Reasoning about Computational Systems using Abella
Kaustuv Chaudhuri, Gopalan Nadathur

To cite this version:
Kaustuv Chaudhuri, Gopalan Nadathur. Reasoning about Computational Systems using Abella. Abella Tutorial, Aug 2015, Berlin, Germany. 2015. hal-01222774

HAL Id: hal-01222774
https://inria.hal.science/hal-01222774
Submitted on 30 Oct 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NoDerivatives 4.0 International License
Reasoning about Computational Systems using Abella

http://abella-prover.org

Kaustuv Chaudhuri\textsuperscript{1} \quad Gopalan Nadathur\textsuperscript{2}

\textsuperscript{1}Inria & LIX/École polytechnique, France
\textsuperscript{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 consist 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 $G$
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

<table>
<thead>
<tr>
<th>Types</th>
<th>$A \to ((B \to C) \to D)$</th>
<th>$A \to (B \to C) \to D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>$(M , N) , (J , K)$</td>
<td>$M , N , (J , K)$</td>
</tr>
<tr>
<td>Abstraction</td>
<td>$\lambda x. , M$</td>
<td>$x , \backslash , M$</td>
</tr>
<tr>
<td></td>
<td>$\lambda x: A. , M$</td>
<td>$(x: A) , \backslash , M$</td>
</tr>
<tr>
<td>Formulas</td>
<td>$\top, \bot$</td>
<td>$true, false$</td>
</tr>
<tr>
<td></td>
<td>$F \land G, \quad F \lor G$</td>
<td>$F , \land , G, \quad F , \lor , G$</td>
</tr>
<tr>
<td></td>
<td>$F \supseteq G$</td>
<td>$F , \supseteq , G$</td>
</tr>
<tr>
<td></td>
<td>$\forall x, y. , F$</td>
<td>$forall , x , y, , F$</td>
</tr>
<tr>
<td></td>
<td>$\exists x : A, y. , F$</td>
<td>$exists , (x: A) , y, , F$</td>
</tr>
<tr>
<td></td>
<td>$M = N$</td>
<td>$M = N$</td>
</tr>
<tr>
<td></td>
<td>$\neg F$</td>
<td>$F , \rightarrow , false$</td>
</tr>
</tbody>
</table>
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

1 – Syntax
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
Ordinary Intuitionistic Logic

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.
  
  \begin{verbatim}
  Kind i  type.
  Type a,b  i.
  
  Theorem eq1 : a = a /\ b = b.
  Theorem eq2 : a = b -> false.
  \end{verbatim}

- 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 $\lambda$-convertible.

Consequences

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

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

Theorem eq1 : a = a \land 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 $\forall x: \text{nat}. \ F$ stands for:

$$[z/x]F \land [s \ z/x]F \land [s (s \ z)/x]F \land \cdots$$
Equality and Extensional Variables

\[
\text{forall} (x: \text{nat}) \ y, \ x = y \rightarrow F \ x \ y
\]

We have:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x = y</th>
<th>x = y \rightarrow F x y</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>z</td>
<td>true</td>
<td>F z z</td>
</tr>
<tr>
<td>z</td>
<td>anything else</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>s z</td>
<td>s z</td>
<td>true</td>
<td>F (s z) (s z)</td>
</tr>
<tr>
<td>s z</td>
<td>anything else</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

In other words, the formula is equivalent to:

\[
\text{forall} (x: \text{nat}), \ F x x
\]
More generally, given an assumption $M = N$:

1. Find all **unifiers** for $M$ and $N$.
   - A unifier of $M$ and $N$ is a substitution of terms for the free variables of $M$ and $N$ that makes them $\lambda$-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
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

<table>
<thead>
<tr>
<th>Functions</th>
<th>Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifies term language</td>
<td>No change to terms</td>
</tr>
<tr>
<td>Modifies equality</td>
<td>No change to equality</td>
</tr>
<tr>
<td>Requires confluence</td>
<td>Can be non-deterministic</td>
</tr>
<tr>
<td>Fixed inputs and output</td>
<td>Modes can vary</td>
</tr>
<tr>
<td>Functional programming</td>
<td>Logic programming</td>
</tr>
</tbody>
</table>
Relational Definitions

Define plus : nat -> nat -> nat -> \text{prop} by

\begin{align*}
\text{plus } z \ N \ N &; \\
\text{plus } (s \ M) \ N \ (s \ K) &:= \text{plus } M \ N \ K.
\end{align*}

- All defined relations must have target type \text{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 \( \text{plus1} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{prop} \) by

\[
\begin{align*}
\text{plus1}\ z\ N\ N\ ; \\
\text{plus1}\ (s\ M)\ N\ (s\ K) &:= \text{plus1}\ M\ N\ K.
\end{align*}
\]

is equivalent to

Define \( \text{plus2} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{prop} \) by

\[
\begin{align*}
\text{plus2}\ M\ N\ K &:= \\
&(M = z \lor N = K) \\
&\lor (\exists M' K', M = s M' \land K = s K' \land \\
&\quad \text{plus2}\ M'\ N\ K').
\end{align*}
\]
Proving Defined Atoms

If \( p \) is a defined relation, then to prove \( p \ M_1 \cdots M_n \):

1. Find a clause whose head matches with \( p \ M_1 \cdots M_n \);
2. Apply the matching substitution to its body;
3. and prove that instance of the body.

Backtracks over clauses and ways to match.
Define  \( \text{plus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{prop} \) by

\[
\begin{align*}
\text{plus} & \ z \ N \ N ; \\
\text{plus} & \ (s \ M) \ N \ (s \ K) := \text{plus} \ M \ N \ K.
\end{align*}
\]

Example: \( \text{plus} \ (s \ z) \ (s \ (s \ z)) \ (s \ (s \ (s \ z))) \):

1. Pick second clause with unifier \( [z / M, s(s \ z) / N, s(s \ z) / K] \).
2. Yields goal: \( \text{plus} \ z \ (s \ (s \ z)) \ (s \ (s \ z)) \).
3. Now pick first clause with unifier \( [s(s \ z) / N] \).
4. Yields goal \textit{true}, and we’re done!
Reasoning About Defined Atoms

To reason about hypothesis $p \ M_1 \ \cdots \ Mn$:

1. Find every way to unify $p \ M_1 \ \cdots \ Mn$ 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 $p \text{ M}_1 \cdots \text{ M}_n$:

1. Find **every** way to unify $p \text{ M}_1 \cdots \text{ M}_n$ 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 \( \text{plus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{prop} \) by
\[
\text{plus } z \ N \ N ; \\
\text{plus } (s \ M) \ N \ (s \ K) := \text{plus } M \ N \ K.
\]

Given hypothesis: \( \text{plus } M \ N \ (s \ K) \):

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

**Theorem** \( \text{plus}_s : \forall M \ N \ K, \text{plus } M \ N \ (s \ K) \rightarrow (\exists J, M = s \ J) \lor (\exists J, N = s \ J) \).
The **case and unfold** Tactics

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 $p : \text{prop}$ by

\[
p := p \rightarrow \text{false}.
\]

- Abella’s stratification condition guarantees consistency.
2.4 - Stratification
The Expressivity of \textit{case} and \textit{unfold}

Consider

\begin{verbatim}
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.
\end{verbatim}

- With \textit{case} and \textit{unfold}, we cannot prove:
  \[
  \forall x, \text{is_nat1 } x \rightarrow \text{is_nat2 } x.
  \]
  
- Abella actually interprets fixed points as \textit{least fixed points}.
- This in turn allows us to perform \textit{induction} on such definitions.
The **induction** tactic

Given a goal

\[
\text{forall } X_1 \ldots X_n, F_1 \to \ldots \to F_k \to \ldots \to G
\]

where \(F_k\) is a defined atom, the invocation

\[
\text{induction on } k.
\]

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

\[
\text{forall } X_1 \ldots X_n, F_1 \to \ldots \to F_k \to \ldots \to G
\]

2. Then **changes** the goal to:

\[
\text{forall } X_1 \ldots X_n, F_1 \to \ldots \to F_k \to \ldots \to G
\]
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 $\land$ and $\lor$;
  - Only the right operand of $\rightarrow$; and
  - The bodies of forall and exists.
Natural Number Induction
Lists of Natural Numbers

2.6 – Lists
Nested and Mutual Induction

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
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.

There is “one binder to ring them all.”

There is no such thing as a free variable.

– cf. Alan Perlis’ epigram #47

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.
Warmup: simple types.

\[ \begin{align*}
\text{Kind} & \quad \text{ty} \quad \text{type}. \\
\text{Type} & \quad \text{bas} \quad \text{ty}. \\
\text{Type} & \quad \text{arrow} \quad \text{ty} \to \text{ty} \to \text{ty}. \\
\end{align*} \]

\[
[b] = \text{bas} \quad \quad [A \rightarrow B] = \text{arrow} [A] [B]
\]
HOAS: Representing the Simply Typed Lambda Calculus

(Closed) $\lambda$-terms

<table>
<thead>
<tr>
<th>Kind</th>
<th>tm</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>app</td>
<td>tm $\rightarrow$ tm $\rightarrow$ tm.</td>
</tr>
<tr>
<td>Type</td>
<td>abs</td>
<td>(tm $\rightarrow$ tm) $\rightarrow$ tm.</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
[M N] &= \text{app} [M][N] \\
[\lambda x. M] &= \text{abs} (x \left[ [x/x]M \right]) \\
[x] &= x
\end{align*}
\]

Examples:

\[
\begin{align*}
[\lambda x. \lambda y. x] &= \text{abs} x \left[ \text{abs} y \left[ x \right] \right] \\
[\lambda x. \lambda y. \lambda z. x z (y z)] &= \text{abs} x \left[ \text{abs} y \left[ \text{abs} z \left[ \text{app} \left( \text{app} x z \right) \left( \text{app} y z \right) \right] \right] \right] \\
[\left( \lambda x. x x \right) \left( \lambda x. x x \right)] &= \text{app} \left( \text{abs} x \left[ \text{app} x x \right] \left[ \text{app} x x \right] \right)
\end{align*}
\]
HOAS: Representing the Typing Relation

\[
\frac{\Gamma, x:A \vdash M : B}{\Gamma \vdash (\lambda x. M) : A \rightarrow B}
\]

\[
\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}
\]

Kind ctx type.

Type emp ctx.

Type add ctx -> tm -> ty -> ctx.
HOAS: Representing the Typing Relation

\[ \frac{\Gamma, x:A \vdash x : A}{\Gamma, x:A \vdash x : A} \quad \frac{\Gamma \vdash (\lambda x. M) : A \rightarrow B}{\Gamma, x:A \vdash M : B} \]

\[ \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \]

Kind ctx type.

Type emp ctx.
Type add ctx -> tm -> ty -> ctx.
HOAS: Representing the Typing Relation

\[ \Gamma, x : A \vdash x : A \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x. M) : A \to B} \]

\[ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \]

Kind `ctx` type.

Type `emp` `ctx`.

Type `add` `ctx` \to `tm` \to `ty` \to `ctx`.
Define $\text{mem} : \text{ctx} \to \text{tm} \to \text{ty} \to \text{prop}$ by

\[
\text{mem} (\text{add} \ G \ X \ A) \ X \ A ; \\
\text{mem} (\text{add} \ G \ Y \ B) \ X \ A := \text{mem} \ G \ X \ A.
\]

\[
\begin{array}{c}
\Gamma, x:A \vdash x : A \\
\hline
\Gamma \vdash (\lambda x. M) : A \to B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash M : A \to B \\
\hline
\Gamma \vdash MN : B
\end{array}
\]

Define $\text{of} : \text{ctx} \to \text{tm} \to \text{ty} \to \text{prop}$ by

\[
\text{of} \ G \ X \ A := \text{mem} \ G \ X \ A ; \\
\text{of} \ G \ (\text{app} \ M \ N) \ B := \\
\quad \exists A, \text{of} \ M \ (\text{arrow} \ A \ B) \ \land \ \text{of} \ N \ A ; \\
\text{of} \ G \ (\text{abs} \ x \ M \ x) \ (\text{arrow} \ A \ B) := \\
\quad \text{of} \ (\text{add} \ G \ ?? \ A) \ (M ??) \ B
\]

\[
\begin{array}{c}
\Gamma, x:A \vdash x : A \\
\hline
\Gamma \vdash (\lambda x. M) : A \to B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash M : A \to B \\
\hline
\Gamma \vdash MN : B
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x:A \vdash x : A \\
\hline
\Gamma \vdash (\lambda x. M) : A \to B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash M : A \to B \\
\hline
\Gamma \vdash MN : B
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x:A \vdash x : A \\
\hline
\Gamma \vdash (\lambda x. M) : A \to B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash M : A \to B \\
\hline
\Gamma \vdash MN : B
\end{array}
\]
Define \( \text{mem} : \text{ctx} \rightarrow \text{tm} \rightarrow \text{ty} \rightarrow \text{prop} \) by
\[
\text{mem} \ (\text{add} \ G \ X \ A) \ X \ A \ ; \\
\text{mem} \ (\text{add} \ G \ Y \ B) \ X \ A \ := \ \text{mem} \ G \ X \ A.
\]

Define \( \text{of} : \text{ctx} \rightarrow \text{tm} \rightarrow \text{ty} \rightarrow \text{prop} \) by
\[
\text{of} \ G \ X \ A \ := \ \text{mem} \ G \ X \ A \ ; \\
\text{of} \ (\text{app} \ M \ N) \ B \ := \\
\quad \exists \ A, \ \text{of} \ M \ (\text{arrow} \ A \ B) \ \land \ \land \ \text{of} \ N \ A ; \\
\text{of} \ (\text{abs} \ x \ M \ x) \ (\text{arrow} \ A \ B) \ := \\
\quad \text{of} \ (\text{add} \ G \ ?? \ A) \ (M \ ??) \ B
\]

\[
\frac{}{\Gamma, x:A \vdash x:A} \quad \frac{\Gamma \vdash \lambda x. M : A \rightarrow B}{\Gamma, x:A \vdash \lambda x. M : A \rightarrow B} \quad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M \ N : B}
\]
What does $\Gamma, x:A$ mean?

- $x \not\in \text{fv}(\Gamma)$
- $x \not\in \text{fv}(A)$

- $(\Gamma, x:A)(y) = \begin{cases} A & \text{if } x = y \\ \Gamma(y) & \text{otherwise} \end{cases}$
Names and the $\forall$ (nabla) Quantifier

$\forall x. \, F$

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

$\forall x. \, F$

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 $\nabla$ vs. $\forall$

- $\nabla x. \nabla y. x \neq y$.
  - For any name $n \not\in \{\}$, it is that $\nabla y. n \neq y$.
  - For any name $n \not\in \{\}$, for any name $m \not\in \{n\}$, 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 \( \lambda \)-terms is restated in terms of \( \nabla \).

\[
(\lambda x. M) = (\lambda x. N) \text{ if and only if } \nabla x. (M = N).
\]

Why not \( \forall \)?

- **Differentiate** between the identity function \( \lambda x. x \) and the constant function \( \lambda x. c \).
- \( \forall x. (x = c) \) is satisfiable.
- \( \nabla x. (x = c) \) is false, i.e., \( \nabla x. (x = c) \) 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 identifier matching the regexp $n[0-9]+$ is considered to be a name.
Let \( \text{supp}(F) \) stand for the free names in \( F \).

\( \forall x. F: \)

[boxed text]

For every term \( M \), it is the case that \( [M/x]F \) is true.
Let $\text{supp}(F)$ stand for the free names in $F$.

$\forall x. \ F$:

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

\( \forall x. F: \)

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

- \( \forall x. \nabla y. p \ x \ y \)
  - For every term \( M \), it is that \( \nabla y. p \ M \ y. \)
  - For every \( M \), for any name \( n \notin \text{fn}(M) \), 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

\[
\begin{align*}
\Gamma, x : A & \vdash x : A \\
\Gamma & \vdash (\lambda x. M) : A \rightarrow B \\
\Gamma & \vdash M : A \rightarrow B \\
\Gamma & \vdash N : A \\
\Gamma & \vdash M N : B
\end{align*}
\]

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) \land 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

\[ \Gamma, x : A \vdash x : A \]
\[ \begin{array}{c}
\Gamma, x : A \vdash M : B \\
\Gamma \vdash (\lambda x. M) : A \rightarrow B
\end{array} \]
\[ \begin{array}{c}
\Gamma \vdash M : A \rightarrow B \\
\Gamma \vdash N : A \\
\Gamma \vdash MN : B
\end{array} \]

Define of : ctx \rightarrow tm \rightarrow ty \rightarrow 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

\[ \nabla \text{of } G \ (\text{abs } x \backslash M \ x) \ (\text{arrow } A \ B) := \]
\[ \nabla x, \ \text{of} \ (\text{add } G \ x \ A) \ (M \ x) \ B \]

means

\[ \forall G \ M \ A \ B, \]
\[ \text{of } G \ (\text{abs } x \backslash M \ x) \ (\text{arrow } A \ B) \leftarrow \]
\[ \nabla x, \ \text{of} \ (\text{add } G \ x \ A) \ (M \ x) \ B. \]

- None of \( G, M, A, B \) can mention \( x \).
- \( M \) can \textbf{indirectly} mention \( x \).
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).
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.
Let us start with intuitionistic minimal logic.

\[ F, G ::= A \mid F \Rightarrow G \mid \Pi x. F \]
\[ \Gamma ::= \cdot \mid \Gamma, F \]

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

\[ \Gamma \vdash F \quad \text{Goal decomposition sequent} \]
\[ \Gamma, [F] \vdash A \quad \text{Backchaining sequent} \]
Small Aside: A Bit of Proof Theory

Let us start with intuitionistic minimal logic.

\[ F, G ::= A \mid F \implies G \mid \Pi x. F \]
\[ \Gamma ::= \cdot \mid \Gamma, F \]

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

\[ \Gamma \vdash F \quad \text{Goal decomposition sequent} \]
\[ \Gamma, [F] \vdash A \quad \text{Backchaining sequent} \]
Focused Proof System

Goal decomposition

\[
\frac{\Gamma, F \vdash G}{\Gamma \vdash F \Rightarrow G} \quad \frac{(x \not\in \Gamma) \quad \Gamma \vdash F}{\Gamma \vdash \Pi x. F}
\]

Decision

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

Backchaining

\[
\frac{\Gamma \vdash F \quad \Gamma, [G] \vdash A}{\Gamma, [F \Rightarrow G] \vdash A} \quad \frac{\Gamma, [[t/x]F] \vdash A}{\Gamma, [\Pi x. F] \vdash A} \quad \frac{\Gamma, [A] \vdash A}{\Gamma, [A] \vdash A}
\]
Focused Proof System

Goal decomposition

\[
\frac{\Gamma, F \vdash G}{\Gamma \vdash F \Rightarrow G} \quad \frac{(x \not\in \Gamma) \quad \Gamma \vdash F}{\Gamma \vdash \Pi x. F}
\]

Decision

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

Backchaining

\[
\frac{\Gamma \vdash F \quad \Gamma, [G] \vdash A}{\Gamma, [F \Rightarrow G] \vdash A} \quad \frac{\Gamma, [[t/x]F] \vdash A}{\Gamma, [\Pi x. F] \vdash A} \quad \frac{\Gamma, [A] \vdash A}{\Gamma, [A] \vdash A}
\]
Focused Proof System

Goal decomposition

\[
\frac{\Gamma, F \vdash G}{\Gamma \vdash F \Rightarrow G} \quad \frac{(x \not\in \Gamma) \quad \Gamma \vdash F}{\Gamma \vdash \Pi x. F}
\]

Decision

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

Backchaining

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


Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

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

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

Consider the result of deciding on $R_1$ and $R_2$.

\[
\begin{align*}
\Gamma \vdash \text{of } M \text{ (arr } A B) & \quad \Gamma \vdash \text{of } N A & \quad \Gamma \vdash \text{of (app } M N \text{ ) } B \vdash C \\
\Gamma, [M/m, N/n, A/a, B/b] \cdots & \Rightarrow \cdots \Rightarrow \cdots \vdash C \\
\Gamma, [R_1] \vdash C & \quad \frac{\Gamma \vdash C \quad \Gamma \vdash \text{of } M \text{ (arr } A B) \quad \Gamma \vdash \text{of } N A}{\Gamma \vdash \text{of (app } M N \text{ ) } B}
\end{align*}
\]
Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

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

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

Consider the result of deciding on $R_1$ and $R_2$.

\[
\frac{\Gamma \vdash \text{of } M \ (\text{arr } A \ B) \quad \Gamma \vdash \text{of } N A \quad \Gamma, [\text{of } (\text{app } M \ N) \ B] \vdash C}{\Gamma, [[M/m, N/n, A/a, B/b] \ldots \Rightarrow \ldots \Rightarrow \ldots] \vdash C}
\]

\[
\frac{\Gamma, [R_1] \vdash C}{\Gamma \vdash C}
\]

\[
\frac{\Gamma \vdash \text{of } M \ (\text{arr } A \ B) \quad \Gamma \vdash \text{of } N A}{\Gamma \vdash \text{of } (\text{app } M \ N) \ B}
\]
Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

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

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

Consider the result of deciding on $R_1$ and $R_2$.

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

\[
\frac{\Gamma \vdash \text{of } M (\text{arr } A \ B) \quad \Gamma \vdash \text{of } N \ A}{\Gamma \vdash \text{of } (\text{app } M \ N) B}
\]
Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

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

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

Consider the result of deciding on $R_1$ and $R_2$.

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

$$
\Gamma \vdash \text{of } M \; (\text{arr } A \; B) \quad \Gamma \vdash \text{of } N \; A
\quad
\Gamma \vdash \text{of } (\text{app } M \; N) \; B
$$
Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

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

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

Consider the result of deciding on $R_1$ and $R_2$. 

$$
\begin{align*}
\Gamma \vdash \text{of } M (\text{arr } A B) & \quad \Gamma \vdash \text{of } N A \\
\Gamma, [\text{of } (\text{app } M N) B] \vdash \text{of } (\text{app } M N) B \\
\Gamma, [[M/m, N/n, A/a, B/b] \Rightarrow \cdots \Rightarrow \cdots] \vdash \text{of } (\text{app } M N) B \\
\Gamma, [R_1] \vdash \text{of } (\text{app } M N) B \\
\Gamma \vdash \text{of } (\text{app } M N) B
\end{align*}
$$

$$
\begin{align*}
\Gamma \vdash \text{of } M (\text{arr } A B) & \quad \Gamma \vdash \text{of } N A \\
\Gamma \vdash \text{of } (\text{app } M N) B
\end{align*}
$$
Synthetic (Derived) Rules

Imagine $\Gamma = R_1, R_2$ where:

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

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

Consider the result of deciding on $R_1$ and $R_2$.

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

\[
\frac{
\Gamma \vdash \text{of } M (\text{arr } A B) \quad \Gamma \vdash \text{of } N A
}{
\Gamma \vdash \text{of } (\text{app } M N) B
}\]
Synthetic (Derived) Rules

Imagine \( \Gamma = R_1, R_2 \) where:

\[ R_1: \ \Pi m, n, a, b. \text{of } m \ (\text{arr } a \ b) \Rightarrow \text{of } n \ a \Rightarrow \text{of } (\text{app } m \ n) \ b. \]

\[ R_2: \ \Pi r, a, b. (\Pi x. \text{of } x \ a \Rightarrow \text{of } (r \ x) \ b) \Rightarrow \text{of } (\text{abs } r) \ (\text{arr } a \ b). \]

Consider the result of deciding on \( R_1 \) and \( R_2 \).

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

\[
\Gamma \vdash \text{of } M \ (\text{arr } A \ B) \quad \Gamma \vdash \text{of } N \ A \\
\Gamma \vdash \text{of } (\text{app } M \ N) \ B
\]
Deciding on \( R_2 \)

\[
\begin{array}{l}
[1] \quad \Gamma, [\text{of (abs } R \text{) (arr } A \text{ B)}] \vdash \text{of (abs } R \text{) (arr } A \text{ B)} \\
\Gamma, [[R/r, A/a, B/b](\Pi x. \cdots \Rightarrow \cdots) \Rightarrow \cdots] \vdash \text{of (abs } R \text{) (arr } A \text{ B)} \\
\Gamma, [R_2] \vdash \text{of (abs } R \text{) (arr } A \text{ B)} \\
\Gamma \vdash \text{of (abs } R \text{) (arr } A \text{ B)}
\end{array}
\]

where \([1]\) is:

\[
(x\#\Gamma) \quad \Gamma, \text{of } xA \vdash \text{of } (Rx) B
\]

\[
\Gamma \vdash \Pi x. \text{of } xA \Rightarrow \text{of } (Rx) B
\]

So:

\[
(x\#\Gamma) \quad \Gamma, \text{of } xA \vdash \text{of } (Rx) B
\]

\[
\Gamma \vdash \text{of (abs } R \text{) (arr } A \text{ B)}
\]
Deciding on $R_2$

\[ \Gamma, [\text{of}(\text{abs } R) \ (\text{arr } A B)] \vdash \text{of}(\text{abs } R) \ (\text{arr } A B) \]

\[ \Gamma, [[R/r, A/a, B/b](\Pi x. \cdots \Rightarrow \cdots) \Rightarrow \cdots] \vdash \text{of}(\text{abs } R) \ (\text{arr } A B) \]

\[ \Gamma, [R_2] \vdash \text{of}(\text{abs } R) \ (\text{arr } A B) \]

\[ \Gamma \vdash \text{of}(\text{abs } R) \ (\text{arr } A B) \]

where \[ \square \] is:

\[ (x\#\Gamma) \quad \Gamma, \text{of } x \ A \vdash \text{of } (R \ x) \ B \]

\[ \Gamma \vdash \Pi x. \text{of } x \ A \Rightarrow \text{of } (R \ x) \ B \]

So:

\[ (x\#\Gamma) \quad \Gamma, \text{of } x \ A \vdash \text{of } (R \ x) \ B \]

\[ \Gamma \vdash \text{of}(\text{abs } R) \ (\text{arr } A B) \]
Synthetic Rules vs. SOS rules

\[
\begin{align*}
\Gamma \vdash M : A \to B & \quad \Gamma \vdash N : A \\
\Gamma \vdash (M \ N) : B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{of } M (\text{arr } A B) & \quad \Gamma \vdash \text{of } N A \\
\Gamma \vdash \text{of } (\text{app } M N) B
\end{align*}
\]

\[
\begin{align*}
\Gamma, x:A \vdash M : B \\
\Gamma \vdash (\lambda x. M) : A \to B
\end{align*}
\]

\[
\begin{align*}
(x\#\Gamma) \quad \Gamma, \text{of } x A \vdash \text{of } (R x) B \\
\Gamma \vdash \text{of } (\text{abs } R) (\text{arr } A B)
\end{align*}
\]

Reasoning about SOS derivations is isomorphic to reasoning about focused derivations for its minimal theory.
Synthetic Rules vs. SOS rules

\[
\begin{align*}
\Gamma \vdash M : A \rightarrow B & \quad \Gamma \vdash N : A \\
\Gamma \vdash (M \ N) : B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{of } M (\text{arr } A \ B) & \quad \Gamma \vdash \text{of } N A \\
\Gamma \vdash \text{of } (\text{app } M \ N) \ B
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : A & \vdash M : B \\
\Gamma \vdash (\lambda x. \ M) : A \rightarrow B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{of } x A & \vdash \text{of } (R \ x) \ B \\
\Gamma \vdash \text{of } (\text{abs } R) (\text{arr } A \ B)
\end{align*}
\]

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 $=>$ $o$ $\rightarrow$ $o$ $\rightarrow$ $o$.
Type $\pi$ $(A \rightarrow o) \rightarrow o$.

Kind olist type

Type $nil$ olist.
Type $::$ $o$ $\rightarrow$ olist $\rightarrow$ olist.

Define member : $o$ $\rightarrow$ olist $\rightarrow$ prop by ...

Sequent Encoding

\(\Gamma \vdash F\) seq L F
\(\Gamma, [F] \vdash A\) bch L F A
Minimal Logic Definable in $G$

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 ...
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.
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 $G$, but formulas of type $\circ$
  - Concrete syntax the same as $\lambda$Prolog

- **Reasoning Logic**
  - Inductive definition of the specification logic proof system
  - Inductive reasoning about specification logic derivations
  - Syntactic sugar:
    
    \[
    \text{seq } L \ F \quad \{L \mid - F\} \\
    \text{bch } L \ F \ A \quad \{L, [F] \mid - A\}
    \]
Example: STLC Specification

3.1 – Typing and Subject Reduction
Uniqueness of Typing

Change to a Church style representation:

```plaintext
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** typeuniq : forall M A B,

{of M A}  ->  {of M B}  ->  A = B.

Need to generalize!

**Theorem** typeuniq_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:

```plaintext
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 : \(\forall M \ A \ B,\) \(\{\text{of } M \ A\} \rightarrow \{\text{of } M \ B\} \rightarrow A = B.\)

Need to generalize!

**Theorem** type_uniq_open : \(\forall L \ M \ A \ B,\) \(\{L \mid \text{of } M \ A\} \rightarrow \{L \mid \text{of } M \ B\} \rightarrow A = B.\)
Uniqueness of Typing

Change to a Church style representation:

\[
\text{type } \text{abs ty} \rightarrow (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm}.
\]

\[
\text{of } (\text{abs A R}) (\text{arr A B}) :- \\
\pi \ x \ \text{of } x \ A \Rightarrow \text{of } (R \ x) \ B.
\]

Want to show that every term has a unique type.

**Theorem** type_uniq : \(\forall M \ A \ B,\) \(\{\text{of } M \ A\} \rightarrow \{\text{of } M \ B\} \rightarrow A = B.\)

Need to generalize!

**Theorem** type_uniq_open : \(\forall L \ M \ A \ B,\) \(\{L \ |- \ \text{of } M \ A\} \rightarrow \{L \ |- \ \text{of } M \ B\} \rightarrow A = B.\)
The typing dynamic context \( L \) is a list of assumptions. Already seen how to inductively define the structure of lists. Therefore:

\[
\text{Define } \text{ctx} : \text{olist} \rightarrow \text{prop} \text{ by }
\begin{align*}
\text{ctx} \text{ nil} ; \\
\text{ctx} \ (\text{of } X \ A :: L) & := \text{ctx} \ L.
\end{align*}
\]

But this does not capture \( x \# L \)!
"∇ In The Head"

Meaning of the second clause:

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

Let us change the “flavor” of x.

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

Equivalent to:

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

This suggests:

```plaintext
Define ctx : olist -> prop by
  ctx nil ;
  nabla x, ctx (of x A :: L) := ctx L.
```
“∇ In The Head”

Meaning of the second clause:

\[
\text{forall } L A X,
\begin{align*}
\text{ctx } L & \rightarrow \text{ctx } (\text{of } X A :: L).
\end{align*}
\]

Let us change the “flavor” of \(x\).

\[
\text{forall } L A, \text{ nabla } x,
\begin{align*}
\text{ctx } L & \rightarrow \text{ctx } (\text{of } x A :: L).
\end{align*}
\]

Equivalent to:

\[
\text{forall } L A, \text{ ctx } L \rightarrow
\begin{align*}
\text{nabla } x, \text{ ctx } (\text{of } x A :: L).
\end{align*}
\]

This suggests:

Define \(\text{ctx} : \text{olist} \rightarrow \text{prop}\) by

\[
\begin{align*}
\text{ctx } \text{nil} ; \\
\text{nabla } x, \text{ ctx } (\text{of } x A :: L) & := \text{ctx } L.
\end{align*}
\]
"∇ In The Head"

Meaning of the second clause:

\[
\forall L A X,\quad \text{ctx } L \rightarrow \text{ctx } (\text{of } X A :: L).
\]

Let us change the “flavor” of \(x\).

\[
\forall L A, \nabla x,\quad \text{ctx } L \rightarrow \text{ctx } (\text{of } x A :: L).
\]

Equivalent to:

\[
\forall L A, \text{ctx } L \rightarrow \nabla x, \text{ctx } (\text{of } x A :: L).
\]

This suggests:

\[
\text{Define } \text{ctx} : \text{olist } \rightarrow \text{prop} \text{ by}
\]
\[
\text{ctx } \text{nil} ;
\]
\[
\nabla x, \text{ctx } (\text{of } x A :: L) := \text{ctx } L.
\]
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 $\nabla$ In Heads

Clause head: $\text{nabla } x, \text{ctx (of } x \text{ A :: L)}$
Assumption: $H : \text{ctx (of } U \text{ B :: LL)}$

- $U$ must be a name ...
- ...that does not occur in $B$ or $LL$!
- Therefore, $\text{case } H$ picks an $n \notin \text{supp}(B) \cup \text{supp}(LL)$ for the unifier for $U$. 
Unification with $\nabla$ In Heads

Clause head: $\nabla x, \text{ctx (of x A :: L)}$

Assumption: $H : \text{ctx (of U B :: LL)}$

- $U$ must be a name ...
- ...that does not occur in $B$ or $LL$!
- Therefore, case $H$ picks an $n \notin \text{supp}(B) \cup \text{supp}(LL)$ for the unifier for $U$. 
Unification with $\nabla$ In Heads

Clause head: $\nabla x, \text{ctx} \ (\text{of} \ x \ A :: L)$
Assumption: $H : \text{ctx} \ (\text{of} \ U B :: LL)$

- $U$ must be a name ...
- ...that does not occur in $B$ or $LL$!
- Therefore, case $H$ picks an $n \notin \text{supp}(B) \cup \text{supp}(LL)$ for the unifier for $U$. 
Unification with $\nabla$ In Heads

Clause head: \texttt{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 $n \notin \text{supp}(B) \cup \text{supp}(LL)$ for the unifier for $U$. 
Unification with $\nabla$ In Heads

Clause head: \[ \nabla x, \text{ctx (of } x \ A :: L) \]
Assumption: \[ H : \text{ctx (of } n1 \ B :: (LL \ n1)) \]
Tactic: \[ \text{case } H. \]

Unification prunes $n1$ from $LL \ n1$.

Clause head: \[ \nabla x, \text{ctx (of } x \ A :: L) \]
Assumption: \[ H : \text{ctx (of } n1 \ B :: kon \ n1) \]
Tactic: \[ \text{case } H. \]

Cannot prune $n1$, so unification fails!
### Unification with $\nabla$ In Heads

**Clause head:** $\nabla x, \text{ctx (of } x A :: L)$

**Assumption:** $H : \text{ctx (of } n1 B :: (\text{LL } n1))$

**Tactic:** `case H`.

Unification *prunes* $n1$ from $\text{LL } n1$.

**Clause head:** $\nabla x, \text{ctx (of } x A :: L)$

**Assumption:** $H : \text{ctx (of } n1 B :: \text{kon } n1)$

**Tactic:** `case H`.

Cannot prune $n1$, so unification fails!
Unification with $\nabla$ In Heads

Clause head: $\nabla x, \text{ctx (of } x \ A :: L)\$
Assumption: $H : \text{ctx (of } n1 \ B :: (LL \ n1))$
Tactic: $\text{case } H$.

Unification prunes $n1$ from $LL \ n1$.

Clause head: $\nabla x, \text{ctx (of } x \ A :: L)\$
Assumption: $H : \text{ctx (of } n1 \ B :: \text{kon } n1)$
Tactic: $\text{case } H$.

Cannot prune $n1$, so unification fails!
Unification with $\nabla$ In Heads

Clause head: $\nabla x$, $\text{ctx (of } x \ A :: L)$
Assumption: $H : \text{ctx (of } n1 \ B :: (L \ L \ n1))$
Tactic: $\text{case } H.$

Unification prunes $n1$ from $L \ L \ n1$.

Clause head: $\nabla x$, $\text{ctx (of } x \ A :: L)$
Assumption: $H : \text{ctx (of } n1 \ B :: \text{kon } n1)$
Tactic: $\text{case } H.$

Cannot prune $n1$, so unification fails!
Some Puzzles

- Define \texttt{name : tm -> prop} that holds only for names.

\[
\text{Define name : tm -> prop by}
\]
\[
\text{nabla x, name x.}
\]

- Define \texttt{fresh : tm -> tm -> prop} such that \texttt{fresh X Y} means \texttt{x} is a name that does not occur in \texttt{Y}.

\[
\text{Define fresh : tm -> tm -> prop by}
\]
\[
\text{nabla x, fresh x Y.}
\]
Some Puzzles

- Define \texttt{name} : \texttt{tm} \rightarrow \texttt{prop} that holds only for names.

  \texttt{Define name : tm -> prop by}
  \begin{verbatim}
  nabla x, name x.
  \end{verbatim}

- Define \texttt{fresh} : \texttt{tm} \rightarrow \texttt{tm} \rightarrow \texttt{prop} such that \texttt{fresh X Y}
  means \texttt{x} is a name that does not occur in \texttt{y}.

  \texttt{Define fresh : tm -> tm -> prop by}
  \begin{verbatim}
  nabla x, fresh x Y.
  \end{verbatim}
Some Puzzles

• Define \texttt{name} : \texttt{tm} \rightarrow \texttt{prop} that holds only for names.

\begin{verbatim}
Define name : tm -> prop by
    nabla x, name x.
\end{verbatim}

• Define \texttt{fresh} : \texttt{tm} \rightarrow \texttt{tm} \rightarrow \texttt{prop} such that \texttt{fresh X Y} means \texttt{x} is a name that does not occur in \texttt{Y}.

\begin{verbatim}
Define fresh : tm -> tm -> prop by
    nabla x, fresh x Y.
\end{verbatim}
Some Puzzles

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

\[
\text{Define } name : \text{tm} \rightarrow \text{prop by} \\
\quad \text{nabla } x, \text{name } x.
\]

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

\[
\text{Define } fresh : \text{tm} \rightarrow \text{tm} \rightarrow \text{prop by} \\
\quad \text{nabla } x, \text{fresh } x \ Y.
\]
Extended Example: Uniqueness of Typing

3.2 - Type Uniqueness
Context Relations

No reason for \texttt{ctx} relations to be unary.

\begin{verbatim}
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.
\end{verbatim}

\begin{verbatim}
Define ctxs : olist -> olist -> prop by
  ctxs nil nil ;
  nabla x, ctxs (term x :: L) (neutral x :: K) :=
  ctxs L K.
\end{verbatim}
No reason for $\texttt{ctx}$ relations to be unary.

**Define** $\texttt{ctx\_len}$ : $\texttt{olist} \to \texttt{nat} \to \texttt{prop}$ by

\[
\texttt{ctx\_len} \texttt{nil} \texttt{z} ; \\
\texttt{nabla} \texttt{x}, \texttt{ctx\_len} \texttt{(of x A :: L)} \texttt{(s N)} := \\
\texttt{ctx\_len} \texttt{L} \texttt{N}.
\]

**Define** $\texttt{ctxs}$ : $\texttt{olist} \to \texttt{olist} \to \texttt{prop}$ by

\[
\texttt{ctxs} \texttt{nil} \texttt{nil} ; \\
\texttt{nabla} \texttt{x}, \texttt{ctxs} \texttt{(term x :: L)} \texttt{(neutral x :: K)} := \\
\texttt{ctxs} \texttt{L} \texttt{K}.
\]
Context Relations

No reason for \texttt{ctx} relations to be unary.

Define \texttt{ctx\_len} : \texttt{olist} \to \texttt{nat} \to \texttt{prop} by
\begin{verbatim}
  ctx\_len \texttt{nil} \texttt{z} ;
  nabla \textit{x}, ctx\_len (of \textit{x} \textit{A} :: \textit{L}) (s \textit{N}) :=
    ctx\_len \textit{L} \textit{N}.
\end{verbatim}

Define \texttt{ctxs} : \texttt{olist} \to \texttt{olist} \to \texttt{prop} by
\begin{verbatim}
  ctxs \texttt{nil} \texttt{nil} ;
  nabla \textit{x}, ctxs (term \textit{x} :: \textit{L}) (neutral \textit{x} :: \textit{K}) :=
    ctxs \textit{L} \textit{K}.
\end{verbatim}
Example: Partitioning of Lambda Terms

3.3 - Partitioning
Extended Example: Relating HOAS and De Bruijn Representations

3.4 – HOAS vs. Indexed
Co-Induction
Interpretations of Co-Induction

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

**Define** $p : \text{prop}$ by
$$p := p.$$  

**Theorem** $p_{th} : p \to \text{false}.$

**CoDefine** $q : \text{prop}$ by
$$q := q.$$  

**Theorem** $q_{th} : q.$
The **coinduction** Tactic

Given a goal

\[
\text{forall } X_1 \ldots X_n, \ F_1 \rightarrow \ldots \rightarrow F_n \rightarrow G
\]

where \( G \) is a co-inductively defined atom, the invocation

**coinduction**

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

\[
\text{forall } X_1 \ldots X_n, \ F_1 \rightarrow \ldots \rightarrow F_n \rightarrow G +
\]

2. Then **changes** the goal to:

\[
\text{forall } X_1 \ldots X_n, \ F_1 \rightarrow \ldots \rightarrow F_n \rightarrow G #.
\]
## Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Place</th>
<th>Tactic</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>hypothesis</td>
<td>case</td>
<td>*</td>
</tr>
<tr>
<td>@</td>
<td>goal</td>
<td>anything</td>
<td>no change</td>
</tr>
<tr>
<td>#</td>
<td>goal</td>
<td>unfold</td>
<td>+</td>
</tr>
<tr>
<td>#</td>
<td>hypothesis</td>
<td>anything</td>
<td>no change</td>
</tr>
</tbody>
</table>
Definition: *q simulates* *p*, written *p* \(\preceq\) *q*, iff:

- for every *p*′, *a* such that *p* \(\xrightarrow{a}\) *p*′,
- there is a *q*′ such that *q* \(\xrightarrow{a}\) *q*′, and
- *p*′ \(\preceq\) *q*′.

Here,

- *q*0 \(\preceq\) *p*0.
- *q*1 \(\preceq\) *p*0.
- *p*0 \(\nprec\) *q*0.
Example: Automata Simulation

4.1 - Automata
Example: Diverging $\lambda$-Terms
Summary So Far

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

- Higher-Order Abstract Syntax and $\nabla$
- 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 $\nabla$
- 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\textit{ is_nat} : \textit{n}at \rightarrow \textit{prop} by
\begin{align*}
is_{\text{nat}} z & ; \\
is_{\text{nat}} (s \; N) & := is_{\text{nat}} N. \\
\end{align*}

Theorem\textit{ nat}\_\textit{str} : \textbf{forall }\textit{x}, \textit{is}\_\textit{nat} \textit{x} \rightarrow \textit{x} = z \; \lor \; \exists y, \textit{is}\_\textit{nat} y \; \land \; \textit{x} = s \; y.

\ldots

But not always!

Define\textit{ is}\_\textit{tm} : \textit{tm} \rightarrow \textit{prop} by
\begin{align*}
is_{\text{tm}} (\text{app} \; M \; N) & := is_{\text{tm}} M \; \land \; is_{\text{tm}} N ; \\
is_{\text{tm}} (\text{abs} \; R) & := \text{\textit{nabla} }x, is_{\text{tm}} x \rightarrow is_{\text{tm}} (R \; x). \\
\end{align*}

This is not stratified.
Re-ifying Typing

Sometimes the typing relation can be reified.

Define \( \text{is_nat} : \text{nat} \rightarrow \text{prop} \) by
\[
\begin{align*}
\text{is_nat } z ; \\
\text{is_nat } (s \ N) & := \text{is_nat } N.
\end{align*}
\]

Theorem \( \text{nat_str} : \forall x, \text{is_nat } x \rightarrow \)
\[
\begin{align*}
x & = z \lor \exists y, \text{is_nat } y \land x = s \ y.
\end{align*}
\]

... But not always!

Define \( \text{is_tm} : \text{tm} \rightarrow \text{prop} \) by
\[
\begin{align*}
\text{is_tm } (\text{app } M \ N) & := \text{is_tm } M \land \text{is_tm } N ; \\
\text{is_tm } (\text{abs } R) & := \text{nabla } x, \text{is_tm } x \rightarrow \text{is_tm } (R \ x).
\end{align*}
\]

This is not stratified.
Re-ifying Typing

Sometimes the typing relation can be reified.

Define \(\text{is_nat} : \text{n} \to \text{prop} \) by
\[
is_{\text{nat}} z ; \\
is_{\text{nat}} (s \; N) := is_{\text{nat}} N.
\]

Theorem \(\text{nat}_\text{str} : \forall x, \text{is}_\text{nat} x \to x = z \lor \exists y, \text{is}_\text{nat} y \land x = s \; y.
\]

... But not always!

Define \(\text{is}_\text{tm} : \text{tm} \to \text{prop} \) by
\[
is_{\text{tm}} (\text{app} \; M \; N) := is_{\text{tm}} M \land is_{\text{tm}} N ; \\
is_{\text{tm}} (\text{abs} \; R) := \text{nabla} \; x, \text{is}_\text{tm} x \to is_{\text{tm}} (R \; x).
\]

This is not stratified.
Two-Level Reification

```plaintext
% 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** : \(\forall x, \{\text{is_nat } x\} \rightarrow x = z \lor \exists y, \{\text{is_nat } y\} \land x = s y.\)

**Theorem tm_str** : \(\forall T, \{\text{is_tm } T\} \rightarrow (\exists M N, \{\text{is_tm } M\} \land \{\text{is_tm } N\} \land T = \text{app } M N) \lor (\exists R, (\forall x, \{\text{is_tm } x\} \rightarrow \{\text{is_tm } R x\}) \land T = \text{abs } R).\)
Two-Level Reification

```plaintext
% 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

```plaintext
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 DT\(\lambda\) terms):

\[
\begin{align*}
\llbracket \Pi x : A. \ U \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket U \rrbracket \\
\llbracket M \ N \rrbracket &= \llbracket M \rrbracket \llbracket N \rrbracket \\
\llbracket a \ M_1 \cdots M_n \rrbracket &= a \ M_1 \cdots M_n \\
\llbracket \lambda x : A. \ M \rrbracket &= \lambda x : \llbracket A \rrbracket. \llbracket M \rrbracket \\
\llbracket \text{type} \rrbracket &= \text{lftype}
\end{align*}
\]

- Encoding typing as specification formulas.

\[
\begin{align*}
\llbracket M : \Pi x : A. \ U \rrbracket &= \Pi x. [x : A] \Rightarrow \llbracket M x : U \rrbracket \\
\llbracket M : P \rrbracket &= \text{hastype} \llbracket M \rrbracket \llbracket P \rrbracket \\
\llbracket A : \text{type} \rrbracket &= \text{istype} \llbracket A \rrbracket
\end{align*}
\]

- Encoding LF signatures

\[
\begin{align*}
\llbracket [c : U] \rrbracket &= \text{type} \ c \llbracket U \rrbracket.
\end{align*}
\]
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/


• Course notes by Gopalan Nadathur for: Specification and Reasoning About Computational Systems

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 $\lambda$Prolog + 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