A Modular Approach to Type Theory Rules

Aleksey Nogin

April 9, 2001

Overview

- Introduction and motivation
- Basic building blocks (modularizing the quotient type)
- Reusing the new building blocks (collections)

Overview

⇒ Introduction and motivation:
  - I propose splitting complex rules and concepts into simpler ones
    - More flexible rules
    - Easier to understand
    - Allow reusing and recombining
  - I concentrate on the quotient type
    - Quotient rules are the most complex ones in NuPRL
    - We had trouble using quotient types efficiently
- Basic building blocks (modularizing the quotient type)
- Reusing the new building blocks (collections)

Notes

- Simpler rules can be used more flexibly which makes it easier to use them. They are also easier to understand which is another reason it is easier to operate with them.
- With modular system we can now use the submodules separately or combine them differently. And with dependency tracking simple rules give better understanding of what concepts a particular theorem actually uses.
- Without a derived rules mechanism, simple rules can be slower, but if we are able to derive the faster complex rules we need, then it is no longer a problem.

- My primary target was the quotient type since it is one of the hardest to use types (probably the hardest one) in the NuPRL type theory. In particular, I was hoping to be able to formalize the collection types that Mark could not formalize in NuPRL.

Overview

- Introduction and motivation
- ⇒ Basic building blocks (modularizing the quotient type)
  - The Squash type
  - The extensional Squash type
  - The explicit nondeterminicity type
  - The intensional quotient type

  Criteria:
  - · "single–step" rules
  - · balanced introduction and elimination rules
  - · reversible rules
- Reusing the new building blocks (collections)

Notes

- In choosing particular rules for the new concepts I introduce I try to keep the rules elim-intro "symmetric" and balanced.
- Reversible rules (e.g. the ones the are valid both "forward" and "backward") are good for automation.

- The first basic building block that we need is the `squash` type.
- Remember, $[A]$ is the same as $\{\mathrm{Unit} \mid A\}$, but since we want to define sets using squash, we can not use this definition.

The Squash Type

$$\frac{\Gamma \vdash A}{\Gamma \vdash [A]} \qquad \textit{(Squash-intro)}$$

$$\frac{\Gamma;\ x : A;\ \Delta \vdash t \in C}{\Gamma;\ x : [A];\ \Delta \vdash t \in C} \qquad \textit{(Squash-elim)}$$

$$\frac{\Gamma \vdash [A]}{\Gamma \vdash \bullet \in [A]} \qquad \textit{(Squash-mem-intro)}$$

$$\frac{\Gamma;\ x : [A];\ \Delta[\bullet] \vdash C[\bullet]}{\Gamma;\ x : [A];\ \Delta[x] \vdash C[x]} \qquad \textit{(Squash-mem-elim)}$$

Notes

- For each intro we have a matching elim.
- All the rules except for (*Squash-intro*) are reversible.
- Note that (*Squash-mem-elim*) allows us not to consider the case x occurs free somewhere in the rule (*Squash-elim*).

THE Squash TYPE A A DERIVED RULE

$$\frac{\Gamma;\ x : A;\ \Delta \vdash [C]}{\Gamma;\ x : [A];\ \Delta \vdash [C]} \qquad \textit{(Squash-elim')}$$

Notes

- The squash type can be immediately reused for many different purposes (other then formalizing the quotient type). One of them is formalizing Markov's principle ($[A \vee \neg A]$) that Alexei showed us last week.
- Another is formalizing the set type properly. For simplicity I only show independent sets and I do not show membership rules, but they all are very similar and have very similar properties.

EXAMPLE: THE INDEPENDENT SET TYPE

$$\frac{\Gamma;\ x : A;\ y : [B];\ \Delta[x] \vdash C[x]}{\Gamma;\ x : \{A \mid B\};\ \Delta[x] \vdash C[x]} \qquad \textit{(Set-elim)}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash [B]}{\Gamma \vdash \{A \mid B\}} \qquad \textit{(Set-intro)}$$

Notes

- Again, we get an elim and an into rule that exactly correspond to each other (in Girard's and Pfenning's "local cut elimination" sense Γ *explain on the board*).
- Note that both rules are "exact" Γ e.g. the conclusion is derivable only if the assumption is derivable. This means that we can freely apply the rule whenever it is applicable (and more importantly Γ we can instruct our tactics to do it). This was not the case for the original set introduction rule.

- The next building block that we need is the extensionality. One of the easy ways to define it is to define an extensional squash type.
- As usual, $[\![A]\!]$ is a type when A is a type.
- Using the extensional quotient type one could define $[\![A]\!]$ as $\texttt{tt} = \texttt{ff} \in (x, y : \mathbb{B} /\!/_{e} ((x = y \in \mathbb{B}) \vee A))$ (this is how Jason defined it in MetaPRL). However, again, we want to define quotient type through esquash, not the other way around.

Extensional Squash Operator

$$\frac{\Gamma \vdash [\![A]\!] \Leftrightarrow [\![B]\!]}{\Gamma \vdash [\![A]\!] = [\![B]\!]} \qquad (\textit{Esquash-equality})$$

$$\frac{\Gamma \vdash [A]}{\Gamma \vdash [\![A]\!]} \qquad (\textit{Esquash-intro})$$

$$\frac{\Gamma \vdash [\![A]\!] \qquad \Gamma \vdash A\ \mathsf{Type}}{\Gamma \vdash [A]} \qquad (\textit{Esquash-elim})$$

$$\frac{\Gamma \vdash [\![A]\!]}{\Gamma \vdash \bullet \in [\![A]\!]} \qquad (\textit{Esquash-mem-intro})$$

$$\frac{\Gamma;\ x : [\![A]\!];\ \Delta[\bullet] \vdash C[\bullet]}{\Gamma;\ x : [\![A]\!];\ \Delta[x] \vdash C[x]} \qquad (\textit{Esquash-mem-elim})$$

Notes

- Again, all the rules except for (*Esquash-intro*) are reversible.

- The next building block that we need is to be able to express the type that has to distinct elements quotiented together. We use the type we call ND. The idea is to make ND similar to $\mathbb{B}/\!\!/\mathsf{True}$.

Explicit Nondeterminism

$$\frac{\Gamma;\ u : A[\mathtt{tt}] = A[\mathtt{ff}];\ y : A[\mathtt{tt}];\ x : \mathsf{ND};\ \Delta[x;y] \vdash C[x;y]}{\Gamma;\ x : \mathsf{ND};\ y : A[x];\ \Delta[x;y] \vdash C[x;y]} \qquad (\textit{ND-elim})$$

$$\frac{\Gamma \vdash C[\mathtt{tt}] = C[\mathtt{ff}] \qquad \Gamma \vdash C[\mathtt{tt}]}{\Gamma;\ x : \mathsf{ND} \vdash C[x]} \qquad (\textit{ND-elim}')$$

Explicit ND A a Derived Rule

$$\frac{\Gamma \vdash t[\mathtt{tt}] = t[\mathtt{ff}] \in A}{\Gamma;\ x : \mathsf{ND} \vdash t[x] \in A} \qquad (\textit{ND-memb})$$

We also need a "choice" operation $\mathsf{ND}_x\{t_1;t_2\}$ such that

$$\mathsf{ND}_{\mathtt{tt}}\{u;v\} \rightsquigarrow u \qquad \mathsf{ND}_{\mathtt{ff}}\{u;v\} \rightsquigarrow v$$

We can use $\mathsf{ND}_x\{t_1;t_2\} := \mathtt{if}\ x\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2\ \mathtt{fi}$

Notes

- We could use the $ifthenelse$ operator for ND...$\{\cdots\}$, or we could create new `tt`, `ff` and ND...$\{\cdots\}$ for ND, it does not really matter.
- Notice that (*ND-elim*) does not completely eliminate ND, in order to eliminate it we need to iterate (*ND-elim*) and then apply (*ND-elim'*).
- The ruleset for ND is intensionally very incomplete. In this context, ND is only meant to be an auxiliary type since our mechanism for expressing rules can not express the result of completely eliminating ND.
- (*ND-memb*) is here to show what (*ND-elim'*) means in a simple particular case where it is especially interesting.
- However, this type seems to also be interesting on its own. If we decide to use it on its own, we'll need more rules about it, but this is outside of the scope of this talk.
- Again, all the rules are reversible.
- Now we have all we need to be able to describe the quotient type.
- Note that NuPRL 4 has intensional set type, but an extensional quotient type. Now that we have esquash, it's easy (*show on the board*) to convert intensional set and quotient types into extensional ones by esquashing the predicate, so I will focus on the intensional quotient type.

Intensional Quotient Type A I

$$\frac{E \text{ is an ER over } A \qquad \Gamma \vdash A = A' \qquad \Gamma;\, x : A;\, y : A \vdash E[x;y] = E'[x;y]}{\Gamma \vdash (A /\!\!/_i E) = (A' /\!\!/_i E')}$$

(*Iquot-equal*)

$$\frac{\Gamma \vdash x \in A \qquad \Gamma \vdash (A /\!\!/_i E) \text{ Type}}{\Gamma \vdash x \in (A /\!\!/_i E)}$$

(*Iquot-mem-intro*)

$$\frac{\Gamma \vdash [\![E[x;y]]\!] \qquad \Gamma \vdash x \in (A /\!\!/_i E) \qquad \Gamma \vdash y \in (A /\!\!/_i E)}{\Gamma \vdash x = y \in (A /\!\!/_i E)}$$

(*Iquot-equal-intro*)

$$\frac{\Gamma;\, u : x = y \in (A /\!\!/_i E);\, v : [\![E[x;y]]\!];\, \Delta[u] \vdash C[u]}{\Gamma;\, u : x = y \in (A /\!\!/_i E);\, \Delta[u] \vdash C[u]}$$

(*Iquot-equal-elim*)

Intensional Quotient Type A II

$$\frac{\Gamma;\, u_1 : A;\, u_2 : A;\, v : E[u_1;u_2];\; x : \text{ND};\, \Delta[\text{ND}_x\{u_1;u_2\}] \vdash [C[\text{ND}_x\{u_1;u_2\}]]}{\Gamma;\, u : A /\!\!/_i E;\, \Delta[u] \vdash [C[u]]}$$

(*Iquot-elim*)

Derived rules:

$$\frac{\Gamma;\, u_1 : A;\, u_2 : A;\, v : E[u_1;u_2];\; x : \text{ND};\, \Delta[\text{ND}_x\{u_1;u_2\}] \vdash t[u_1] = t[u_2] \in C}{\Gamma;\, u : A /\!\!/_i E;\, \Delta[u] \vdash t[u] \in C}$$

(*Iquot-elim2*)

$$\frac{\Gamma;\, u : x = y \in (A /\!\!/_i E);\, v : [E[x;y]];\, \Delta[u] \vdash C[u] \qquad \Gamma;\, u : x = y \in (A /\!\!/_i E);\, \Delta[u] \vdash E[x;y] \text{ Type}}{\Gamma;\, u : x = y \in (A /\!\!/_i E);\, \Delta[u] \vdash C[u]}$$

(*Iquot-equal-elim2*)

Notes

- It's interesting to note that even for intensional quotient type we need the extensional squash operator. Indeed, in (*Iquot-equal-intro*), x and y could potentially range over elements of an equivalence class of A, but E is not necessary functional over these classes, it's only functional over A. Of course, we could restrict this rule to the case when E is indeed functional Γ this is what (*Iquot-equal-elim2*) does and what the corresponding NuPRL rule does. However, (*Iquot-equal-elim2*) is weaker then (*Iquot-equal-intro*) and sometimes it is insufficient.
- (*Iquot-elim2*) shows what (*Iquot-elim*) means when the conclusion is a membership. This rule is very similar to what NuPRL has (although NuPRL has two versions of it and (*Iquot-elim*) is a generalization of *both* of them.
- Mark wanted a type of collections of elements of arbitrary type T. We could use either $I : \mathbb{U}_i \times (I \to T)$ or just $T \to \mathbb{P}_i$, they are isomorphic.

Overview

- Introduction and motivation
- Basic building blocks (modularizing the quotient type)

⇒ Reusing the new building blocks (collections)
  - The problem
  - A solution

Collections A the Problem

$$\mathsf{Col}_i\{T\} := c_1, c_2 : (T \to \mathbb{P}_i)/\!\!/_i(\forall t : T.\, c_1(t) \Leftrightarrow c_2(t))$$

We want to define `mem` and `union`:

1. $\mathtt{mem}: \ \mathsf{Col}_i\{T\} \to T \to \mathbb{P}_i$
2. $\mathtt{union}: \ I : \mathbb{U}_i \to (I \to \mathsf{Col}_i\{T\}) \to \mathsf{Col}_i\{T\}$
3. $\forall c : T \to \mathbb{U}_i.\, \forall t : T.\, c(t) \Rightarrow \mathtt{mem}(c; t)$
4. $\forall c : T \to \mathbb{U}_i.\, \forall t : T.\, \neg c(t) \Rightarrow \neg\mathtt{mem}(c; t)$
5. $\forall c_1, c_2 : \mathsf{Col}_i\{T\}.$ $$\Big(\Big(\forall t : T.\, \big(\mathtt{mem}(c_1; t) \Leftrightarrow \mathtt{mem}(c_2; t)\big)\Big) \Leftrightarrow c_1 = c_2 \in \mathsf{Col}_i\{T\}\Big)$$
6. $\forall I : \mathbb{U}_i.\, \forall C : I \to \mathsf{Col}_i\{T\}.\, \forall t : T.\, \big(\exists i : I.\, \mathtt{mem}(C(i); t) \Rightarrow \mathtt{mem}(\bigcup_{i:I} C(i); t)\big)$
7. $\forall I : \mathbb{U}_i.\, \forall C : I \to \mathsf{Col}_i\{T\}.\, \forall t : T.$ $$\Big(\neg\Big(\exists i : I.\, \mathtt{mem}\big(C(i); t\big)\Big) \Rightarrow \neg\Big(\mathtt{mem}\big(\bigcup_{i:I} C(i); t\big)\Big)\Big)$$

Notes

- Note in 4 that we can not require $\mathtt{mem} \Rightarrow c$ since that would mean that we'll have to be able to reconstruct proofs of c which is not realistic. Same in 7.
- In 5, the $\Leftarrow$ direction is easy Γ it follows from 1.
- Problems: the solutions we considered would either give us `mem` in $\mathbb{P}_i/\!\!/Lra$, or `mem` and/or `union` would raise the universe level, or something else would go wrong.

- And now the solution.

Collections A a Solution

$$\mathtt{mem}(c; t) := [\![c(t)]\!]$$

$$\bigcup_{i:I} C(i) := \lambda t.\exists i : I.\mathtt{mem}(C(i); t)$$