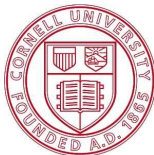


Towards a Formally Verified Proof Assistant

Abhishek Anand Vincent Rahli



July 10, 2016

Our initial motivation

How do we know that our systems are sound? How do we safely extend them?

Our initial motivation

How do we know that our systems are sound? How do we safely extend them?

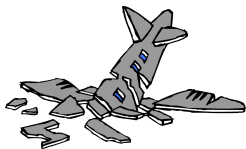
Formalization of air traffic controllers

Formal verification of banking protocols

Our initial motivation

How do we know that our systems are sound? How do we safely extend them?

Formalization of air traffic controllers



Formal verification of banking protocols



Our initial motivation

How do we know that our systems are sound? How do we safely extend them?

- ▶ Proofs mostly carried out on paper.
- ▶ Not carried out in full detail.
- ▶ Spread over several papers/PhD theses.
- ▶ Precise metatheory, precise account of Nuprl.
- ▶ No better way than using a proof assistant.

Our initial motivation

Agda & Coq

➤ 2013/2014: bug in their termination checker.

Our initial motivation

Agda & Coq

↪ 2013/2014: bug in their termination checker.

Nuprl

Inconsistencies related to types and rules, e.g.,

- ▶ Mendler's recursive type,
- ▶ LEM is inconsistent with Base

How can we be sure that these rules are valid?

Our initial motivation

Agda & Coq

↪ 2013/2014: bug in their termination checker.

Nuprl

Inconsistencies related to types and rules, e.g.,

- ▶ Mendler's recursive type,
- ▶ LEM is inconsistent with Base

How can we be sure that these rules are valid?

Nuprl's PER semantics in Coq (and Agda).

Mechanization and Experimentation!

Mechanization



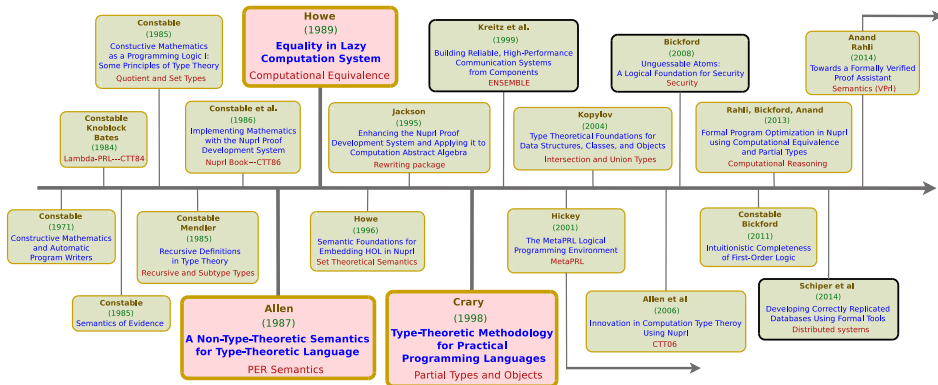
- Less error prone
- Easier to propagate changes
- Positive feedback loop
- Additive

Experimentation



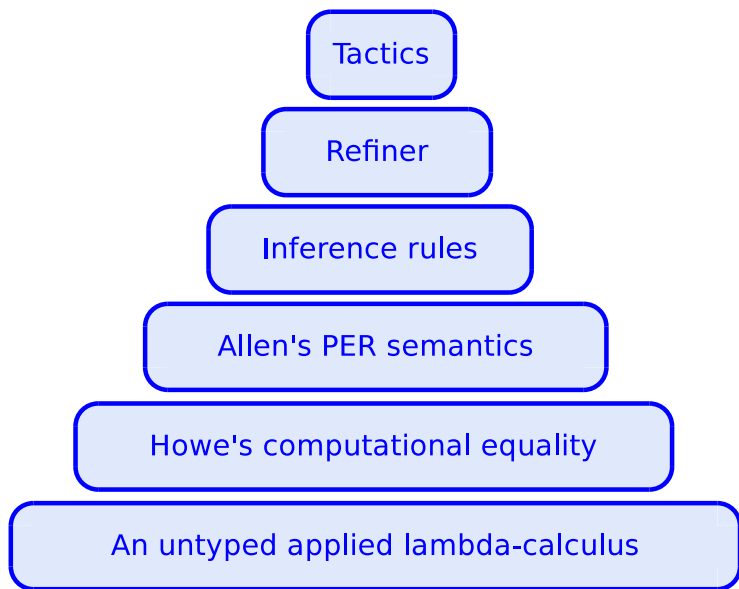
- Adding new computations
- Adding new types
- Exploring type theory
- Changing the theory

What do we cover?



Stuart Allen had his own meta-theory that was meant to be meaningful on its own and needs not be framed into type theory. We chose to use Coq and Agda.

Nuprl — Stack



Nuprl — Environment

Distributed

Structure editor

Tactic language: Classic ML

Runs in the cloud

Shared library

Nuprl — Types

Equality: $a = b \in T$

Dependent function: $a:A \rightarrow B[a]$

Dependent product: $a:A \times B[a]$

Universe: \mathbb{U}_i

Nuprl — Types

Partial: \bar{A}

Intersection: $\cap_{a:A}.B[a]$

Subset: $\{a : A \mid B[a]\}$

Computational equivalence: $t_1 \sim t_2$

Image: $\text{Img}(A, f)$

PER: $\text{per}(R)$, with R a partial equivalence relation.

Nuprl — Types

- Rich type language facilitates specification
- Makes type checking harder

Inductive types?

➤ Using W types.

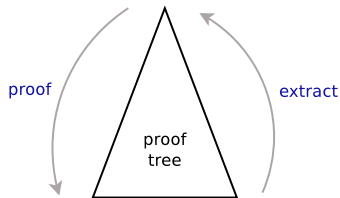
- In Nuprl, we used to define inductive types using Mendler's recursive types. PER semantics?
- We now use Brouwer's bar induction rule to define W types. Validity?

Nuprl — Trusted core

Nuprl's proof engine is called a refiner.

A generic goal directed reasoner:

- a rule interpreter
- a proof manager

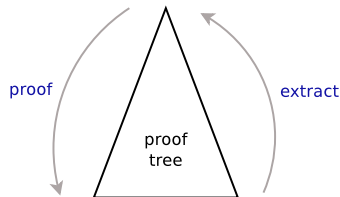


Nuprl — Trusted core

Nuprl's proof engine is called a refiner.

A generic goal directed reasoner:

- a rule interpreter
- a proof manager

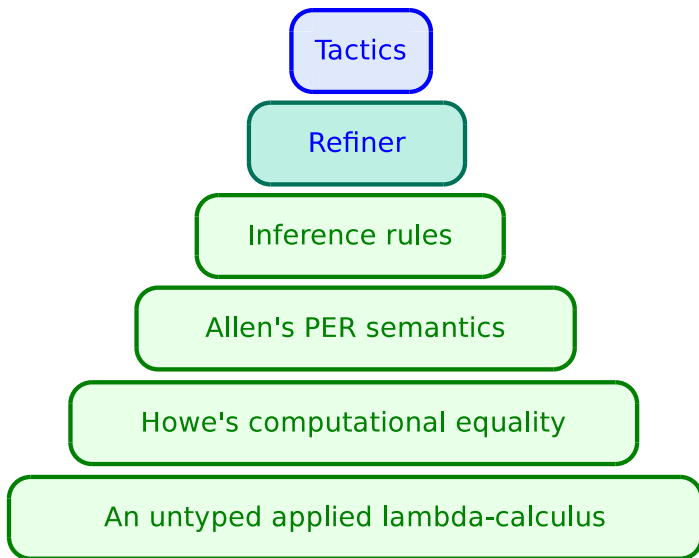


Parameterized by a collection of rules

➤ We proved that Nuprl's rules are valid

➤ Next step is to build a verified refiner

What we implemented in Coq



What we implemented in Coq

Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped applied lambda-calculus



An untyped lambda-calculus

A “nominal” approach:

```
Inductive NTerm : Set :=  
| vterm: NVar → NTerm  
| oterm: Opid → list BTerm → NTerm  
with BTerm : Set :=  
| bterm: list NVar → NTerm → BTerm.
```

For example:

```
oterm (Can NLambda) [bterm [nvar 0] (vterm (nvar 0))]
```

represents a λ -term of the form $\lambda x.x$.

An untyped lambda-calculus

We have the usual computation rules

with a β -reduction rule, pair and injection destructors, a call-by-value operator, a fix operator, exceptions, ...

Provides a generic framework for defining and reasoning about programming languages using a “nominal” style

➤ See **Abhishek's LFMTTP talk on Thursday**

What we have to implement

Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped applied lambda-calculus



What we have to implement

Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped applied lambda-calculus



Howe's computational equality

One can think of `approx` as the greatest fixpoint of the following operator on binary relations:

Definition `close_compute`

$$(R : \text{NTerm} \rightarrow \text{NTerm} \rightarrow \text{Type})$$
$$(a \ b : \text{NTerm}) : \text{Type} :=$$

`programs` [a , b]

$\times \forall (c : \text{CanonicalOp}) (as : \text{list BTerm}),$

$a \Downarrow \text{oterm} (\text{Can } c) \ as$

$\rightarrow \{bs : \text{list BTerm}$

$\ \& (b \Downarrow \text{oterm} (\text{Can } c) \ bs)$

$\ \times \text{lift} (\text{olift } R) \ as \ bs \}$.

Howe's computational equality

One would like to define

```
CoInductive approx (a b : NTerm) : Type :=  
| approx_fold: close_compute approx a b → approx a b.
```

Unfortunately, because of `cofix`'s conservative productivity checking, we had to use parametrized coinduction.

Definition `cequiv a b := approx a b × approx b a`.

`approx (\preceq)` and `cequiv (\sim)` are congruences

Constructive domain theory (Crary)

Let \perp be `fix`($\lambda x.x$).

Constructive domain theory (Crary)

Let \perp be $\text{fix}(\lambda x.x)$.

Least element

$\forall t. \text{approx } \perp t.$

Constructive domain theory (Crary)

Let \perp be $\text{fix}(\lambda x.x)$.

Least element

$\forall t. \text{approx } \perp t.$

Least upper bound principle

$\forall G f. G(\text{fix}(f))$ is the lub of the (approx) chain $G(f^n(\perp))$ for $n \in \mathbb{N}$.

Constructive domain theory (Crary)

Let \perp be $\text{fix}(\lambda x.x)$.

Least element

$\forall t. \text{approx } \perp t.$

Least upper bound principle

$\forall G f. G(\text{fix}(f))$ is the lub of the (**approx**) chain $G(f^n(\perp))$ for $n \in \mathbb{N}$.

Compactness

if $G(\text{fix}(f))$ converges, then there exists a natural number n such that $G(f^n(\perp))$ converges.

What we have to implement

Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped applied lambda-calculus



What we have to implement

Tactics

Refiner

Inference rules

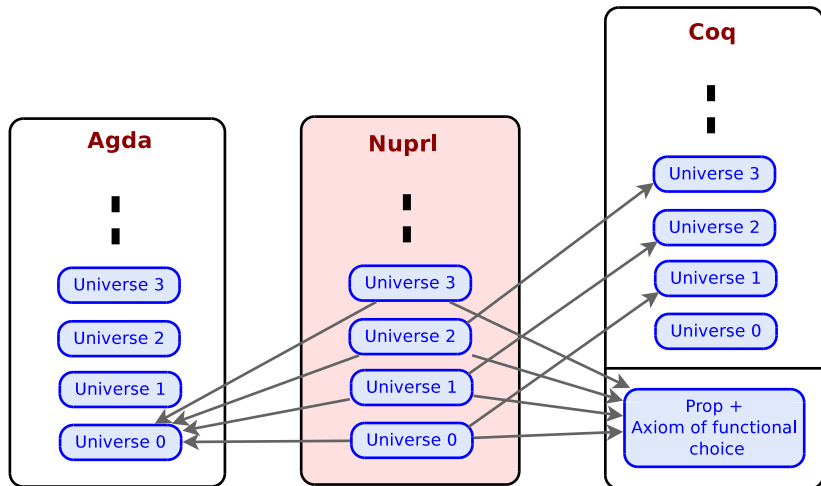
Allen's PER semantics

Howe's computational equality

An untyped applied lambda-calculus



Allen's PER semantics



Allen's PER semantics

$$f_1 \equiv f_2 \in x:A \rightarrow B$$

$$\text{type}((x:A \rightarrow B)) \wedge \forall a_1, a_2. a_1 \equiv a_2 \in A \Rightarrow \\ f_1(a_1) \equiv f_2(a_2) \in B[x \setminus a_1]$$

Allen's PER semantics

$$f_1 \equiv f_2 \in x:A \rightarrow B$$

$$\text{type}((x:A \rightarrow B)) \wedge \forall a_1, a_2. a_1 \equiv a_2 \in A \Rightarrow f_1(a_1) \equiv f_2(a_2) \in B[x \setminus a_1]$$

$$t_1 \equiv t_2 \in \text{Base}$$

$$t_1 \sim t_2$$

$$Ax \equiv Ax \in (a = b \in A)$$

$$\text{type}((a = b \in A)) \wedge a \equiv b \in A$$

$$t_1 \equiv t_2 \in \bar{A}$$

$$\text{type}((\bar{A})) \wedge (t_1 \Downarrow \iff t_2 \Downarrow) \wedge (t_1 \Downarrow \Rightarrow t_1 \equiv t_2 \in A)$$

Allen's PER semantics

$$x_1:A_1 \rightarrow B_1 \equiv x_2:A_2 \rightarrow B_2$$

$$A_1 \equiv A_2 \wedge \forall a_1, a_2. a_1 \equiv a_2 \in A_1 \Rightarrow B_1[x_1 \setminus a_1] \equiv B_2[x_2 \setminus a_2]$$

Allen's PER semantics

$$x_1:A_1 \rightarrow B_1 \equiv x_2:A_2 \rightarrow B_2$$

$$A_1 \equiv A_2 \wedge \forall a_1, a_2. a_1 \equiv a_2 \in A_1 \Rightarrow B_1[x_1 \setminus a_1] \equiv B_2[x_2 \setminus a_2]$$

$$\text{Base} \equiv \text{Base}$$

$$(a_1 = a_2 \in A) \equiv (b_1 = b_2 \in B)$$

$$A \equiv B \wedge (a_1 \equiv b_1 \in A \vee a_1 \sim b_1) \wedge (a_2 \equiv b_2 \in A \vee a_2 \sim b_2)$$

$$\overline{A} \equiv \overline{B}$$

$$A \equiv B \wedge (\forall a. a \in A \Rightarrow a \Downarrow)$$

Allen's PER semantics

This definition can be made formal using induction-recursion

Simple induction mechanisms such as in Coq are not enough

↪ Definition is non-strictly-positive

Allen suggests that the definition should be valid because it is “half-positive” (achieved by induction-recursion)

Instead of using induction-recursion, Allen defines ternary relations between types and equalities

↪ Translation of a mutually inductive-recursive definition to a single inductive definition (Capretta).

Allen's PER semantics

Ternary relations

candidate type systems:

$$cts = CTerm \rightarrow CTerm \rightarrow per \rightarrow Univ$$

where $per = CTerm \rightarrow CTerm \rightarrow Univ$

Allen's PER semantics

Ternary relations

candidate type systems:

$$\text{cts} = \text{CTerm} \rightarrow \text{CTerm} \rightarrow \text{per} \rightarrow \text{Univ}$$

where $\text{per} = \text{CTerm} \rightarrow \text{CTerm} \rightarrow \text{Univ}$

Type constructors

Definition $\text{per_function} (ts : \text{cts}) : \text{cts} := \dots$

Allen's PER semantics

Ternary relations

candidate type systems:

$$\text{cts} = \text{CTerm} \rightarrow \text{CTerm} \rightarrow \text{per} \rightarrow \text{Univ}$$

where $\text{per} = \text{CTerm} \rightarrow \text{CTerm} \rightarrow \text{Univ}$

Type constructors

Definition $\text{per_function} (ts : \text{cts}) : \text{cts} := \dots$

Closure

Inductive $\text{close} (ts : \text{cts}) : \text{cts} := \dots$

Allen's PER semantics

Ternary relations

candidate type systems:

$$\text{cts} = \text{CTerm} \rightarrow \text{CTerm} \rightarrow \text{per} \rightarrow \text{Univ}$$

where $\text{per} = \text{CTerm} \rightarrow \text{CTerm} \rightarrow \text{Univ}$

Type constructors

Definition $\text{per_function} (ts : \text{cts}) : \text{cts} := \dots$

Closure

Inductive $\text{close} (ts : \text{cts}) : \text{cts} := \dots$

Universes

Fixpoint $\text{univ}_i (i : \text{nat}) : \text{cts} := \dots$

Allen's PER semantics

```
Fixpoint univi (i : nat) (T T' : CTerm) (eq : per) : Prop :=  
  match i with  
  | 0 => False  
  | S n =>  
    ...  
    eq  $\Leftarrow$ 2 $\Rightarrow$  (fun A A' => {eqa : per, close (univi n) A A' eqa})  
    ...  
end.
```

Has to be in **Prop**, otherwise we can only define a finite number of universes

Allen's PER semantics

Definition $\text{univ } T T' \text{ eq} := \{i : \text{nat} , \text{univ } i T T' \text{ eq}\}$.

Definition $\text{nuprl} := \text{close univ}$.

$t_1 \equiv t_2 \in T = \{ \text{eq} : \text{per} , \text{nuprl } T T \text{ eq} \times \text{eq } t_1 t_2 \}$

$T \equiv T' = \{ \text{eq} : \text{per} , \text{nuprl } T T' \text{ eq} \}$

Interesting fact: $n : \mathbb{N} \rightarrow \mathbb{U}(n)$ is a Nuprl type

What we have to implement

Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped applied lambda-calculus



What we have to implement

Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped applied lambda-calculus



Inference rules

We verified over 70 rules

The more rules the better

- Expose more of the metatheory
- Encode Mathematical knowledge

Gives us the basis to formally define a refiner

What next?

Adding new types

Adding new computations

Write a parser

Build a proof assistant

What about Mendler's
recursive types?

Extend our formalization with
a library of definitions

Build a verified refiner

Type checker/type inferencer?

Implement Allen's semantics
of Atoms

What can you do with it?