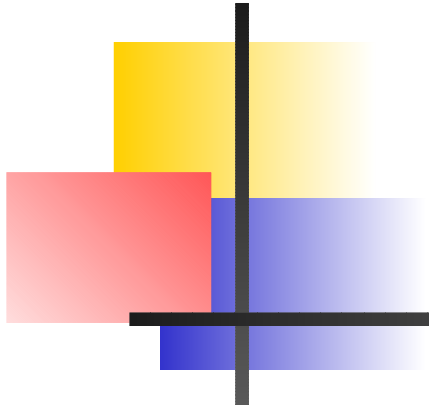


Formalizing Type Operations Using the “Image” Type Constructor

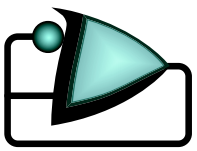
Aleksey Nogin and Alexei Kopylov

California Institute of Technology

WoLLIC, 2006

Introduction

- Set theory constructors
  - Subset constructor: $\{x : A | P[x]\}$
  - Image constructor: $\{f(x) | x \in A\}$
- We do have the *subset* type constructor in the Constructive Type Theory (e.g., in NuPRL, MetaPRL)
- Can we define the *image* type constructor?



Easy...

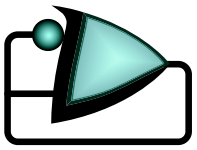
- Let A and B be types.

  Let f be a function of the type $A \to B$.

  Define

$$Img(A; B; f) = \{y : B | \exists x : A \,.\, y = f(x) \in B\}$$

- Here "$t = s \in T$" stands for "t and s are two elements of type T that are equal according to T"



But...

It is not what we are looking for.

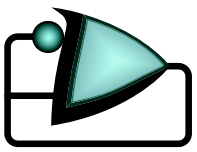
- This type depends on B:

$$Img(A; B; f) = \{y : B | \exists x : A \,.\, y = f(x) \in B\}$$

B is the image of f, which we were trying to define in the first place!

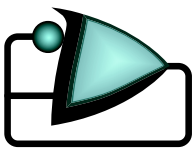
- We can't take the largest type instead of B.
  - In CTT there is the largest type called Top.
  - Top contains all elements, and all elements are equal in Top.
  - Thus, $Img(A; \text{Top}; f)$ is always empty or Top.

- In CTT even if $B \subseteq B'$ and $f \in A \to B$, then $Img(A, B, f)$ is not necessarily equal to $Img(A, B, f)$!



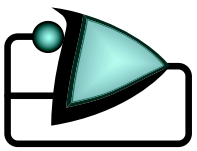
Conclusion

- We want to define $Img(A; f)$ type.
- We can't define it using existing type constructors.
- We can extend our type theory with the image type.



Overview

- Constructive Type Theory Overview.
- Why the definition of the image type is not trivial.
- The definition and semantics of the image type.
- Examples of usage of the image type.
- Main example: Definition of Higher Order Abstract Syntax.



Constructive Type Theory

The Constructive Type Theory (variants of CTT are used in NuPRL and MetaPRL theorem provers) is an extension of Martin-Löf type theory.

CTT includes:

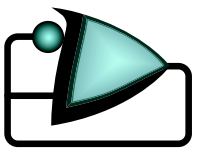
- Equality $a = b \in A$

  Example: the equality on functional types is *extensional*.

  $$\lambda x.|x| = \lambda x.x \in \mathbb{N} \to \mathbb{N}$$

- Martin-Löf type constructors, e.g $A + B$, $A \times B$, $A \to B$

- Other type constructors: e.g., Top, $A \bigcup B$, $\{x : A | P[x]\}$



PER semantics

The standard semantics for CTT is the PER semantics.

- Each type is interpreted as a partial equivalence relation on closed terms.
- This partial equivalence relation provides equality on elements of this type ($a = b \in A$).
- $a \in A \triangleq a = a \in A$

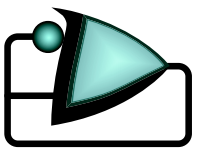
Example 1. Product type: $A \times B$.

$$p = p' \in A \times B \quad \text{iff} \quad p \equiv (a, b),\ p' \equiv (a', b'),\ a = a' \in A,\ b = b' \in B$$

where $t \equiv s$ is a computational equality.

Example 2. Set type: $\{x : A | P[x]\}$.

$$a = a' \in \{x : A | P[x]\} \quad \text{iff} \quad a = a' \in A,\ P[a]$$



Example 3. Top type: $a = b \in \text{Top}$.

Image type is tricky

Example. ***Singleton type:***

$$\{a\} = Img(Unit; \lambda x.a)$$

What are the rules for the singleton type?

- Well-formedness:

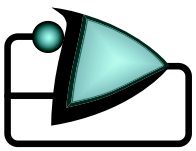
$$\frac{}{\Gamma \vdash \{a\}\ \mathrm{Type}}$$

- Introduction:

$$\frac{}{\Gamma \vdash a \in \{a\}}$$

- Elimination:

$$\frac{\Gamma; \Delta \vdash C[a]}{\Gamma; y : \{a\}; \Delta \vdash C[y]}$$



But these rules lead to contradiction!

- According to the above rules, we can prove that

$$x : \text{Top} \vdash \{x\}\ \text{Type}$$

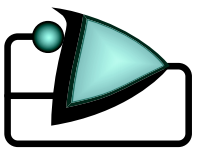
- In CTT the above implies that $\{x\}$ respects the equalities of the Top, namely

$$x : \text{Top}; y : \text{Top}; x = y \in \text{Top} \vdash \{x\} = \{y\}$$

- But all elements are equal in Top, therefore

$$\vdash \{1\} = \{2\}$$

- That leads to $1 = 2$.



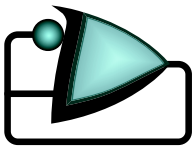
Solution

- Singleton type $\{a\}$ is not always a well-formed type.
  We say that $\{a\}$ is well-formed only when a is a constant (*i.e.* a closed expression).
- MetaPRL uses the *sequent schema* for the inference rules.
  Sequent schema notation

$$\Gamma \vdash f\ \langle\rangle$$

  prohibits f from containing free occurrences of variables declared in Γ.
- The right well formed-rule for singleton:

$$\frac{}{\Gamma \vdash \{a\ \langle\rangle\}\ \mathrm{Type}}$$



Rules for singleton

Rules for singleton

- Well-formedness:

$$\frac{}{\Gamma \vdash \{a\langle\rangle\}\ \mathrm{Type}}$$

- Introduction:

$$\frac{}{\Gamma \vdash a\langle\rangle \in \{a\langle\rangle\}}$$

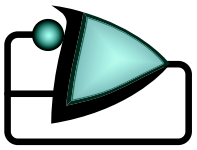
- Elimination:

$$\frac{\Gamma; \Delta \vdash C[a\langle\rangle]}{\Gamma; y : \{a\langle\rangle\}; \Delta \vdash C[y]}$$

Remark. Although, *a priory* $\{a\}$ Type is only true for constants, we can still *derive* $\{a\}$ Type for some non-constant expressions a.

Examples:

- $x : \mathbb{B} \vdash \{x\}$ Type is provable by splitting into two cases: $\{\mathrm{t}\}$ Type and $\{\mathrm{f}\}$ Type.
- $x : \mathrm{Top} \vdash \{x\}$ Type is not provable.



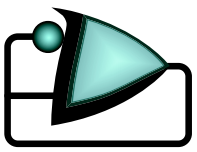
Constructivism

- We are working in the *Constructive* Type Theory.

  If we know that $y \in Img(A; f)$ then we do not necessarily know how to construct an $x \in A$, s.t. $y = f(x)$.

- Therefore the following elimination rule is wrong in the Constructive Type Theory:

$$\frac{\Gamma; x : A; \Delta \vdash C[f(x)]}{\Gamma; y : Img(A; f); \Delta \vdash C[y]}$$



Squash-stable statement

Definition. *A statement A is* squash-stable *iff we can find a witness of A just knowing that there is one.*

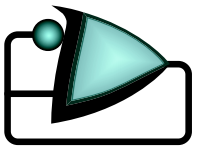
Example. In CTT, membership is squash-stable: $x \in T$ always has trivial witness it.

- The elimination rule for the image type

$$\frac{\Gamma; x : A; \Delta \vdash C[f(x)]}{\Gamma; y : Img(A; f); \Delta \vdash C[y]}$$

  is true if $C[y]$ is squash-stable.

- It is sufficient to axiomatize this rule only for membership statements.



Inference Rules for The Image Type Constructor

New rules:

- Well-formedness:

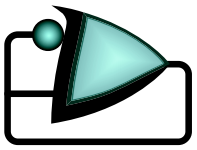
$$\frac{\Gamma \vdash A\,\mathrm{Type}}{\Gamma \vdash Img(A; f\,\langle\rangle)\,\mathrm{Type}}$$

- Introduction:

$$\frac{\Gamma \vdash a \in A}{\Gamma \vdash f[a] \in Img(A; f\,\langle\rangle)}$$

- Elimination:

$$\frac{\Gamma;\, x : A \vdash t \in T[f(x)]}{\Gamma;\, y : Img(A; f\,\langle\rangle) \vdash t \in T[y]}$$

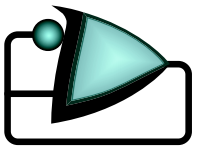


PER Semantics for Image Type

We define the semantics of the Img constructor as follows:

- The closed term $Img(A; f)$ is a well-formed type *if and only if* A is a type.
- $Img(A_1; f_1) = Img(A_2; f_2)$ *iff* $A_1 = A_2$ and $f_1 \equiv f_2$.
- The equality relation on $Img(A; f)$ is the smallest PER s.t.
  - it respects the $\equiv$ relation (e.g. $t = s \in Img(A; f)$ if $t \equiv s$);
  - $f(a) = f(b) \in Img(A; f)$ whenever $a = b \in A$.
- In particular, this means that $t \in Img(A; f)$ *iff* $t \equiv f(a)$ for some $a \in A$.

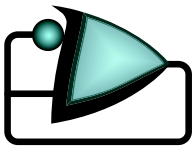
Theorem. The rules from the previous slide are valid under this semantics.



Example

$$
\begin{array}{ccccccc}
 & A & & & & Img(A;f) & \\
a & & & & f(a) & & \\
 & & & f & \| & & \\
b & = & c & \longrightarrow & f(b) & = & f(c) \\
 & & & & & & \| \\
 & & d & & & & f(d)
\end{array}
$$

In this case $f(a) = f(d) \in Img(A;f)$ although $a \neq d \in A$.

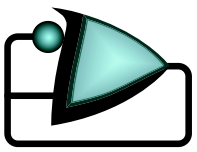


Example: Deriving Union Type Constructor

- The union type $\bigcup_{x:A} B[x]$ is the least common supertype of $B[x]$'s for $x \in A$.
- Now we can define the union type:

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

- Note that the equivalence relation of the $\bigcup_{x:A} B[x]$ is the *transitive closure* of the union of the equivalence relations of types $B[x]$.
  - Surprisingly in CTT, $\mathbb{Z}_4 \cup \mathbb{Z}_6 = \mathbb{Z}_2$.

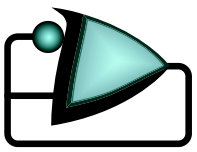


Example: Deriving Set Type Constructor

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

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

- From this definition we are able to derive all the rules that are traditionally postulated with this type constructor.

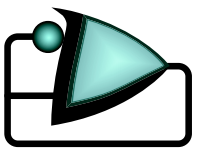


Example: Deriving Squash Type Constructor

- The squash type $\lfloor A \rfloor$ "forgets" the witnesses of A.
  - For any type A, the type $\lfloor A \rfloor$ is empty *if and only if* A is empty and contains a single canonical element it when A is inhabited.
- Now we can define it as

$$\lfloor A \rfloor \triangleq Img(A;\ \lambda x.it)$$

- We can derive all the rules about the squash type that used to be postulated.

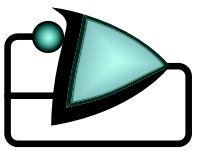


High Order Abstract Syntax

- We want to be able to define languages with bindings in CTT.
- We want to *preserve* bindings.
- We need two basic constructors:
  - $\mathbf{b}x.t[x]$ (the binding constructor)
  - $\mathrm{mkterm}_{op}(t_1; \ldots; t_n)$ (the constructor of terms with an n-ary operator op)

Example. $\lambda x.(x + x)$ is represented as

$$\mathrm{mkterm}_{\lambda}(\mathbf{b}x.\mathrm{mkterm}_{+}([x; x]))$$

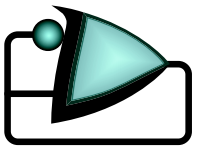


HOAS: The Definition of Syntax

It is easy to define the syntax constructors:

- $\mathbf{b}x.t[x] \triangleq \texttt{inl}\,\lambda x.t$
- $\mathrm{mkterm}_{op}(t_1;\ldots;t_n) \triangleq \texttt{inr}\,(op,[t_1;\ldots;t_n])$

But how do we define the type of terms?



HOAS: Exotic Terms

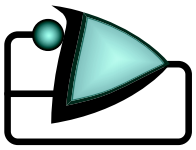
Q: Can we define the type of terms $Term$ inductively s.t.,

$$bind \in (Term \to Term) \to Term$$
$$mkterm \in Operator \times (Term\,\text{List}) \to Term$$

A: No.

- This requires to find the fix point of *non-monotone* operation.
- The induction principle is unclear.
- There are exotic terms, e.g.,

$$\mathbf{b}x.\texttt{if}\ \ x = 0\ \ \texttt{then}\ \ \ 1\ \ \texttt{else}\ \ \ 2.$$



HOAS: The Type of Terms

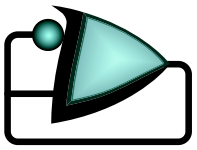
We need to define the type $Term$ to be the minimum type, s.t.

- $Term$ contains variables:

$$\frac{}{\mathbf{b}x_1, \ldots, x_n.x_i \in Term}$$

- $Term$ is closed under the $mkterm_{op}$ constructor, i.e.,

$$\frac{op \in Operator \qquad t_i \in Term}{\mathrm{mkterm}_{op}(t_1; \ldots; t_n) \in Term}$$



HOAS: The Definition of Type

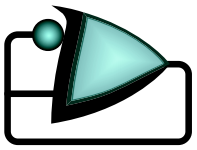
We can define this type using image type.

- $var(n, i) = \mathbf{b}x_1, \ldots, x_n.x_i$ (cf. deBruijn's notations)
- $Var = Img(\mathbb{N} \times \mathbb{N}; x.var(\pi_1 x, \pi_2 x))$
- $TermStep(T) = Img(Operator \times T\,\mathrm{List}; x.\mathrm{mkterm}_{\pi_1 x}(\pi_2 x))$
- $Term = \mu T.(Var \cup TermStep(T))$

This is a natural recursive definition of the type of terms.
It does have the intended induction principle.

Our approach allows us to combine the power of HOAS with the simplicity of the deBruijn's definition:

- We do preserve bindings.
- We have the induction principle and do not have exotic terms.



Summary

Our contributions:

- We add a new primitive constructor for the image type
- We define some old primitive constructors using this new one
  - Thus we simplify the theory.
- We use the definition of HOAS in our big project Reflection.
- This work is implemented in the MetaPRL theorem prover.