Reasoning about Computational Systems using Abella

http://abella-prover.org

Kaustuv Chaudhuri[1]   Gopalan Nadathur[2]

[1]Inria & LIX/École polytechnique, France

[2]Department of Computer Science and Engineering
University of Minnesota, Minneapolis, USA

2015-08-02

Overview

Overview of Abella

Abella is an interactive tactics-based theorem prover for a logic with the following features

- its underlying substrate is an intuitionistic first-order logic over simply typed lambda terms
- it incorporates a mechanism for interpreting atoms through fixed-point definitions
- it allows for inductive and co-inductive forms of reasoning
- it includes logical devices for analyzing binding structure

Abella also builds in a special ability for reasoning about specifications expressed in a separate executable logic

Abella and Computational Systems

Abella offers intriguing capabilities for reasoning about syntax-directed and rule-based specifications

- such specifications can be formalized succinctly through fixed-point definitions
- formalizations adopt a natural and flexible relational style as opposed to a computational style
- the formalizations allow specifications to be interpreted either inductively or co-inductively in the reasoning process
- binding structure in object systems can be treated via a well-restricted and effective form of higher-order syntax
- a two-level logic approach allows intuitions about the object systems to be reflected into the reasoning process

Objectives for the Tutorial

We aim to accomplish at least the following goals through the tutorial

- to expose the novel features of the logic underlying Abella
- to provide a feel for Abella so that you will be able to (and interested in) experimenting with it on your own
- to show the applicability of Abella in mechanizing the meta-theory of formal systems
- to indicate the benefits of a special brand of higher-order abstract syntax in treating object-level binding structure

We will assume a basic familiarity with sequent-style logical systems and with intuitionistic logic

The Structure of the Tutorial

The tutorial will consists of the following conceptual parts

- an exposure to the syntax of formulas in Abella and the basic theorem proving environment
- a presentation of the special logical features of Abella with examples of their use
- an exposition of the two-level logic approach a la Abella to formalization and reasoning
- extensions to reasoning about specifications in a dependently typed lambda calculus

Outline

1. Setup
2. The Reasoning Logic $\mathcal{G}$
3. The Two-Level Logic Approach
4. Co-Induction
5. Extensions

Setup

How to Run Abella in your Web-Browser

Go to:

`http://abella-prover.org/try`

- Everything runs inside your browser
- Interface reminiscent of ProofGeneral

Running Abella Offline

- You will need a working OCaml toolchain + OPAM
- **`opam install abella`**
- To get ProofGeneral support, read the instructions on:

  `http://abella-prover.org/tutorial/`

Code for This Tutorial

`http://abella-prover.org/tutorial/try`

Special on-line version just for this tutorial

Some Concrete Syntax

Types	$A \to ((B \to C) \to D)$	`A -> (B -> C) -> D`
Application	$(M\ N)\ (J\ K)$	`M N (J K)`
Abstraction	$\lambda x.\ M$	`x\ M`
	$\lambda x{:}A.\ M$	`(x:A)\ M`
Formulas	$\top, \bot$	`true`, `false`
	$F \wedge G, \quad F \vee G$	`F /\ G,  F \/ G`
	$F \supset G$	`F -> G`
	$\forall x, y.\ F$	`forall x y, F`
	$\exists x{:}A, y.\ F$	`exists (x:A) y, F`
	$M = N$	`M = N`
	$\neg F$	`F -> false`

Declaring Basic Types and Term Constructors

- New basic types are introduced with **`Kind`** declarations.

```
Kind nat    type.
Kind bt     type.
Kind tm,ty  type.
```

Reserved: **`o`**, **`olist`**, and **`prop`**.

- New term constructors are introduced with **`Type`** declarations.

```
Type z       nat.
Type s       nat -> nat.

Type leaf    nat -> bt.
Type node    bt -> bt -> bt.

Type app     tm -> tm -> tm.
Type abs    (tm -> tm) -> tm.
```

Theorems and Proofs

$\boxed{\texttt{1 - Syntax}}$

The Reasoning Logic $\mathcal{G}$

The Reasoning Logic $\mathcal{G}$

Outline:

1. Ordinary Intuitionistic Logic
2. Equality
3. Fixed Point Definitions
4. Induction
    - Inductive data: lists
    - Kinds of induction: simple, mutual, nested
5. Higher-Order Abstract Syntax
    - Example: subject reduction for STLC

$\boxed{\texttt{2.1 - Basic Logic}}$

Equality

> *For closed terms M and N, the formula $M = N$ is true if and only if M and N are $\alpha\beta\eta$-convertible.*

Consequences

- Two closed first-order terms are equal iff they are identical.

```
Kind i     type.
Type a,b   i.

Theorem eq1 : a = a /\ b = b.
Theorem eq2 : a = b -> false.
```

- Different constants are distinct.

Equality

> *For closed terms M and N, the formula $M = N$ is true if and only if M and N are λ-convertible.*

Consequences

- Two closed first-order terms are equal iff they are identical.

```
Kind i     type.
Type a,b   i.

Theorem eq1 : a = a /\ b = b.
Theorem eq2 : a = b -> false.
```

- Different constants are distinct.

The Nature of Variables

Terminology: *variable*, *eigenvariable*, and *universal variable* used interchangably in Abella.

> *Variables are interpreted extensionally in the term model of the underlying logic.*

In other words, a variable stands for all its possible instances.

```
Kind nat   type.
Type z     nat.
Type s     nat -> nat.
```

The formula $\forall x{:}\texttt{nat}.\ F$ stands for:

$$[\texttt{z}/x]F \quad \wedge \quad [\texttt{s z}/x]F \quad \wedge \quad [\texttt{s (s z)}/x]F \quad \wedge \quad \cdots$$

Equality and Extensional Variables

```
forall (x:nat) y, x = y -> F x y
```

We have:

`x`	`y`	`x = y`	`x = y -> F x y`
`z`	`z`	`true`	`F z z`
`z`	anything else	`false`	`true`
`s z`	`s z`	`true`	`F (s z) (s z)`
`s z`	anything else	`false`	`true`

⋮

In other words, the formula is equivalent to:

```
forall (x:nat), F x x
```

Equality-Left

More generally, given an assumption `M = N`:

1. Find all unifiers for **M** and **N**.
   - A unifier of **M** and **N** is a subsitution of terms for the free variables of **M** and **N** that makes them λ-convertible.
2. For each unifier, apply the unifier to the rest of the subgoal to generate a new subgoal.

Notes:

- There may be infinitely many unifiers
- Unification in the general case is undecidable
- In practice we work with complete sets of unifiers (csu) that cover all possibilities; csus are often finite, even singletons.

Equality Assumptions on Open Terms

Example:

```
Kind i  type

Type f  i -> i -> i.
Type g  i -> i.

Theorem eq3 : forall x y z,
  f x (g y) = f (g y) z  ->  x = z.
```

- A csu of `f x (g y)` and `f (g y) z` is the singleton set $\{[$`(g y)`$/$`x`, `(g y)`$/$`z`$]\}$.
- This substitution turns `x = z` into `g y = g y`, which is `true`.

Equality Example: Peano’s Axioms

$\boxed{\texttt{2.2 - Peano}}$

Functions vs. Relations

Say you want to define addition on natural numbers.

- Functional approach:
  - Declare a new symbol:

```
Type sum   nat -> nat -> nat.
```

  - Define a closed set of computational rules:

```
Rule sum z N = N.
Rule sum (s M) N = s K where sum M N = K.
```

- Relational approach:
  - Declare a new predicate:

```
Type plus  nat -> nat -> nat -> prop.
```

  - Declare a closed set of properties of the predicate:

```
forall M, plus z M M.
forall M N K, plus M N K -> plus (s M) N (s K).
```

Functions vs. Relations

Functions	Relations
Modifies term language	No change to terms
Modifies equality	No change to equality
Requires confluence	Can be non-deterministic
Fixed inputs and output	Modes can vary
Functional programming	Logic programming

Relational Definitions

type of the relation

clauses

```
Define plus : nat -> nat -> nat -> prop by
  plus z N N ;
  plus (s M) N (s K) := plus M N K.
```

head

body

- All defined relations must have target type **`prop`**.
- Clauses are universally closed over the capitalized identifiers.
- The body implies the head in each clause.
- An omitted body stands for **`true`**.
- The set of clauses is closed.

Multiple Clauses vs. Single Clause

```
Define plus1 : nat -> nat -> nat -> prop by
  plus1 z N N ;
  plus1 (s M) N (s K) := plus1 M N K.
```

is equivalent to

```
Define plus2 : nat -> nat -> nat -> prop by
  plus2 M N K :=
        (M = z /\ N = K)
   \/ (exists M’ K’, M = s M’ /\ K = s K’ /\
                          plus2 M’ N K’).
```

Proving Defined Atoms

If `p` is a defined relation, then to prove `p M1 ··· Mn`:

1. Find a clause whose head matches with `p M1 ··· Mn`;
2. Apply the matching substitution to its body;
3. and prove that instance of the body.

Backtracks over clauses and ways to match.

Proving Defined Atoms: Example

```
Define plus : nat -> nat -> nat -> prop by
  plus z N N ;
  plus (s M) N (s K) := plus M N K.
```

Example: `plus (s z) (s (s z)) (s (s (s z)))`:

1. Pick second clause with unifier [`z`/`M`, `s (s z)`/`N`, `s (s z)`/`K`].
2. Yields goal: `plus z (s (s z)) (s (s z))`.
3. Now pick first clause with unifier [`s (s z)`/`N`].
4. Yields goal `true`, and we're done!

Reasoning About Defined Atoms

To reason about hypothesis **`p M1`** $\cdots$ **`Mn`**:

1. Find every way to unify **`p M1`** $\cdots$ **`Mn`** with some head;
2. Separately reason about each corresponding instance of the body as a new hypothesis.

Generates one premise (subgoal) per unification solution.

Observe the analogy with equality assumptions!

Reasoning About Defined Atoms

To reason about hypothesis `p M1` $\cdots$ `Mn`:

1. Find every way to unify `p M1` $\cdots$ `Mn` with some head;
2. Separately reason about each corresponding instance of the body as a new hypothesis.

Generates one premise (subgoal) per unification solution.

Observe the analogy with equality assumptions!

Reasoning About Defined Atoms: Example

```
Define plus : nat -> nat -> nat -> prop by
  plus z N N ;
  plus (s M) N (s K) := plus M N K.
```

Given hypothesis: `plus M N (s K)`:

1. Generate one subgoal for the first clause and unifier $[\texttt{z}/\texttt{M}, \texttt{s K}/\texttt{N}]$;
2. Another subgoal for the second clause and unifier $[\texttt{s M'}/\texttt{M}]$

```
Theorem plus_s : forall M N K, plus M N (s K) ->
  (exists J, M = s J) \/ (exists J, N = s J).
```

The `case` and `unfold` Tactics

$\boxed{\texttt{2.3 - case and unfold}}$

Consistency of Relational Definitions

- Relational definitions are given a fixed point interpretation.
- That is, every defined atom is considered to be equivalent to the disjunction of its unfolded forms.
- Such an equivalence can introduce inconsistencies.

```
Define p : prop by
  p := p -> false.
```

- Abella's stratification condition guarantees consistency.

Stratification

$$\boxed{\texttt{2.4 - Stratification}}$$

The Expressivity of `case` and `unfold`

Consider

```
Define is_nat1 : nat -> prop by
  is_nat1 z ;
  is_nat1 (s N) := is_nat1 N.

Define is_nat2 : nat -> prop by
  is_nat2 z ;
  is_nat2 (s N) := is_nat2 N.
```

- With `case` and `unfold`, we cannot prove:

  ```
  forall x, is_nat1 x -> is_nat2 x.
  ```

- Abella actually interprets fixed points as least fixed points.
- This in turn allows us to perform induction on such definitions.

The `induction` tactic

Given a goal

```
forall X1 ... Xn, F1 -> ... -> Fk -> ... -> G
```

where `Fk` is a defined atom, the invocation

```
induction on k.
```

1. Adds an inductive hypothesis (`IH`):

   ```
   forall X1 ... Xn, F1 -> ... -> Fk * -> ... -> G
   ```

2. Then changes the goal to:

   ```
   forall X1 ... Xn, F1 -> ... -> Fk @ -> ... -> G
   ```

Inductive Annotations

Meaning of `F*`

> *`F` has resulted from at least one application of `case` to an assumption of the form `F'@`.*

- These annotations are only maintained on defined atoms.
- Applying `case` to `F@` changes the annotation to `*` for the resulting bodies in every subgoal.
- The `*` annotation percolates to:
  - Both operands of `/\` and `\/`;
  - Only the right operand of `->`; and
  - The bodies of `forall` and `exists`.

Natural Number Induction

$\boxed{\texttt{2.5 - Natural Numbers}}$

Lists of Natural Numbers

$\boxed{\texttt{2.6 - Lists}}$

Nested and Mutual Induction

$\boxed{\texttt{2.7 - Nested and Mutual Induction}}$

The Reasoning Logic $\mathcal{G}$

Outline:

1. Ordinary Intuitionistic Logic
2. Equality
3. Fixed Point Definitions
4. Induction
   - Inductive data: lists
   - Kinds of induction: simple, mutual, nested
5. Higher-Order Abstract Syntax
   - Example: subject reduction for STLC

The Reasoning Logic $\mathcal{G}$

Outline:

1. Ordinary Intuitionistic Logic
2. Equality
3. Fixed Point Definitions
4. Induction
   - Inductive data: lists
   - Kinds of induction: simple, mutual, nested
5. Higher-Order Abstract Syntax
   - Example: subject reduction for STLC

Principles of Abstract Syntax

[Miller 2015]

1. *The names of bound variables should be treated as the same kind of fiction as we treat white space: they are artifacts of how we write expressions and have no semantic content.*
2. *There is "one binder to ring them all."*
3. *There is no such thing as a free variable.*
   - *cf.* Alan Perlis' epigram #47
4. *Bindings have mobility and the equality theory of expressions must support such mobility [...].*

Higher-Order Abstract Syntax

Also known as: λ-Tree Syntax

- Binding constructs in syntax are represented with term constructors of higher-order types.
- The normal forms of the representation are in bijection with the syntactic constructs.
- Syntactic substitution is for free – part of the λ-converibility inherent in equality.

HOAS: Representing the Simply Typed Lambda Calculus

Warmup: simple types.

```
Kind ty     type.

Type bas    ty.
Type arrow  ty -> ty -> ty.
```

$$[\![b]\!] = \texttt{bas} \qquad\qquad [\![A \to B]\!] = \texttt{arrow}\ [\![A]\!]\ [\![B]\!]$$

HOAS: Representing the Simply Typed Lambda Calculus

(Closed) λ-terms

```
Kind tm     type.

Type app    tm -> tm -> tm.
Type abs   (tm -> tm) -> tm.
```

$$[\![M\ N]\!] = \mathbf{app}\ [\![M]\!]\ [\![N]\!]$$

$$[\![\boldsymbol{\lambda} x.\, M]\!] = \mathbf{abs}\ \ \mathbf{(x\backslash}\ [\![[\mathbf{x}/x]M]\!]\mathbf{)}$$

$$[\![\mathbf{x}]\!] = \mathbf{x}$$

Examples:

$$[\![\boldsymbol{\lambda} x.\, \boldsymbol{\lambda} y.\, x]\!] = \texttt{abs x\textbackslash\ abs y\textbackslash\ x}$$

$$[\![\boldsymbol{\lambda} x.\, \boldsymbol{\lambda} y.\, \boldsymbol{\lambda} z.\, x\, z\, (y\, z)]\!] = \texttt{abs x\textbackslash\ abs y\textbackslash\ abs z\textbackslash\ app (app x z) (app y z)}$$

$$[\![(\boldsymbol{\lambda} x.\, x\, x)\ (\boldsymbol{\lambda} x.\, x\, x)]\!] = \texttt{app (abs x\textbackslash\ app x x) (abs x\textbackslash\ app x x)}$$

HOAS: Representing the Typing Relation

$$\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\boldsymbol{\lambda} x.\, M) : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

```
Kind ctx    type.

Type emp    ctx.
Type add    ctx -> tm -> ty -> ctx.
```

HOAS: Representing the Typing Relation

$$\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\lambda x.\, M) : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

```
Kind ctx    type.

Type emp    ctx.
Type add    ctx -> tm -> ty -> ctx.
```

HOAS: Representing the Typing Relation

$$\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\boldsymbol{\lambda} x.\, M) : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

```
Kind ctx    type.

Type emp    ctx.
Type add    ctx -> tm -> ty -> ctx.
```

HOAS: Representing Typing Contexts

```
Define mem : ctx -> tm -> ty -> prop by
  mem (add G X A) X A ;
  mem (add G Y B) X A := mem G X A.
```

$$\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\lambda x.\, M) : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

```
Define of : ctx -> tm -> ty -> prop by
  of G X A := mem G X A ;

  of G (app M N) B :=
    exists A, of M (arrow A B) /\ of N A ;

  of G (abs x\ M x) (arrow A B) :=
    of (add G ?? A) (M ??) B
```

HOAS: Representing Typing Contexts

```
Define mem : ctx -> tm -> ty -> prop by
  mem (add G X A) X A ;
  mem (add G Y B) X A := mem G X A.
```

$$\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\boldsymbol{\lambda} x.\, M) : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

```
Define of : ctx -> tm -> ty -> prop by
  of G X A := mem G X A ;

  of G (app M N) B :=
    exists A, of M (arrow A B) /\ of N A ;

  of G (abs x\ M x) (arrow A B) :=
    of (add G ?? A) (M ??) B
```

Contexts

What does $\Gamma, x{:}A$ mean?

- $x \notin \mathrm{fv}(\Gamma)$
- $x \notin \mathrm{fv}(A)$
- $(\Gamma, x{:}A)(y) = \begin{cases} A & \text{if } x = y \\ \Gamma(y) & \text{otherwise} \end{cases}$

Names and the ∇ (`nabla`) Quantifier

$\forall x.\ F$

> *For every term M, it is the case that $[M/x]F$ is true.*

$\nabla x.\ F$

> *For any name n that is not free in F, it is the case that $[n/x]F$ is true.*

Every type is inhabited by an infinite set of names.

Terminology: sometimes we say *nominal constant* instead of *name*.

Some Properties of ∇ vs. $\forall$

- $\nabla x.\, \nabla y.\, x \neq y$.
  - For any name $n \notin \{\}$, it is that $\nabla y.\, n \neq y$.
  - For any name $n \notin \{\}$, for any name $m \notin \{n\}$, it is that $n \neq m$.
- $\forall x.\, \forall y.\, x \neq y$ is not provable.
  - Given any term M, it must be that $M = M$.
- $(\forall x.\, \forall y.\, p\, x\, y) \supset (\forall z.\, p\, z\, z)$.
- $(\nabla x.\, \nabla y.\, p\, x\, y) \supset (\nabla z.\, p\, z\, z)$ is not provable.
  - $\nabla x.\, \nabla y.\, p\, x\, y$ means that p holds for any two distinct names.
  - $\nabla z.\, p\, z\, z$ means that p holds for any name, repeated.

Mobility of Binding

The equational theory of λ-terms is restated in terms of ∇.

$$\boxed{(\boldsymbol{\lambda} x.\, M) = (\boldsymbol{\lambda} x.\, N) \textit{ if and only if } \boldsymbol{\nabla} x.\, (M = N).}$$

Why not $\forall$?

- Differentiate between the identity function $\boldsymbol{\lambda} x.\, x$ and the constant function $\boldsymbol{\lambda} x.\, c$.
- $\forall x.\, (x = c)$ is satisfiable.
- $\boldsymbol{\nabla} x.\, (x = c)$ is false, i.e., $\neg \boldsymbol{\nabla} x.\, (x = c)$ is provable.

Names and Equivariance

- Formulas are considered equivalent up to a permutation of their free names, known as equivariance.
- Example: if m and n are distinct names, then:
  - $p\ m \equiv p\ n$.
  - $p\ m\ n \equiv p\ n\ m$.
  - $p\ m\ m \not\equiv p\ m\ n$.
- Note: terms are not equal up to equivariance!
- In Abella, any identifer matching the regexp `n[0-9]+` is considered to be a name.

Raising

Let supp(F) stand for the free names in F.

$\forall x.\ F$:

$$\boxed{\textit{For every term } M\textit{, it is the case that } [M/x]F \textit{ is true.}}$$

Raising

Let $\mathrm{supp}(F)$ stand for the free names in F.

$\forall x.\ F$:

> *For every term M with $\mathrm{supp}(M) = \{\}$, it is the case that $[M\ \mathrm{supp}(F)/x]F$ is true.*

Raising

$\forall x.\, F$:

> *For every term M with $\mathrm{supp}(M) = \{\}$, it is the case that $[M\ \mathrm{supp}(F)/x]F$ is true.*

- $\forall x.\, \nabla y.\, p\ x\ y$
  - For every term M, it is that $\nabla y.\, p\ M\ y$.
  - For every M, for any name $n \notin \mathrm{fn}(M)$, it is that $p\ M\ n$.
  - Therefore M cannot mention n.
- $\nabla y.\, \forall x.\, p\ x\ y$
  - For any name $n \notin \{\}$, it is that $\forall x.\, p\ x\ n$.
  - For any name n, for every term M, it is that $p\ (M\ n)\ n$.
  - In other words, M is of the form $\lambda x.\, M'$ where M' can have x free.
  - Therefore, M can (indirectly) mention n.

Back to HOAS: The Typing Relation

$$\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\boldsymbol{\lambda} x.\, M) : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

```
Define of : ctx -> tm -> ty -> prop by
  of G X A := mem G X A ;

  of G (app M N) B :=
    exists A, of M (arrow A B) /\ of N A ;

  of G (abs x\ M x) (arrow A B) :=
    nabla x, of (add G x A) (M x) B
```

Back to HOAS: The Typing Relation

$$\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\lambda x.\, M) : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

```
Define of : ctx -> tm -> ty -> prop by
  of G X A := mem G X A ;

  of G (app M N) B :=
    exists A, of M (arrow A B) /\ of N A ;

  of G (abs x\ M x) (arrow A B) :=
    nabla x, of (add G x A) (M x) B
```

∇ in the Body of a Clause

```
of G (abs x\ M x) (arrow A B) :=
  nabla x, of (add G x A) (M x) B
```

means

```
forall G M A B,
  of G (abs x\ M x) (arrow A B) <-
    nabla x, of (add G x A) (M x) B.
```

- None of **`G`**, **`M`**, **`A`**, **`B`** can mention **`x`**.
- **`M`** can indirectly mention **`x`**.

HOAS: Typing Relation

$\boxed{\texttt{2.8 - Properties of the Typing Relation}}$

HOAS: Substitution

The main promise of HOAS: substitution “for free”

```
Define eval : tm -> tm -> prop by
  eval (abs R) (abs R) ;
  eval (app M N) V :=
    exists R, eval M (abs R) /\ eval (R N) V.
```

Notes:

- `(R N)` may be arbitrarily larger than `(app M N)`.
- However, proving `(eval (R N) V)` will require strictly fewer unfolding steps than `(eval (app M N) V)`.

HOAS: Subject Reducton (Extended Example)

$$\boxed{\texttt{2.9 - Subject Reduction}}$$

INTERMISSION

The Two-Level Logic Approach

Outline

1. Focused Minimal Intuitionistic Logic
2. Two-Level Logic Approach
3. Context Structure
4. Examples

Meta-Theorems

- We have just seen several examples of meta-theorems:
  - Cut (for substituting in contexts)
  - Instantiation (for replacing names with terms)
  - Weakening

- Such theorems can be seen as instances of similar meta-theorems for a proof system

- If we can isolate this proof system and prove the meta-theorems once and for all, we can avoid a lot of boilerplate.

Small Aside: A Bit of Proof Theory

Let us start with intuitionistic minimal logic.

$$
\begin{array}{rcccccc}
F, G & ::= & A & \mid & F \Rightarrow G & \mid & \Pi x.\, F \\
\Gamma & ::= & \cdot & \mid & \Gamma, F & &
\end{array}
$$

We are going to build a focused proof system for this logic.

$\Gamma \vdash F$	Goal decomposition sequent
$\Gamma, [F] \vdash A$	Backchaining sequent

Small Aside: A Bit of Proof Theory

Let us start with intuitionistic minimal logic.

$$\begin{array}{rcl} F, G & ::= & A \quad | \quad F \Rightarrow G \quad | \quad \Pi x.\, F \\ \Gamma & ::= & \cdot \quad | \quad \Gamma, F \end{array}$$

We are going to build a focused proof system for this logic.

$\Gamma \vdash F$	Goal decomposition sequent
$\Gamma, [F] \vdash A$	Backchaining sequent

Focused Proof System

Goal decomposition

$$\frac{\Gamma, F \vdash G}{\Gamma \vdash F \Rightarrow G} \qquad \frac{(x \# \Gamma) \quad \Gamma \vdash F}{\Gamma \vdash \Pi x.\, F}$$

Decision

$$\frac{\Gamma, F, [F] \vdash A}{\Gamma, F \vdash A}$$

Backchaining

$$\frac{\Gamma \vdash F \quad \Gamma, [G] \vdash A}{\Gamma, [F \Rightarrow G] \vdash A} \qquad \frac{\Gamma, [[t/x]F] \vdash A}{\Gamma, [\Pi x.\, F] \vdash A} \qquad \frac{}{\Gamma, [A] \vdash A}$$

Focused Proof System

Goal decomposition

$$\frac{\Gamma, F \vdash G}{\Gamma \vdash F \Rightarrow G} \qquad \frac{(x \# \Gamma) \quad \Gamma \vdash F}{\Gamma \vdash \Pi x.\, F}$$

Decision

$$\frac{\Gamma, F, [F] \vdash A}{\Gamma, F \vdash A}$$

Backchaining

$$\frac{\Gamma \vdash F \quad \Gamma, [G] \vdash A}{\Gamma, [F \Rightarrow G] \vdash A} \qquad \frac{\Gamma, [[t/x]F] \vdash A}{\Gamma, [\Pi x.\, F] \vdash A} \qquad \frac{}{\Gamma, [A] \vdash A}$$

Focused Proof System

Goal decomposition

$$\frac{\Gamma, F \vdash G}{\Gamma \vdash F \Rightarrow G} \qquad \frac{(x \# \Gamma) \quad \Gamma \vdash F}{\Gamma \vdash \Pi x.\, F}$$

Decision

$$\frac{\Gamma, F, [F] \vdash A}{\Gamma, F \vdash A}$$

Backchaining

$$\frac{\Gamma \vdash F \quad \Gamma, [G] \vdash A}{\Gamma, [F \Rightarrow G] \vdash A} \qquad \frac{\Gamma, [[t/x]F] \vdash A}{\Gamma, [\Pi x.\, F] \vdash A} \qquad \frac{}{\Gamma, [A] \vdash A}$$

Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

R_1: $\Pi m, n, a, b.\ \texttt{of}\ m\ (\texttt{arr}\ a\ b) \Rightarrow \texttt{of}\ n\ a \Rightarrow \texttt{of}\ (\texttt{app}\ m\ n)\ b.$

R_2: $\Pi r, a, b.\ (\Pi x.\ \texttt{of}\ x\ a \Rightarrow \texttt{of}\ (r\ x)\ b) \Rightarrow \texttt{of}\ (\texttt{abs}\ r)\ (\texttt{arr}\ a\ b).$

Consider the result of deciding on R_1 and R_2.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{of}\ M\ (\texttt{arr}\ A\ B) \quad \Gamma \vdash \texttt{of}\ N\ A \quad \Gamma, [\texttt{of}\ (\texttt{app}\ M\ N)\ B] \vdash C}{\Gamma, [[M/m, N/n, A/a, B/b] \cdots \Rightarrow \cdots \Rightarrow \cdots] \vdash C}}{\Gamma, [R_1] \vdash C}}{\Gamma \vdash C}$$

$$\dfrac{\Gamma \vdash \texttt{of}\ M\ (\texttt{arr}\ A\ B) \quad \Gamma \vdash \texttt{of}\ N\ A}{\Gamma \vdash \texttt{of}\ (\texttt{app}\ M\ N)\ B}$$

Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

R_1: $\Pi m, n, a, b.\ \mathtt{of}\ m\ (\mathtt{arr}\ a\ b) \Rightarrow \mathtt{of}\ n\ a \Rightarrow \mathtt{of}\ (\mathtt{app}\ m\ n)\ b.$

R_2: $\Pi r, a, b.\ (\Pi x.\ \mathtt{of}\ x\ a \Rightarrow \mathtt{of}\ (r\ x)\ b) \Rightarrow \mathtt{of}\ (\mathtt{abs}\ r)\ (\mathtt{arr}\ a\ b).$

Consider the result of deciding on R_1 and R_2.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \quad \Gamma \vdash \mathtt{of}\ N\ A \quad \dfrac{}{\Gamma, [\mathtt{of}\ (\mathtt{app}\ M\ N)\ B] \vdash C}}{\Gamma, [[M/m, N/n, A/a, B/b] \cdots \Rightarrow \cdots \Rightarrow \cdots] \vdash C}}{\Gamma, [R_1] \vdash C}}{\Gamma \vdash C}$$

$$\dfrac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \quad \Gamma \vdash \mathtt{of}\ N\ A}{\Gamma \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}$$

Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

R_1: $\Pi m, n, a, b.\ \mathtt{of}\ m\ (\mathtt{arr}\ a\ b) \Rightarrow \mathtt{of}\ n\ a \Rightarrow \mathtt{of}\ (\mathtt{app}\ m\ n)\ b.$

R_2: $\Pi r, a, b.\ (\Pi x.\ \mathtt{of}\ x\ a \Rightarrow \mathtt{of}\ (r\ x)\ b) \Rightarrow \mathtt{of}\ (\mathtt{abs}\ r)\ (\mathtt{arr}\ a\ b).$

Consider the result of deciding on R_1 and R_2.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \quad \Gamma \vdash \mathtt{of}\ N\ A \quad \dfrac{}{\Gamma, [\mathtt{of}\ (\mathtt{app}\ M\ N)\ B] \vdash C}}{\Gamma, [[M/m, N/n, A/a, B/b] \cdots \Rightarrow \cdots \Rightarrow \cdots] \vdash C}}{\Gamma, [R_1] \vdash C}}{\Gamma \vdash C}$$

$$\dfrac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \quad \Gamma \vdash \mathtt{of}\ N\ A}{\Gamma \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}$$

Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

R_1: $\Pi m, n, a, b.\ \texttt{of}\ m\ (\texttt{arr}\ a\ b) \Rightarrow \texttt{of}\ n\ a \Rightarrow \texttt{of}\ (\texttt{app}\ m\ n)\ b.$

R_2: $\Pi r, a, b.\ (\Pi x.\ \texttt{of}\ x\ a \Rightarrow \texttt{of}\ (r\ x)\ b) \Rightarrow \texttt{of}\ (\texttt{abs}\ r)\ (\texttt{arr}\ a\ b).$

Consider the result of deciding on R_1 and R_2.

$$\dfrac{\dfrac{\Gamma \vdash \texttt{of}\ M\ (\texttt{arr}\ A\ B) \quad \Gamma \vdash \texttt{of}\ N\ A \quad \overline{\Gamma, [\texttt{of}\ (\texttt{app}\ M\ N)\ B] \vdash C}}{\Gamma, [[M/m, N/n, A/a, B/b] \cdots \Rightarrow \cdots \Rightarrow \cdots] \vdash C}}{\dfrac{\Gamma, [R_1] \vdash C}{\Gamma \vdash C}}$$

$$\dfrac{\Gamma \vdash \texttt{of}\ M\ (\texttt{arr}\ A\ B) \quad \Gamma \vdash \texttt{of}\ N\ A}{\Gamma \vdash \texttt{of}\ (\texttt{app}\ M\ N)\ B}$$

Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

R_1: $\Pi m, n, a, b.\ \mathtt{of}\ m\ (\mathtt{arr}\ a\ b) \Rightarrow \mathtt{of}\ n\ a \Rightarrow \mathtt{of}\ (\mathtt{app}\ m\ n)\ b.$

R_2: $\Pi r, a, b.\ (\Pi x.\ \mathtt{of}\ x\ a \Rightarrow \mathtt{of}\ (r\ x)\ b) \Rightarrow \mathtt{of}\ (\mathtt{abs}\ r)\ (\mathtt{arr}\ a\ b).$

Consider the result of deciding on R_1 and R_2.

$$\dfrac{\dfrac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \qquad \Gamma \vdash \mathtt{of}\ N\ A \qquad \dfrac{}{\Gamma, [\mathtt{of}\ (\mathtt{app}\ M\ N)\ B] \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}}{\dfrac{\Gamma, [[M/m, N/n, A/a, B/b] \cdots \Rightarrow \cdots \Rightarrow \cdots] \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}{\Gamma, [R_1] \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}}}{\Gamma \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}$$

$$\dfrac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \qquad \Gamma \vdash \mathtt{of}\ N\ A}{\Gamma \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}$$

Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

R_1: $\Pi m, n, a, b.\ \mathtt{of}\ m\ (\mathtt{arr}\ a\ b) \Rightarrow \mathtt{of}\ n\ a \Rightarrow \mathtt{of}\ (\mathtt{app}\ m\ n)\ b.$

R_2: $\Pi r, a, b.\ (\Pi x.\ \mathtt{of}\ x\ a \Rightarrow \mathtt{of}\ (r\ x)\ b) \Rightarrow \mathtt{of}\ (\mathtt{abs}\ r)\ (\mathtt{arr}\ a\ b).$

Consider the result of deciding on R_1 and R_2.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \quad \Gamma \vdash \mathtt{of}\ N\ A \quad \dfrac{}{\Gamma, [\mathtt{of}\ (\mathtt{app}\ M\ N)\ B] \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}}{\Gamma, [[M/m, N/n, A/a, B/b] \cdots \Rightarrow \cdots \Rightarrow \cdots] \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}}{\Gamma, [R_1] \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}}{\Gamma \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}$$

$$\dfrac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \quad \Gamma \vdash \mathtt{of}\ N\ A}{\Gamma \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}$$

Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

R_1: $\Pi m, n, a, b.\ \texttt{of}\ m\ (\texttt{arr}\ a\ b) \Rightarrow \texttt{of}\ n\ a \Rightarrow \texttt{of}\ (\texttt{app}\ m\ n)\ b.$

R_2: $\Pi r, a, b.\ (\Pi x.\ \texttt{of}\ x\ a \Rightarrow \texttt{of}\ (r\ x)\ b) \Rightarrow \texttt{of}\ (\texttt{abs}\ r)\ (\texttt{arr}\ a\ b).$

Consider the result of deciding on R_1 and R_2.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{of}\ M\ (\texttt{arr}\ A\ B) \quad \Gamma \vdash \texttt{of}\ N\ A \quad \dfrac{}{\Gamma, [\texttt{of}\ (\texttt{app}\ M\ N)\ B] \vdash \texttt{of}\ (\texttt{app}\ M\ N)\ B}}{\Gamma, [[M/m, N/n, A/a, B/b] \cdots \Rightarrow \cdots \Rightarrow \cdots] \vdash \texttt{of}\ (\texttt{app}\ M\ N)\ B}}{\Gamma, [R_1] \vdash \texttt{of}\ (\texttt{app}\ M\ N)\ B}}{\Gamma \vdash \texttt{of}\ (\texttt{app}\ M\ N)\ B}$$

$$\dfrac{\Gamma \vdash \texttt{of}\ M\ (\texttt{arr}\ A\ B) \quad \Gamma \vdash \texttt{of}\ N\ A}{\Gamma \vdash \texttt{of}\ (\texttt{app}\ M\ N)\ B}$$

Deciding on R_2

$$\dfrac{\dfrac{\dfrac{\boxed{1} \qquad \Gamma, [\mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)] \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}{\Gamma, [[R/r, A/a, B/b](\Pi x.\ \cdots \Rightarrow \cdots) \Rightarrow \cdots] \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}}{\Gamma, [R_2] \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}}{\Gamma \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}$$

where $\boxed{1}$ is:

$$\dfrac{(x \# \Gamma) \qquad \Gamma, \mathtt{of}\ x\ A \vdash \mathtt{of}\ (R\ x)\ B}{\Gamma \vdash \Pi x.\ \mathtt{of}\ x\ A \Rightarrow \mathtt{of}\ (R\ x)\ B}$$

So:

$$\dfrac{(x \# \Gamma) \qquad \Gamma, \mathtt{of}\ x\ A \vdash \mathtt{of}\ (R\ x)\ B}{\Gamma \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}$$

Deciding on R_2

$$\dfrac{\dfrac{\dfrac{\boxed{1} \qquad \Gamma, [\mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)] \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}{\Gamma, [[R/r, A/a, B/b](\Pi x.\ \cdots \Rightarrow \cdots) \Rightarrow \cdots] \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}}{\Gamma, [R_2] \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}}{\Gamma \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}$$

where $\boxed{1}$ is:

$$\dfrac{(x\#\Gamma) \qquad \Gamma, \mathtt{of}\ x\ A \vdash \mathtt{of}\ (R\ x)\ B}{\Gamma \vdash \Pi x.\ \mathtt{of}\ x\ A \Rightarrow \mathtt{of}\ (R\ x)\ B}$$

So:

$$\dfrac{(x\#\Gamma) \qquad \Gamma, \mathtt{of}\ x\ A \vdash \mathtt{of}\ (R\ x)\ B}{\Gamma \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}$$

Synthetic Rules vs. SOS rules

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M\, N) : B} \qquad \frac{\Gamma \vdash \texttt{of}\ M\ (\texttt{arr}\ A\ B) \qquad \Gamma \vdash \texttt{of}\ N\ A}{\Gamma \vdash \texttt{of}\ (\texttt{app}\ M\ N)\ B}$$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\boldsymbol{\lambda} x.\, M) : A \to B} \qquad \frac{(x \# \Gamma) \qquad \Gamma, \texttt{of}\ x\ A \vdash \texttt{of}\ (R\ x)\ B}{\Gamma \vdash \texttt{of}\ (\texttt{abs}\ R)\ (\texttt{arr}\ A\ B)}$$

Reasoning about SOS derivations is isomorphic to reasoning about focused derivations for its minimal theory.

Synthetic Rules vs. SOS rules

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\,N) : B} \qquad \frac{\Gamma \vdash \mathtt{of}\ M\ (\mathtt{arr}\ A\ B) \quad \Gamma \vdash \mathtt{of}\ N\ A}{\Gamma \vdash \mathtt{of}\ (\mathtt{app}\ M\ N)\ B}$$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\boldsymbol{\lambda} x.\, M) : A \to B} \qquad \frac{(x \# \Gamma) \quad \Gamma, \mathtt{of}\ x\ A \vdash \mathtt{of}\ (R\,x)\ B}{\Gamma \vdash \mathtt{of}\ (\mathtt{abs}\ R)\ (\mathtt{arr}\ A\ B)}$$

Reasoning about SOS derivations is isomorphic to reasoning about focused derivations for its minimal theory.

Minimal Logic Definable in $\mathcal{G}$

```
Kind o        type.

Type =>       o -> o -> o.
Type pi       (A -> o) -> o.

Kind olist    type

Type nil      olist.
Type ::       o -> olist -> olist.

Define member : o -> olist -> prop by ...
```

Sequent	Encoding
$\Gamma \vdash F$	`seq L F`
$\Gamma, [F] \vdash A$	`bch L F A`

Minimal Logic Definable in $\mathcal{G}$

```
Kind o       type.

Type =>      o -> o -> o.
Type pi      (A -> o) -> o.

Kind olist   type

Type nil     olist.
Type ::      o -> olist -> olist.

Define member : o -> olist -> prop by ...
```

Sequent	**Encoding**
$\Gamma \vdash F$	`seq L F`
$\Gamma, [F] \vdash A$	`bch L F A`

Focused Minimal Sequent Calculus in $\mathcal{G}$

```
Define seq : olist -> o -> prop,
       bch : olist -> o -> o -> prop by

  % goal reduction
  seq L (F => G)    := seq (F :: L) G ;
  seq L (pi F)      := nabla x, seq L (F x) ;

  % decision
  seq L A           :=
    exists F, member F L /\ bch L F A ;

  % backchaining
  bch L (F => G) A := seq L F /\ bch L G A ;
  bch L (pi F) A    := exists T, bch L (F T) A
  bch L A A.
```

Meta-Theory of Minimal Sequent Calculus

```
Theorem cut : forall L C F,
  seq L C -> seq (C :: L) F -> seq L F.

Theorem inst : forall L F, nabla x,
  seq (L x) (F x) ->
    forall T, seq (L T) (F T).

Theorem monotone : forall L1 L2 F,
  %% L1 ⊆ L2
  (forall G, member G L1 -> member G L2) ->
    seq L1 F -> seq L2 F.
```

The Two Level Logic Approach of Abella

- Specification Logic
  - Focused sequent calculus for minimal intuitionistic logic
  - Shares the type system of $\mathcal{G}$, but formulas of type **o**
  - Concrete syntax the same as λProlog

- Reasoning Logic
  - Inductive definition of the specification logic proof system
  - Inductive reasoning about specification logic derivations
  - Syntactic sugar:

```
seq L F         {L |- F}
bch L F A       {L, [F] |- A}
```

Example: STLC Specification

$\boxed{\texttt{3.1 - Typing and Subject Reduction}}$

Uniqueness of Typing

Change to a Church style representation:

```
type  abs   ty -> (tm -> tm) -> tm.
----
of (abs A R) (arr A B) :-
  pi x\ of x A => of (R x) B.
```

Want to show that every term has a unique type.

```
Theorem type_uniq : forall M A B,
  {of M A} -> {of M B} -> A = B.
```

Need to generalize!

```
Theorem type_uniq_open : forall L M A B,
  {L |- of M A} -> {L |- of M B} -> A = B.
```

Uniqueness of Typing

Change to a Church style representation:

```
type  abs   ty -> (tm -> tm) -> tm.
----
of (abs A R) (arr A B) :-
  pi x\ of x A => of (R x) B.
```

Want to show that every term has a unique type.

```
Theorem type_uniq : forall M A B,
  {of M A} -> {of M B} -> A = B.
```

Need to generalize!

```
Theorem type_uniq_open : forall L M A B,
  {L |- of M A} -> {L |- of M B} -> A = B.
```

Uniqueness of Typing

Change to a Church style representation:

```
type  abs   ty -> (tm -> tm) -> tm.
----
of (abs A R) (arr A B) :-
  pi x\ of x A => of (R x) B.
```

Want to show that every term has a unique type.

```
Theorem type_uniq : forall M A B,
  {of M A} -> {of M B} -> A = B.
```

Need to generalize!

```
Theorem type_uniq_open : forall L M A B,
  {L |- of M A} -> {L |- of M B} -> A = B.
```

Structure of Contexts

- The typing dynamic context **`L`** is a list of **`of`** assumptions.
- Already seen how to inductively define the structure of lists.
- Therefore:

```
Define ctx : olist -> prop by
  ctx nil ;
  ctx (of X A :: L) := ctx L.
```

- But this does not capture **`X`**#**`L`**!

"∇ In The Head"

Meaning of the second clause:

```
forall L A X,
  ctx L -> ctx (of X A :: L).
```

Let us change the "flavor" of x.

```
forall L A, nabla x,
  ctx L -> ctx (of x A :: L).
```

Equivalent to:

```
forall L A, ctx L ->
  nabla x, ctx (of x A :: L).
```

This suggests:

```
Define ctx : olist -> prop by
  ctx nil ;
  nabla x, ctx (of x A :: L) := ctx L.
```

“∇ In The Head”

Meaning of the second clause:

```
forall L A X,
  ctx L -> ctx (of X A :: L).
```

Let us change the “flavor” of **x**.

```
forall L A, nabla x,
  ctx L -> ctx (of x A :: L).
```

Equivalent to:

```
forall L A, ctx L ->
  nabla x, ctx (of x A :: L).
```

This suggests:

```
Define ctx : olist -> prop by
  ctx nil ;
  nabla x, ctx (of x A :: L) := ctx L.
```

"∇ In The Head"

Meaning of the second clause:

```
forall L A X,
  ctx L -> ctx (of X A :: L).
```

Let us change the "flavor" of **x**.

```
forall L A, nabla x,
  ctx L -> ctx (of x A :: L).
```

Equivalent to:

```
forall L A, ctx L ->
  nabla x, ctx (of x A :: L).
```

This suggests:

```
Define ctx : olist -> prop by
  ctx nil ;
  nabla x, ctx (of x A :: L) := ctx L.
```

"∇ In The Head"

Meaning of the second clause:

```
forall L A X,
  ctx L -> ctx (of X A :: L).
```

Let us change the "flavor" of **x**.

```
forall L A, nabla x,
  ctx L -> ctx (of x A :: L).
```

Equivalent to:

```
forall L A, ctx L ->
  nabla x, ctx (of x A :: L).
```

This suggests:

```
Define ctx : olist -> prop by
  ctx nil ;
  nabla x, ctx (of x A :: L) := ctx L.
```

Unification with ∇ In Heads

Clause head: `nabla x, ctx (of x A :: L)`
Assumption: `H : ctx (of U B :: LL)`

- `U` must be a name ...
- ...that does not occur in `B` or `LL`!
- Therefore, `case H` picks an $n \notin \mathrm{supp}(\texttt{B}) \cup \mathrm{supp}(\texttt{LL})$ for the unifier for `U`.

Unification with ∇ In Heads

Clause head: `nabla x, ctx (of x A :: L)`
Assumption: `H : ctx (of U B :: LL)`

- `U` must be a name ...
- ...that does not occur in `B` or `LL`!
- Therefore, `case H` picks an $n \notin \mathrm{supp}(\texttt{B}) \cup \mathrm{supp}(\texttt{LL})$ for the unifier for `U`.

Unification with ∇ In Heads

Clause head: `nabla x, ctx (of x A :: L)`
Assumption: `H : ctx (of U B :: LL)`

- `U` must be a name …
- …that does not occur in `B` or `LL`!
- Therefore, `case H` picks an $n \notin \mathrm{supp}(\texttt{B}) \cup \mathrm{supp}(\texttt{LL})$ for the unifier for `U`.

Unification with ∇ In Heads

Clause head: `nabla x, ctx (of x A :: L)`
Assumption: `H : ctx (of U B :: LL)`

- **U** must be a name ...
- ...that does not occur in **B** or **LL**!
- Therefore, `case H` picks an $n \notin \mathrm{supp}(\mathtt{B}) \cup \mathrm{supp}(\mathtt{LL})$ for the unifier for **U**.

Unification with ∇ In Heads

Clause head:	`nabla x, ctx (of x A :: L)`
Assumption:	`H : ctx (of n1 B :: (LL n1))`
Tactic:	`case H.`

Unification prunes `n1` from `LL n1`.

Clause head:	`nabla x, ctx (of x A :: L)`
Assumption:	`H : ctx (of n1 B :: kon n1)`
Tactic:	`case H.`

Cannot prune `n1`, so unification fails!

Unification with ∇ In Heads

Clause head:	`nabla x, ctx (of x A :: L)`
Assumption:	`H : ctx (of n1 B :: (LL n1))`
Tactic:	`case H.`

Unification prunes `n1` from `LL n1`.

Clause head:	`nabla x, ctx (of x A :: L)`
Assumption:	`H : ctx (of n1 B :: kon n1)`
Tactic:	`case H.`

Cannot prune `n1`, so unification fails!

Unification with ∇ In Heads

Clause head:	`nabla x, ctx (of x A :: L)`
Assumption:	`H : ctx (of n1 B :: (LL n1))`
Tactic:	`case H.`

Unification prunes `n1` from `LL n1`.

Clause head:	`nabla x, ctx (of x A :: L)`
Assumption:	`H : ctx (of n1 B :: kon n1)`
Tactic:	`case H.`

Cannot prune `n1`, so unification fails!

Unification with ∇ In Heads

Clause head:	`nabla x, ctx (of x A :: L)`
Assumption:	`H : ctx (of n1 B :: (LL n1))`
Tactic:	`case H.`

Unification prunes `n1` from `LL n1`.

Clause head:	`nabla x, ctx (of x A :: L)`
Assumption:	`H : ctx (of n1 B :: kon n1)`
Tactic:	`case H.`

Cannot prune `n1`, so unification fails!

Some Puzzles

- Define `name : tm -> prop` that holds only for names.

```
Define name : tm -> prop by
  nabla x, name x.
```

- Define `fresh : tm -> tm -> prop` such that `fresh X Y` means `X` is a name that does not occur in `Y`.

```
Define fresh : tm -> tm -> prop by
  nabla x, fresh x Y.
```

Some Puzzles

- Define `name : tm -> prop` that holds only for names.

```
Define name : tm -> prop by
  nabla x, name x.
```

- Define `fresh : tm -> tm -> prop` such that `fresh X Y` means `X` is a name that does not occur in `Y`.

```
Define fresh : tm -> tm -> prop by
  nabla x, fresh x Y.
```

Some Puzzles

- Define `name : tm -> prop` that holds only for names.

```
Define name : tm -> prop by
  nabla x, name x.
```

- Define `fresh : tm -> tm -> prop` such that `fresh X Y` means `X` is a name that does not occur in `Y`.

```
Define fresh : tm -> tm -> prop by
  nabla x, fresh x Y.
```

Some Puzzles

- Define `name : tm -> prop` that holds only for names.

```
Define name : tm -> prop by
  nabla x, name x.
```

- Define `fresh : tm -> tm -> prop` such that `fresh X Y` means `X` is a name that does not occur in `Y`.

```
Define fresh : tm -> tm -> prop by
  nabla x, fresh x Y.
```

Extended Example: Uniqueness of Typing

$\boxed{\texttt{3.2 - Type Uniqueness}}$

Context Relations

No reason for **`ctx`** relations to be unary.

```
Define ctx_len : olist -> nat -> prop by
  ctx_len nil z ;
  nabla x, ctx_len (of x A :: L) (s N) :=
    ctx_len L N.
```

```
Define ctxs : olist -> olist -> prop by
  ctxs nil nil ;
  nabla x, ctxs (term x :: L) (neutral x :: K) :=
    ctxs L K.
```

Context Relations

No reason for `ctx` relations to be unary.

```
Define ctx_len : olist -> nat -> prop by
  ctx_len nil z ;
  nabla x, ctx_len (of x A :: L) (s N) :=
    ctx_len L N.
```

```
Define ctxs : olist -> olist -> prop by
  ctxs nil nil ;
  nabla x, ctxs (term x :: L) (neutral x :: K) :=
    ctxs L K.
```

Context Relations

No reason for **`ctx`** relations to be unary.

```
Define ctx_len : olist -> nat -> prop by
  ctx_len nil z ;
  nabla x, ctx_len (of x A :: L) (s N) :=
    ctx_len L N.
```

```
Define ctxs : olist -> olist -> prop by
  ctxs nil nil ;
  nabla x, ctxs (term x :: L) (neutral x :: K) :=
    ctxs L K.
```

Example: Partitioning of Lambda Terms

```
3.3 – Partitioning
```

Extended Example: Relating HOAS and De Bruijn Representations

$\boxed{\texttt{3.4 - HOAS vs. Indexed}}$

Co-Induction

Interpretations of Co-Induction

- Non-termination
- Greatest Fixed Point
- Dual of Induction

```
Define p : prop by
  p := p.

Theorem pth : p -> false.

CoDefine q : prop by
  q := q.

Theorem qth : q.
```

The `coinduction` Tactic

Given a goal

```
forall X1 ... Xn, F1 -> ... -> Fn -> G
```

where `G` is a co-inductively defined atom, the invocation

```
coinduction
```

1. Adds a co-inductive hypothesis (`CH`):

   ```
   forall X1 ... Xn, F1 -> ... -> Fn -> G +
   ```

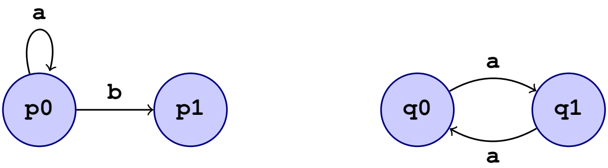
2. Then changes the goal to:

   ```
   forall X1 ... Xn, F1 -> ... -> Fn -> G #.
   ```

Annotations

Annotation	Place	Tactic	Result
@	hypothesis	`case`	*
@	goal	anything	no change
#	goal	`unfold`	+
#	hypothesis	anything	no change

Example: Automata Simulation



Definition: q simulates p, written $p \precsim q$, iff:

- for every p', a such that $p \xrightarrow{a} p'$,
- there is a q' such that $q \xrightarrow{a} q'$, and
- $p' \precsim q'$.

Here,

- **q0** $\precsim$ **p0**.
- **q1** $\precsim$ **p0**.
- **p0** $\not\precsim$ **q0**.

Example: Automata Simulation

$\boxed{\texttt{4.1 – Automata}}$

Example: Diverging λ-Terms

$\boxed{\texttt{4.2 - Divergence}}$

Summary So Far

You have now seen the *headline features* of Abella.

- Higher-Order Abstract Syntax and ∇
- Inductive and Co-Inductive Definitions
- Two-Level Logic Approach

Next:

- Re-ification of the type system
- Beyond simple types
- Automation

Summary So Far

You have now seen the *headline features* of Abella.

- Higher-Order Abstract Syntax and ∇
- Inductive and Co-Inductive Definitions
- Two-Level Logic Approach

Next:

- Re-ification of the type system
- Beyond simple types
- Automation

Extensions

Reasoning about typing

Abella's induction mechanism has two simple principles:

- Every inductive proof is based on an inductive definition
- All inductive definitions are explicit, fixed, and finite

Consequences:

- Typing is not itself inductive
- Signatures can always be extended

```
Type z nat.
Type s nat -> nat.

Theorem nat_str : forall (x:nat),
  x = z \/ exists (y:nat), x = s y.
% not provable
skip.

Type p nat -> nat -> nat.
```

Is nat_str still true?

Reasoning about typing

Abella's induction mechanism has two simple principles:

- Every inductive proof is based on an inductive definition
- All inductive definitions are explicit, fixed, and finite

Consequences:

- Typing is not itself inductive
- Signatures can always be extended

```
Type z nat.
Type s nat -> nat.

Theorem nat_str : forall (x:nat),
  x = z \/ exists (y:nat), x = s y.
% not provable
skip.

Type p nat -> nat -> nat.
```

Is `nat_str` still true?

Re-ifying Typing

Sometimes the typing relation can be reified.

```
Define is_nat : nat -> prop by
  is_nat z ;
  is_nat (s N) := is_nat N.

Theorem nat_str : forall x, is_nat x ->
  x = z \/ exists y, is_nat y /\ x = s y.
...
```

But not always!

```
Define is_tm : tm -> prop by
  is_tm (app M N) := is_tm M /\ is_tm N ;
  is_tm (abs R) := nabla x, is_tm x -> is_tm (R x).
```

This is not stratified.

Re-ifying Typing

Sometimes the typing relation can be reified.

```
Define is_nat : nat -> prop by
  is_nat z ;
  is_nat (s N) := is_nat N.

Theorem nat_str : forall x, is_nat x ->
  x = z \/ exists y, is_nat y /\ x = s y.
...
```

But not always!

```
Define is_tm : tm -> prop by
  is_tm (app M N) := is_tm M /\ is_tm N ;
  is_tm (abs R) := nabla x, is_tm x -> is_tm (R x).
```

This is not stratified.

Re-ifying Typing

Sometimes the typing relation can be reified.

```
Define is_nat : nat -> prop by
  is_nat z ;
  is_nat (s N) := is_nat N.

Theorem nat_str : forall x, is_nat x ->
  x = z \/ exists y, is_nat y /\ x = s y.
...
```

But not always!

```
Define is_tm : tm -> prop by
  is_tm (app M N) := is_tm M /\ is_tm N ;
  is_tm (abs R) := nabla x, is_tm x -> is_tm (R x).
```

This is not stratified.

Two-Level Reification

```
% typing.sig
type is_nat nat -> o.
type is_tm  tm -> o.
----
% typing.mod
is_nat z.
is_nat (s N) :- is_nat N.

is_tm (app M N) :- is_tm M, is_tm N.
is_tm (abs R) :- pi x\ is_tm x => is_tm (R x).
```

Then

```
Theorem nat_str : forall x, {is_nat x} ->
  x = z \/ exists y, {is_nat y} /\ x = s y.

Theorem tm_str : forall T, {is_tm T} ->
   (exists M N, {is_tm M} /\ {is_tm N} /\ T = app M N)
     \/
   (exists R, (forall x, {is_tm x} -> {is_tm R x})
            /\ T = abs R).
```

Two-Level Reification

```
% typing.sig
type is_nat nat -> o.
type is_tm  tm -> o.
----
% typing.mod
is_nat z.
is_nat (s N) :- is_nat N.

is_tm (app M N) :- is_tm M, is_tm N.
is_tm (abs R) :- pi x\ is_tm x => is_tm (R x).
```

Then

```
Theorem nat_str : forall x, {is_nat x} ->
  x = z \/ exists y, {is_nat y} /\ x = s y.

Theorem tm_str : forall T, {is_tm T} ->
   (exists M N, {is_tm M} /\ {is_tm N} /\ T = app M N)
     \/
   (exists R, (forall x, {is_tm x} -> {is_tm R x})
              /\ T = abs R).
```

Beyond Simple Types: LF (a.k.a. $\lambda\Pi$)

`http://abella-prover.org/lf`

- All kinds of typing relations can be reified.
- Encoding dependent types (and DTλ terms):

$$
\begin{array}{rclcrcl}
[\![\Pi x{:}A.\ U]\!] & = & [\![A]\!] \to [\![U]\!] & \qquad & [\![M\ N]\!] & = & [\![M]\!]\ [\![N]\!] \\
[\![a\ M_1\ \cdots\ M_n]\!] & = & a\ M_1\ \cdots\ M_n & & [\![\boldsymbol{\lambda} x{:}A.\ M]\!] & = & \boldsymbol{\lambda} x{:}[\![A]\!].\ [\![M]\!] \\
[\![\mathtt{type}]\!] & = & \mathtt{lftype} & & & &
\end{array}
$$

- Encoding typing as specification formulas.

$$
\begin{array}{rcl}
[\![M : \Pi x{:}A.\ U]\!] & = & \Pi x.\ [\![x : A]\!] \Rightarrow [\![M\,x : U]\!] \\
[\![M : P]\!] & = & \mathtt{hastype}\ [\![M]\!]\,[\![P]\!] \\
[\![A : \mathtt{type}]\!] & = & \mathtt{istype}\ [\![A]\!]
\end{array}
$$

- Encoding LF signatures

$$
[[c : U]] = \quad \begin{array}{l} \mathtt{type}\ c\ [\![U]\!]. \\ \mathtt{----} \\ [\![c : U]\!]\,. \end{array}
$$

Abella/LF Examples

Automation

- Many theorems about contexts are:
  - Tedious, and
  - Predictable
- This is particularly the case for regular contexts.
- We have a proof of concept for some rather sophisticated and certifying automation procedures (LFMTP 2014)
- Look out for it in Abella 2.1!

More Resources

Related Material

- See list on:

  `http://abella-prover.org/tutorial/`

- Extensive tutorial document: Abella: A System for Reasoning About Relational Specifications, J. Formalized Reasoning, 2014.
- Course notes by Gopalan Nadathur for: Specification and Reasoning About Computational Systems
- Book – Dale Miller and Gopalan Nadathur: Programming in Higher-Order Logic, CUP, 2012

Some Work in Progress

That I Know Of

- Compiler verification project in λProlog + Abella
  - Using step-indexed logical relations
  - Yuting Wang, Gopalan Nadathur
- ORBI-to-Abella
  - Alberto Momigliano & his student(s)
- Certified procedures for type checkers
  - Yuting Wang, Kaustuv Chaudhuri
- Polymorphism and reasoning modules
  - Polymorphic definitions and theorems already part of the upcoming Abella 2.0.4.
  - Polymorphic data being worked on by Yuting Wang
- Declarative proof language
  - Kaustuv Chaudhuri
- Exporting Abella proofs + model checking
  - Roberto Blanco, Quentin Heath, Dale Miller

Some Work in Progress

That I Know Of

- Compiler verification project in λProlog + Abella
  - Using step-indexed logical relations
  - Yuting Wang, Gopalan Nadathur
- ORBI-to-Abella
  - Alberto Momigliano & his student(s)
- Certified procedures for type checkers
  - Yuting Wang, Kaustuv Chaudhuri
- Polymorphism and reasoning modules
  - Polymorphic definitions and theorems already part of the upcoming Abella 2.0.4.
  - Polymorphic data being worked on by Yuting Wang
- Declarative proof language
  - Kaustuv Chaudhuri
- Exporting Abella proofs + model checking
  - Roberto Blanco, Quentin Heath, Dale Miller