an inhabitation checking algorithm and sometimes depends on our needs in practice. The author implemented a decidable sub-set by a sound algorithm, *i.e.* if the algorithm says a type is inhabitable, then there is an object of the type under the empty context. In the implementation, most commonly used functions (which the author knows of) can be introduced.

Principle 2. Type checking must be decidable.

This principle is really a matter of choice. In the system NuPRL [C$^+$86], type checking is undecidable. However, in most of the type systems such as Lego [LP92] and Coq [B$^+$00], type checking is decidable. The main reason that one choose decidable systems is because the implementation of an undecidable system is much more difficult than that of a decidable one. And for this same reason, the author chooses the principle for the system *i.e.* type checking must be decidable.

In dependent type systems, type checking goes down to testing the conversion of two terms, and this is closely related to the properties of Strong Normalisation and Church-Rosser. The author also chooses these two properties as basic requirements for the system. It is worth remarking that type checking does not involve any computation in simply typed systems. This is why partially defined functions and non-terminating programs are accepted in strongly typed programming languages such as ML [MTH90] and Haskell [Tho99].

2 Logical Framework

We start from this section to formally present the type theory. This section will focus on the logical framework.

Definition 1. ***(Terms and Kinds)***

- *Terms*
  1. *a variable is a term,*
  2. *$\lambda x : K.M$ is a term if x is a variable, K is a kind and M is a term,*
  3. *MN is a term if M and N are terms.*
- *Kinds*
  1. *Type is a kind,*
  2. *$El(A)$ is a kind if A is a term,*
  3. *$(x : K_1)K_2$ is a kind if K_1 and K_2 are kinds,*

Remark 1. Terms and kinds are mutually and recursively defined.

Notation: We sometimes write $f(e)$ for fe, $f(e_1, e_2)$ or fe_1e_2 for $(fe_1)e_2$ and so on. We also write A for $El(A)$ when no confusion may occur, and $(K)K'$ or $K \to K'$ for $(x : K)K'$ if x is not free in K'. $[N/x]M$ stands for the expression obtained from M by substituting N for the free occurrences of variable x in M. $FV(M)$ is the set of free variables in M, and $FV(x_1 : K_1, ..., x_n : K_n) = \{x_1, ..., x_n\}$.

Judgement Forms

The explanation of the symbols in a judgement form are the following:

- $\mathit{\Gamma}$ is a sequence of assumptions of the form $x_1 : K_1, ..., x_n : K_n$. $\mathit{\Gamma}$ is also called context if no confusion occurs.
- C is a sequence of constants of the form $c_1 : K_1, ..., c_n : K_n$. The constants include the inductive data types and their constructors.
- F is a sequence of functions of the form $f_1 : (x : K_1)K_1', ..., f_n : (x : K_n)K_n'$. The functions also include the eliminators of inductive data types.
- R is a sequence of equations of the form

$$(\mathit{\Delta}_1)p_1 = t_1 : T_1, ..., (\mathit{\Delta}_n)p_n = t_n : T_n$$

where T_i is Type or of the form $El(A_i)$, and $\mathit{\Delta}_i$ is of the form $x_1 : K_1, ..., x_m : K_m$.

Notation We shall use x, y, z for arbitrary variables, and T for Type or $El(A)$ for some A, and K for an arbitrary kind.

The judgement forms are the following:

- $\mathit{\Gamma} \vdash_R^{C;F}$ valid, which means $\mathit{\Gamma} \vdash_R^{C;F}$ is a valid context. We use ";" to separate the sequences of constants and functions (*i.e.* C and F)
- $\mathit{\Gamma} \vdash_R^{C;F} K$ kind, which means K is a kind in $\mathit{\Gamma} \vdash_R^{C;F}$.
- $\mathit{\Gamma} \vdash_R^{C;F} k : K$, which means k is an object of kind K in $\mathit{\Gamma} \vdash_R^{C;F}$.
- $\mathit{\Gamma} \vdash_R^{C;F} K = K'$, which means K and K' are equal kinds in $\mathit{\Gamma} \vdash_R^{C;F}$.
- $\mathit{\Gamma} \vdash_R^{C;F} k = k' : K$, which means k and k' are equal objects of kind K in $\mathit{\Gamma} \vdash_R^{C;F}$.

Notation We shall write $\mathit{\Gamma} \vdash_R^{C;F} \overline{k} : \overline{K}$ for the sequence of judgements $\mathit{\Gamma} \vdash_R^{C;F} k_1 : K_1$, $\mathit{\Gamma} \vdash_R^{C;F} k_2 : [k_1/x_1]K_2$,...,$\mathit{\Gamma} \vdash_R^{C;F} k_n : [k_{n-1}/x_{n-1}, ..., k_1/x_1]K_n$, and $\mathit{\Gamma} \vdash_R^{C;F} \overline{k = k'} : \overline{K}$ for the sequence $\mathit{\Gamma} \vdash_R^{C;F} k_1 = k_1' : K_1$, $\mathit{\Gamma} \vdash_R^{C;F} k_2 = k_2' : [k_1/x_1]K_2$,...,$\mathit{\Gamma} \vdash_R^{C;F} k_n = k_n' : [k_{n-1}/x_{n-1}, ..., k_1/x_1]K_n$. And $[\overline{k}/\overline{x}]M$ stands for $[k_n/x_n, ..., k_1/x_1]M$.

Commands

We also have four commands; "Assume $x : K$" adds a new assumption in $\mathit{\Gamma}$; "Inductive *sch*" specifies an inductive data type by adding constants in C and eliminator in F and computation rules in R; "Function $f : K$" introduces a new function symbol; and "Equation $(\mathit{\Delta})p = t : T$" adds a new equation rule for a function.

Contexts and assumptions:

$$\frac{}{<>\vdash^{<>;<>}_{<>} \ \text{valid}}$$

$$\frac{\Gamma \vdash^{C;F}_{R} \ \text{valid} \qquad x : K \in \Gamma \cup C \cup F}{\Gamma \vdash^{C;F}_{R} x : K}$$

$$\frac{\Gamma \vdash^{C;F}_{R} K \ \text{kind} \qquad \text{Assume } x : K \qquad x \notin FV(\Gamma \cup C \cup F)}{\Gamma, x : K \vdash^{C;F}_{R} \ \text{valid}}$$

The kind *Type*:

$$\frac{\Gamma \vdash^{C;F}_{R} \ \text{valid}}{\Gamma \vdash^{C;F}_{R} Type \ \text{kind}} \qquad \frac{\Gamma \vdash^{C;F}_{R} A : Type}{\Gamma \vdash^{C;F}_{R} El(A) \ \text{kind}}$$

Dependent product kinds:

$$\frac{\Gamma \vdash^{C;F}_{R} K \ \text{kind} \qquad \Gamma, x : K \vdash^{C;F}_{R} K' \ \text{kind}}{\Gamma \vdash^{C;F}_{R} (x : K)K' \ \text{kind}}$$

$$\frac{\Gamma, x_1 : K_1, ..., x_n : K_n \vdash^{C;F}_{R} M : T}{\Gamma \vdash^{C;F}_{R} \lambda x_1 : K_1 ... \lambda x_n : K_n.M : (x_1 : K_1)...(x_n : K_n)T}$$

$$\frac{\Gamma \vdash^{C;F}_{R} M : (x_1 : K_1)...(x_n : K_n)T \qquad \Gamma \vdash^{C;F}_{R} \overline{k} : \overline{K}}{\Gamma \vdash^{C;F}_{R} Mk_1...k_n : [\overline{k}/\overline{x}]T}$$

Figure 1. Inference rules for logical framework

Inference rules

The inference rules for Logical Framework (LF) are presented in Figure 1, 2, 3 and 4.

Remark 2. We have the following remarks.

- The Logical Framework (LF) presented here is very much similar to the Martin-löf's Logical Framework (MLF) [ML84,NPS90,Luo94] and the Edinburgh Logical Framework (ELF) [HHP87,HHP92] and PAL^+ [Luo03], although the judgement forms are obviously different from all of them. In the logical framework (LF), for $\Gamma \vdash^{C;F}_{R}$, only Γ changes. The others, C, F and R, keep as empty sequences, but they will increase later when new rules are added.

Substitution rules:

$$\frac{\varGamma, x : K, \varGamma' \vdash^{C;F}_{R} \text{ valid} \qquad \varGamma \vdash^{C;F}_{R} k : K}{\varGamma, [k/x]\varGamma' \vdash^{C;F}_{R} \text{ valid}}$$

$$\frac{\varGamma, x : K, \varGamma' \vdash^{C;F}_{R} K' \text{ kind} \qquad \varGamma \vdash^{C;F}_{R} k : K}{\varGamma, [k/x]\varGamma' \vdash^{C;F}_{R} [k/x]K' \text{ kind}}$$

$$\frac{\varGamma, x : K, \varGamma \vdash^{C;F}_{R} K' \text{ kind} \qquad \varGamma \vdash^{C;F}_{R} k = k' : K}{\varGamma, [k/x]\varGamma' \vdash^{C;F}_{R} [k/x]K' = [k'/x]K'}$$

$$\frac{\varGamma, x : K, \varGamma' \vdash^{C;F}_{R} k' : K' \qquad \varGamma \vdash^{C;F}_{R} k : K}{\varGamma, [k/x]\varGamma' \vdash^{C;F}_{R} [k/x]k' : [k/x]K'}$$

$$\frac{\varGamma, x : K, \varGamma' \vdash^{C;F}_{R} k' : K' \qquad \varGamma \vdash^{C;F}_{R} k_1 = k_2 : K}{\varGamma, [k_1/x]\varGamma' \vdash^{C;F}_{R} [k_1/x]k' = [k_2/x]k' : [k_1/x]K'}$$

$$\frac{\varGamma, x : K, \varGamma' \vdash^{C;F}_{R} K' = K'' \qquad \varGamma \vdash^{C;F}_{R} k : K}{\varGamma, [k/x]\varGamma' \vdash^{C;F}_{R} [k/x]K' = [k/x]K''}$$

$$\frac{\varGamma, x : K, \varGamma' \vdash^{C;F}_{R} k' = k'' : K' \qquad \varGamma \vdash^{C;F}_{R} k : K}{\varGamma, [k/x]\varGamma' \vdash^{C;F}_{R} [k/x]k' = [k/x]k'' : [k/x]K'}$$

Figure 2. Inference rules for logical framework—substitution

- The well-typed terms in LF are different from those in MLF and ELF, but similar to the let-expressions in PAL^+. In LF, if the term $\lambda x : K.y$ is well-typed, then y must not be a function. And if a function is applied, it must be fully applied.
- The η equation rule, which looks like

  $$\lambda x_1 : K_1 ... \lambda x_n : K_n . y x_1 ... x_n = y$$

  where y is a variable, is different from the normal one. There are three reasons why this rule is correct and sufficient. First of all, we must be able to prove that, if $yx_1...x_n$ is well-typed then $y \neq x_i$ for any i. Secondly, if y is changed to an application, η equation rule will not be applicable. For example,

  $$\lambda x_1 : K_1 ... \lambda x_n : K_n . (yz) x_1 ... x_n \neq yz$$

  because yz is not fully applied. Thirdly, if y is changed to a λ-abstraction, then β equation rule will be enough.

General equation rules:

$$\frac{\Gamma \vdash_R^{C;F} K \text{ kind}}{\Gamma \vdash_R^{C;F} K = K} \qquad \frac{\Gamma \vdash_R^{C;F} A = B : Type}{\Gamma \vdash_R^{C;F} El(A) = El(B)}$$

$$\frac{\Gamma \vdash_R^{C;F} K = K'}{\Gamma \vdash_R^{C;F} K' = K} \qquad \frac{\Gamma \vdash_R^{C;F} K = K' \quad \Gamma \vdash_R^{C;F} K' = K''}{\Gamma \vdash_R^{C;F} K = K''}$$

$$\frac{\Gamma \vdash_R^{C;F} k : K}{\Gamma \vdash_R^{C;F} k = k : K} \qquad \frac{\Gamma \vdash_R^{C;F} k = k' : K}{\Gamma \vdash_R^{C;F} k' = k : K}$$

$$\frac{\Gamma \vdash_R^{C;F} k = k' : K \quad \Gamma \vdash_R^{C;F} k' = k'' : K}{\Gamma \vdash_R^{C;F} k = k'' : K} \qquad \frac{\Gamma \vdash_R^{C;F} k : K \quad \Gamma \vdash_R^{C;F} K = K'}{\Gamma \vdash_R^{C;F} k : K'}$$

$$\frac{\Gamma \vdash_R^{C;F} k = k' : K \qquad \Gamma \vdash_R^{C;F}\vdash K = K'}{\Gamma \vdash_R^{C;F} k = k' : K'}$$

$$\frac{\Gamma \vdash_R^{C;F} K_1 = K_2 \qquad \Gamma, x : K_1 \vdash_R^{C;F} K_1' = K_2'}{\Gamma \vdash_R^{C;F}\vdash (x : K_1)K_1' = (x : K_2)K_2'}$$

$$\frac{\begin{array}{c}\Gamma, x_1 : K_1, ..., x_n : K_n \vdash_R^{C;F} M = M' : T \\ \Gamma \vdash_R^{C;F} K_1 = K_1' \quad ...\ ... \quad \Gamma, x_1 : K_1, ..., x_{n-1} : K_{n-1} \vdash_R^{C;F} K_n = K_n'\end{array}}{\Gamma \vdash_R^{C;F} \lambda x_1 : K_1 ... \lambda x_n : K_n.M = \lambda x_1 : K_1' ... \lambda x_n : K_n'.M' : (x_1 : K_1)...(x_n : K_n)T}$$

$$\frac{\Gamma \vdash_R^{C;F} M = M' : (x_1 : K_1)...(x_n : K_n)T \qquad \Gamma \vdash_R^{C;F} \overline{k = k'} : \overline{K}}{\Gamma \vdash_R^{C;F} Mk_1...k_n = M'k_1'...k_n' : [\overline{k}/\overline{x}]T}$$

Figure 3. Inference rules for logical framework

- We assume that the good properties such as strong normalisation, Church-Rosser and subject reduction hold for LF. The author has proved strong normalisation for the terms which have correct arities [Luo05b], and proved Church-Rosser by introducing a new η-reduction [Luo05a]. The proof methods can be applied here.

Example 2. In this example, we give a derivation to demonstrate how to use the last equation rule in Figure 4. Suppose $(x : Nat)pred(succ(x)) = x : Nat \ \in \ R$

Core equation rules:

$$(\beta)\quad \frac{\Gamma, x_1 : K_1, ..., x_n : K_n \vdash_R^{C;F} M : T \qquad \Gamma \vdash_R^{C;F} \overline{k} : \overline{K}}{\Gamma \vdash_R^{C;F} (\lambda x_1 : K_1 ... \lambda x_n : K_n . M) k_1 ... k_n = [\overline{k}/\overline{x}]M : [\overline{k}/\overline{x}]T}$$

$$(\eta)\quad \frac{\Gamma \vdash_R^{C;F} y : (x_1 : K_1)...(x_n : K_n)T}{\Gamma \vdash_R^{C;F} \lambda x_1 : K_1 ... \lambda x_n : K_n . y x_1 ... x_n = y : (x_1 : K_1)...(x_n : K_n)T}$$

$$\frac{\begin{array}{c}\Gamma \vdash_R^{C;F} \text{valid} \qquad (\Delta)p = t : T \ \in \ R \\ (\Delta)p = t : T \ =_\alpha \ (\Delta')p' = t' : T' \\ FV(\Delta' \cap \Gamma \cap C \cap F) = \emptyset\end{array}}{\Gamma, \Delta' \vdash_R^{C;F} p' = t' : T'}$$

Figure 4. Inference rules for logical framework

and x is fresh. Then we have the following derivation.

$$\frac{\dfrac{(x : Nat)pred(succ(x)) = x : Nat \ \in \ R}{x : Nat \vdash_R^{C;F} pred(succ(x)) = x : Nat} \qquad \dfrac{zero : Nat \ \in \ C}{\vdash_R^{C;F} zero : Nat}}{\vdash_R^{C;F} pred(succ(zero)) = zero : Nat}$$

3 Inductive data types

Now, we start to specify type theory in LF. In this section, we briefly demonstrate how to specify inductive data types by inductive schemata. Logic and universes are omitted here.

The details of inductive schemata can be found in [Luo94,Luo04]. When the inductive schemata are given for an inductive data type, we have algorithms to add new constants, eliminator and computation rules. We shall call these algorithms $algC$, $algF$ and $algR$ respectively.

Example 3. Now, we give some examples to show how the algorithms work.

1. The Empty type:

$$sch_1 =_{df} (Empty, <>, <>, <>)$$

   We normally have

$$\begin{array}{l} Empty : Type \\ \mathcal{E}_{Empty} : (C : (Empty)Type)(z : Empty)C(z) \end{array}$$

So,

$$\begin{aligned}algC(sch_1) &= < Empty : Type >\\ algF(sch_1) &= < \mathcal{E}_{Empty} : (C : (Empty)Type)(z : Empty)C(z) >\\ algR(sch_1) &= <>\end{aligned}$$

2. The boolean type:

$$sch_2 = (Bool, <>, <>, < (True, X), (False, X) >)$$

We normally have

$$\begin{aligned}Bool &: Type\\ True &: Bool\\ False &: Bool\end{aligned}$$

$$\begin{aligned}\mathcal{E}_{Bool} : &(C : (Bool)Type)(f_1 : C(True))\\ &(f_2 : C(False))(z : Bool)C(z)\end{aligned}$$

$$\begin{aligned}\mathcal{E}_{Bool}(C, f_1, f_2, True) &= f_1 : C(Ture)\\ \mathcal{E}_{Bool}(C, f_1, f_2, False) &= f_2 : C(False)\end{aligned}$$

So,

$$\begin{aligned}algC(sch_2) &= < Bool : Type,\ True : Bool,\ False : Bool >\\ algF(sch_2) &= < \mathcal{E}_{Bool} : (C : (Bool)Type)(f_1 : C(True))\\ &\qquad (f_2 : C(False))(z : Bool)C(z) >\\ algR(sch_2) &= < ((\Delta)\mathcal{E}_{Bool}(C, f_1, f_2, True) = f_1 : C(True),\\ &\qquad (\Delta)\mathcal{E}_{Bool}(C, f_1, f_2, False) = f_2 : C(False) >\end{aligned}$$

where $\Delta \equiv < C : (Bool)Type, f_1 : C(True), f_2 : C(False) >$

3. The type of natural numbers:

$$sch_3 = (Nat, <>, <>, < (zero, X), (succ, (X)X) >)$$

We have

$$\begin{aligned}&Nat : Type\\ &zero : Nat\\ &succ : (Nat)Nat\end{aligned}$$

$$\begin{aligned}\mathcal{E}_{Nat} : &(C : (Nat)Type)(c : C(zero))\\ &(f : (n : Nat)(C(n))C(S(n)))\\ &(z : Nat)C(z)\end{aligned}$$

$$\begin{aligned}\mathcal{E}_{Nat}(C, c, f, zero) &= c : C(zero)\\ \mathcal{E}_{Nat}(C, c, f, succ(n)) &= f(n, \mathcal{E}_N(C, c, f, n)) : C(succ(n))\end{aligned}$$

The results of $algC(sch_3)$ and $algF(sch_3)$ can be given accordingly.

$$\begin{aligned} algR(sch_3) = <\ & (\Delta)\mathcal{E}_{Nat}(C, c, f, zero) = c : C(zero),\\ & (\Delta, n : Nat)\mathcal{E}_{Nat}(C, c, f, succ(n)) =\\ & f(n, \mathcal{E}_N(C, c, f, n)) : C(succ(n)) > \end{aligned}$$

where

$$\begin{aligned}\Delta \equiv <\ & C : (Nat)Type, c : C(zero),\\ & f : (n : Nat)(C(n))C(S(n)) >\end{aligned}$$

4. The type of lists:

$$sch_4 = (List, < A : Type >, <>, < (nil, X), (cons, (A)(X)X) >)$$

We have

$$\begin{aligned} List &: (A : Type)Type\\ nil &: (A : Type)List(A)\\ cons &: (A : Type)(a : A)(l : List(A))List(A)\end{aligned}$$

$$\begin{aligned}\mathcal{E}_{List} :\ & (A : Type)(C : (List(A))Type)(C(nil(A)))\\ & ((a : A)(l : List(A))(C(l))C(cons(A, a, l)))\\ & (z : List(A))C(z)\end{aligned}$$

$$\begin{aligned}\mathcal{E}_{List}(A, C, c, f, nil(A)) &= c : C(nil(A))\\ \mathcal{E}_{List}(A, C, c, f, cons(A, a, l)) &= f(a, l, \mathcal{E}_{List}(A, C, c, f, l))\\ & \quad : C(cons(A, a, l))\end{aligned}$$

The results of $algC(sch_4)$, $algF(sch_4)$ and $algR(sch_4)$ can be given accordingly.

5. The type of vectors:

$$\begin{aligned} sch_5 = (& Vec, < A : Type >, < n : Nat >,\\ & < (vnil, X(zero)), (vcons, (n : Nat)(A)(X(n))X(succ(n))) >)\end{aligned}$$

We have

$$\begin{aligned} Vec &: (A : Type)(n : Nat)Type\\ vnil &: (A : Type)Vec(A, zero)\\ vcons &: (A : Type)(n : Nat)(a : A)(l : Vec(A, n))Vec(A, succ(n))\end{aligned}$$

$$\begin{aligned}\mathcal{E}_{Vec} : &(A : Type)(C : (n : Nat)(Vec(A, n))Type)\\ &(C(zero, vnil(A)))\\ &((n : Nat)(a : A)(l : Vec(A, n))\\ &(C(n, l))C(succ(n), vcons(A, n, a, l)))\\ &(n : Nat)(l : Vec(A, n))C(n, l)\end{aligned}$$

$$\begin{aligned}\mathcal{E}_{Vec}(A, C, c, f, zero, vnil(A)) &= c : C(zero, vnil(A))\\ \mathcal{E}_{Vec}(A, C, c, f, succ(n), vcons(A, n, a, l)) &= f(n, a, l, \mathcal{E}_{Vec}(A, C, c, f, n, l))\\ &\qquad : C(succ(n), vcons(A, n, a, l))\end{aligned}$$

The results of $algC(sch_5)$, $algF(sch_5)$ and $algR(sch_5)$ can be given accordingly.

In general, the rule for specifying inductive data types is the following.

Rule for inductive data types:

$$\frac{\Gamma \vdash_{R}^{C;F} \text{ valid} \qquad \text{Inductive } sch}{\Gamma \vdash_{R \cup algR(sch)}^{C \cup algC(sch); F \cup algF(sch)} \text{ valid}}$$

Figure 5. Inference rule for inductive data types

Remark 3. We have the following remarks.

- There are obvious restrictions for inductive data types, for example, the names of types and constructors must be all fresh.
- The type theory presented up to now is intuitively equivalent to UTT [Luo94] except the logic and universes, which can be presented here in the same manner. Since the purpose of this paper is about partially defined functions, we omit the details of the logic and universes.
- We assume that the good properties such as strong normalisation and Church-Rosser and subject reduction still hold after introducing inductive data types. In Goguen's thesis [Gog94], he proved such properties for the system UTT, but the author has doubt whether the proof of Lemma 4.9.19 in the thesis is convincing when the computation rules for inductive data types are applied.

Many functions can be defined by the eliminators of inductive data types. For example, one can define addition $(+) : (Nat)(Nat)Nat$ as follows.

$$\begin{aligned}(+) =_{df} &\ \lambda m : Nat.\lambda n : Nat.\\ &\quad \mathcal{E}_{Nat}(\lambda x : Nat.Nat, m, \lambda x : Nat.\lambda y : Nat.succ(y), n)\end{aligned}$$

However, many functions is not easy, if possible, to be defined in this way (see Example 1). In the following two sections, we shall introduce functions in a more flexible and powerful way.

4 Function symbols

As mentioned in the introduction section, functions are defined by existing symbols, mostly eliminators, in conventional type theories. In this section, we introduce new function symbols which will not simply be definitions.

Inhabitable kinds

Definition 2. ***(Inhabitable kinds)*** *If* $\vdash_R^{C;F} K\ \mathrm{kind}$ *is derivable and there is at least one* k *such that* $\vdash_R^{C;F} k : K$ *is derivable, then we say* K *is inhabitable in* $\vdash_R^{C;F}$*. If* $\mathcal{I}$ *is a set of kinds and every kind in the set is inhabitable in* $\vdash_R^{C;F}$*, then we say* $\mathcal{I}$ *is an inhabitable set in* $\vdash_R^{C;F}$*.*

The kind $(A : Type)(A)A$ is inhabitable in any valid context, in fact, we have $\vdash_R^{C;F} \lambda A : Type.\lambda x : A.x : (A : Type)(A)A$. So, an inhabitable set $\mathcal{I}$ exists for any valid context.

Example 4. We give some inhabitable and non-inhabitable kinds.

- $(Empty)Empty$ is inhabitable.
- $(Empty)Nat$ is inhabitable.
- $(Nat)Empty$ is non-inhabitable.
- $(A : Type)(List(A))A$ is problematic. If a function f is of this kind, then there is an element of the type $Empty$ (see Introduction for details).

The purpose of inhabitable sets is to guarantee the consistency of the type theory. We need an algorithm to check the inhabitability of a kind when we introduce a new function symbol. If the algorithm says it is inhabitable, then it is allowed. If it says no or uncertain, the function is not allowed. Although inhabitability is undecidable in general, we can have algorithms to decide some sub-sets of kinds. How big these sub-sets will be often depends on the complexity of the algorithms and user's needs. The author has implemented an algorithm which can decide most commonly used kinds.

The rule for introducing new function symbols is in Figure 6.

5 Pattern matching

In this section, we introduce new equations by means of pattern matching.

Definition 3. ***(Patterns)***

- *A variable is a pattern.*

Rule for function symbols:

$$\frac{\begin{array}{c}\Gamma \vdash_R^{C;F} \text{ valid} \quad \vdash_R^{C;F} (x:K)K' \text{ kind} \quad \text{Function } f:(x:K)K' \\ f \notin FV(\Gamma \cup C \cup F) \qquad (x:K)K' \in \mathcal{I}\end{array}}{\Gamma \vdash_R^{C;F,f:(x:K)K'} \text{ valid}}$$

where $\mathcal{I}$ is an inhabitable set in $\vdash_R^{C;F}$.

Figure 6. Inference rule for function symbols

- *$c(p_1, ...p_n)$ is a pattern if c is a constructor and p_1,..., p_n are patterns and c is fully applied.*

Example 5. $zero$ and $succ(x)$ are patterns but $succ$ is not.

Remark 4. This is a simple definition of patterns. Some definitions include more patterns, for example, 'a closed term is a pattern'. Since the idea of partially defined functions and the implication in practice may be very new for those who study dependent types, we shall concentrate on presenting the main idea through this simple definition.

Notation We shall write p for arbitrary patterns.

Definition 4. *Free variables of a pattern is defined as follows.*

- *$FV(x) = \{x\}$;*
- *$FV(c(p_1, ..., p_n)) = FV(p_1) \cup ... \cup FV(p_n)$.*

Let $< p_1, ..., p_n >$ be a sequence of patterns. If all the free variables only appear once in the sequence, then we say that $< p_1, ..., p_n >$ is a sequence of **linear patterns**. In this paper, we only consider linear patterns. Non-linear patterns are useful but we use linear patterns to demonstrate the main idea.

Definition 5. ***(Disjointness)*** *Let p and q be patterns. If $disjoint(p, q) = true$, then we say p and q are disjoint. Let $< p_1, ..., p_n >$ and $< q_1, ..., q_n >$ be two sequences of patterns. If one pair of p_i and q_i are disjoint, then these two sequences are disjoint.*

$$\begin{aligned} disjoint(x, -) &= false \\ disjoint(c(p_1, ..., p_n), x) &= false \\ disjoint(c(p_1, ..., p_n), c'(q_1, ..., q_n)) &= true \text{ } if \text{ } c \neq c' \\ disjoint(c(p_1, ..., p_n), c(q_1, ..., q_n)) &= disjoint(p_1, q_1) \vee ... \vee \\ &\quad\; disjoint(p_n, q_n) \end{aligned}$$

where $\vee$ is the usual logic operator "or".

Example 6. $succ(x)$ and $zero$ are disjoint but $succ(x)$ and $succ(succ(y))$ are not.

Remark 5. The purpose of disjointness is to prevent us from giving different value to the same arguement. It is one of the ways to guarantee the property of Church-Rosser.

Definition 6. ***(Structural smallness)*** *Let $p \equiv c(r_1, ..., r_n)$ be a pattern and q a term. We say that q is structurally smaller than p if*

- *q is one of r_1,..., r_n; or*
- *q is structurally smaller than one of r_1,..., r_n.*

Example 7. x is structurally smaller than $succ(succ(x))$ but $succ(zero)$ is not structurally smaller than $succ(x)$.

Remark 6. The purpose of structural smallness is to prevent us from defining non-terminating functions. It is one of the ways to guarantee the property of strong normalisation.

It is worth remarking that we sometimes don't directly compare the arguments. For example, in the following equation, the arguement of f in both sides are $c(g)$ and $g(x)$, but we compare $c(g)$ with g, and g is structurally smaller than $c(g)$. One may regard that g is equivalent to $\lambda x : A.g(x)$.

$$f(c(g)) = c'(\lambda x : A.f(g(x)))$$

Structural smallness does not involve any computation, but only syntactical comparison. For example, $(\lambda y : Nat.y)x$ is not structurally smaller than $succ(x)$ although x is.

Definition 7. ***(Absolute smallness)*** *Let $< p_1, ..., p_n >$ be a sequence of patterns and $< q_1, ..., q_n >$ a sequence of terms. If there is at least one pair p_i and q_i such that q_i is structurally smaller than p_i, and for all other pairs such that q_i is structurally smaller than p_i or $q_i \equiv p_i$, then we say that the sequence $< q_1, ..., q_n >$ is* ***absolutely smaller than*** *the sequence $< p_1, ..., p_n >$.*

Example 8. $< x, succ(y) >$ and $< succ(x), y >$ are absolutely smaller than $< succ(x), succ(y) >$.

Remark 7. Although the concept of structural smallness is a sort of common sense, the concept of absolute smallness is not well-understood or clearly defined in literature. For example, the following equation is not allowed in [Coq92], but it is allowed in this paper.

$$f(succ(x), succ(y)) = f(x, succ(y)) + f(succ(x), y)$$

And the following definition makes it precise.

Definition 8. *Let $P \equiv < p_1, ..., p_n >$ be a sequence of patterns and $q \equiv < q_1, ...q_n >$ a sequence of terms, and Q a set of sequences of terms as follows.*

$$Q \equiv \left\{ \begin{array}{c} < q_{11}, ..., q_{1n} > \\ ... \; ... \\ < q_{m1}, ..., q_{mn} > \end{array} \right\}$$

- *Q is structurally smaller than P, notation $Q <^i P$ if there is one i ($i \leq n$) such that q_{1i}, q_{2i},..., q_{mi} are structurally smaller than p_i.*
- *$Q \cup \{q\} <^i P$ if $Q <^i P$ and q is absolutely smaller than P.*

Example 9. We give two examples to help us to understand the above definition. Suppose we have an equation as follows.

$$\begin{aligned} f(succ(x), succ(y), succ(z)) = {} & f(x, succ(succ(y)), z) + \\ & f(succ(succ(x)), y, z) + \\ & f(x, succ(y), succ(z)) + \\ & f(x, y, succ(z)) \end{aligned}$$

There are four recursive calls in the right hand side. The first two have the third term z which is structurally smaller than the third pattern in the left hand side ($succ(z)$). The last two sequences in the right hand side are absolutely smaller than the sequence in the left hand side.

For the last equation of the Ackermann function in Example 1,

$$ack(succ(x), succ(y)) = ack(x, ack(succ(x), y))$$

there are two recursive calls in the right hand side. So, we should compare $< succ(x), succ(y) > \equiv P$ with $< x, ack(succ(x), y) > \equiv q_1$ and $< succ(x), y > \equiv q_2$. The second, q_2, is absolutely smaller than P, and for the first, we have $q_1 <^1 P$. So, we have $\{q_1, q_2\} <^1 P$.

When we introduce a new equation for a function, we should also take all other existing equations for the function into consideration. Firstly, the pattern in the left hand side of the new equation must be disjoint with all the patterns in the left hand side of the existing equations. Secondly, structural smallness has to be considered carefully. We shall not consider the mutually defined functions at the moment in order to keep the presentation simple and concentrate on the main point — partially defined functions.

Definition 9. ***(Legal equation)*** *Let (P_1, Q_1),..., (P_n, Q_n) are the patterns and terms got from the existing equations, where P_1,..., P_n got from the left hand side and Q_1,..., Q_n got from the right hand side. Let (P_{n+1}, Q_{n+1}) are the patterns and terms got from the new equation which we want to introduce. Then, we say the new equation is* ***legal*** *if the following four conditions are satisfied.*

1. *There is no mutual calls for other functions, and the defined function must be always fully applied, and P_{n+1} is a sequence of linear patterns.*
2. *P_1,..., P_n are all disjoint with P_{n+1}.*
3. *There is a i such that $Q_1 <^i P_1$,..., $Q_n <^i P_n$ and $Q_{n+1} <^i P_{n+1}$.*
4. *Not all of the patterns in P_{n+1} are variables.*

Remark 8. Mutually defined functions are not specified here, not because the structural smallness is conceptually different but because a formal definition would be a notational nightmare. For example, when the last equation for *odd* is introduced in Example 1, we start to know that *odd* and *even* are mutually defined. Then whether $odd(succ(x)) = even(x)$ is legal depends on whether the following two equations are legal.

$$odd(succ(zero)) = true$$
$$odd(succ(succ(x))) = odd(x)$$

Let's also explain more about the forth condition. This condition can only be applied to the case that there is no existing equations. Otherwise the second condition would fail. The property of Church-Rosser may fail without the forth condition. For example, if the first equation for the function $f : (Nat)(Nat)Nat$ is $f(x, y) = zero$, which means that, for any natural numbers k_1 and k_2, $f(k_1, k_2) = zero$. For the term $\lambda x : Nat.\lambda y : Nat.f(x, y)$, there are two different normal form $\lambda x : Nat.\lambda y : Nat.zero$ and f.

Now, we are ready to introduce a new rule for equations.

Rule for equations:

$$\frac{\begin{array}{c} \varGamma \vdash_R^{C;F} \text{ valid} \qquad f : (x_1 : K_1)...(x_n : K_n)T \;\in\; F \\ \varDelta \vdash_R^{C;F} f(p_1, ..., p_n) : T' \qquad \varDelta \vdash_R^{C;F} t : T' \\ FV(\varDelta) \equiv FV(p_1) \cup ... \cup FV(p_n) \qquad f(p_1, ..., p_n) = t \text{ is legal} \\ \text{Equation } \; (\varDelta)f(p_1, ..., p_n) = t : T' \end{array}}{\varGamma \vdash^{C;F}_{R,(\varDelta)f(p_1,...,p_n)=t:T'} \text{ valid}}$$

Figure 7. Inference rule for equations

6 Meta-theoretic Properties

As mentioned in the introduction section, when the author designs the system, there are two principles which are always in his mind. So, we also expect the system has the following nice meta-theoretic properties after it is presented.

- Strong normalisation.
- Church-Rosser.
- Decidability of type-checking.
- Consistency, *e.g.* *Empty* type is not inhabitable, $\not\vdash_R^{C;F} M : Empty$ for any C, F, R and M.

Proving these properties is not an easy matter at all. There are a lot of papers and theses about the meta-theoretic properties [Bar92,Luo90,Alt93][MW96,Gog94]

[Geu93,Wer92][Bar84,Tak95,Nip01]. Goguen employed "typed operational semantics" in his thesis [Gog94] to prove the meta-theoretic properties for the system UTT. Since the system here is close to UTT, the techniques for UTT might also be applied here. Studying meta-theoretic properties is a very important part of future work for the system.

Reductions

The one-step reduction rules for β and η are the same with many other systems. We give only two rules for η-reduction and omit others.

$$\frac{}{\lambda x_1 : K_1 ... \lambda x_n : K_n . y x_1 ... x_n \longrightarrow_\eta y}$$

$$\frac{M \longrightarrow_\eta M' \qquad M \text{ is the final body}}{\lambda x_1 : K_1 ... \lambda x_n : K_n . M \longrightarrow_\eta \lambda x_1 : K_1 ... \lambda x_n : K_n . M'}$$

where y is a variable, and M is the final body of λ-abstractions, that is, M itself cannot be of the form $\lambda x : K.N$.

The pattern matching and one-step reduction rules for functions are also the same with our common understanding. We omit the details here.

$\beta\eta$ Weak Head Normal Form and Normal Form

Now, we present a new definition of weak head normal form which is very different from the conventional one in literature. One of the purposes of weak head normal form is to have an efficient algorithm to test conversion of two terms. The definitions here for weak head normal form and normal form are only suitable for logical framework and $\beta\eta$-reduction when we test conversion of two terms. If we add new computation rules, the definitions should be modified accordingly. Details are omitted here.

Definition 10. ***($\beta\eta$ Weak Head Normal Form)*** *A term M is in $\beta\eta$ Weak Head Normal Form* ***(whnf)*** *if and only if*

- *$M \equiv x e_1 ... e_n$ where x is a variable; or*
- *$M \equiv \lambda x_1 : K_1 ... \lambda x_n : K_n . y e_1 ... e_m$ where $n \neq 0$ and y is a variable and,*
  - *$n \neq m$; or*
  - *$n = m$ and $e_1, ..., e_m$ are in whnf and $[e_1, ..., e_m] \not\equiv [x_1, ..., x_n]$.*

Remark 9. The conventional definition of whnf does not take η-reduction into consideration and any λ-abstraction ($\lambda x : K.M$) is in whnf.

Definition 11. ***(Normal Form)*** *A term or kind M is in Normal Form* ***(nf)****, notation $M \in NF$, if and only if*

- *$M \equiv x e_1 ... e_n$ where x is a variable and $e_i \in NF$; or*
- *$M \equiv \lambda x_1 : K_1 ... \lambda x_n : K_n . y e_1 ... e_m$ where $n \neq 0$ and y is a variable and $K_i \in NF$ and $e_j \in NF$, and $[e_1, ..., e_m] \not\equiv [x_1, ..., x_n]$;*

- *or* $M \equiv Type$*; or*
- $M \equiv El(A)$ *where* $A \in NF$*; or*
- $M \equiv (x : K_1)K_2$ *where* $K_i \in NF$*.*

Lemma 1. *If a term or kind M is in normal form, then there is no one-step β or η-reduction can be applied to M and* vice versa*.*

Proof. This property can be verified by induction on the definition of normal form.

Lemma 2. *If the normal form of M ($nf(M)$) is of the form $xe_1...e_n$ then the $\beta\eta$ weak head normal form of M ($whnf(M)$) is of the same form. Similarly, if $nf(M)$ is of the form $\lambda x_1 : K_1 ... \lambda x_n : K_n.ye_1...e_m$ then $whnf(M)$ is of the same form.*

Proof. Suppose $whnf(M)$ is of the form $\lambda x_1 : K_1...\lambda x_n : K_n.ye_1...e_m$. By the definition of whnf, no matter how we reduce this term, the λ-abstraction will never be wiped off.

Conversion testing algorithm The following is a core algorithm of testing conversion of two terms and two kinds. If M and N are convertible, we denote $M \simeq N$, otherwise $M \not\simeq N$.

1. Given two terms M and N. Compute M to a whnf M' and N to a whnf N'.
   (a) $M' \equiv xa_1...a_n$ and $N' \equiv yb_1...b_m$
   if $x \neq y$ then $M \not\simeq N$
   else if $m = n$ and $a_i \simeq b_i$ then $M \simeq N$ else $M \not\simeq N$;
   (b) $M' \equiv \lambda x : K_1.M''$ and $N' \equiv \lambda x : K_2.N''$
   if $K_1 \simeq K_2$ and $M'' \simeq N''$ then $M \simeq N$ else $M \not\simeq N$;
   (Note that if the bound variables are different, they ought to be changed to the same)
   (c) Any other case, $M \not\simeq N$.
2. Given two kinds K_1 and K_2.
   (a) $K_1 \equiv Type$ and $K_2 \equiv Type$
   then $K_1 \simeq K_2$;
   (b) $K_1 \equiv El(A)$ and $K_2 \equiv El(B)$
   if $A \simeq B$ then $K_1 \simeq K_2$ else $K_1 \not\simeq K_2$;
   (c) $K_1 \equiv (x : K_{11})K_{12}$ and $K_2 \equiv (x : K_{21})K_{22}$
   if $K_{11} \simeq K_{21}$ and $K_{12} \simeq K_{22}$ then $K_1 \simeq K_2$ else $K_1 \not\simeq K_2$;
   (Note that if the bound variables are different, they ought to be changed to the same)
   (d) Any other case, $K_1 \not\simeq K_2$.

Remark 10. The conventional definition of whnf is based on β-reduction only. When a system has β and η-reductions, an algorithm of testing conversion of two terms is quite complicate and involves η-expansion, if one takes the conventional definition. The significance of this new definition (*i.e.* $\beta\eta$ weak head normal form) is that an algorithm of testing conversion of two terms can be very straightforward and easier. As for the complexity, the author believes that the algorithm based on the new definition is not slower than the one based on the old if not faster. This topic is out of the scope of the paper.

Theorem 1. ***(Soundness and Completeness)*** *The above algorithm is sound and complete if the logical framework has the properties of strong normalisation and Church-Rosser.*

Proof. Since the logical framework has the properties of strong normalisation and Church-Rosser, if $\Gamma \vdash M = N : K$ or $\Gamma \vdash M = N$ where M and N are kinds, is derivable, then M and N have the same normal form. The proof proceeds by Lemma 2 and by analysing the cases of normal forms.

7 Discussion and conclusion

7.1 More patterns

As mentioned before, we employed a simple version of patterns to convey the main idea of the paper, partially defined functions in a type theory and the implication in practice. In the following, we give examples to illustrate some more flexible ways of introducing functions. Many questions may arise for the examples in this section. A more detailed and formal presentation will be in a forthcoming paper.

For example, we want to introduce the following function.

$$f(x) = \begin{cases} 1 & \text{if } x = 100 \\ 0 & \text{otherwise} \end{cases}$$

It is very tedious indeed to introduce this function in type theories, since we have to list 102 different cases. However, we can introduce the function as follows.

$$\begin{aligned} f(100) &= succ(zero) \\ f(x) &= zero \end{aligned}$$

where $100 \equiv succ(...succ(zero)...)$ and there are 100 *succ*. Although the patterns 100 and x are not disjoint, the last equation is still allowed. But it cannot stand alone, and it must be bound with other equations of the function. The pattern matching algorithm now not only give "yes" or "no", but has three answers: "matching", "uncertain" and "impossible". For example, if the algorithm is $pma(t, p)$ where t is a term and p is a pattern, then

$$\begin{aligned} pma(100, 100) &= matching \\ pma(succ(x), zero) &= impossible \\ pma(x, 100) &= uncertain \\ pma(succ(x), 100) &= uncertain \\ pma(zero, 100) &= impossible \end{aligned}$$

So, we can compute $f(100)$ to $succ(zero)$ because of $pma(100, 100) = matching$, $f(zero)$ to $zero$ because of $pma(zero, 100) = impossible$, and $f(succ(x))$ cannot be computed further because of $pma(succ(x), 100) = uncertain$.

In many programming languages such as Haskell [Tho99], the requirement of disjointness is not needed because the patterns for a function have orders. If a closed term matches the first pattern, the algorithm will not try to match other patterns. This makes sense because Haskell only computes closed terms.

Non-linear patterns Since the properties of strong normalisation and Church-Rosser are taken as basic requirements, we may also allow the following examples.

$$f : Nat \to Nat \to Bool$$

$$f(x, x) = True$$

$$h : (Nat \to Nat) \to (Nat \to Nat) \to Bool$$

$$h(g, g) = True$$

The way of computing for this sort of functions is that, for example, for the term $h(g_1, g_2)$, if g_1 is computationally equal to g_2, then it is computed to $True$, otherwise, it cannot be computed further.

Another example often found in mathematics and physics has non-linear pattern.

$$\overrightarrow{\oplus} : (n : Nat)(Vec(Nat, n))(Vec(Nat, n))Vec(Nat, n)$$

$$\overrightarrow{\oplus}(zero, vnil(Nat), vnil(Nat)) = vnil(Nat)$$

$$\begin{gathered}\overrightarrow{\oplus}(succ(m), vcons(Nat, m, x, xs), vcons(Nat, m, y, ys)) \\ = vcons(Nat, m, x + y, \overrightarrow{\oplus}(m, xs, ys))\end{gathered}$$

Closed terms can also be patterns For example, $\lambda x : Nat.x$ can be a pattern because it is a closed term. Let's examine the following example.

$$f : (Nat \to Nat) \to Bool$$

$$f(\lambda x : Nat.x) = True$$

For the term $f(g)$, if g is computationally equal to $\lambda x : Nat.x$, then it is computed to $True$, otherwise it cannot be computed further.

7.2 A comparison

Here is a brief comparison with Coquand's paper on his home page.

- This paper allows partially defined functions, while his paper doesn't. As mentioned before, totality checking is undecidable in general and this is a very big problem.

- This paper treat pattern matching as an extra power to define functions. Many other aspects such as logic, universes, equality rules, inductive data types are introduced in the same way as in normal type theories.
- In Coquand's paper, $g(t)$ is structurally smaller than $c(g)$ for any term t. This is neither natural nor necessary. In this paper, g is structurally smaller than $c(g)$ as discussed in Remark 6.
- In this paper, the concept of absolute smallness is presented in Definition 7. More functions can be introduced as remarked in Remark 7.

7.3 Conclusion

The purpose of the paper is to present a type theory which can introduce more functions by allowing partially defined functions. This paper crosses the limitation of the elimination rules, which only are primitive recursions. By allowing partially defined functions, we avoid an undecidable problem in dependent type theories, totality checking of patterns. The implication of the idea in practice is significant, for example, a non-primitive function, Ackermann function, can be defined easily. This paper also challenges a conventional philosophical understanding, functions must be totally defined in type theories. And the author believes this paper will make a big impact in the community of theoreticians and philosophers of type theories.

References

[Alt93] Th. Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, Edinburgh University, 1993.

[B$^+$00] B. Barras et al. *The Coq Proof Assistant Reference Manual (Version 6.3.1).* INRIA-Rocquencourt, 2000.

[Bar84] H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North-Holland, revised edition, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, 1992.

[C$^+$86] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Pretice-Hall, 1986.

[Coq92] Th. Coquand. Pattern matching with dependent types. Talk given at the BRA workshop on Proofs and Types, Bastad, 1992.

[Geu93] Herman Geuvers. *Logics and Type Systems.* PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[Gog94] H. Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, 1994.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Proc. 2nd Ann. Symp. on Logic in Computer Science. IEEE*, 1987.

[HHP92] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of ACM*, 40(1):143–184, 1992.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[Luo90] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990. Also as Report CST-65-90/ECS-LFCS-90-118, Department of Computer Science, University of Edinburgh.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[Luo03] Z. Luo. PAL$^+$: a lambda-free logical framework. *Journal of Functional Programming*, 13(2):317–338, 2003.

[Luo04] Y. Luo. *Coherence and Transitivity in Coercive Subtyping.* PhD thesis, Department of Computer Science, University of Durham, 2004.

[Luo05a] Y. Luo. New eta-reduction and Church-Rosser. *http://www.cs.kent.ac.uk/people/staff/yl41/*, 2005.

[Luo05b] Y. Luo. Yet Another Normalisation Proof for Martin-Löf's Logical Framework—Terms with correct arities are strongly normalising. *http://www.cs.kent.ac.uk/people/staff/yl41/*, 2005.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[MM04] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT, 1990.

[MW96] P. Mellies and B. Werner. A generic normalisation proof for pure type systems, 1996.

[Nip01] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.

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

[Tak95] Masako Takahashi. Parallel redeuctions in λ-calculus. *Journal of Information and Computation*, 118:120–127, 1995.

[Tho99] Simon Thompson. *Haskell : the craft of functional programming.* Harlow : Addison Wesley, 2nd edition, 1999.

[Wer92] B. Werner. A normalization proof for an impredicative type system with large eliminations over integers. In *Workshop on Logical Frameworks*, 1992.