



Formalizing Type Operations Using the “Image” Type Constructor

Aleksey Nogin and Alexei Kopylov

*Computer Science Department
California Institute of Technology
Pasadena, CA 91125
Email: {nogin, kopylov}@cs.caltech.edu*

Abstract

In this paper we introduce a new approach to formalizing certain type operations in type theory. Traditionally, many type constructors in type theory are independently axiomatized and the correctness of these axioms is argued semantically. In this paper we introduce a notion of an “image” of a given type under a mapping that captures the spirit of many of such semantical arguments. This allows us to use the new “image” type to formalize within the type theory a large range of type constructors that were traditionally formalized via postulated axioms.

We demonstrate the ability of the “image” constructor to express “forgetful” types by using it to formalize the “squash” and “set” type constructors. We also demonstrate its ability to handle types with non-trivial equality relations by using it to formalize the union type operator. We demonstrate the ability of the “image” constructor to express certain inductive types by showing how the type of lists and a higher-order abstract syntax type can be naturally formalized using the new type constructor.

The work presented in this paper have been implemented in the [MetaPRL](#) proof assistant and all the derivations checked by [MetaPRL](#).

Keywords: Type theory, type constructor, [NuPRL](#), [MetaPRL](#), [Coq](#), Calculus of constructions

1 Introduction

1.1 Type Theory

The work presented here focuses on the [NuPRL](#) type theory [7] and its variations, including the [MetaPRL](#) implementation [12] of the [NuPRL](#) type theory. [NuPRL](#) type theory in an extension of the Martin-Löf’s type theory [15, 16], which differs from many other type theories in its treatment of equality. In [Coq](#)’s Calculus of Constructions, for example, there is a single global equality relation which is not the desired one for many types (e.g. function types). The desired equalities have to be handled explicitly, which can be quite burdensome. In Martin-Löf type theory each type comes with its own equality relation (the extensional one in the case of functions), and the typing rules guarantee that well-typed terms respect these equalities.

Martin-Löf’s type theory has the following judgments:

A Type A is a well-formed type

$A = B$ A and B are (intentionally) equal types (presupposes A Type and B Type)

$a \in A$ a has type A (presupposes A Type)

$a = b \in A$ a and b are equal as elements of type A (presupposes $a \in A$ and $b \in B$).

Martin-Löf’s type theory allows dependent types. This means that type expression may contain free variables ranging over arbitrary types. For example, we can form a *family* of types $T[x] = [0..x]$ which represents an initial sequent of natural numbers. This family is well-formed when x ranges over \mathbb{N} .

Martin-Löf’s type theory includes type constructors such as disjoint unions $A+B$ (that is inhabited by elements of the form $\text{inl}(a)$ for $a \in A$ as well as $\text{inr}(b)$ for $b \in B$) and dependent products $x : A \times P[x]$ (that contains pairs $\langle a, p \rangle$, where $a \in A$ and $p \in P[a]$).

The **NuPRL** type theory also has subtyping relation. Although it is not essential for our work, we should mention that membership and subtyping in **NuPRL** are extensional. For example, $A \subseteq B$ does not say anything about structure of these types, but only means that $t_1 = t_2 \in A$ implies $t_1 = t_2 \in B$. As a result the type checking and subtyping are undecidable. On the other hand, type equality ($A = B$) is intensional.

1.2 Notation

We will write $f[x_1, \dots, x_n]$ for a term that may contain free occurrences of variables x_1, \dots, x_n , and $f[t_1, \dots, t_n]$ for the substitution of terms t_i ’s for *all* of the free occurrences of x_i ’s in $f[x_1, \dots, x_n]$.

We will be presenting the axioms and rules of type theory in the form of Gentzen-style single-conclusion calculus. When referring to sequents, it is often more convenient to reason about the sequences of hypotheses as a whole instead of concentration on individual hypotheses. We will write $\gamma : \Gamma \vdash C[\gamma]$ for a sequent $x_1 : A_1; x_2 : A_2[x_1]; \dots; x_n : A_n[x_1; \dots; x_{n-1}] \vdash C[x_1; \dots; x_n]$. Then if γ is a list of terms $t_1; \dots; t_n$ then we will use $C[\gamma]$ for $C[t_1; \dots; t_n]$. We will also write “ $\forall \gamma \in \Gamma$ ” as an abbreviation for “for all list of terms $\gamma = t_1; \dots; t_n$, s.t. for all $i = 1..n$, $t_i \in A_i[t_1; \dots; t_{i-1}]$ ”.

We will also write “ $u = v \in A = B$ ” as an abbreviation for “ $A = B$ and $u = v \in A$ ” and “ $\forall \gamma = \gamma' \in \Gamma = \Gamma'$ ” as an abbreviation for “for all list of terms $\gamma = t_1 \dots; t_n$ and for all list of terms $\gamma' = t'_1 \dots; t'_n$ s.t. for all $i = 1..n$, $t_i = t'_i \in A_i[t_1; \dots; t_{i-1}] = A'_i[t'_1; \dots; t'_{i-1}]$ ”.

1.3 PER Models of Type Theory

The most commonly used semantics of the NuPRL type theory (and some other type theories as well) is the PER (partial equivalence relations) semantics [21, 2, 1]. In PER semantics each type is identified with a set of objects (often — a set of closed terms) together with an equivalence relation on that set that serves as an *equality relation* for objects (closed terms) of that type. This causes the equality predicate to be three-place: “ $a = b \in C$ ” stands for “ a and b are equal elements of type C ”, or, semantically, “ a and b are related by the equality relation of type C ”.

Remark 1.1 *Note that in this approach an objects is an element of a type iff it is equal to itself in that type. This allows us to identify $a \in A$ with $a = a \in A$.*

According to PER approach, whenever something ranges over a certain type, it not only has to span the whole type, it also has to respect the equality of that type.

Example 1.2 *In order for a function f to be considered a function from type A to type B , not only for every $a \in A$, $f(a)$ has to be in B , but also whenever a and a' are equal in the type A , $f(a)$ should be equal to $f(a')$ in the type B . Note that in this example the second condition is sufficient since it actually implies the first one. However it is often useful to consider the first condition separately.*

Since the type theory contains dependent types, the notion of a “*well-formed family of types*” plays an important role in defining well-formedness of dependent types. In PER approach, in order for the family of types $T[x]$ over type A to be considered well-formed, not only for every $a \in A$, $T[a]$ has to be a well-formed type, but also for every $a_1 = a_2 \in A$, the types $T[a_1]$ and $T[a_2]$ have to be equal.

Example 1.3 *Consider a set type $T := \{x : A \mid B[x]\}$ (cf. Section 3.2) or a dependent product type $T := x : A \times B[x]$. Similarly to Example 1.2 above, in order for T to be a well-formed type, not only $B[a]$ has to be a well-formed type for any $a \in A$, but also for any $a = a' \in A$ it should be the case that $B[a]$ and $B[a']$ are equal types.*

The meaning of sequent judgments is defined in the same fashion as well. In order for a sequent $\gamma : \Gamma \vdash C[\gamma]$ to be considered valid, not only it has to be the case that $\forall \gamma \in \Gamma. C[\gamma]$, but also t and C has to respect all equalities that are respected by Γ . That is, $\forall \gamma_1 = \gamma_2 \in \Gamma[\gamma_1] = \Gamma[\gamma_2]. C[\gamma_1] = C[\gamma_2]$.

Example 1.4 *Provided A is a well-formed type, the sequents “ $\vdash \lambda x.t \in A \rightarrow B$ ” and “ $x : A \vdash t \in B$ ” are equivalent. In other words, the definition of the meaning of sequents corresponds exactly to the other functionality requirements, such as the one for functions that we gave in Example 1.2.*

1.4 Propositions-as-Types and Computational Type Theory

Martin-Löf’s type theory adheres to the *propositions-as-types principle*. This principle means that a proposition is identified with the type of all its witnesses. A proposition is considered true if the corresponding type is inhabited and is considered false otherwise. Similarly a sequent $\Gamma \vdash C$ is considered valid iff $\Gamma \vdash t \in C$

is valid for some t .

In this paper we will use words “*type*” and “*proposition*” interchangeably; same with “*witness*” and “*member*”. However it is important to note that the notion of “*witness*” is *not* the same as the notion of “*proof*”. For example, the membership type $t \in T$ contains a single canonical element \bullet ¹.

Because the NuPRL type theory is often interpreted computationally, the members of a type are also sometimes called the “*computational content*” of that type. In fact, NuPRL type theory is sometimes called a *computational type theory* and the notion of *computation* is a very basic one in it. In addition to the four judgments of the Martin-Löf theory listed in Section 1.1, the NuPRL type theory also includes the following one:

$a \sim b$ a and b are *computationally equal*

This computational equality [14] is a transitive symmetric closure of a relation that includes both purely computational relations (such as beta reduction) and definitional equalities. In NuPRL type theory a term may be replaced with a computationally equal one in *any context* [14].

1.5 Motivation and Overview

At first glance, it might seem that the image type constructor would be trivial to define — at least semantically. Given a type A and a function $\lambda x.f[x]$, the type $\text{Img}\{A; x.f[x]\}$ would be a type containing all the elements $f[a]$, for $a \in A$. However, this naïve definition fails to account for equalities. In order to make it work, one would have to specify not only the members of the type $\text{Img}\{A; x.f[x]\}$, but also its equality relation.

Another obstacle is that one would also have to specify when two $\text{Img}\{\}$ types are equal. Normally two types are considered equal if they are constructed from the equal parts. For example, two product types $A_1 \times B_1$ and $A_2 \times B_2$ are considered equal when $A_1 = A_2$ and $B_1 = B_2$. However in case of the image type we can not follow the same model and state that $\text{Img}\{A_1; x.f_1[x]\} = \text{Img}\{A_2; x.f_2[x]\}$ when $A_1 = A_2$ and for all $x \in A$ $f_1[x] = f_2[x] \in B$, since that would require knowing the image B of functions f_i , which is what we were trying to define in the first place.

As we will see in this paper, these problems can be solved. And once these problems are solved and the “Image” type is defined, it turns out this constructor allows capturing an essential feature of the PER semantics for the type theory. It is well known [3, 4, 5] that there is a common theme to the standard type constructors and their axiomatization in type theory. When a new type constructor is added to the theory, there is usually a standard pattern to defining its semantics, formulating axioms and arguing (semantically) that the new axioms are consistent with the canonical PER model of the theory. As it turns out, the “Image” type constructor allows capturing this pattern within the theory itself. And now many standard type constructors can be defined using the “Image” constructor and the corresponding

¹ This canonical element is sometimes denoted as the unit element $()$, sometimes called “it” (MetaPRL), ax (NuPRL), or Triv.

rules can now be formally derived instead of having to add them as axioms and justify them semantically.

This distinction between the in-theory derivation and a semantical argument is especially important in context of an automated theorem proving environment, where the in-theory derivation are formal and can be automatically checked, while the semantical arguments are informal, written on paper, and can only be checked manually. The work presented in this paper has been implemented in the **MetaPRL** proof assistant [10, 11, 13] and all the derivations were checked by **MetaPRL**.

This paper is structured as follows. In Section 2 we will introduce the “Image” type constructor, define its meaning in the PER semantics, present the inference rules and establish their validity in PER semantics. Next, in Section 3 we will show how some very basic type operations that are traditionally formalized in **NuPRL**-style type theories by postulated axioms can now be derived using the new type constructor. Finally, in Section 4 we show how the new type constructor can be used to formalized some inductive types, including one that have been previously considered impossible to adequately formalize.

2 The Image Type

We will introduce a new type constructor $Img\{A; x.f[x]\}$. As expected, the elements of the type $Img\{A; x.f[x]\}$ are all expressions of the form $f[a]$, where $a \in A$.

Example 2.1 *We can define a singleton type as $\{a\} := Img\{\mathbf{Unit}; x.a\}$ (where any non-empty type may be used instead of **Unit**). This type contains only one element a .*

In PER approach, when defining semantics for a new type constructor we are required to define its equality relation. It is clear that whenever $a_1 = a_2 \in A$, it ought to be the case that $f[a_1] = f[a_2] \in Img\{A; x.f[x]\}$. Therefore we will define the equality on the image type as the transitive closure of the equality induced by the equality relation on A (together with the computational equality \sim that all the equality relations in **NuPRL**’s PER models are required to respect).

As we have briefly mentioned in Section 1.5, defining the equality on $Img\{\}$ types can be tricky. Consider, for example, the singleton type constructor from Example 2.1 above. It is easy to see that type families involving this constructor will not always be well-formed. Consider a type that has two distinct elements equal in it. For example, the type $\mathbb{B} // \mathbf{True}$ ² that has $\mathbf{tt} = \mathbf{ff} \in \mathbb{B} // \mathbf{True}$. It is clear that $\{a\}$ is not a well-formed type family over $a \in \mathbb{B} // \mathbf{True}$. Indeed, $\{\mathbf{tt}\}$ and $\{\mathbf{ff}\}$ are clearly different types, so $\{a\}$ does not respect the equality $\mathbf{tt} = \mathbf{ff} \in \mathbb{B} // \mathbf{True}$. This demonstrates that the following naïve well-formedness rule

$$\frac{\Gamma \vdash A \text{Type}}{\Gamma \vdash Img\{A; x.f[x]\} \text{Type}}$$

² \mathbb{B} is a type of booleans that contains only two elements: \mathbf{tt} and \mathbf{ff} . $\mathbb{B} // \mathbf{True}$ is the \mathbb{B} type quotiented over a constant relations so that all the elements of the quotient become equal.

is invalid. Instead, we will state that $Img\{A; x.f[x]\}$ is well-formed when f does not have any parameters, i.e., $f[x]$ has only x as a free variable.

2.1 Inference Rules

We add the following axioms to our type theory:

$$\begin{array}{l} \textbf{Well-formedness rule:} \quad \frac{\Gamma \vdash A \text{ Type}}{\Gamma \vdash Img\{A; x.f\langle \rangle[x]\} \text{ Type}} \\ \textbf{Introduction rule:} \quad \frac{\Gamma \vdash a \in A}{\Gamma \vdash f[a] \in Img\{A; x.f\langle \rangle[x]\}} \\ \textbf{Elimination rule:} \quad \frac{\Gamma; x : A \vdash t \in T[f[x]]}{\Gamma; y : Img\{A; x.f\langle \rangle[x]\} \vdash t \in T[y]} \end{array}$$

where $f\langle \rangle[x]$ is a *sequent schema* [18] notation prohibiting f from containing free occurrences of variables other than x .

Remark 2.2 Note that the elimination rule requires the conclusion to be an equality. This has to do with the computational nature of the theory. As we have mentioned in Section 1.4, $\Gamma; x : A \vdash C[f[x]]$ is valid only when $\Gamma; x : A \vdash t[x] \in C[f[x]]$ for some t . And since in general we can not compute x from y such that $y = f[x]$, in general there is no way to construct a witness t' such that $\Gamma; y : Img\{A; x.f\langle \rangle[x]\}; \Delta[y] \vdash t'[y] \in C[y]$. Therefore validity of $\Gamma; x : A \vdash C[f[x]]$ would not always imply validity of $\Gamma; y : Img\{A; x.f\langle \rangle[x]\}; \Delta[y] \vdash t'[y] \in C[y]$. On the other hand, as we have stated in Section 1.4, the equalities in our type theory always have a canonical witness, so this problem does not occur when C is an equality.

Also note that while the elimination rule is stated with t not depending on x or y , this is not essential — a more general form of this rule can be derived from the simpler one.

2.2 Formal Semantics

According to [2], to give the semantics for a type expression E we need to determine when this expression is a well-formed type, describe when two such types are equal, and specify the partial equivalence relation that would act as an equality relations for this type ($a = b \in E$).

We define the semantics of the $Img\{\}$ constructor as follows:

- The closed term $Img\{A; x.f[x]\}$ is a well-formed type *if and only if* A is a type.
- $Img\{A_1; x.f_1[x]\} = Img\{A_2; x.f_2[x]\}$ *iff* $A_1 = A_2$ and $\lambda x.f_1[x] \sim \lambda x.f_2[x]$.
- The equality relation on $Img\{A; x.f[x]\}$ is the smallest PER such that $f[a] = f[b] \in Img\{A; x.f\langle \rangle[x]\}$ whenever $a = b \in A$ that respects the \sim relation. (In particular, this means that $t \in Img\{A; x.f\langle \rangle[x]\}$ *iff* $t \sim f[a]$ for some $a \in A$.)

Theorem 2.3 The rules of Section 2.1 are valid under this semantics.

Proof.

Well-formedness rule. Suppose $\gamma : \Gamma \vdash A[\gamma]$ Type is a true judgment. That is,

$$\forall \gamma_1 = \gamma_2 \in \Gamma. A[\gamma_1] = A[\gamma_2]$$

Since $f\langle \rangle[x]$ is a closed term that can not depend on γ , we immediately get that $Img\{A[\gamma_1]; x.f[x]\} = Img\{A[\gamma_2]; x.f[x]\}$ and therefore

$$\forall \gamma_1 = \gamma_2 \in \Gamma. Img\{A[\gamma_1]; x.f[x]\} = Img\{A[\gamma_2]; x.f[x]\}$$

This means that $\gamma : \Gamma \vdash Img\{A[\gamma]; x.f[x]\}$ Type is a true judgment.

Introduction rule. Suppose $\gamma : \Gamma \vdash a[\gamma] \in A[\gamma]$ is a true judgment. That is,

$$\forall \gamma_1 = \gamma_2 \in \Gamma. a[\gamma_1] = a[\gamma_2] \in A[\gamma_1] = A[\gamma_2]$$

Then by the definition of the image type,

$$\forall \gamma_1 = \gamma_2 \in \Gamma. f[a[\gamma_1]] = f[a[\gamma_2]] \in Img\{A[\gamma_1]; x.f[x]\} = Img\{A[\gamma_2]; x.f[x]\}$$

Therefore, $\gamma : \Gamma \vdash f[a[\gamma]] \in Img\{A[\gamma]; x.f[x]\}$ is a true judgment.

Elimination rule. Suppose

$$\gamma : \Gamma; x : A[\gamma] \vdash t[\gamma] \in T[\gamma; f[x]] \quad (2.I)$$

is a true judgment. We need to show that $\gamma : \Gamma; y : Img\{A[\gamma]; x.f[x]\} \vdash t[\gamma] \in T[\gamma; y]$ holds. That is, assuming

$$\gamma_1 = \gamma_2 \in \Gamma \quad \text{and} \quad y_1 = y_2 \in Img\{A[\gamma_1]; x.f[x]\} = Img\{A[\gamma_2]; x.f[x]\},$$

we need to show the witness of $t[\gamma_1] \in T[\gamma_1; y_1]$ and prove that

$$(t[\gamma_1] \in T[\gamma_1; y_1]) = (t[\gamma_2] \in T[\gamma_2; y_2]). \quad (2.II)$$

First, if $y_1 \in Img\{A[\gamma_1]; x.f[x]\}$ then there is $x_1 \in A$, such that $y_1 \sim f[x_1]$. Then because $\gamma_1 = \gamma_2 \in \Gamma$ and $x_1 = x_1 \in A[\gamma_1]$, we can use (2.I) to conclude that $t[\gamma_1] \in T[\gamma_1; f[x_1]]$ and $(t[\gamma_1] \in T[\gamma_1; f[x_1]]) = (t[\gamma_2] \in T[\gamma_2; f[x_1]])$. Since all judgements are congruent w.r.t. \sim relation, we can say that $t[\gamma_1] \in T[\gamma_1; y_1]$ and

$$(t[\gamma_1] \in T[\gamma_1; y_1]) = (t[\gamma_2] \in T[\gamma_2; y_1]). \quad (2.III)$$

By the definition, the true membership also has a witness \bullet . So we have the witness of $t[\gamma_1] \in T[\gamma_1; y_1]$.

Let R be the relation on $Img\{A[\gamma]; x.f[x]\}$ induced by $A[\gamma_2]$. That is $y_1 R y_2$ iff there are x_1 and x_2 such that $x_1 = x_2 \in A[\gamma_2]$ and $y_1 \sim f[x_1]$ and $y_2 \sim f[x_2]$. We are going to show that then

$$(t[\gamma_2] \in T[\gamma_2; y_1]) = (t[\gamma_2] \in T[\gamma_2; y_2]). \quad (2.IV)$$

Because $\gamma_2 = \gamma_2 \in \Gamma$ and $x_1 = x_2 \in A[\gamma_1] = A[\gamma_2]$, we can use (2.I) to conclude that $(t[\gamma_2] \in T[\gamma_2; f[x_1]]) = (t[\gamma_2] \in T[\gamma_1; f[x_2]])$. Then (2.IV) holds.

Since equality on $Img\{A[\gamma_2]; x.f[x]\}$ is the transitive closure of R , (2.IV) holds for any $y_1 = y_2 \in Img\{A[\gamma_2]; x.f[x]\}$. Now (2.II) follows from (2.III) and (2.IV). \square

3 Deriving Simple Type Constructors

It turns out that some very basic type operations that are traditionally formalized in NuPRL-style type theories by postulated axioms can now be derived using the new type constructor.

3.1 Deriving the Squash Type Constructor

Our type theory used to contain a primitive type constructor called squash [7, 17]. The squash type $[A]$ “forgets” the witnesses of A . For any type A , the type $[A]$ (“squashed A ” or “hidden A ”) is empty *if and only if* A is empty and contains a single canonical element \bullet when A is inhabited. Informally one can think of $[A]$ as a proposition that says that A is a *non-empty type*, but “squashes down to a point” all the information on *why* A is non-empty.

Using the Image type we can now define squash type as simple as

$$[A] := Img\{A; x.\bullet\}$$

Using this definition, all the rules that are normally postulated for this type [17] can be derived. As all the other derivations in this paper, these derivations were performed in and checked by the MetaPRL proof assistant.

3.2 Deriving the Set Type Constructor

The set type type $\{x : A|P[x]\}$ is a type that contains all elements of the type A for which the condition $P[x]$ holds, hiding the information on *why* it holds.

Using the image type constructor, we can define the set type operator as

$$\{x : A|P[x]\} := Img\{x : A \times P[x]; x.\pi_1(x)\}$$

where π_1 is the first element projection. From this definition we were able to derive all the rules that are traditionally postulated with this type constructor.

3.3 Deriving the Union Type Constructor

The union type [20], denoted $\bigcup_{x:A} B[x]$, is the least common supertype of $B[x]$ ’s for $x \in A$. It contains all elements of types $B[x]$. The equivalence relation of the $\bigcup_{x:A} B[x]$ is the *transitive closure* of the union of the equivalence relations of types $B[x]$. The inference rules for the union type are presented in Table 1.

$$\frac{\Gamma \vdash A \text{Type} \quad \Gamma; x : A \vdash B[x] \text{Type}}{\Gamma \vdash \left(\bigcup_{x:A} B[x]\right) \text{Type}}$$

$$\frac{\Gamma \vdash a \in A \quad \Gamma \vdash b \in B[a] \quad \Gamma; x : A \vdash B[x] \text{Type}}{\Gamma \vdash b \in \bigcup_{x:A} B[x]} \quad \frac{\Gamma; x : A; y : B[x] \vdash [C[y]]}{\Gamma; y : \bigcup_{x:A} B[x] \vdash [C[y]]}$$

Table 1
Inference rules for the union type

$$\frac{\Gamma \vdash a \sim b \quad \Gamma; \Delta[a] \vdash C[a]}{\Gamma \vdash b \in \{a \langle \rangle\}} \quad \frac{\Gamma; \Delta[a] \vdash C[a]}{\Gamma; x : \{a \langle \rangle\}; \Delta[x] \vdash C[x]}$$

Table 2
Inference rules for the singleton type

We can define the union type constructor as follows:

$$\bigcup_{x:A} B[x] := \text{Img}\{x : A \times B[x]; x.\pi_2(x)\}$$

where π_2 is the second element projection. All the rules in Table 1 can be derived from this definition.

3.4 Deriving the Singleton Type Constructor

It is possible in the type theory without the Image type to define a singleton subtype of a given type. That is, for a given type A and given element a of type A we can define a subtype of A that contains only a :

$$\{a\}_A := \{x : A \mid x = a \in A\}.$$

This type actually contains not only a but other elements that are not distinguishable from a by A . To define singleton type of the element a we need to provide a “host” type A that contains a .

However, as we saw in Example 2.1, using Image type we can define singleton type $\{a\}$ without knowing the type where a comes from:

$$\{a\} := \text{Img}\{\text{Unit}; x.a\}$$

Table 2 shows the inference rule for this type derived in [MetaPRL](#). Note that the type $\{a\}$ contains only one element a with respect to squiggle equality \sim . Thus we can derive a more powerful elimination rule for this type than for $\{a\}_A$. As we mentioned earlier this type has a restriction that a should be a constant. (Remember that $a \langle \rangle$ means that a could not contain free variables).

4 Implementing Inductive Types

It turns out that the image type constructor allows formalizing some interesting inductive types. We will start with a simple example of the type of finite lists. This type constructor is defined in Martin-Löf’s type theory using W -type [16]. We will see the list type can also be implemented using just disjoint unions, products, natural numbers and unions. Here the “contribution” of the image type constructor is limited to allowing us to derive the union type constructor instead of having to use the postulated one, however it introduces an approach that we will use to implement a higher-order abstract syntax type that was previously considered impossible to adequately formalize in a NuPRL-style type theory (and a number of other type theories as well). The latter implementation is a much more involved one, and we will only give a brief overview. See [19] for a detailed description and extensive discussion of using the image type constructor for reasoning about syntax.

4.1 Example: The List Type

The A List type is the type of all the finite lists of the form $a_1 :: a_2 :: \dots a_n :: []$ where $a_i \in A$, $::$ is a binary **cons** operator that concatenates an item to a list, and $[]$ is the empty list (“**nil**”) that is common to all the list types.

Using the disjoint union type (*cf.* Section 1.1), we can recursively define the types of lists of concrete length as follows:

$$\begin{aligned} A\text{List}_0 &:= \text{Void} + \text{Unit} \\ A\text{List}_{i+1} &:= (A \times A\text{List}_i) + \text{Void} \end{aligned}$$

where Void is the empty type, $[]$ is implemented as $\text{inr}(\bullet)$ and $h :: t$ is implemented as $\text{inl}(\langle h, t \rangle)$.

Now the type of arbitrary finite lists can be defined by simply taking the union of all the concrete-length list types:

$$A\text{List} := \bigcup_{i:\mathbb{N}} A\text{List}_i .$$

All the standard rules for the list type [12] can be derived from this definition.

4.2 Example: Higher Order Abstract Syntax

Suppose we want to define a type of λ -expressions where variables are handled abstractly and in an alpha-invariant way. Namely, for some type var , we want to define **Term** recursively as

$$\begin{aligned} \text{Term} &:= \text{Var of var} \\ &\quad \text{Apply of Term} \times \text{Term} \\ &\quad \text{Lambda of Term} \rightarrow \text{Term} \end{aligned}$$

where the function space in the `Lambda` case is restricted to only *substitution functions* — *i.e.* functions that treat their argument as a “black box”, only applying `Term` constructors to it and never applying destructors. The traditional approaches to formalizing such Higher-Order Abstract Syntax types would either result in types where some “exotic” terms are left in the type (for example, [8] defines an approach where most exotic terms are eliminated, but those that are extensionally equal to the real ones remain) or would require special modalities that would allow expressing restricted function spaces [9].

With the image type constructor, however, it is possible to define the `Term` type so that there are no “exotic” terms in it. It turns out that if one formalizes the deBruijn representation of the λ -expressions and the easily definable function that transforms from deBruijn representation into the `Term` one, then the image of that type under the translation function is exactly the `Term` type, *sans* exotic terms.

References

- [1] Allen, S. F., *A Non-type-theoretic Definition of Martin-Löf’s Types*, in: D. Gries, editor, *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science* (1987), pp. 215–224.
- [2] Allen, S. F., “A Non-Type-Theoretic Semantics for Type-Theoretic Language,” Ph.D. thesis, Cornell University (1987).
- [3] Backhouse, R., *On the meaning and construction of the rules in Martin-Löf’s theory of types*, in: A. Avron, editor, *Workshop on General Logic, Edinburgh, February 1987*, number 88-52 in ECS-LFCS (1988).
- [4] Backhouse, R. C., P. Chisholm and G. Malcolm, *Do-it-yourself type theory (part II)*, *EATCS Bulletin* **35** (1988), pp. 205–245.
- [5] Backhouse, R. C., P. Chisholm, G. Malcolm and E. Saaman, *Do-it-yourself type theory*, *Formal Aspects of Computing* **1** (1989), pp. 19–84.
- [6] Carreño, V. A., C. A. Muñoz and S. Tahar, editors, “Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002),” *Lecture Notes in Computer Science* **2410**, Springer-Verlag, 2002.
- [7] Constable, R. L., S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki and S. F. Smith, “Implementing Mathematics with the `NuPRL` Proof Development System,” Prentice-Hall, NJ, 1986.
- [8] Despeyroux, J. and A. Hirschowitz, *Higher-order abstract syntax with induction in Coq*, in: *LPAR ’94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, *Lecture Notes in Computer Science* **822** (1994), pp. 159–173, also appears as [INRIA research report RR-2292](#).
- [9] Despeyroux, J., F. Pfenning and C. Schürmann, *Primitive recursion for higher-order abstract syntax*, in: R. Hindley, editor, *Proceedings of the International Conference on Typed Lambda Calculus and its Applications (TLCA’97)*, *Lecture Notes in Computer Science* **1210** (1997), pp. 147–163, an extended version is available as [Technical Report CMU-CS-96-172](#), Carnegie Mellon University.
- [10] Hickey, J., A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. N. Krupski, L. Lorigo, S. Schmitt, C. Witty and X. Yu, *MetaPRL — A modular logical environment*, in: D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, *Lecture Notes in Computer Science* **2758** (2003), pp. 287–303.
URL <http://nogin.org/papers/metaprl.html>
- [11] Hickey, J. J., “The `MetaPRL` Logical Programming Environment,” Ph.D. thesis, Cornell University, Ithaca, NY (2001).
URL <http://www.nuprl.org/documents/Hickey/Hickeythesis.html>
- [12] Hickey, J. J., B. Aydemir, Y. Bryukhov, A. Kopylov, A. Nogin and X. Yu, *A listing of MetaPRL theories*.
URL <http://metaprl.org/theories.pdf>

- [13] Hickey, J. J., A. Nogin, A. Kopylov et al., *MetaPRL home page*.
URL <http://metaprl.org/>
- [14] Howe, D. J., *Equality in lazy computation systems*, in: *Proceedings of the 4th IEEE Symposium on Logic in Computer Science* (1989), pp. 198–203.
- [15] Martin-Löf, P., *Constructive mathematics and computer programming*, in: *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science* (1982), pp. 153–175.
- [16] Martin-Löf, P., “Intuitionistic Type Theory,” Number 1 in *Studies in Proof Theory*, Lecture Notes, Bibliopolis, Napoli, 1984.
- [17] Nogin, A., *Quotient types: A modular approach*, in: Carreño et al. [6], pp. 263–280.
URL <http://nogin.org/papers/quotients.html>
- [18] Nogin, A. and J. Hickey, *Sequent schema for derived rules*, in: Carreño et al. [6], pp. 281–297.
URL http://nogin.org/papers/derived_rules.html
- [19] Nogin, A., A. Kopylov, X. Yu and J. Hickey, *A computational approach to reflective meta-reasoning about languages with bindings*, in: *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding* (2005), pp. 2–12, an extended version is available as [California Institute of Technology technical report CaltechCSTR:2005.003](http://www.cse.cmu.edu/~nogin/papers/reflective-meta-reasoning.pdf).
- [20] Pierce, B. C., *Programming with intersection types, union types, and polymorphism*, Technical Report CMU-CS-91-106, Carnegie Mellon University (1991).
- [21] Troelstra, A. S., “Metamathematical Investigation of Intuitionistic Mathematics,” *Lecture Notes in Mathematics* **344**, Springer-Verlag, 1973.