Program Refinement Logics

Daniel Gratzer

July 3, 2017

Outline

- The Underling Type Theory of Nuprl
- Building a Proof Assistant for Type Theory
- A Guided Exploration of `miniprl`

Type Theory

- Nuprl's underlying type theory is adapted from an early version of MLTT.
- It's built around an intended model which the proof theory is subordinate to.

Type Theory

- Nuprl's underlying type theory is adapted from an early version of MLTT.
- It's built around an intended model which the proof theory is subordinate to.

What is this intended model?

Type Theory

- The model is a mathematical formalization of Martin-Löf's meaning explanation.
- Intended to be a mathematization of the intuitionistic philosophy.
- The model presupposes some underlying programming language $\mathcal{L}$ and an evaluation relation on it $- \Downarrow -$.
- $\mathcal{L}$ will include all of the terms and types we intend to use for our type theory.

Type Theory

- The model is a mathematical formalization of Martin-Löf's meaning explanation.
- Intended to be a mathematization of the intuitionistic philosophy.
- The model presupposes some underlying programming language $\mathcal{L}$ and an evaluation relation on it $- \Downarrow -$.
- $\mathcal{L}$ will include all of the terms and types we intend to use for our type theory.

The fundamental and primitive nature of computation is what distinguishes computational type theory from the proof-theoretic approach found in Coq, Agda, etc

Type Theory: What is a Type?

To *know* that a canonical form A is a type is to *know* its canonical inhabitants and when they're equal.

To *know* that $A = B$ type is to *know* that they judge the same canonical forms to be equal.

Type Theory: What is a Type?

To *know* that a canonical form A is a type is to *know* its canonical inhabitants and when they're equal.

To *know* that $A = B$ type is to *know* that they judge the same canonical forms to be equal.

We have a PER on $\mathcal{L}$ for types $\approx$ and a $\approx$-indexed PER on values $\approx_{-}$.

$$A \text{ type} \triangleq A \approx A \qquad A = B \text{ type} \triangleq A \approx B$$

Type Theory: What is Typehood?

To know that $a \in A$ presupposing that A is a type is to know that $a \Downarrow v$ so that v is a canonical witness of A

Type Theory: What is Typehood?

To know that $a \in A$ presupposing that A is a type is to know that $a \Downarrow v$ so that v is a canonical witness of A

To formalize this, we just use the $\approx$-indexed PER describing values:

$$e \in A \triangleq \exists v.\ e \Downarrow v \wedge v \approx_A v$$

$$e_1 = e_2 \in A \triangleq \exists v_1, v_2.\ e_1 \Downarrow v_1 \wedge e_2 \Downarrow v_2 \wedge v_1 \approx_A v_2$$

Type Theory: What is Typehood?

To know that $a \in A$ presupposing that A is a type is to know that $a \Downarrow v$ so that v is a canonical witness of A

To formalize this, we just use the $\approx$-indexed PER describing values:

$$e \in A \triangleq \exists v.\ e \Downarrow v \wedge v \approx_A v$$

$$e_1 = e_2 \in A \triangleq \exists v_1, v_2.\ e_1 \Downarrow v_1 \wedge e_2 \Downarrow v_2 \wedge v_1 \approx_A v_2$$

Notice that this is a *behavioral* condition. We make no assumptions or restrictions on the form that e takes.

Type Theory: Products

As an example, we can define products in this setup.

$$A_1 \times B_1 \approx A_2 \times B_2 \iff A_1 \approx A_2 \wedge B_1 \approx B_2$$

Type Theory: Products

As an example, we can define products in this setup.

$$A_1 \times B_1 \approx A_2 \times B_2 \iff A_1 \approx A_2 \wedge B_1 \approx B_2$$

The relation on terms is the standard logical relations construction.

$$v_1 \approx_{A \times B} v_2 \iff \pi_1(v_1) \approx_A \pi_1(v_2) \wedge \pi_2(v_1) \approx_B \pi_2(v_2)$$

Type Theory: Functions

Now let us define functions.

$$A_1 \to B_1 \approx A_2 \to B_2 \iff A_1 \approx A_2 \land B_1 \approx B_2$$

Type Theory: Functions

Now let us define functions.

$$A_1 \to B_1 \approx A_2 \to B_2 \iff A_1 \approx A_2 \wedge B_1 \approx B_2$$

The relation on terms is the standard logical relations construction.

$$v_1 \approx_{A \to B} v_2 \iff \forall a_1 \approx_A a_2.\ v_1(a_1) \approx_B v_2(a_2)$$

Type Theory: Generalizing to Open Terms

One generalizes to open terms similarly to logical relations.

$$x : B \gg e \in A \triangleq \forall v_1 \approx_B v_2.\ [v_1/x]e \approx_{[v_1/x]A} [v_2/x]e$$

Type Theory: Generalizing to Open Terms

One generalizes to open terms similarly to logical relations.

$$x : B \gg e \in A \triangleq \forall v_1 \approx_B v_2.\ [v_1/x]e \approx_{[v_1/x]A} [v_2/x]e$$

We demand that all terms respect the PER of types that they depend upon.

Type Theory: Growing a Type System

- By defining $\approx$ and $\approx_{-}$ we can quickly define a type system as a family of PERs on $\mathcal{L}$.
- There are various methods (Allen, Harper, Angiuli et al) for closing these type systems under all type formers.

Type Theory: Monotone Operators

- Type systems as a pair $(\approx, \approx_{-})$ form a complete lattice.
- We can define monotone operators on this lattice which add the appropriate type operators.
- We take the fixed point of the operator adding all the type formers we want.

Type Theory: Properties of CTT

- Canonicity and other results are immediate *by definition* of the type system.
- Correspondingly, most of the work is showing the validity of the definition, not proving metatheorems.

Type Theory: Stepping Back for a Minute

- Thus far we have only defined a semantics for our type theory.
- For the sake of usability and convenience, we wish to define various rules known to be validated by the semantics.
- Since the definition of our type system is $\approx$ and $\approx_{-}$, these rules are entirely supplementary.

Type Theory: Stepping Back for a Minute

- Thus far we have only defined a semantics for our type theory.
- For the sake of usability and convenience, we wish to define various rules known to be validated by the semantics.
- Since the definition of our type system is $\approx$ and $\approx_{-}$, these rules are entirely supplementary.

How should a proof assistant implement this?

Implementing CTT

We need to have some way of checking the judgment $\Gamma \gg e \in A$

Implementing CTT

We need to have some way of checking the judgment $\Gamma \gg e \in A$

- The quantifier complexity is potentially arbitrarily high!
- Clearly we need to carve out a reasonably behaved subset of this judgment that we can implement.

Implementing CTT: Coq versus a PRL

There are two ways that we could go with this.

Implementing CTT: Coq versus a PRL

There are two ways that we could go with this.

- We could carve out a decidable subset of this judgment.
    - This subset would satisfy Gentzen's inversion principles.
    - Decidability leads us into the well-studied area of Coq-like type theories.

Implementing CTT: Coq versus a PRL

There are two ways that we could go with this.

- We could carve out a decidable subset of this judgment.
  - This subset would satisfy Gentzen's inversion principles.
  - Decidability leads us into the well-studied area of Coq-like type theories.
- In a PRL, we *don't* construct an implementation the judgment $\Gamma \gg e \in A$.
- Instead we implement $\Gamma \gg A \rightsquigarrow e$.

Implementing CTT: Program Synthesis

- In an implementation of a PRL, the user sees only Γ and A of $\Gamma \gg A \rightsquigarrow e$.
- e is mechanically *extracted*.
- There is no constraint that the rules governing the manipulation of $\Gamma \gg A \rightsquigarrow e$ is decidable.
- The only constraint we impose on $\Gamma \gg A \rightsquigarrow e$ is that it implies $\Gamma \gg e \in A$.

Implementing CTT: Components of the Implementation

We are now in a position to state the main components of our implementation of a PRL

1. The *goal*: $\Gamma \gg A$

Implementing CTT: Components of the Implementation

We are now in a position to state the main components of our implementation of a PRL

1. The *goal*: $\Gamma \gg A$
2. The *tactic* script, $\mathcal{T}$, describing how to solve the goal.

Implementing CTT: Components of the Implementation

We are now in a position to state the main components of our implementation of a PRL

1. The *goal*: $\Gamma \gg A$
2. The *tactic* script, $\mathcal{T}$, describing how to solve the goal.
3. The *derivation*, $\mathcal{D}$, produced by the tactic script.

Implementing CTT: Components of the Implementation

We are now in a position to state the main components of our implementation of a PRL

1. The *goal*: $\Gamma \gg A$
2. The *tactic* script, $\mathcal{T}$, describing how to solve the goal.
3. The *derivation*, $\mathcal{D}$, produced by the tactic script.
4. The *extract*, e, produced from $\mathcal{D}$.

Implementing CTT: Components of the Implementation

We are now in a position to state the main components of our implementation of a PRL

1. The *goal*: $\Gamma \gg A$
2. The *tactic* script, $\mathcal{T}$, describing how to solve the goal.
3. The *derivation*, $\mathcal{D}$, produced by the tactic script.
4. The *extract*, e, produced from $\mathcal{D}$.

This is chosen so that if $\mathcal{T} \rightsquigarrow \mathcal{D}$ and $\mathcal{D}$ extracts to e then $\Gamma \gg A \rightsquigarrow e$ holds and therefore $\Gamma \gg e \in A$.

Implementing CTT: Some Essential Differences

- Attempting to prove a goal does *not* require that it is a priori sensible.
- The user of a PRL has no explicit interaction with proof terms.
- Proof complexity is not indicative of realizer complexity.

Implementing CTT: Aren't Derivations just Proof Terms?

Derivations seem like the obvious corresponding object to Coq-style proof terms. However there are important differences.

- Derivations are *not* necessary; in Nuprl they're an implementation detail!
- Derivations usually reflect a far more intensional structure of the tactic script.
- They are mainly useful for independently checkable proofs and niceties in the implementation.

Implementing CTT: Aren't Derivations just Proof Terms?

Derivations seem like the obvious corresponding object to Coq-style proof terms. However there are important differences.

- Derivations are *not* necessary; in Nuprl they're an implementation detail!
- Derivations usually reflect a far more intensional structure of the tactic script.
- They are mainly useful for independently checkable proofs and niceties in the implementation.

In short, really the main objects of concern are the goals and the extracts. They are what have type-theoretic significance.

Implementing CTT: What Advantages Does This Possess

- We've divorced the notion of proof from realizer.
- There is no reason why our realizers cannot possess different structure than our proofs!
- Generally speaking, CTT is a pleasant setting for working with subset types, quotient types, and other things that lead to unpleasantness in other settings.

Implementing CTT: What Disadvantages Does This Possess

- We've divorced the notion of proof from realizer.
- Since all of the onus in establishing the validity of a goal is placed on the user, tedious goals can arise quite easily.

`miniprl`

Let's step away from the theory to a concrete implementation: `miniprl`.