Towards a Formally Verified Proof Assistant

Abhishek Anand Vincent Rahli

CORNELL UNIVERSITY FOUNDED A.D. 1865

July 11, 2014

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?

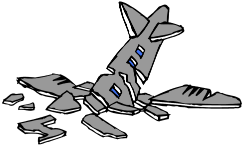
Formalization of air traffic controllers

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!

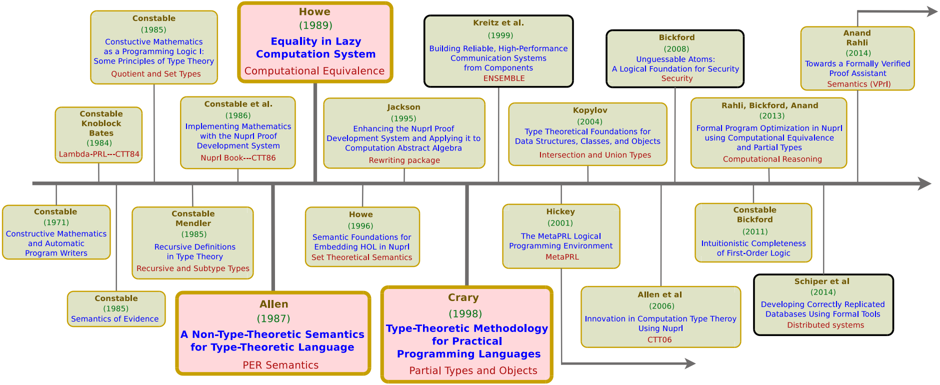
Mechanization

- ➲ 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?



Constable
(1971)
Constructive Mathematics and Automatic Program Writers

Constable Knoblock Bates
(1984)
Lambda-PRL---CTT84

Constable
(1985)
Constuctive Mathematics as a Programming Logic I: Some Principles of Type Theory
Quotient and Set Types

Constable Mendler
(1985)
Recursive Definitions in Type Theory
Recursive and Subtype Types

Constable
(1985)
Semantics of Evidence

Constable et al.
(1986)
Implementing Mathematics with the Nuprl Proof Development System
Nuprl Book---CTT86

Allen
(1987)
A Non-Type-Theoretic Semantics for Type-Theoretic Language
PER Semantics

Howe
(1989)
Equality in Lazy Computation System
Computational Equivalence

Jackson
(1995)
Enhancing the Nuprl Proof Development System and Applying it to Computation Abstract Algebra
Rewriting package

Howe
(1996)
Semantic Foundations for Embedding HOL in Nuprl
Set Theoretical Semantics

Crary
(1998)
Type-Theoretic Methodology for Practical Programming Languages
Partial Types and Objects

Kreitz et al.
(1999)
Building Reliable, High-Performance Communication Systems from Components
ENSEMBLE

Hickey
(2001)
The MetaPRL Logical Programming Environment
MetaPRL

Kopylov
(2004)
Type Theoretical Foundations for Data Structures, Classes, and Objects
Intersection and Union Types

Allen et al
(2006)
Innovation in Computation Type Thery Using Nuprl
CTT06

Bickford
(2008)
Unguessable Atoms: A Logical Foundation for Security
Security

Constable Bickford
(2011)
Intuitionistic Completeness of First-Order Logic

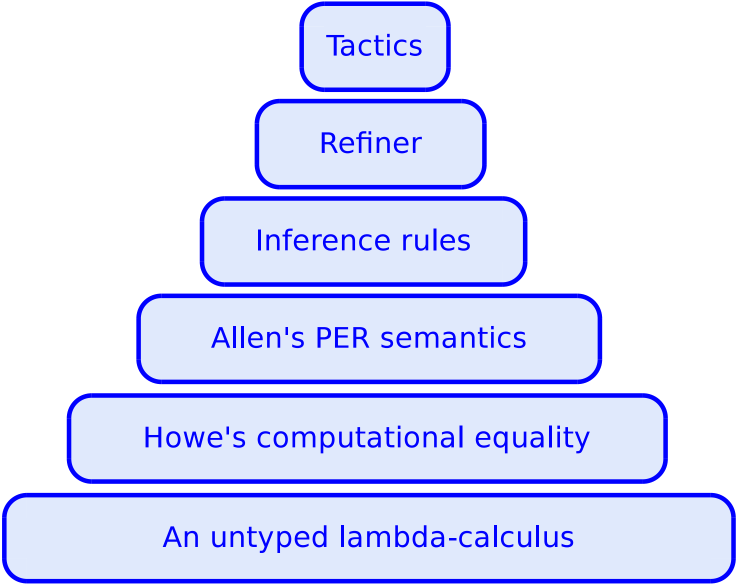
Rahli, Bickford, Anand
(2013)
Formal Program Optimization in Nuprl using Computational Equivalence and Partial Types
Computational Reasoning

Schiper et al
(2014)
Developing Correctly Replicated Databases Using Formal Tools
Distributed systems

Anand Rahli
(2014)
Towards a Formally Verified Proof Assistant
Semantics (VPrl)

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



Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped lambda-calculus

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: $\overline{A}$

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

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

Computational equivalence: $t_1 \sim t_2$

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

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

Nuprl — Types

- **Rich type language facilitates specification**
- **Makes type checking harder**

Inductive types?

- **Using W types.**

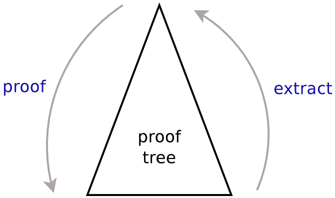
- In Nuprl, we used to define inductive types using Mendler's recursive types. PER semantics?
- We now use Brouwer's bar induction rule to define W types. Validity?

Nuprl — Trusted core

Nuprl's proof engine is called a refiner.

A generic goal directed reasoner:

- **a rule interpreter**
- **a proof manager**

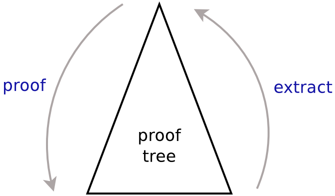


Nuprl — Trusted core

Nuprl's proof engine is called a refiner.

A generic goal directed reasoner:

- **a rule interpreter**
- **a proof manager**

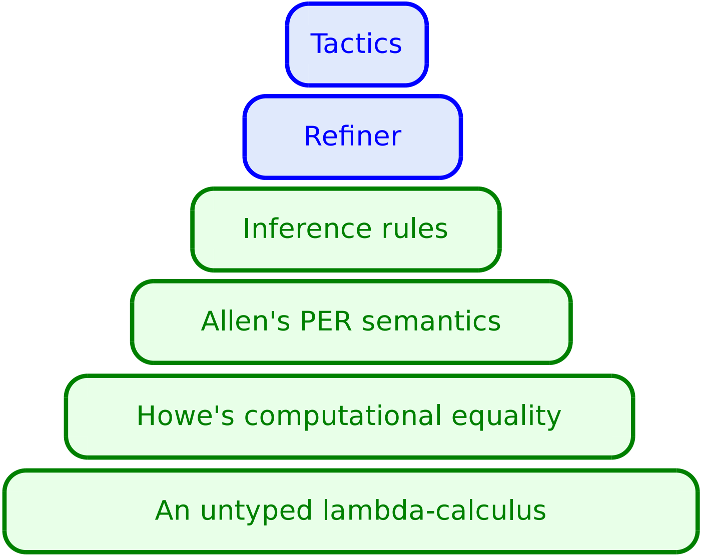


Parameterized by a collection of rules

- **We proved that Nuprl's rules are valid**

- **Next step is to build a verified refiner**

What we implemented in Coq



Tactics

Refiner

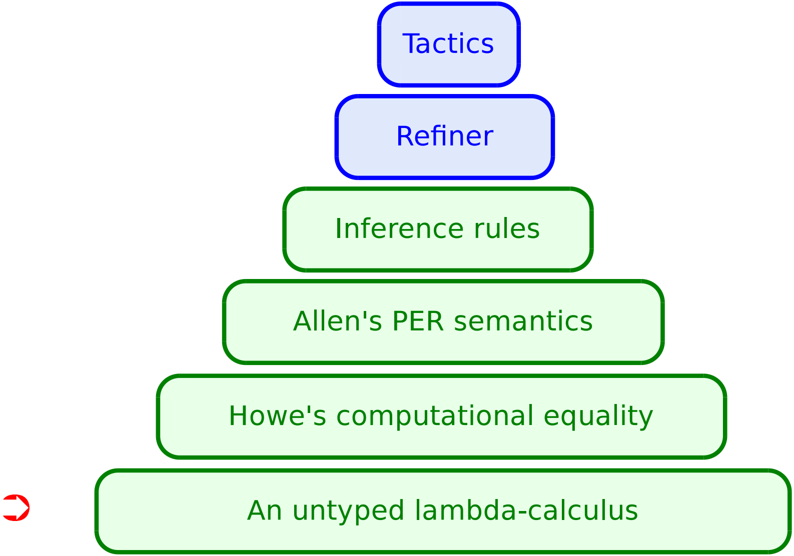
Inference rules

Allen's PER semantics

Howe's computational equality

An untyped lambda-calculus

What we implemented in Coq



Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

➲ An untyped 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

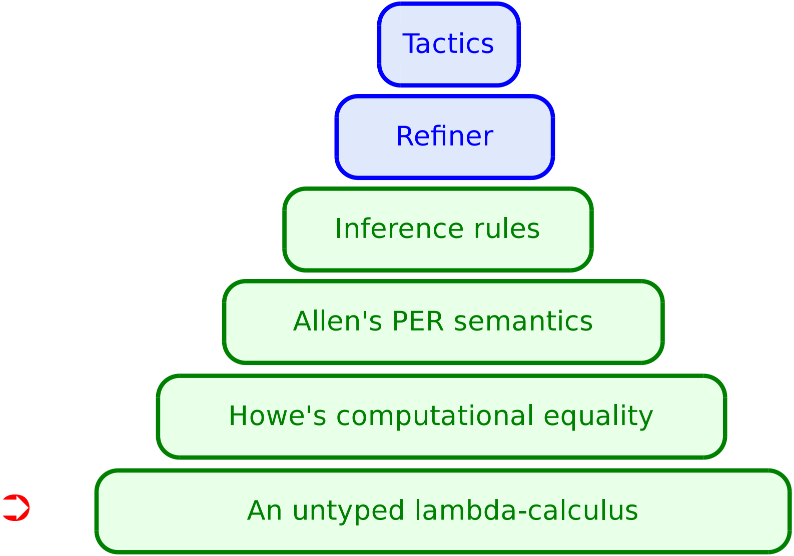
We have the usual computation rules

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 LFMTP talk on Thursday**

What we have to implement



Tactics

Refiner

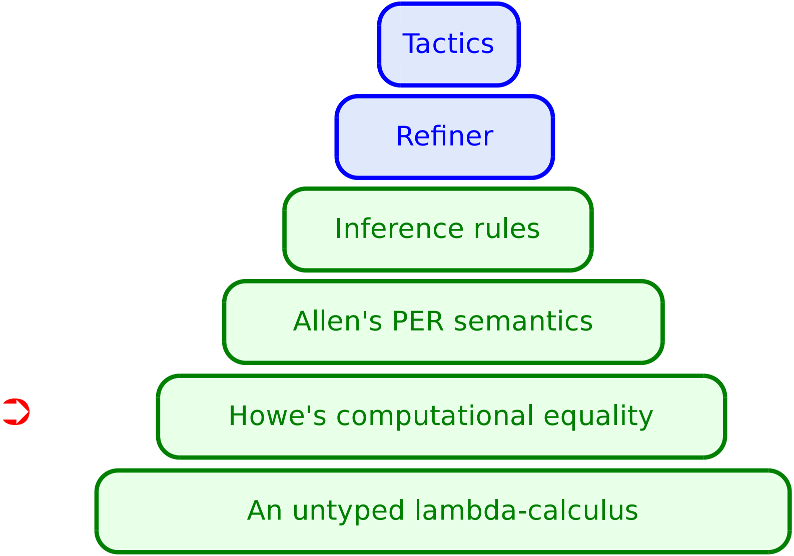
Inference rules

Allen's PER semantics

Howe's computational equality

An untyped lambda-calculus

What we have to implement



Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped 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 : NTerm→ NTerm →Type)
              (a b : NTerm) : Type :=
programs [ a, b ]
× ∀ (c : CanonicalOp) (as : list BTerm),
  a ⇓ oterm (Can c) as
  → {bs : list BTerm
      & (b ⇓ oterm (Can c) bs)
      × lblift (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 $\mathtt{fix}(\lambda x.x)$.

Constructive domain theory (Crary)

Let $\bot$ be $\texttt{fix}(\lambda x.x)$.

Least element

$\forall t.$ approx $\bot$ $t.$

Constructive domain theory (Crary)

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

Least element

$\forall t.$ approx $\perp$ t.

Least upper bound principle

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

Constructive domain theory (Crary)

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

Least element

$\forall t.$ approx $\perp$ t.

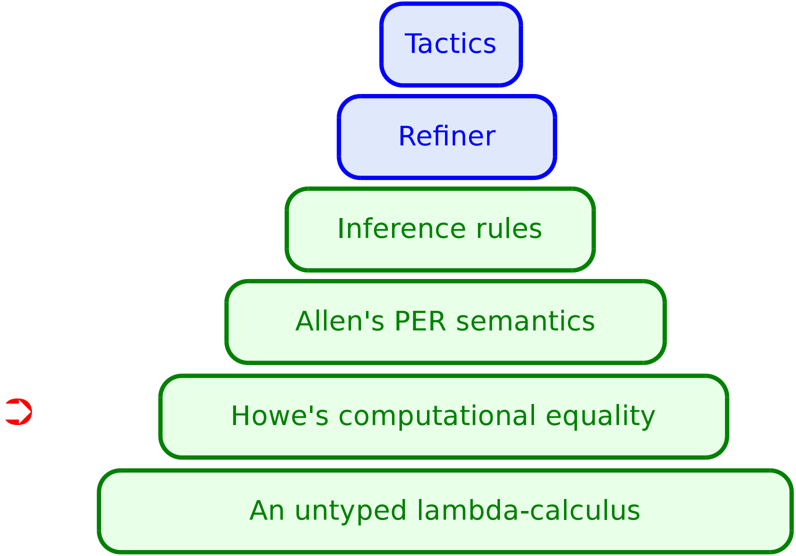
Least upper bound principle

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

Compactness

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

What we have to implement



Tactics

Refiner

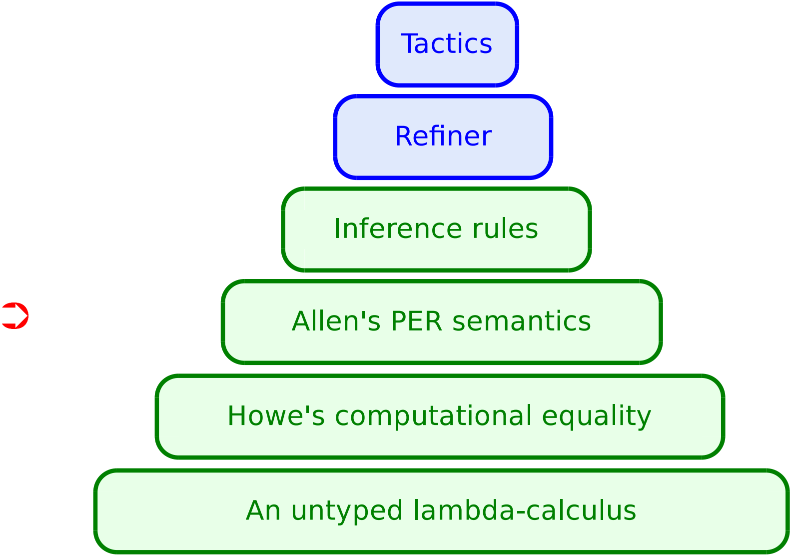
Inference rules

Allen's PER semantics

Howe's computational equality

An untyped lambda-calculus

What we have to implement



Tactics

Refiner

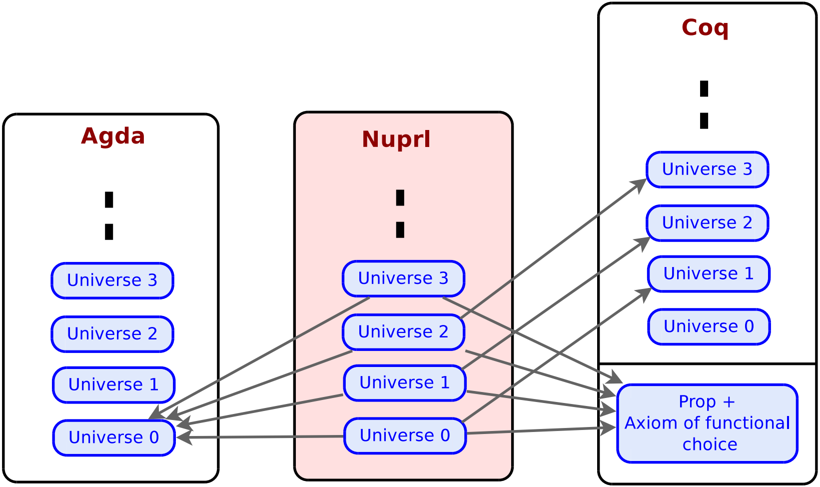
Inference rules

Allen's PER semantics

Howe's computational equality

An untyped lambda-calculus

Allen's PER semantics



Allen's PER semantics

$$f_1 \equiv f_2 \in x{:}A \to B$$

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

Allen's PER semantics

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

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

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

$$t_1 \sim t_2$$

$$\mathtt{Ax} \equiv \mathtt{Ax} \in (a = b \in A)$$

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

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

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

Allen's PER semantics

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

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

Allen's PER semantics

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

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

$$\mathtt{Base} \equiv \mathtt{Base}$$

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

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

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

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

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:

$$\mathtt{cts} = \mathrm{CTerm} \to \mathrm{CTerm} \to \mathrm{per} \to \mathtt{Univ}$$

where $\mathrm{per} = \mathrm{CTerm} \to \mathrm{CTerm} \to \mathtt{Univ}$

Allen's PER semantics

Ternary relations

candidate type systems:

$$\mathtt{cts} = \mathrm{CTerm} \to \mathrm{CTerm} \to \mathrm{per} \to \mathtt{Univ}$$

where $\mathrm{per} = \mathrm{CTerm} \to \mathrm{CTerm} \to \mathtt{Univ}$

Type constructors

$\mathtt{Definition}\ \mathrm{per\_function}\ (\mathit{ts} : \mathtt{cts}) : \mathtt{cts} := \ldots$

Allen's PER semantics

Ternary relations

candidate type systems:

$$\texttt{cts} = \mathrm{CTerm} \to \mathrm{CTerm} \to \mathrm{per} \to \texttt{Univ}$$

where $\mathrm{per} = \mathrm{CTerm} \to \mathrm{CTerm} \to \texttt{Univ}$

Type constructors

$\texttt{Definition}\ \mathrm{per\_function}\ (\mathit{ts} : \texttt{cts}) : \texttt{cts} := \ldots$

Closure

$\texttt{Inductive}\ \mathsf{close}\ (\mathit{ts} : \texttt{cts}) : \texttt{cts} := \ldots$

Allen's PER semantics

Ternary relations

candidate type systems:

$$\texttt{cts} = \mathrm{CTerm} \rightarrow \mathrm{CTerm} \rightarrow \mathrm{per} \rightarrow \texttt{Univ}$$

where $\mathrm{per} = \mathrm{CTerm} \rightarrow \mathrm{CTerm} \rightarrow \texttt{Univ}$

Type constructors

$\texttt{Definition}\ \mathrm{per\_function}\ (\mathit{ts} : \texttt{cts}) : \texttt{cts} := \ldots$

Closure

$\texttt{Inductive}\ \mathsf{close}\ (\mathit{ts} : \texttt{cts}) : \texttt{cts} := \ldots$

Universes

$\texttt{Fixpoint}\ \mathrm{univi}\ (i : \mathsf{nat}) : \texttt{cts} := \ldots$

Allen's PER semantics

```
Fixpoint univi (i : nat) (T T' : CTerm) (eq : per) : Prop :=
  match i with
  | 0 ⇒ False
  | S n ⇒
    ...
    eq ⇐2⇒ (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` univ $T\ T'\ eq := \{i : \mathsf{nat}\ ,\ \mathrm{univi}\ i\ T\ T'\ eq\}$.

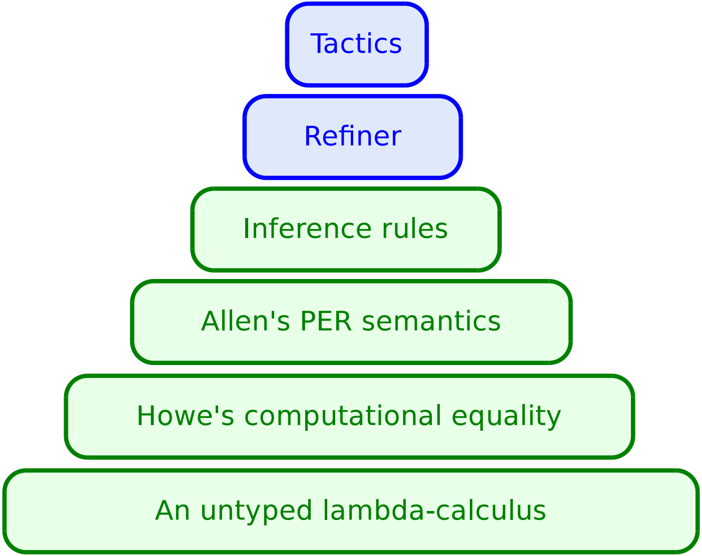
`Definition` nuprl := close univ.

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

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

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

What we have to implement



Tactics

Refiner

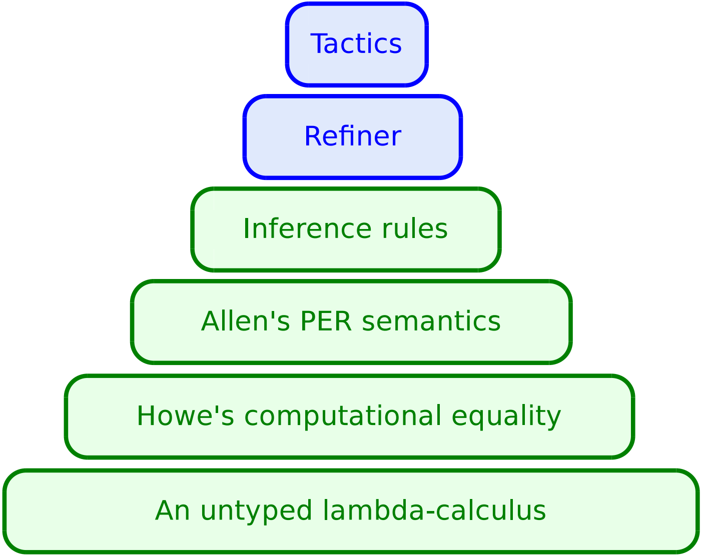
Inference rules

Allen's PER semantics

Howe's computational equality

An untyped lambda-calculus

What we have to implement



Tactics

Refiner

Inference rules

Allen's PER semantics

Howe's computational equality

An untyped 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

Extend our formalization with a library of definitions

Adding new computations

Build a verified refiner

Write a parser

Type checker/type inferencer?

Build a proof assistant

Implement Allen's semantics of Atoms

What about Mendler's recursive types?

What can you do with it?