Mechanized metatheory revisited

Dale Miller

Inria Saclay & LIX, École Polytechnique
Palaiseau, France

TYPES 2016, Novi Sad
25 May 2016

Theory vs Metatheory

When formalizing programming languages, we often have to deal with *theorems* such as

- $\vdash M \Downarrow V$,
- $\vdash \Gamma$ context,
- $\Gamma \vdash M \colon \tau$, and
- $\vdash$cps $M\ \ \hat{M}$.

Such provability judgments are generally given inductively using inference rules encoding *structured operational semantics* and *typing rules*.

Of course, the real prize is proving *metatheorems* about entire programming languages or specification languages.

- $\vdash \forall M, V, U.\ (\vdash M \Downarrow V) \supset (\vdash M \Downarrow U) \supset U = V$
- $\vdash \forall M, V, T.\ (\vdash M \Downarrow V) \supset (\vdash M : T) \supset (\vdash M : V)$

Metatheory is unlike other domains

Formalizing metatheory requires dealing with *linguistic* items (e.g., types, terms, formulas, proofs, programs, etc) which are not typical data structures (e.g., integers, trees, lists, etc).

The authors of the POPLmark challenge tried metatheory problems on existing systems and urged the developers of proof assistants to make improvements:

> *Our conclusion [...] is that the relevant technology has developed almost to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research [TPHOLS 2005]*

That is: existing systems are close but need additional engineering.

A major obstacle: bindings

Linguistic expressions generally involve bindings. Our formal tools need to

- acknowledge that bindings are special aspects of parsed syntax and
- provide support for bindings in syntax within proof principles (e.g., induction and pattern matching).

In the 11 years since the POPLmark challenge, several approaches to binding syntax have been made within mature theorem provers:

- locally nameless,
- nominal reasoning, and
- parametric higher-order abstract syntax.

None seem canonical.

Sometimes additional engineering is not enough

An analogy: Early and mature programming languages provided treatments of concurrency and distributed computing in ways:

- thread packages,
- remote procedure calls, and
- tuple space (Linda).

Such approaches addressed important needs. None-the-less, early pioneers (Dijkstra, Hoare, Milner, Petri) considered new ways to express and understand concurrency via formalisms such as CCS, CSP, Petri Nets, π-calculus, etc. None seem canonical.

In a similar spirit, we examine here an approach to metatheory that is not based on extending mature theorem proving platforms.

We keep the *scope* but not the *approach* of the POPLmark challenge.

Major first step: Drop mathematics as an intermediate

A traditional approach to formalizing metatheory.

1. Implement mathematics
   - Pick a rich logic: intuitionistic higher-order logic, classical first-order logic, set theory, etc.
   - Provide abstractions such as sets, functions, etc.
2. Model computation via mathematical structures:
   - via denotational semantics and/or
   - via inductively defined data types and proof systems.

What could be wrong with this approach? Isn't mathematics the universal language?

Major first step: Drop mathematics as an intermediate

A traditional approach to formalizing metatheory.

1. Implement mathematics
   - Pick a rich logic: intuitionistic higher-order logic, classical first-order logic, set theory, etc.
   - Provide abstractions such as sets, functions, etc.
2. Model computation via mathematical structures:
   - via denotational semantics and/or
   - via inductively defined data types and proof systems.

What could be wrong with this approach? Isn't mathematics the universal language?

Various "intensional aspects" of computational specifications — bindings, names, resource accounting, etc — are challenges to this approach to reasoning about computation.

Examples of intensional aspects of expressions

Consider *algorithms*: two sort programs describe the same function but should not be replaced in all contexts.

A more explicit example: Is the following a theorem?

$$\forall w_i \; \neg(\lambda x.x = \lambda x.w) \qquad (*)$$

Examples of intensional aspects of expressions

Consider *algorithms*: two sort programs describe the same function but should not be replaced in all contexts.

A more explicit example: Is the following a theorem?

$$\forall w_i \neg(\lambda x.x = \lambda x.w) \tag{$*$}$$

If λ-abstractions denote functions, $(*)$ is equivalent to

$$\forall w_i \neg \forall x(x = w).$$

This is not a theorem (consider the singleton model).

Examples of intensional aspects of expressions

Consider *algorithms*: two sort programs describe the same function but should not be replaced in all contexts.

A more explicit example: Is the following a theorem?

$$\forall w_i \; \neg(\lambda x.x = \lambda x.w) \tag{$*$}$$

If λ-abstractions denote functions, $(*)$ is equivalent to

$$\forall w_i \; \neg \; \forall x(x = w).$$

This is not a theorem (consider the singleton model).

If λ-abstractions denote syntactic expressions, then $(*)$ should be a theorem since no (capture avoiding) substitution of an expression of type i for the w in $\lambda x.w$ can yield $\lambda x.x$.

Two Type Theories of Church [JSL 1940]

Tension between a logic for metatheory and for mathematics.

Axioms 1-6: Elementary Type Theory (ETT). Foundations for a higher-order predicate calculus.

Two Type Theories of Church [JSL 1940]

Tension between a logic for metatheory and for mathematics.

Axioms 1-6: Elementary Type Theory (ETT). Foundations for a higher-order predicate calculus.

Axioms 7-11: Simple Theory of Types (STT)

- non-empty domains
- Peano's axioms,
- axioms of description and choice, and
- extensionality for functions.

Adding these gives us a foundations for much of mathematics.

Two Type Theories of Church [JSL 1940]

Tension between a logic for metatheory and for mathematics.

Axioms 1-6: Elementary Type Theory (ETT). Foundations for a higher-order predicate calculus.

Axioms 7-11: Simple Theory of Types (STT)

- non-empty domains
- Peano's axioms,
- axioms of description and choice, and
- extensionality for functions.

Adding these gives us a foundations for much of mathematics.

With extensionality, description, and choice, STT goes too far for our interests in metatheory.

We keep to ETT and eventually extend it for our metatheory needs.

Simple types as syntactic categories

The type o (omicron) is the type of formulas.

Other primitive types provide for multisorted terms.

The arrow type denotes the syntactic category of one syntactic category over another.

For example, the universal quantifier $\forall_\tau$ is not applied to a term of type τ and a formula (of type o) but rather to an abstraction of type $\tau \rightarrow o$.

Both $\forall_\tau$ and $\exists_\tau$ belong to the syntactic category $(\tau \rightarrow o) \rightarrow o$.

Typing in this sense is essentially the same as Martin-Löf's notion of *arity types*.

Proof theory for induction and coinduction

Following Gentzen, proof theory for both intuitionistic and classical versions of ETT have been studied.

Recent work adds to ETT equality, induction, and coinduction.

- 2000: R. McDowell & M, "Cut-Elimination for a Logic with Definitions and Induction", TCS.
- 2004: A. Tiu, "A Logical Framework for Reasoning about Logical Specifications", PhD.
- 2008: D. Baelde, "A linear approach to the proof-theory of least and greatest fixed points", PhD.
- 2011: A. Gacek, M, G. Nadathur "Nominal abstraction", I&C.

(The last three papers also deal with the ∇-quantifier.)

A framework for the metatheory of programming languages

A framework for metatheory should accommodate the following features.

1. Relational specifications, not functional specifications, appear to be primitive: for example, $M \Downarrow V$ and $\Gamma \vdash M : \tau$.
2. Semantic specification as inference rules (e.g., SOS, typing, etc).
3. Inductive and co-inductive reasoning about provability.
4. Variable binding and their concomitant operations need to be supported.

We will eventually show that all these features are treated within a single logic: ETT plus induction, coinduction, ∇-quantification.

Semantics as inference rules

Both the dynamic and static semantics of programming languages are generally given using relations and inference rules.

CCS and π-calculus transition system:

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad\qquad \frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}} \quad \begin{array}{l} y \neq x \\ w \notin fn((y)P') \end{array}$$

Functional programming evaluation:

$$\frac{M \Downarrow \lambda x.R \quad N \Downarrow U \quad S \Downarrow V}{(M\ N) \Downarrow V} \quad S = R[N/x]$$

Typing of terms:

$$\frac{\Gamma, x\colon \tau \vdash t\colon \sigma}{\Gamma \vdash \lambda x.t\colon \tau \to \sigma} \quad x \notin fn(\Gamma)$$

How abstract is your syntax?

Gödel and Church did their formal metatheory on string representation of formulas! Today, we parse strings into *abstract syntax* (a.k.a *parse trees*). But how abstract is that syntax?

Principle 1: *The names of bound variables should be treated as the same kind of fiction as white space.*

Principle 2: *There is "one binder to ring them all."*[1]

Principle 3: *There is no such thing as a free variable. (Alan Perlis's epigram 47.)*

Principle 4: *Bindings have* mobility *and the equality theory of expressions must support such mobility.*

[1] A scrambling of J. R. R. Tolkien's "One Ring to rule them all, ... and in the darkness bind them."

α, β_0, and η conversions

β_0-conversion rule

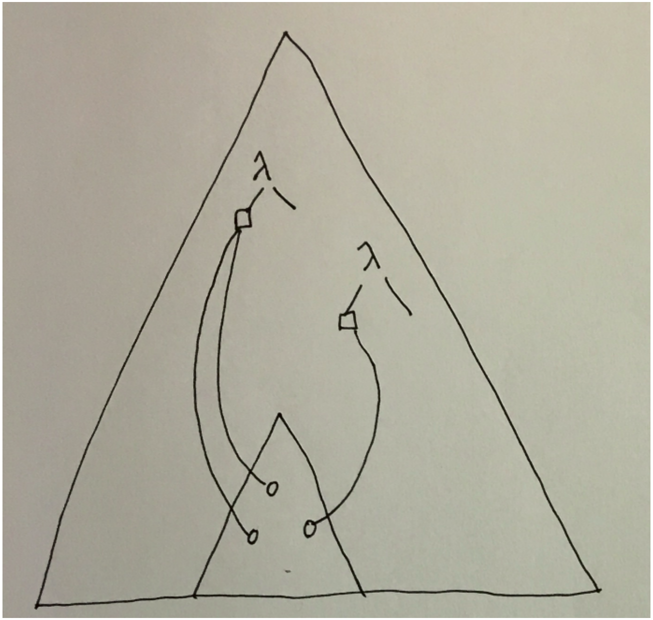
- $(\lambda x.t)x = t$ or equivalently
- $(\lambda y.t)x = t[x/y]$, provided that x is not free in $\lambda y.t$.

β_0 reduction makes terms smaller. Mobility: an internal bound variable y is replaced by an external (bound) variable x.

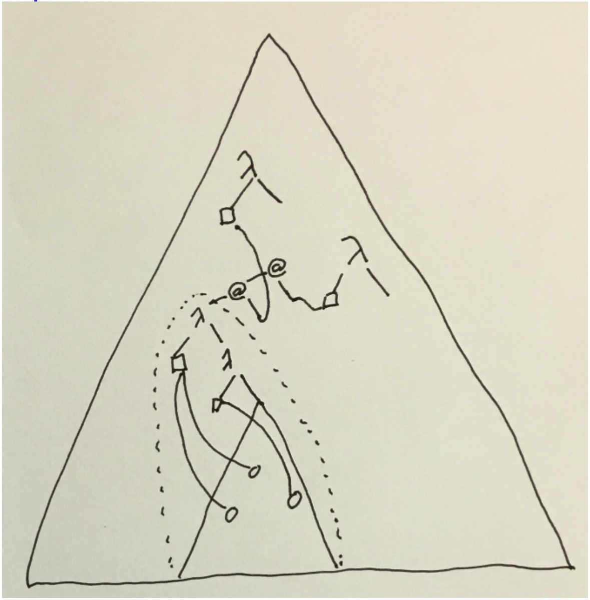
Note the symmetry:

- if t is a term over the signature $\Sigma \cup \{x\}$ then $\lambda x.t$ is a term over the signature Σ and
- if $\lambda x.s$ is a term over the signature Σ then the β_0 reduction of $((\lambda x.s)\ y)$ is a term over the signature $\Sigma \cup \{y\}$.

Rewriting a subterm with external bound variables

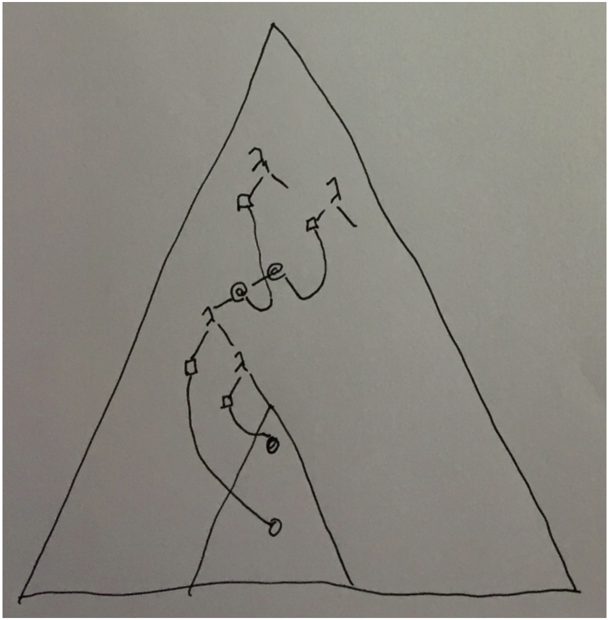


β_0-expansion

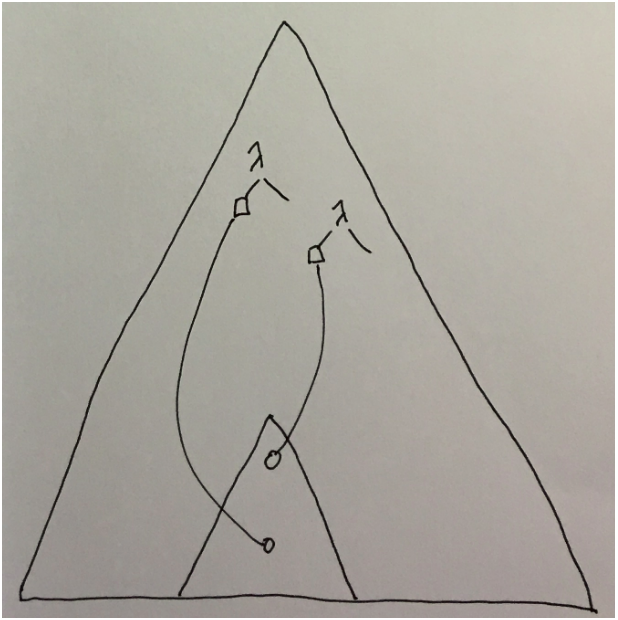


Replace $t(x, y)$ with $(\lambda u \lambda v.t(u, v))\ x\ y$

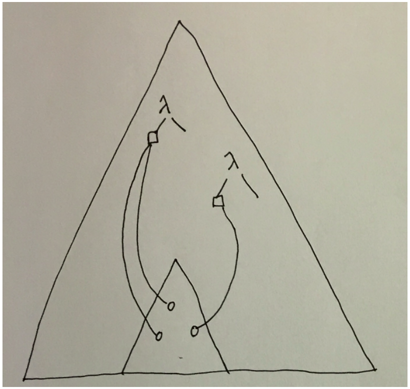
Replacement of abstracted subterm

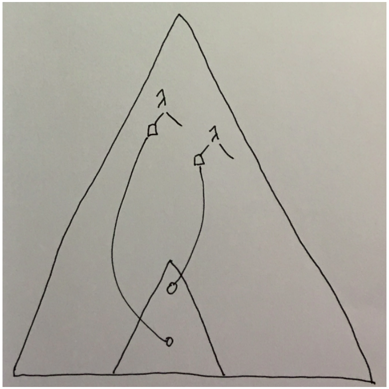


β_0-reduction



One step rewriting modulo β_0





The contextual modal type theory of Nanevski, Pfenning, and Pientka [2008] provides another approach to binder mobility.

Unification of λ-terms

Since β_0 is such a weak rule, unification of simply typed λ-terms modulo α, β_0, and η is decidable.

Higher-order pattern unification has the restriction that meta-variables can be applied to only *distinct bound variables*. With that restriction, unification modulo $\beta_0\eta$ is complete for unification modulo $\beta\eta$.

Such unification does not require type information. Thus, it can be moved to many different typed settings.

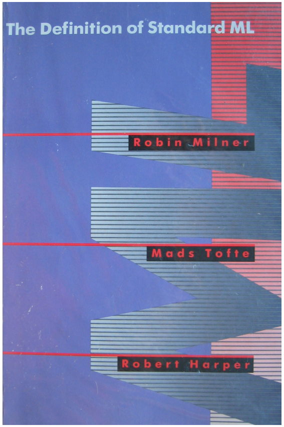
In the π-calculus literature there is a notion of “internal mobility” captured by the π_I-calculus of Sangiorgi [1996]. In this fragment, β_0 is the only form of β that is needed to bind input variables to outputs.

HOAS vs λ-tree syntax

Higher-order abstract syntax is a technique that maps object language bindings to meta-language bindings [Pfenning & Schürmann, CADE99]. Given that programming languages differ greatly, this identified is ambiguous.

HOAS vs λ-tree syntax

Higher-order abstract syntax is a technique that maps object language bindings to meta-language bindings [Pfenning & Schürmann, CADE99]. Given that programming languages differ greatly, this identified is ambiguous.



In functional programming, HOAS implies using function spaces to denote bindings.

Thus, there is no built-in notion of equality for HOAS.

This is semantically odd approach to syntax.

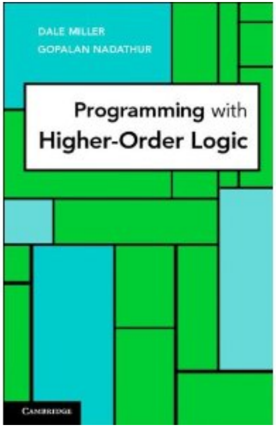
HOAS vs λ-tree syntax

Higher-order abstract syntax is a technique that maps object language bindings to meta-language bindings [Pfenning & Schürmann, CADE99]. Given that programming languages differ greatly, this identified is ambiguous.

In logic programming, HOAS implies using term-level bindings, which are available in, say, λProlog.

Built-in equality incorporates α-conversion. Capture-avoiding substitution is provided by β-reduction.

We use *λ-tree syntax* to denote this approach to encoding (also in Isabelle, Twelf, Minlog, Beluga, ...)



λ-tree syntax illustrated

Encode the rule $$\frac{M \Downarrow \lambda x.R \quad N \Downarrow U \quad S \Downarrow V}{(M\ N) \Downarrow V} \quad S = R[N/x]$$

as $$\frac{M \Downarrow (abs\ R) \qquad N \Downarrow U \qquad (R\ U) \Downarrow V}{(app\ M\ N) \Downarrow V}.$$

In λProlog syntax:

```
kind tm type.
type abs (tm -> tm) -> tm.
type app tm -> tm -> tm.

eval (app M N) V :-
    eval M (abs R), eval N U, eval (R U) V.
```

Binding a variable in a proof

When proving a universal quantifier, one uses a “new” or “fresh” variable.

$$\frac{B_1, \ldots, B_n \vdash Bv}{B_1, \ldots, B_n \vdash \forall x_\tau . Bx}\ \forall\mathcal{R},$$

provided that v is a “new” variable (not free in the lower sequent). Gentzen called such new variables *eigenvariables*.

But this violates the “Perlis principle.” Instead, we write

$$\frac{\Sigma, v : \tau\ :\ B_1, \ldots, B_n \vdash Bv}{\Sigma\ :\ B_1, \ldots, B_n \vdash \forall x_\tau . Bx}\ \forall\mathcal{R},$$

The variables in the signature context are bound in the sequent.

Eigenvariables are proof-level bindings.

Dynamics of binders during proof search

During proof search, binders can be *instantiated* (using β implicitly)

$$\frac{\Sigma : \Delta, \mathit{typeof}\ c\ (\mathit{int} \to \mathit{int}) \vdash C}{\Sigma : \Delta, \forall\alpha(\mathit{typeof}\ c\ (\alpha \to \alpha)) \vdash C}\ \forall\mathcal{L}$$

They also have *mobility* (they can move):

$$\cfrac{\cfrac{\Sigma, x : \Delta, \mathit{typeof}\ x\ \alpha \vdash \mathit{typeof}\ \lceil B \rceil\ \beta}{\Sigma : \Delta \vdash \forall x(\mathit{typeof}\ x\ \alpha \supset \mathit{typeof}\ \lceil B \rceil\ \beta)}\ \forall\mathcal{R}}{\Sigma : \Delta \vdash \mathit{typeof}\ \lceil \lambda x.B \rceil\ (\alpha \to \beta)}$$

In this case, the binder named x moves from *term-level* (λx) to *formula-level* ($\forall x$) to *proof-level* (as an eigenvariable in Σ, x).

Only β_0 conversion is needed for mobility.

An example: call-by-name evaluation and simple typing

We want to do more than "animate" or "execute" a specification.

We want to prove properties about the specifications. We illustrate with a proof of type preservation (subject-reduction).

$$(\mathit{eval}\ \ M\ (\mathit{abs}\ R) \land \mathit{eval}\ \ (R\ \ N)\ V) \supset \mathit{eval}\ \ (\mathit{app}\ M\ \ N)\ V$$

$$\mathit{eval}\ \ (\mathit{abs}\ R)\ (\mathit{abs}\ R)$$

$$(\mathit{typeof}\ M\ (\mathit{arr}\ A\ B) \land \mathit{typeof}\ N\ A) \supset \mathit{typeof}\ (\mathit{app}\ M\ \ N)\ B$$

$$\forall x[\mathit{typeof}\ x\ A \supset \mathit{typeof}\ (R\ x)\ B] \supset \mathit{typeof}\ (\mathit{abs}\ R)\ (\mathit{arr}\ A\ B)$$

The first three clauses are Horn clauses; the fourth is not.

Proof of type preservation

Theorem: If $\vdash \mathit{eval}\ \ P\ V$ and $\vdash \mathit{typeof}\ P\ \ T$ then $\vdash \mathit{typeof}\ V\ \ T$.
Proof: By structural induction on a proof of $\mathit{eval}\ \ P\ V$.

There are two ways to prove $\vdash \mathit{eval}\ \ P\ V$.

Proof of type preservation

Theorem: If $\vdash \textit{eval}\ \ P\ V$ and $\vdash \textit{typeof}\ P\ \ T$ then $\vdash \textit{typeof}\ V\ \ T$.
Proof: By structural induction on a proof of $\textit{eval}\ \ P\ V$.

There are two ways to prove $\vdash \textit{eval}\ \ P\ V$.

Case eval-abs: Thus P and V are equal to $(\textit{abs}\ \ R)$, for some R.
The consequent is immediate.

Proof of type preservation

Theorem: If $\vdash \mathit{eval}\ \ P\ V$ and $\vdash \mathit{typeof}\ P\ \ T$ then $\vdash \mathit{typeof}\ V\ \ T$.
Proof: By structural induction on a proof of $\mathit{eval}\ \ P\ V$.

There are two ways to prove $\vdash \mathit{eval}\ \ P\ V$.

Case eval-abs: Thus P and V are equal to $(\mathit{abs}\ \ R)$, for some R. The consequent is immediate.

Case eval-app: P is of the form $(\mathit{app}\ M\ \ N)$ and for some R, there are shorter proofs of $\vdash \mathit{eval}\ \ M\ (\mathit{abs}\ R)$ and $\vdash \mathit{eval}\ \ (R\ N)\ V$.

Proof of type preservation

Theorem: If $\vdash \mathit{eval}\ \ P\ V$ and $\vdash \mathit{typeof}\ P\ T$ then $\vdash \mathit{typeof}\ V\ T$.
Proof: By structural induction on a proof of $\mathit{eval}\ \ P\ V$.

There are two ways to prove $\vdash \mathit{eval}\ \ P\ V$.

Case eval-abs: Thus P and V are equal to $(\mathit{abs}\ \ R)$, for some R. The consequent is immediate.

Case eval-app: P is of the form $(\mathit{app}\ M\ N)$ and for some R, there are shorter proofs of $\vdash \mathit{eval}\ \ M\ (\mathit{abs}\ R)$ and $\vdash \mathit{eval}\ \ (R\ N)\ V$.

Since $\vdash \mathit{typeof}\,(\mathit{app}\ M\ N)\ T$ there must be a U such that $\vdash \mathit{typeof}\ M\ (\mathit{arr}\ U\ T)$ and $\vdash \mathit{typeof}\ N\ U$.

Proof of type preservation

Theorem: If $\vdash eval\ \ P\ V$ and $\vdash typeof\, P\ T$ then $\vdash typeof\, V\ T$.
Proof: By structural induction on a proof of $eval\ \ P\ V$.

There are two ways to prove $\vdash eval\ \ P\ V$.

Case eval-abs: Thus P and V are equal to $(abs\ \ R)$, for some R.
The consequent is immediate.

Case eval-app: P is of the form $(app\ M\ N)$ and for some R, there are shorter proofs of $\vdash eval\ \ M\ (abs\ R)$ and $\vdash eval\ \ (R\, N)\ V$.

Since $\vdash typeof\,(app\ M\ N)\ T$ there must be a U such that
$\vdash typeof\, M\ (arr\ U\ T)$ and $\vdash typeof\, N\ U$.

Using the inductive hypothesis, we have $\vdash typeof\,(abs\ R)\ (arr\ U\ T)$ and, hence, $\vdash \forall x.[typeof\, x\ U \supset typeof\,(R\, x)\ T]$.

Proof of type preservation

Theorem: If $\vdash eval\ \ P\ V$ and $\vdash typeof\, P\ T$ then $\vdash typeof\, V\ T$.
Proof: By structural induction on a proof of $eval\ \ P\ V$.

There are two ways to prove $\vdash eval\ \ P\ V$.

Case eval-abs: Thus P and V are equal to $(abs\ \ R)$, for some R. The consequent is immediate.

Case eval-app: P is of the form $(app\, M\ N)$ and for some R, there are shorter proofs of $\vdash eval\ \ M\ (abs\, R)$ and $\vdash eval\ \ (R\, N)\ V$.

Since $\vdash typeof\, (app\, M\ N)\ T$ there must be a U such that $\vdash typeof\, M\ (arr\, U\ T)$ and $\vdash typeof\, N\ U$.

Using the inductive hypothesis, we have $\vdash typeof\, (abs\, R)\ (arr\, U\ T)$ and, hence, $\vdash \forall x.[typeof\, x\ U \supset typeof\, (R\, x)\ T]$.

By properties of logic, we can instantiate this quantifier with N and use cut (modus ponens) to conclude that $\vdash typeof\, (R\, N)\ T$.
(A substitution lemma for free!)

Proof of type preservation

Theorem: If $\vdash \mathit{eval}\ \ P\ V$ and $\vdash \mathit{typeof}\ P\ \ T$ then $\vdash \mathit{typeof}\ V\ \ T$.
Proof: By structural induction on a proof of $\mathit{eval}\ \ P\ V$.

There are two ways to prove $\vdash \mathit{eval}\ \ P\ V$.

Case eval-abs: Thus P and V are equal to $(\mathit{abs}\ \ R)$, for some R. The consequent is immediate.

Case eval-app: P is of the form $(\mathit{app}\ M\ \ N)$ and for some R, there are shorter proofs of $\vdash \mathit{eval}\ \ M\ (\mathit{abs}\ R)$ and $\vdash \mathit{eval}\ \ (R\ N)\ V$.

Since $\vdash \mathit{typeof}\,(\mathit{app}\ M\ \ N)\ T$ there must be a U such that $\vdash \mathit{typeof}\ M\ (\mathit{arr}\ U\ \ T)$ and $\vdash \mathit{typeof}\ N\ \ U$.

Using the inductive hypothesis, we have $\vdash \mathit{typeof}\,(\mathit{abs}\ R)\ (\mathit{arr}\ U\ \ T)$ and, hence, $\vdash \forall x.[\mathit{typeof}\ x\ \ U \supset \mathit{typeof}\,(R\,x)\ \ T]$.

By properties of logic, we can instantiate this quantifier with N and use cut (modus ponens) to conclude that $\vdash \mathit{typeof}\,(R\,N)\ \ T$.
(A substitution lemma for free!)

Using the inductive hypothesis again yields $\vdash \mathit{typeof}\ V\ \ T$. QED

A fully formal proof in Abella

```
Theorem type-preserve :  forall E V T,
 {|- eval E V} -> {|- typeof E T} -> {|- typeof V T}.

induction on 1. intros. case H1.
  search.
  case H2. apply IH to H3 H5. case H7.
           inst H8 with n1 = N.
           cut H9 with H6.
           apply IH to H4 H10. search.
```

The `inst` command instantiates $\forall x.[\mathit{typeof}\, x\ U \supset \mathit{typeof}\,(R\, x)\ T]$ to get $[\mathit{typeof}\, N\ U \supset \mathit{typeof}\,(R\, N)\ T]$.

The `cut` command applies that implication to the hypothesis $\mathit{typeof}\, N\ U$.

Something is missing

Type preservation theorems are too simple, given that substitution lemmas are free. Turn to simple but more general meta-theoretic questions.

Consider the following problem about reasoning with an object-logic. The formula

$$\forall u \forall v [q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle]$$

is provable from the assumptions

$$\mathcal{H} = \{\forall x \forall y [q\ x\ x\ y], \quad \forall x \forall y [q\ x\ y\ x], \quad \forall x \forall y [q\ y\ x\ x]\}$$

only if terms t_2 and t_3 are

Something is missing

Type preservation theorems are too simple, given that substitution lemmas are free. Turn to simple but more general meta-theoretic questions.

Consider the following problem about reasoning with an object-logic. The formula

$$\forall u \forall v [q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle]$$

is provable from the assumptions

$$\mathcal{H} = \{\forall x \forall y [q\ x\ x\ y], \quad \forall x \forall y [q\ x\ y\ x], \quad \forall x \forall y [q\ y\ x\ x]\}$$

only if terms t_2 and t_3 are *equal*.

We would like to prove a meta-level formula like

$$\forall t_1, t_2, t_3. \{\mathcal{H} \vdash (\forall u \forall v [q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle])\} \supset t_2 = t_3$$

It seems we need a treatment of "new" or "fresh" variables.

A stronger form of the ξ rule

The usual form of the ξ rule is given as

$$\frac{t = s}{\lambda x.t = \lambda x.s}$$

As written, this violates the "Perlis principle". If we fix this with

$$\frac{\forall x.t = s}{\lambda x.t = \lambda x.s}$$

then $(\forall x.t = s) \equiv (\lambda x.t = \lambda x.s)$ which is not appropriate for reasoning with λ-tree syntax since we want $\forall w_i\ \neg(\lambda x.x = \lambda x.w)$ to be provable. The ∇-quantifier addresses this problem:

$$\frac{\nabla x.t = s}{\lambda x.t = \lambda x.s}$$

The formula $\forall w_i\ \neg\nabla x.x = w$ is provable [M & Tiu, LICS 2003].

A new quantifier ∇

∇-quantification is similar to Pitt's freshness quantifier [FAC 2002]. Both are self dual $\nabla x \neg Bx \equiv \neg \nabla x Bx$ and in weak settings (roughly Horn clauses), they do coincide [Gacek, PPDP 2010].

To accommodate a new quantifier, we need a new place to which a binding can move.

Sequents will have one *global* signature (the familiar Σ) and several *local* signatures.

$$\Sigma : \sigma_1 \triangleright B_1, \ldots, \sigma_n \triangleright B_n \vdash \sigma_0 \triangleright B_0$$

σ_i is a list of variables, locally scoped over the formula B_i.

The expression $\sigma_i \triangleright B_i$ is called a *generic judgment*.

The sequent calculus rules for ∇

The ∇-introduction rules modify the local contexts.

$$\frac{\Sigma : (\sigma, y_\gamma) \triangleright B[y/x], \Gamma \vdash \mathcal{C}}{\Sigma : \sigma \triangleright \nabla x_\gamma . B, \Gamma \vdash \mathcal{C}} \ \nabla\mathcal{L} \qquad \frac{\Sigma : \Gamma \vdash (\sigma, y_\gamma) \triangleright B[y/x]}{\Sigma : \Gamma \vdash \sigma \triangleright \nabla x_\gamma . B} \ \nabla\mathcal{R}$$

Since these rules are the same on the left and the right, this quantifier is *self-dual*.

$$\begin{array}{cc}
\nabla x \neg Bx \equiv \neg \nabla x Bx & \nabla x (Bx \wedge Cx) \equiv \nabla x Bx \wedge \nabla x Cx \\
\nabla x (Bx \vee Cx) \equiv \nabla x Bx \vee \nabla x Cx & \nabla x (Bx \Rightarrow Cx) \equiv \nabla x Bx \Rightarrow \nabla x Cx \\
\nabla x \forall y Bxy \equiv \forall h \nabla x Bx(hx) & \nabla x \exists y Bxy \equiv \exists h \nabla x Bx(hx) \\
\nabla x \forall y Bxy \Rightarrow \forall y \nabla x Bxy & \nabla x . \top \equiv \top, \quad \nabla x . \bot \equiv \bot
\end{array}$$

Implementing proof search in the presence of ∇ does not require new unification since ∇'s can be mini-scoped and since $\nabla x_1 \cdots \nabla x_n . t = s$ is equivalence to $\lambda x_1 \cdots \lambda x_n . t = \lambda x_1 \cdots \lambda x_n . s$.

Example: encoding π calculus

There are two syntactic categories *processes* and *names* and we use the primitive types p and n for these. The syntax is the following:

$$P := 0 \mid \tau.P \mid x(y).P \mid \bar{x}y.P \mid (P \mid P) \mid (P + P) \mid (x)P \mid [x = y]P$$

There are two binding constructors here.
The *restriction* operator $(x)P$ is encoded using a constant of type $(n \to p) \to p$.
The *input* operator $x(y).P$ is encoded using a constant of type $n \to (n \to p) \to p$.

Encoding π-calculus transitions

Processes can make transitions via various *actions*. There are three constructors for actions: $\tau : a$ for *silent* actions, $\downarrow : n \to n \to a$ for *input* actions, and $\uparrow : n \to n \to a$ for *output* actions.

$\downarrow xy : a$ denotes the action of inputting y on channel x
$\uparrow xy : a$ denotes the action of outputting y on channel x
$\uparrow x : n \to a$ denotes outputting of an abstracted name, and
$\downarrow x : n \to a$ denotes inputting of an abstracted variable.

One-step transitions are encoded as two different predicates:

$P \xrightarrow{A} Q$ free or silent action, $A : a$
$P \xrightarrow{\downarrow x} M$ bound input action, $\downarrow x : n \to a$, $M : n \to p$
$P \xrightarrow{\uparrow x} M$ bound output action, $\uparrow x : n \to a$, $M : n \to p$

π-calculus: operational semantics

Three example inference rules defining the semantics of π-calculus.

$$\frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \qquad \frac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'}\ y \notin \mathrm{n}(\alpha)$$

$$\begin{array}{lrcl}
\text{OUTPUT-ACT}: & \bar{x}y.P \xrightarrow{\bar{x}y} P & \triangleq & \top \\
\text{MATCH}: & [x=x]P \xrightarrow{\alpha} P' & \triangleq & P \xrightarrow{\alpha} P' \\
\text{RES}: & (x)Px \xrightarrow{\alpha} (x)P'x & \triangleq & \nabla x.(Px \xrightarrow{\alpha} P'x)
\end{array}$$

Consider the process $(y)[x=y]\bar{x}z.0$. It cannot make any transition since y has to be "new"; that is, it cannot be x.

The following statement is provable.

$$\forall x \forall Q \forall \alpha.[((y)[x=y](\bar{x}z.0) \xrightarrow{\alpha} Q) \supset \bot]$$

Encoding simulation in the (finite) π-calculus

Simulation for the (finite) π-calculus is defined simply as:

$$
\begin{array}{l}
\mathit{sim}\ P\ Q \triangleq \\
\qquad \forall A, P'\ [P \xrightarrow{A} P' \Rightarrow \exists Q'.Q \xrightarrow{A} Q' \wedge \mathit{sim}\ P'\ Q'] \wedge \\
\qquad \forall X, P'\ [P \xrightharpoonup{\downarrow X} P' \Rightarrow \exists Q'.Q \xrightharpoonup{\downarrow X} Q' \wedge \forall w.\mathit{sim}\ (P'w)\ (Q'w)] \wedge \\
\qquad \forall X, P'\ [P \xrightharpoonup{\uparrow X} P' \Rightarrow \exists Q'.Q \xrightharpoonup{\uparrow X} Q' \wedge \nabla w.\mathit{sim}\ (P'w)\ (Q'w)]
\end{array}
$$

Bisimulation is easy to encode (just add additional cases).

Bisimulation corresponds to *open* bisimulation. If the meta-logic is made classical, then *late* bisimulation is captured. The difference can be reduced to the excluded middle $\forall x \forall y.\ x = y \vee x \neq y$.
[Tiu & M, ToCL 2010]

The Abella theorem prover

Abella is an interactive theorem prover that is based on the pieces of logic described in this talk.

- elementary type theory (the impredicative and intuitionistic subset)
- least and greatest fixed points with inference rules for induction and coinduction.
- the ∇-quantifier
- an encoding of an object-logic (a subset of λProlog) with tactics related to its meta-theory

The proof theory of this logic (called $\mathcal{G}$) has been studied in [Gacek, M, & Nadathur, 2011].

Abella is written in OCaml and version 2.0.3 is available via OPAM and GitHub.

A tutorial appears in the J. of Formalized Reasoning 2014.

Conclusions

I have described an extension of ETT targeting metatheory and not mathematics.

The resulting logic provides for λ-tree syntax in a direct fashion, via binder-mobility, ∇-quantification, and the unification of λ-terms.

Induction over syntax containing bindings is available: in its richest setting, such induction is done over sequent calculus proofs of typing derivations.

Operational semantics and typing judgments are often encoded directly.

The Abella system has been used to successfully capture important aspects of the metatheory of the λ-calculus, π-calculus, programming languages, and object-logics.