CS 671, Automated Reasoning

Lesson 10: *Refinement Logic* February 20, 2001

Now that we have introduced the basic principles of building type theory, discussed how to use type theory for the formalization of very interesting parts of computational mathematics, and demonstrated how to make use of formal theories in an interactive system, it is time to extend our theoretical foundations agin.

The set of proof rules that we have defined so far is sound and complete with respect to the Martin-Löf semantics of our small theory of natural numbers (and of function types). Nevertheless, it makes sense to modify and expand it again to make it more useful for reasoning about computations. In the process we will discuss a few more design principles for building type theory. In the next week we will then use these design principles to add more data structures to our formal language.

Refinement Logic and Proof Objects

When we introduced the refinement style for reasoning about mathematical statements, we did so mostly to support a top-down proof style that helps interactive users to stay focused. Refinement logic proceeds by *decomposing* a proof task into "smaller" tasks until the tasks are so trivial that they can be knocked off by referring to axioms of the theory, existing hypotheses, or already proven lemmata.

The form of sequents we use in our formal proofs is designed to fit that methodology. A sequent of the form

$$x_1{:}T_1,\ \ldots x_n{:}T_n \vdash e{=}e' \ \in\ T$$

is usually read as "*under the assumption that the x_i are variables of type T_i we conclude that e is equal to e in type T*". Our rules show how to decompose e, e', and T – occasionally introducing new hypotheses – until we can apply the `hypotheses` rule (or call a lemma).

But so far, we haven't really taken full advantage of the refinement style. The proofs that we can handle with our theory so far require us to provide e, e', and T beforehand. But refinement usually means that we start with something coarse, something not completely specified, and step by step introduce new information until all the details that we need have been specified (this is how Nikolaus Wirth originally defined refinement). How can we adopt this style of reasoning to our theory?

Well, in programming you would proceed by stating that you want to develop some program that has a certain property, e.g. a being a function from $\mathbb{N}$ to $\mathbb{N}$ that computes the integer square root of an input. In type theory, we would state that we want to construct *some* element of a certain type – and eventually our type theory will be so expressive that we can express *all* program properties through a type.

So, instead of providing the element right at the beginning and then proving that it does in fact belong to the given type, we would state goals of the form

$$x_1{:}T_1,\ \ldots x_n{:}T_n \vdash T$$

and read them as "*under the assumption that the x_i are variables of type T_i we can construct a member of the type T*". This modification obviously affects the the way we do proofs. Instead of showing membership explicitly as in

$\vdash$ `suc(0)` $\in \mathbb{N}$ **by** `suc-rule`
  $\vdash$ `0` $\in \mathbb{N}$ **by** `0-rule`

we would *construct the element of* $\mathbb{N}$ *implicitly during the proof* and proceed as follows

$\vdash \mathbb{N}$ **by** `suc-formation`
  $\vdash \mathbb{N}$ **by** `0-formation`

That means in the first step we decide that we want to create the successor of something and then in the second we determine this something to become zero. To support this style of implicit reasoning, we need to adjust the formulation of our proof rules as well. Instead of

$\bar{H} \vdash \texttt{suc}(e) \in \mathbb{N}$ **by** `suc-rule`
  $\bar{H} \vdash e' \in \mathbb{N}$

we now write the rule for reasoning about the successor as

$\bar{H} \vdash \mathbb{N}$ $\lfloor$ext $\texttt{suc}(e)\rfloor$ **by** `suc-formation`
  $\bar{H} \vdash \mathbb{N}$ $\lfloor$ext $e\rfloor$

The term $\lfloor$ext $\texttt{suc}(e)\rfloor$ means that by applying the rule `suc-formation` we will create an expression of the form $\texttt{suc}(e)$ where e still needs to be determined in further refinement steps. Furthermore, the rule guarantees that the created term is a member of $\mathbb{N}$ whenever e is.

Logically, this rule states pretty much the same as before – it reflects the judgment about the successor term – but in practice it does not require us to provide this successor term in full detail when we initialize our proof. If we adopt the full refinement style to all rules of our small theory of natural numbers, we get the following set of rules.

1. $\bar{H} \vdash$ `type` $\lfloor$ext $\mathbb{N}\rfloor$ **by** $\mathbb{N}$`-formation`
2. $\bar{H} \vdash \mathbb{N}$ $\lfloor$ext 0$\rfloor$ **by** `0-formation`
3. $\bar{H} \vdash \mathbb{N}$ $\lfloor$ext $\texttt{suc}(e)\rfloor$ **by** `suc-formation`
   $\bar{H} \vdash \mathbb{N}$ $\lfloor$ext $e\rfloor$
4. $\bar{H} \vdash T$ $\lfloor$ext $\texttt{ind}(e;\mathit{base};n,x.\mathit{up})\rfloor$ **by** `ind-formation` n x
   $\bar{H} \vdash \mathbb{N}$ $\lfloor$ext $e\rfloor$
   $\bar{H} \vdash T$ $\lfloor$ext $\mathit{base}\rfloor$
   $\bar{H},\ n{:}\mathbb{N},\ x{:}T \vdash T$ $\lfloor$ext $\mathit{up}\rfloor$
5. $\bar{H}_1,\ x{:}T,\ \bar{H}_2 \vdash T$ $\lfloor$ext $x\rfloor$ **by** `hypothesis` i

Note that the refinement style now require us to provide parameters for some of the rules, which formerly could be determined by looking at the term. For the formation of the induction expression two new variables must be created while the hypothesis rule now needs the index of the hypothesis that shall be used for creating a member of the type T. Both parameters could be determined heuristically, but this is a matter of tactic programming and should not be part of the proof theory.

Once a proof is complete, the proof rules tell us exactly how to construct a term for each proof node. We start at the leaves, where the `hypothesis` rule introduces basic expressions and then assemble *proof terms* bottom up as described by the rules. As NUPRL allows us to extract the top-level term from a proven theorem object, we usually call this proof term the *extract term* of the proof.

The above refinement rules are fine for reasoning about typehood and membership, but what about equality and computation? Obviously the rules do not support that, so we must preserve the rules for reasoning about equality as well. But that creates a certain dilemma. The refinement style forces us to deal with sequents of the form

$x_1{:}T_1,\ \ldots x_n{:}T_n \vdash T \quad \lfloor\text{ext } e\rfloor$

while reasoning about equality requires sequents to have the form

$x_1{:}T_1,\ \ldots x_n{:}T_n \vdash e{=}e' \in T$

Certainly we do not want a proof theory where we have to deal with two types of sequents.

How can we unify these two forms?

Technically, the simplest answer to that is to add extract terms to the latter kind of sequents as well, that is to express reasoning about equality by sequents of the form

$x_1{:}T_1,\ \ldots x_n{:}T_n \vdash e{=}e' \in T \quad \lfloor\text{ext } p\rfloor$

where p is some proof term that provides evidence for the equality $e{=}e' \in T$. Now before we do that, let's think about the consequences of such a step.

1. Adding an extract term to an equality means that we treat an equality *as if it were a type*, and not a logical proposition. Formally, there are no objections against doing that. After all, our proof theory is just a formal representation of a semantical concept and nothing should prevent us from using the same representation for two different concepts if there are no essential differences in the way we formulate proofs about them. Formal systems can't capture the full meaning of a concept anyway but only the laws for handling it.

   So proof theoretically, declaring equality to be a type is not a big deal. Philosophers of science, however, might jump up and down at this, since logical propositions and data types are really not the same. So, whether we want to introduce equality types or not, is really a *design decision* with far-reaching consequences, but not a matter of being right or wrong.

   One of the main reasons for introducing an equality type is that it does not only unify implicit and explicit reasoning, but also allows us to state equality *assumptions* in a sequent and expand our capability for reasoning about equality beyond decomposition and evaluation. We can, for instance, state and prove the following

   $\texttt{\%1:i=j} \in \mathbb{N},\ \texttt{\%2:j=k} \in \mathbb{N},\ \texttt{f:}\mathbb{N}\to\mathbb{N} \vdash \texttt{f(i)=f(k)} \in \mathbb{N} \quad \lfloor\text{ext ?}\rfloor$

   which brings practically useful concepts like *transitivity* and *extensionality* into our theory. These considerations eventually lead to the following descign decision:

   Equality $e{=}e' \in T$ is a type in NUPRL's type theory.

   We will discuss the implications of that decision in the next lecture.

2. Once we have decided to view an expression of the form e=$e' \in T$ as type, we must determine what the members of such a type should be. What computational content does an equality have? What extract terms would we assign to a rule like the following?

   $\bar{H} \vdash$ `suc(`e`)=suc(`e'`)` $\in \mathbb{N}$ $\lfloor$ext ??$\rfloor$ **by** `suc-rule`

   $\quad\bar{H} \vdash e$=$e' \in \mathbb{N}$ $\lfloor$ext ?$\rfloor$

   While the implicit refinement rules are intended to construct an actual member of a type that can be used in computations, the purpose of a rule like the above is primarily to verify that a certain equality holds. We are not really interested in extracting a term that describes why the equality holds – if we want to know that, we can look at the proof. And we certainly don't want to run any computations with that term. All we want to know is "*do we have evidence that the equality holds?*" and the proof term should give us exactly that information.

   Again, there is a design decision to be made. We could build members of equality types from constructors for the equality of `zero`, `suc`, `ind`, `lambda`, `apply`, and also `transitivity`, `symmetry`, and of course all the types and members that we will introduce in the future. Or we could simply say "*let's ignore all these details – just tell me if the equality holds at all*", and provide a single expression that will represent the only element of an equality.

   NUPRL's type theory has favored the latter. So equality comes with only one element, denoted by `Ax` (axiom), which is an element of the type e=$e' \in T$ if and only if that equality holds. All the proof rules that deal with equalities will create just this term as extract term and nothing else.

Doing so affects our theory as follows:

Syntax: Two new expressions represent the theory of equality:

e=$e' \in T$ and `Ax`

Evaluation: `Ax` is a canonical term that cannot be evaluated further.

Semantics: Two new judgments is needed to link the equality type to the equality judgment

7. e=$e' \in T$ is a type if T is a type and e and e' are elements of T
8. `Ax` is an element of e=$e' \in T$ if e and e' are equal elements of T

Recall that "e is an element of T" is an abbreviation for "e is equal to e in T". Note also, that e=$e' \in T$ is not even wellformed, if either e or e' is not an element of T. This is very strict, but otherwise all kinds of nonsense could creep into type theory.

Proof Theory: Most rules for reasoning about equality are tied to a specific type like $\mathbb{N}$ or the function type. However, we need to add a few more rules that deal specifically with equality.

- $\bar{H}$ ⊢ type ⌊ext e=e' ∈ T⌋ **by** equality-formation T
  $\bar{H}$ ⊢ T type ⌊ext Ax⌋
  $\bar{H}$ ⊢ T ⌊ext e⌋
  $\bar{H}$ ⊢ T ⌊ext e'⌋
- $\bar{H}$ ⊢ e=e' ∈ T type ⌊ext Ax⌋ **by** equality-R
  $\bar{H}$ ⊢ T type ⌊ext Ax⌋
  $\bar{H}$ ⊢ e=e ∈ T ⌊ext Ax⌋
  $\bar{H}$ ⊢ e'=e' ∈ T ⌊ext Ax⌋
- $\bar{H}$ ⊢ Ax ∈ e=e' ∈ T ⌊ext Ax⌋ **by** axiom-Eq
  $\bar{H}$ ⊢ e=e' ∈ T ⌊ext Ax⌋
- $\bar{H}_1$, x:T, $\bar{H}_2$ ⊢ x=x ∈ T ⌊ext Ax⌋ **by** hypothesisEq i
- $\bar{H}$ ⊢ $C[e/x]$ ⌊ext t⌋ **by** substitution e=e' ∈ T x C
  $\bar{H}$ ⊢ e=e' ∈ T ⌊ext Ax⌋
  $\bar{H}$ ⊢ $C[e'/x]$ ⌊ext t⌋
- $\bar{H}$ ⊢ e_1=e_2∈T ⌊ext Ax⌋ **by** symmetry
  $\bar{H}$ ⊢ e_2=e_1∈T ⌊ext Ax⌋

Note that the substitution rule has a fairly complex parameter list. We have to give the equality and must indicate which subterms of our conclusion shall be replaced, as there may be several instances of e in C. We do so by introducing a new variable x and replacing all occurrences of e in C that we want to see substituted by that variable. Obviously there are tactics that can figure that out, and there could be editing features where we could mark the terms by point-and-click. But again, the proof theory should not provide such heuristics. The actual version of the substitution rule is a bit more complex as we also need to show that C is globally well-formed, an issue that is trivial at this point but will become relevant when we extend the theory.

1. *We do not need an explicit transitivity rule,*

   $\bar{H}$ ⊢ e_1=e_2∈T ⌊ext Ax⌋ **by** transitivity e_3
   $\bar{H}$ ⊢ e_1=e_3∈T ⌊ext Ax⌋
   $\bar{H}$ ⊢ e_3=e_2∈T ⌊ext Ax⌋

   show how this rule can be simulated by the others rules.
2. *What would happen if we were to drop the symmetry rule as well and would try to simulate it?*
3. *Why is there no rule for reflexivity?*

The rules for reasoning about equality and computation have to be changed by adding `Ax` as extract term to each goal. For our small theory of natural numbers this results in the following set of rules

- $\bar{H} \vdash \mathbb{N}\ \mathtt{type}\quad \lfloor\text{ext Ax}\rfloor\quad$ **by** $\mathbb{N}$-R
- $\bar{H} \vdash \mathtt{0{=}0} \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor\quad$ **by** 0-Eq
- $\bar{H} \vdash \mathtt{suc}(e)\mathtt{=suc}(e') \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor\quad$ **by** suc-Eq
  $\bar{H} \vdash e{=}e' \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor$
- $\bar{H} \vdash \mathtt{ind}(e;base;n,x.up) = \mathtt{ind}(e';base';n',x'.up') \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor\quad$ **by** ind-Eq
  $\bar{H} \vdash e{=}e' \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor$
  $\bar{H} \vdash base{=}base' \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor$
  $\bar{H},\ n{:}\mathbb{N},\ x{:}\mathbb{N} \vdash up{=}up'[n,x\,/\,n',x'] \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor$
- $\bar{H} \vdash \mathtt{ind}(0;base;n,x.up) = e' \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor\quad$ **by** compute 1
  $\bar{H} \vdash base = e' \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor$
- $\bar{H} \vdash \mathtt{ind}(\mathtt{suc}(e);base;n,x.up) = e' \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor\quad$ **by** compute 1
  $\bar{H} \vdash up[\mathtt{e'},\mathtt{ind}(e';base;x.n,up)\ /\ n,x] = e' \in \mathbb{N}\quad \lfloor\text{ext Ax}\rfloor$

Note that the old `hypotheses` rule is now covered by the rule `hypothesisEq` of the equality type. The rules for the function type have to be modified in the same way.

Supporting both a refinement style for implicitly creating elements of types and explicit reasoning about equality and computation doubles the number of proof rules in type theory. But because of the symmetries between these rules, it does not make the task of proving the proof rules sound and complete more difficult than before.

Furthermore, an interactive user of the NUPRL system does not have to memorize all the names of the primitive inferences. Almost all formation (i.e. implicit) rules are covered by the tactic `D`, while `MemD` and `EqD` cover the rules for explicit reasoning about membership and equality.

Type theory also provides one additional rule that links the implicit and explicit forms of reasoning. When constructing an element e of a type T, we may explicitly provide that element and then only have to prove that e is actually a member of T. This rule is useful for forcing very efficient programs into the extract term of a theorem that are difficult to construct via implicit reasoning. We use this, for instance, in theorems about efficient induction forms.

$\bar{H} \vdash T\quad \lfloor\text{ext } e\rfloor\quad$ **by** introduction e
$\quad\bar{H} \vdash e{=}e \in T\quad \lfloor\text{ext Ax}\rfloor$

Decomposing Assumptions

Our proof theory so far is sound and complete wrt. Martin-Löf's semantics of natural numbers and functions in the sense that it allows us to reason about expressions that correspond to natural number constants. However, it does not support reasoning about *variables* that are of type $\mathbb{N}$ in the same way. For instance, we may be able to prove

$\vdash n{+}m = m{+}n \in \mathbb{N}$

(where $n{+}m$ is defined as `ind` $(n;\ m;$ `i, sum.  suc(sum)`$)$ for each concrete instance of n and m, but we cannot prove a goal of the form

$\texttt{n}{:}\mathbb{N},\ \texttt{m}{:}\mathbb{N} \vdash \texttt{n+m = m+n} \in \mathbb{N}$

Typically, such a goal is proven by induction over `n` and `m`, i.e. one would want to proceed as follows.

$\texttt{n}{:}\mathbb{N},\ \texttt{m}{:}\mathbb{N} \vdash \texttt{n+m = m+n} \in \mathbb{N}$ **by** `induction 1`
1. $\texttt{n}{:}\mathbb{N},\ \texttt{m}{:}\mathbb{N} \vdash \texttt{0+m = m+0} \in \mathbb{N}$
2. $\texttt{n}{:}\mathbb{N},\ \texttt{m}{:}\mathbb{N},\ \texttt{n+m=m+n} \in \mathbb{N} \vdash \texttt{suc(n)+m = m+suc(n)} \in \mathbb{N}$

To support this style of reasoning, we need to add an induction rule for natural numbers to our type theory. Basically, this rule would have the following form.

$\bar{H}_1,\ n{:}\mathbb{N},\ \bar{H}_2 \vdash C$ **by** `induction` i
$\quad \bar{H}_1,\ n{:}\mathbb{N},\ \bar{H}_2 \vdash C[0/n]$
$\quad \bar{H}_1,\ n{:}\mathbb{N},\ \bar{H}_2,\ C \vdash C[suc(n)/n]$

But this form is not yet complete. We need to turn the assumption C into a declaration, by adding a label x to it, and must provide appropriate extract terms.

What extract terms can we meaningfully assign to that rule?

If we call the extract term for the base case *base* and the one for the step case *up*, then we know that *up* may depend on n and the label x for the assumption C. The proof term for the main goal must express that we get the term *base* if n evaluates to 0, and that we inductively apply *up* if n is a successor term. This is exactly what our induction term `ind`$(n;\ base;\ n,x.up)$ expresses, so the rule will be as follows.

$\bar{H}_1,\ n{:}\mathbb{N},\ \bar{H}_2 \vdash C\ \lfloor \text{ext ind}(n; base; n,x.up) \rfloor$ **by** `natElim` i
$\quad \bar{H}_1,\ n{:}\mathbb{N},\ \bar{H}_2 \vdash C[0/n]\ \lfloor \text{ext } base \rfloor$
$\quad \bar{H}_1,\ n{:}\mathbb{N},\ \bar{H}_2,\ x{:}C \vdash C[suc(n)/n]\ \lfloor \text{ext } up \rfloor$

Note that we called the rule `natElim` instead of induction, as we will define similar rules for analyzing variables of other types and want to keep a uniform naming scheme.

As the `natElim` rule can be used for the formation of the induction expression, the rule `ind-formation` has become redundant.

$\bar{H} \vdash T\ \lfloor \text{ext ind}(e; base; n,x.up) \rfloor$ **by** `ind-formation` $n\ x$
$\quad \bar{H} \vdash \mathbb{N}\ \lfloor \text{ext } e \rfloor$
$\quad \bar{H} \vdash T\ \lfloor \text{ext } base \rfloor$
$\quad \bar{H},\ n{:}\mathbb{N},\ x{:}T \vdash T\ \lfloor \text{ext } up \rfloor$

To generate $\texttt{ind}(e\,;\,base\,;\,n\,,x\,.\,up)$ it suffices to create the expression $(\lambda n\,.\,\texttt{ind}(n\,;\,base\,;\,n\,,x\,.\,up))$ e using `app-formation` $\mathbb{N}{\rightarrow}T$, λ-`Formation` n, and **`natElim`**.

In a similar way, we introduce a rule for the use of variables that are declared to be of a function type. The rule

$$\begin{array}{l} \bar{H}_1\,,\; f\!:\!S{\rightarrow}T\,,\; \bar{H}_2\;\vdash\; C \quad \lfloor\text{ext } e[f\,s/y]\rfloor \quad \textbf{by}\ \texttt{functionElim}\ i\ y \\ \quad\quad \bar{H}_1\,,\; f\!:\!S{\rightarrow}T\,,\; \bar{H}_2\;\vdash\; S \quad \lfloor\text{ext } s\rfloor \\ \quad\quad \bar{H}_1\,,\; f\!:\!S{\rightarrow}T\,,\; \bar{H}_2\,,\; y\!:\!T\;\vdash\; C \quad \lfloor\text{ext } e\rfloor \end{array}$$

subsumes the rule `app-formation`

$$\begin{array}{l} \bar{H}\;\vdash\; T \quad \lfloor\text{ext } f\ \ e\rfloor \quad \textbf{by}\ \texttt{app-formation}\ S{\rightarrow}T \\ \quad\quad \bar{H}\;\vdash\; S{\rightarrow}T \quad \lfloor\text{ext } f\rfloor \\ \quad\quad \bar{H}\;\vdash\; S \quad \lfloor\text{ext } e\rfloor \end{array}$$

Here is the complete set of rules for the function type

- $\begin{array}{l} \bar{H}\;\vdash\; \texttt{type} \quad \lfloor\text{ext } S{\rightarrow}T\rfloor \quad \textbf{by}\ {\rightarrow}\texttt{-formation} \\ \quad\quad \bar{H}\;\vdash\; \texttt{type} \quad \lfloor\text{ext } S\rfloor \\ \quad\quad \bar{H}\;\vdash\; \texttt{type} \quad \lfloor\text{ext } T\rfloor \end{array}$
- $\begin{array}{l} \bar{H}\;\vdash\; S{\rightarrow}T \quad \lfloor\text{ext } \lambda x.e\rfloor \quad \textbf{by}\ \lambda\texttt{-Formation}\ x \\ \quad\quad \bar{H}\,,\; x\!:\!S\;\vdash\; T \quad \lfloor\text{ext } e\rfloor \\ \quad\quad \bar{H}\;\vdash\; S\ \texttt{type} \quad \lfloor\text{ext Ax}\rfloor \end{array}$
- $\begin{array}{l} \bar{H}_1\,,\; f\!:\!S{\rightarrow}T\,,\; \bar{H}_2\;\vdash\; C \quad \lfloor\text{ext } e[f\,s/y]\rfloor \quad \textbf{by}\ \texttt{functionElim}\ i\ y \\ \quad\quad \bar{H}_1\,,\; f\!:\!S{\rightarrow}T\,,\; \bar{H}_2\;\vdash\; S \quad \lfloor\text{ext } s\rfloor \\ \quad\quad \bar{H}_1\,,\; f\!:\!S{\rightarrow}T\,,\; \bar{H}_2\,,\; y\!:\!T\;\vdash\; C \quad \lfloor\text{ext } e\rfloor \end{array}$
- $\begin{array}{l} \bar{H}\;\vdash\; S{\rightarrow}T\ \texttt{type} \quad \lfloor\text{ext Ax}\rfloor \quad \textbf{by}\ {\rightarrow}\texttt{-R} \\ \quad\quad \bar{H}\;\vdash\; S\ \texttt{type} \quad \lfloor\text{ext Ax}\rfloor \\ \quad\quad \bar{H}\;\vdash\; T\ \texttt{type} \quad \lfloor\text{ext Ax}\rfloor \end{array}$
- $\begin{array}{l} \bar{H}\;\vdash\; \lambda x\,.\,e\ \texttt{=}\ \lambda x'\,.\,e'\ \in\ S{\rightarrow}T \quad \lfloor\text{ext Ax}\rfloor \quad \textbf{by}\ \lambda\texttt{-Eq} \\ \quad\quad \bar{H}\,,\; x\!:\!S\;\vdash\; e\,\texttt{=}\,e'[x/x']\ \in\ T \quad \lfloor\text{ext Ax}\rfloor \\ \quad\quad \bar{H}\;\vdash\; S\ \texttt{type} \quad \lfloor\text{ext Ax}\rfloor \end{array}$
- $\begin{array}{l} \bar{H}\;\vdash\; f\ \ e\ \texttt{=}\ f'\ \ e'\ \in\ T \quad \lfloor\text{ext Ax}\rfloor \quad \textbf{by}\ \texttt{app-Eq}\ S{\rightarrow}T \\ \quad\quad \bar{H}\;\vdash\; f\texttt{=}f'\ \in\ S{\rightarrow}T \quad \lfloor\text{ext Ax}\rfloor \\ \quad\quad \bar{H}\;\vdash\; e\texttt{=}e'\ \in\ S \quad \lfloor\text{ext Ax}\rfloor \end{array}$
- $\begin{array}{l} \bar{H}\;\vdash\; (\lambda x\,.\,e)\ \ e'\ \texttt{=}\ e^*\ \in\ T \quad \lfloor\text{ext Ax}\rfloor \quad \textbf{by}\ \texttt{compute 1} \\ \quad\quad \bar{H}\;\vdash\; e'[e/x]\ \texttt{=}\ e^*\ \in\ T \quad \lfloor\text{ext Ax}\rfloor \end{array}$

Towards "real" mathematics

The proof theory that we have presented here is theoretically clean and small enough to be proven sound and complete with respect to a (formal version) Martin-Löf semantics of natural numbers, functions, and equality. However, for practical reasoning it is somewhat tedious to use, as many intuitively well-understood concepts must be added as defined concepts and standard reasoning patterns must be programmed as large tactics packages. Therefore NUPRL's type theory differs from this theory in the following ways.

1. Instead of a small theory of natural numbers consisting of the type $\mathbb{N}$, and the expressions 0, `suc(`e`)`, and `ind(`e`;` $base$`;` i`,`x`.`up`)`, NUPRL's type theory is based on integers.

   $\mathbb{Z}$ is the basic type.

   All integers `0, 1, -1, 2, -2, ...` in decimal representation are canonical elements.

   The standard operations `-`e, e`+`e', e`-`e', e`*`e', $e\div e'$, e `rem` e' are predefined.

   Induction `ind(`e`;` i`,`x`.`$down$`;` $base$`;` i`,`x`.`up`)` proceeds both upwards towards positive numbers and downwards towards the negative ones.

   There are two atomic tests `if` i`=`j `then` e `else` e' and `if` i`<`j `then` e `else` e'.

   Furthermore, the less-than relation i`<`j is a type in the same way as equality is, with `Ax` as its only element. The other inequalities can easily be defined through that type.

   The proof theory provides formation and equality rules for all these concepts and in addition to that a decision procedure `arith` that can decide all arithmetical propositions that involve `+`, `-`, `*`, $\div$, the equality and less-than relations, and a limited form of substitution.

2. For the equality type, the symmetry rule is replaced by a decision procedure `equality` that is capable of reasoning about symmetry, transitivity, and limited substitution.

3. To support the evaluation of terms, NUPRL provides a `direct_computation` rule, that allows a user to tag subterms of an expressionand to advise the system on how many evaluation steps it shall perform on that term.

   $\bar{H} \vdash C$ $\lfloor$ext $e\rfloor$ **by** `direct_computation` $taggedC$

   $\quad\bar{H} \vdash C{\downarrow_{taggedC}}$ $\lfloor$ext $e\rfloor$

   In this rule $taggedC$ denotes a variant of C where certain subterms have been replaced by their tagged version. To tag an expression e, one enters the object identifier `tag` into a term slot, which creates a term `[[natural]:[Term]]`, copies e into the slot marked by `[Term]` and enters a natural number into the other slot. Editing mechanisms for doing this in-place are being provided.

   Similar rules exist for computations within a hypothesis and for reverse computations. Proving the direct computation rules consisten with respect to the remaining theory was one of the greatest challenges.

All these extensions make formal reasoning more feasible and are a prerequisite for serious applications of automated reasoning.

Remarks: