The Open Calculus of Constructions and its Semantics

Mark-Oliver Stehr
University of Illinois at Urbana-Champaign

Overview

- Introduction
  - Syntax and Basic Semantics
  - Modified Semantics for Impredicative Instances
- Typing and Computation Rules
  - Equational Specifications
  - Dependent Types
  - Rewrite Specifications
- Further Examples
  - Higher-Order Abstract Syntax
  - Generalized Behavioral Specifications
- Conclusions

Introduction

The Open Calculus of Constructions (OCC) integrates ideas from:

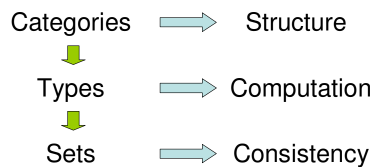
- Type Theory (Automath, Nuprl)
  - Dependent Types, Universes, Propositions as Types
- Calculus of Constructions, LF (Lego,Coq,Twelf)
  - Typechecking based on Computation
- Equational Logic (OBJ-3, Maude)
  - Executable Specs, Flexible Notion of Computation
- Rewriting Logic (Maude, ELAN)
  - Beyond Equational Computation

Pragmatic Approach

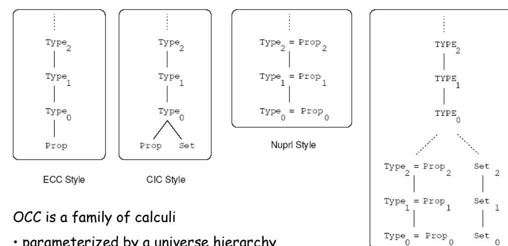
OCC =

Universes
+ Dependent Function Types
+ Conditional Rewriting Modulo
+ Goal-directed Search

Pragmatic Approach



Universe Hierarchies



OCC is a family of calculi

- parameterized by a universe hierarchy,
- with distingushed predicative and impredicative universes

Basic Term Syntax

Universes:	s
Function Application:	F M
Function Abstraction:	[X : S] M
Function Type:	S -> T
Dependent Function Type:	{ X : S } T
Type Assertion:	M : T
Equality:	M = N
Subtyping:	M <: N
Operational Propositions:	|| P, !! P, or ?? P

Basic Semantics of Terms

$$[\![s]\!] = U_s$$
$$[\![M\ N]\!] = [\![M]\!]([\![N]\!])$$
$$[\![[X : S]\ M]\!] = \lambda\ \alpha \in [\![S]\!] .\ [\![[X := \alpha]M]\!]$$
$$[\![\{X : S\}\ T]\!] = \Pi\ \alpha \in [\![S]\!] .\ [\![[X := \alpha]T]\!]$$
$$[\![M : S]\!] = [\![M]\!] \text{ if } [\![M]\!] \in [\![S]\!]$$
$$[\![M = N]\!] = \mathbb{T} \text{ if } [\![M]\!] = [\![N]\!]$$
$$[\![M = N]\!] = \mathbb{F} \text{ if } [\![M]\!] \neq [\![N]\!]$$
$$[\![\tau\ P]\!] = [\![P]\!] \quad \text{for } \tau \in \{\ ||, !!, ??\ \}$$
$$[\![\alpha]\!] = \alpha$$

We define $\mathbb{B} = \{\mathbb{F}, \mathbb{T}\}$ with $\mathbb{F} = \emptyset$ and $\mathbb{T} = \{\bullet\} = \{\emptyset\}$

Impredicativity

Assume Prop is an impredicative universe.

$A : \mathsf{Prop} \vdash [X : A]\ X : \{X : A\}\ A : \mathsf{Prop}$
$\vdash [A : \mathsf{Prop}][X : A]\ X : \{A : \mathsf{Prop}\}\{X : A\}\ A : \mathsf{Prop}$

It is well known that λ-calculus with impredicative polymorphism does not admit non-trivial, classical set-theoretic models [Reynolds 84].

Standard solution:
- full set-theoretic semantics for types in predicative universes and
- trivial set-theoretic semantics for types in impredicative universes (proof irrelevance)
  - $\Rightarrow$ semantics given by case analysis
  - $\Rightarrow$ nonuniform, dependency on the formal system

Question:
Is it possible to give a uniform interpretation to all terms such that $[\![\mathsf{Prop}]\!] = \mathbb{B} = \{\mathbb{F}, \mathbb{T}\}$?.
Answer: Yes !

Impredicativity

$A : \mathsf{Prop} \vdash [X : A]\ X : \{X : A\}\ A : \mathsf{Prop}$
$\vdash [A : \mathsf{Prop}][X : A]\ X : \{A : \mathsf{Prop}\}\{X : A\}\ A : \mathsf{Prop}$

$[\![\mathsf{Prop}]\!] = \mathbb{B} = \{\ \mathbb{F}, \mathbb{T}\ \} = \{\ \emptyset, \{\bullet\}\ \} = \{\emptyset, \{\emptyset\}\ \}$

$[\![[X : A]\ X]\!] = \emptyset$ for $[\![A]\!] = \mathbb{F}$
$[\![[X : A]\ X]\!] = \{(\bullet, \bullet)\}$ for $[\![A]\!] = \mathbb{T}$
$[\![[A : \mathsf{Prop}][X : A]\ X]\!] = \{\ (\mathbb{F}, \emptyset), (\mathbb{T}, \{(\bullet, \bullet)\})\ \}$

$[\![\{X : A\}\ A]\!] = \{\ \emptyset\ \}$ if $[\![A]\!] = \mathbb{F}$
$[\![\{X : A\}\ A]\!] = \{\ \{(\bullet, \bullet)\}\ \}$ if $[\![A]\!] = \mathbb{T}$
$[\![\{A : \mathsf{Prop}\}\{X : A\}\ A]\!] = \{\ \{\ (\mathbb{F}, \emptyset), (\mathbb{T}, \{(\bullet, \bullet)\})\ \}\ \}$

$\{X : A\}\ A : \mathsf{Prop}$ but $[\![\{X : A\}A]\!] \notin [\![\mathsf{Prop}]\!]$
$\{A : \mathsf{Prop}\}\{X : A\}\ A : \mathsf{Prop}$ but $[\![\{A : \mathsf{Prop}\}\{X : A\}\ A]\!] \notin [\![\mathsf{Prop}]\!]$

Modified Semantics

For any set-theoretic function β we define:
$\Downarrow\beta = \{\ (\alpha,\alpha') \in \beta \mid \alpha' \neq \emptyset\ \}$ (recall that $\bullet = \emptyset$)
$\beta(\alpha) = \emptyset$ for all $\alpha \notin \mathrm{Dom}(\beta)$

Obviously, we have:
$\Downarrow(\emptyset) = \emptyset$ and $\Downarrow(\bullet) = \bullet$
$\Downarrow(\beta) = \beta$ if $\beta(\alpha) \neq \emptyset$ for all $\alpha \in \mathrm{Dom}(\beta)$
$(\Downarrow\beta)(\alpha) = \beta(\alpha)$ for all α

Finally, $\Downarrow$ is extended to sets of functions as follows:
$\Downarrow\gamma = \{\ \Downarrow\beta \mid \beta \in \gamma\ \}$

Modified Semantics

$$[\![s]\!] = U_s$$
$$[\![M\ N]\!] = [\![M]\!]\ (\ [\![N]\!]\ )$$
$$[\![[X : S]\ M]\!] = \Downarrow \lambda\ \alpha \in [\![S]\!] .\ [\![[X := \alpha]M]\!]$$
$$[\![\{X : S\}\ T]\!] = \Downarrow \Pi\ \alpha \in [\![S]\!] .\ [\![[X := \alpha]T]\!]$$
$$[\![M : S]\!] = [\![M]\!] \text{ if } [\![M]\!] \in [\![S]\!]$$
$$[\![M = N]\!] = \mathbb{T} \text{ if } [\![M]\!] = [\![N]\!]$$
$$[\![M = N]\!] = \mathbb{F} \text{ if } [\![M]\!] \neq [\![N]\!]$$
$$[\![\tau\ P]\!] = [\![P]\!] \quad \text{for } \tau \in \{\ ||, !!, ??\ \}$$
$$[\![\alpha]\!] = \alpha$$

Impredicativity

A : Prop ⊢ [X : A] X : {X : A} A : Prop
⊢ [A : Prop][X : A] X : {A : Prop}{X : A} A : Prop

⟦ Prop ⟧ = 𝔹 = { 𝔽, 𝕋 } = {∅, {•} } = {∅, {∅} }

⟦ [X : A] X ⟧ = ⇓ ∅ = ∅ for ⟦ A ⟧ = 𝔽
⟦ [X : A] X ⟧ = ⇓ {(•, •)} = ∅ for ⟦ A ⟧ = 𝕋
⟦ [A : Prop][X : A] X ⟧ = ⇓ { (𝔽, ∅), (𝕋, ∅) } = ∅

⟦ {X : A} A ⟧ = ⇓ { ∅ } = { ∅ } if ⟦ A ⟧ = 𝔽
⟦ {X : A} A ⟧ = ⇓ { {(•, •)} } = { ∅ } if ⟦ A ⟧ = 𝕋
⟦ {A : Prop}{X : A} A ⟧ = ⇓ { { (𝔽, ∅), (𝕋, ∅) } } = { ∅ }

{X : A} A : Prop and ⟦ {X : A}A ⟧ ∈ ⟦ Prop ⟧
{A : Prop}{X : A} A : Prop and ⟦ {A : Prop}{X : A} A ⟧ ∈ ⟦ Prop ⟧

Propositions as ?

Propositions as Sets:
- ⟦ False ⟧ = some empty set
- ⟦ True ⟧ = some nonempty set
- ⟦ ∀ x ∈ S . T ⟧ = $\Pi_{x \in [\![S]\!]}$ ⟦ T ⟧

Propositions as Types:
- ⟦ False ⟧ = some empty type
- ⟦ True ⟧ = some nonempty type
- ⟦ ∀ x ∈ S . T ⟧ = { x : ⟦ S ⟧ } ⟦ T ⟧

Higher-Order Logic

```
True : Prop .
True_intro : True .

False : Prop .
False_elim : False -> ( {A : Prop} A ) .

And : Prop -> Prop -> Prop .
And_intro : {A,B : Prop} A -> B -> (And A B) .
And_elim : {A,B,C : Prop} (And A B) -> (A -> B -> C) -> C .

Or : Prop -> Prop -> Prop .
Or_intro_l : {A,B : Prop} A -> (Or A B) .
Or_intro_r : {A,B : Prop} B -> (Or A B) .
Or_elim : {A,B,C : Prop} (Or A B) -> (A -> C) -> (B -> C) -> C .
```

Higher-Order Logic

```
Not : Prop -> Prop .
Not_intro : {A : Prop}(A -> False) -> (Not A) .
Not_elim : {A : Prop}(Not A) -> A -> False .

All : {T : Type} (T -> Prop) -> Prop .
All_intro : {T : Type}{P : (T -> Prop)}
    ({x : T} (P x)) -> (All T P) .
All_elim : {T : Type}{P : (T -> Prop)}
    (All ? P) -> {x : T} (P x) .

Ex : {T : Type} (T -> Prop) -> Prop .
Ex_intro : {T : Type}{x : T}{P : (T -> Prop)}
    (P x) -> (Ex T P) .
Ex_elim : {T : Type}{P : (T -> Prop)}
    (Ex ? P) -> {A : Prop} ( { x : T } (P x) -> A ) -> A .
```

Basic Judgements

Typeinference:	M ->: T
Typechecking:	M : T
Structural Equality:	|| (M = N)
Assertional Proposition:	?? A
Assertional Equality:	?? (M = N)
Assertional Subtyping:	?? (S <: T)
Computational Equality:	!! (M = N)
Computational Relation:	!! (M => N)

Basic Judgements

Typeinference:	M ->: T
Typechecking:	M : T
Structural Equality:	|| M = N
Assertional Proposition:	?? A
Assertional Equality:	?? M = N
Assertional Subtyping:	?? S <: T
Computational Equality:	!! M = N
Computational Relation:	!! M => N

Semantics of Contexts

$[\![\,]\!] = \{()\}$

$[\![X : S, \Gamma]\!] = \Sigma\, \alpha \in [\![S]\!] \,.\, [\![[X := \alpha]\, \Gamma]\!]$

Note that: $(\alpha_1, \alpha_2, \ldots, \alpha_n) = (\alpha_1, (\alpha_2, (\ldots, (\alpha_n, ()))))$

Semantics of Judgements

$\models M : T$ iff $\downarrow [\![T]\!] \Rightarrow [\![M]\!] \in [\![T]\!]$

$\Gamma \models J$ iff $\forall\, \gamma \in [\![\Gamma]\!] \,.\, \models [X := \gamma]\, J$
for $\Gamma = X : S$

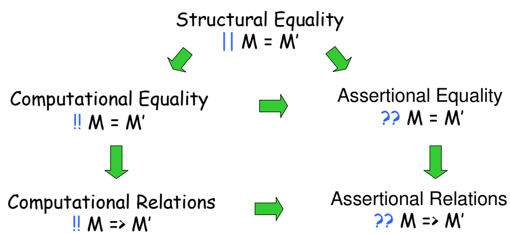
Typing Rules

Syntax directed rules for Γ ⊢ M ->: T plus

$$\frac{\Gamma \vdash M \text{ ->: } T \qquad \Gamma \vdash \text{!! } T = T'}{\Gamma \vdash M \text{ ->: } T'}$$

$$\frac{\Gamma \vdash M \text{ ->: } S \qquad \Gamma \vdash \text{?? } S \text{ <: } T}{\Gamma \vdash M : T}$$

Hierarchy of Judgements



Equational Specifications

Structural Equality

$$\frac{\text{|| } \{X : T\}\ M = M' \text{ in } \Gamma \qquad \Gamma \vdash N : T}{\Gamma \vdash \text{|| } [X := N]\, M = [X := N]\ M'}$$

$$\frac{}{\text{|| } M = M} \qquad \frac{\text{|| } M = N}{\text{|| } N = M} \qquad \frac{\text{|| } P = Q \qquad \text{|| } Q = R}{\text{|| } P = R}$$

$$\frac{\text{|| } M = M' \qquad \text{|| } N = N'}{\text{|| } M\, N = M'\, N'}$$

Structurally equal terms are identified

in all the following rules !

Equational Specifications

```
nat : Type .

0 : nat .
1 : nat .
plus : nat -> nat -> nat .

plus_comm : || {i,j : nat} (plus i j) = (plus j i) .

plus_assoc : || {i,j,k : nat} (plus i (plus j k)) = (plus (plus i j) k) .

plus_id : || {i : nat} (plus i 0) = i .

ver (plus (plus (plus 0 1)  1) 1) = (plus 1 (plus 1 1)) .
OK
```

Computational Equality

$$\frac{!!\ \{X : T\}\ C \to M = M' \text{ in } \Gamma \qquad \Gamma \vdash N : T \quad \Gamma \vdash ??\ [X := N]\ C}{\Gamma \vdash !!\ [X := N]\ M = [X := N]\ M'}$$

$$\frac{}{!!\ M = M} \qquad \frac{!!\ P = Q \quad !!\ Q = R}{!!\ P = R} \qquad \frac{!!\ M = M' \quad !!\ N = N'}{!!\ M\ N = M'\ N'}$$

$$\frac{M : T}{!!\ (M : T) = M} \qquad \frac{N : T}{!!\ ([X : T]\ M)\ N = [X := N]\ M}$$

All modulo structural equality !

But: No α-conversion !

Equational Specifications

```
nat : Type .

0 : nat .
suc : nat -> nat .

1 := (suc 0) .
2 := (suc 1) .
...

minus : nat -> nat -> nat .

minus_eq_1 : !! {i : nat} (minus i 0) = i)
minus_eq_2 : !! {i,j : nat} (minus (suc i) (suc j)) = (minus i j) .

red (minus 4 2) .
2
```

Assertional Equality

$$\frac{??\ \{X : T\}\ C \to M = M' \text{ in } \Gamma \qquad \Gamma \vdash N : T \quad \Gamma \vdash ??\ [X := N]\ C}{\Gamma \vdash ??\ [X := N]\ M = [X := N]\ M'}$$

$$\frac{M =_{\alpha} M'}{??\ M = M'} \qquad \frac{??\ M = N}{??\ N = M} \qquad \frac{??\ M = M' \quad ??\ N = N'}{??\ M\ N = M'\ N'}$$

$$\frac{!!\ M = M' \quad ??\ M' = N}{??\ M = N} \qquad \frac{!!\ N = N' \quad ??\ M = N'}{??\ M = N}$$

All modulo structural equality !

Assertional Propositions

$$\frac{??\ \{X : T\}\ C \to P \text{ in } \Gamma \qquad \Gamma \vdash N : T \quad \Gamma \vdash ??\ [X := N]\ C}{\Gamma \vdash ??\ [X := N]\ P}$$

$$\frac{\Gamma, X : T \vdash ??\ P}{\Gamma \vdash ??\ \{X : T\}\ P}$$

$$\frac{!!\ P = P' \quad ??\ P'}{??\ P}$$

All modulo structural equality !

Conditional Equational Specs

```
le : nat -> nat -> Prop .

le_1 : ?? {j : nat}(le 0 j) .
le_2 : ?? {i,j : nat}(le i j) -> (le (suc i) (suc j)) .

ver (le 2 3) .
verification succeeded

max : nat -> nat -> nat .

max_1 : !! {i,j : nat}(le i j) -> (max i j) = j .
max_2 : !! {i,j : nat}(le j i) -> (max i j) = i .

red (max 2 3) .
3
```

Equational Logic (OBJ-3, Maude)

sort Nat.	Nat? : Type . Nat : Nat? -> Prop .
sort NzNat . subsort NzNat < Nat.	NzNat : Nat? -> Prop . subsort_as : ?? {n : Nat?}(NzNat n) -> (Nat n) .
op 0 : -> Nat.	0 : Nat? . 0_as : ?? (Nat 0) .
op suc : Nat -> NzNat.	suc : Nat? -> Nat? . suc_as : ?? {n : Nat?}(Nat n) -> (NzNat (suc n)) .
op pred : NzNat -> Nat.	pred : Nat? -> Nat? . pred_as : ?? {n : Nat?}(NzNat n) -> (Nat (pred n)) .
var n : Nat . eq pred(suc(n)) = n.	pred_eq : !! {i : Nat?}(Nat i) -> (pred (suc i)) = i .

Higher-Order Equational Specs

```
list_nat : Type .

nil : list_nat .
cons : nat -> list_nat -> list_nat .

append : list_nat -> list_nat -> list_nat .

append_eq_1 : !! {l : list_nat}
    (append nil l) = l .

append_eq_2 : !! {l : list_nat}{h : nat}{t : list_nat}
    (append (cons h t) l) = (cons h (append t l)) .
```

Higher-Order Equational Specs

```
map : (nat -> nat) -> list_nat -> list_nat .

map_eq_1 : !! {f : (nat -> nat)}
    (map f nil) = nil .

map_eq_2 : !! {f : (nat -> nat)}{x : nat}{l : list_nat}
    (map f (cons x l)) = (cons (f x) (map f l)) .

red (map suc (cons 0 (cons 1 (cons 2 nil)))) .
(cons 1 (cons 2 (cons 3 nil)))
```

Higher-Order Equational Specs

```
select : (nat -> Prop) -> list_nat -> list_nat .

select_eq_1 : !! {P : (nat -> Prop)}
    (select P nil) = nil .

select_eq_2 : !! {P : (nat -> Prop)}{x : nat}{l : list_nat} (P x) ->
    (select P (cons x l)) = (cons x (select P l)) .

select_eq_3 : !! {P : (nat -> Prop)}{x : nat}{l : list_nat} (Not (P x)) ->
    (select P (cons x l)) = (select P l) .

red (select ([n : nat] (n = 1)) (cons 1 (cons 2 (cons 1 nil)))) .
(cons 1 (cons 1 nil))
```

Dependent Types

Type Operators

```
list : Type -> Type .

nil : {T : Type} (list T) .
cons : {T : Type} T -> (list T) -> (list T) .

map : {U,V : Type} (U -> V) -> (list U) -> (list V) .

map_eq_1 : !! {U,V : Type}{f : (U -> V)}
    (map U V f (nil U)) = (nil V) .

map_eq_2 : !! {U,V : Type}{n : nat}{f : (U -> V)}{x : U}{l : (list U n)}
    (map U V f (cons U x l)) =
    (cons V (f x) (map U V f l)) .
```

General Dependent Types

```
list : Type -> nat -> Type .

nil : {T : Type} (list T 0) .
cons : {T : Type}{n : nat} T -> (list T n) -> (list T (suc n)) .

map : {U,V : Type}{n : nat} (U -> V) -> (list U n) -> (list V n) .

map_eq_1 : !! {U,V : Type}{f : (U -> V)}
    (map U V 0 f (nil U)) = (nil V) .

map_eq_2 : !! {U,V : Type}{n : nat}{f : (U -> V)}{x : U}{l : (list U n)}
    (map U V (suc n) f (cons U n x l)) =
    (cons V n (f x) (map U V n f l)) .
```

Implicit Arguments

```
list : Type -> nat -> Type .

nil : {T | Type} (list T 0) .
cons : {T | Type}{n | nat} T -> (list T n) -> (list T (suc n)) .

map : {U,V | Type}{n | nat} (U -> V) -> (list U n) -> (list V n) .

map_eq_1 : !! {U,V : Type}{f : (U -> V)}
    (map f nil) = nil .

map_eq_2 : !! {U,V : Type}{n : nat}{f : (U -> V)}{x : U}{l : (list U n)}
    (map f (cons x l)) =
    (cons (f x) (map f l)) .
```

Polymorphic Multisets

```
fms : Type -> Type .

empty : {T | Type} (fms T) .
single : {T | Type} T -> (fms T) .
union : {T | Type} (fms T) -> (fms T) -> (fms T) .

union_comm : || {T : Type}{m1,m2 : (fms T)}
  (union m1 m2) = (union m2 m1) .

union_assoc : || {T : Type}{m1,m2,m3 : (fms T)}
  ((union m1 (union m2 m3)) = (union (union m1 m2) m3)) .

union_id : || {T : Type}{m : (fms T)}
  ((union m (empty | T)) = m) .
```

Polyary Functions

```
fun : nat -> Type -> Type .

fun_eq_1 : !! {T : Type} ((fun 0 T) = T) .
fun_eq_2 : !! {T : Type} {i : nat} ((fun (suc i) T) = (T -> (fun i T))) .

union* : {T | Type} {i : nat} (fun i (fms T)) .

union*_eq_1 : !! {T : Type} (union* 0) = (empty | T) .
union*_eq_2 : !! {T : Type}{s : (fms T)} (union* (suc 0) s) = s .
union*_eq_3 : !! {T : Type}{s,s' : (fms T)}
    (union* (suc 1) s s') = (union s s') .
union*_eq_4 : !! {T : Type}{i : nat}{s,s' : (fms T)}
    (union* (suc (suc (suc i))) s s') = (union* (suc (suc i)) (union s s')) .

red (union* 2 (union* 3 (single 1) (single 2) (single 3))
              (union* 2 (single 3) (single 4)))  .
(union (single 1) (single 2) (single 3) (single 3) (single 4))
```

Computational Relations

!! {X : T} C -> M => M' in Γ

Γ ⊢ N : T   Γ ⊢ ?? [X := N] C

Γ ⊢ !! [X := N] M => [X := N] M'

!! M = M'   !! M' => M''   !! M'' = M'''

!! M => M'''

!! M => M'   ?? M = M''

?? M => M''

All modulo structural equality !

Rewriting Logic (Maude)

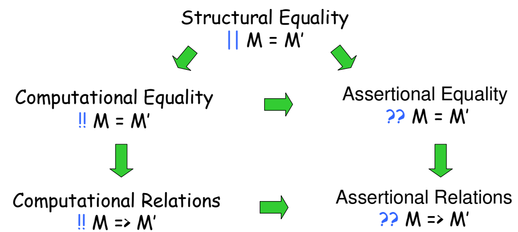
```
sort st .                              st : Type .
op loop : nat -> st .                  loop : nat -> st .
op stop : -> st .                      stop : st .

var i : nat .
crl [a1] : loop(i) => loop(minus(i,1))  a1 : !! {i : nat} (lt 0 i) ->
          if lt(0,i) .                        (loop i) => (loop (minus i 1))) .

rl [a2] : loop(0) => stop .            a2 : !! (loop 0) => stop .

rew loop(3) .                          rew (loop 3) .
stop                                   stop
```

Hierarchy of Judgements



Higher-Order Abstract Syntax

Higher-Order Abstract Syntax

```
term : Type .

const : term .
app : term -> term -> term .
abs : (term -> term) -> term .
```

Example:

(abs ([x : term] (app x x)))

represents

λ x . (x x)

Counting Bound Variables

```
cv : term -> nat .

cv-con-eq : !! (cv const) = 0 .

cv-app-eq : !! {M,N:term}
        (cv (app M N)) = (plus (cv M) (cv N)) .

cv-abs-eq : !! {B : (term -> term)}{n : nat}
        ({X : term} (!! (cv X) = 1) -> (n := (cv (B X)))) ->
        (cv (abs B)) = n .

red (cv (abs ([x : term] (app x x))))
2
```

Copying Terms with Binders

```
cp : term -> term .

cp-con-eq : !! (cp const) = const .

cp-app-eq : !! {M,N:term}
        (cp (app M N)) = (app (cp M) (cp N)) .

cp-abs-eq : !! {B : (term -> term)}{R : term -> term}
        ({X : term} (!! (cp X) = X) -> ((R X) := (cp (B X)))) ->
        (cp (abs B)) = (abs R) .

red (cp (abs ([y : term] (app const (abs ([x : term] x))))))
(abs ([X : term] ((app const) (abs ([X : term] X)))))
```

Warning

```
term : Type .

const : term .
app : term -> term -> term .
abs : (term -> term) -> term .
```

Domain of abs contains too many objects under the classical set-theoretic semantics.

⇒ Classical HOAS is a possible solution (not part of this talk)

Generalized Behavioral Specifications

Computational Equivalence

!! {X : T} C -> M === M' in Γ
Γ ⊢ N : T   Γ ⊢ ?? [X := N] C
——————————————
Γ ⊢ !! [X := N] M === [X := N] M'

!! P === Q   !! Q === R
——————————
!! P === R

——————
!! M = M

!! M = M'   !! M' === M''   !! M'' = M'''
——————————————
!! M === M'''

?? M = N
——————
?? M === N

?? M === N
——————
?? N === M

!! M === M'   ?? M' === N
——————————
?? M === N

!! N === N'   ?? M === N'
——————————
?? M === N

Hidden Algebra (BOBJ, BMaude)

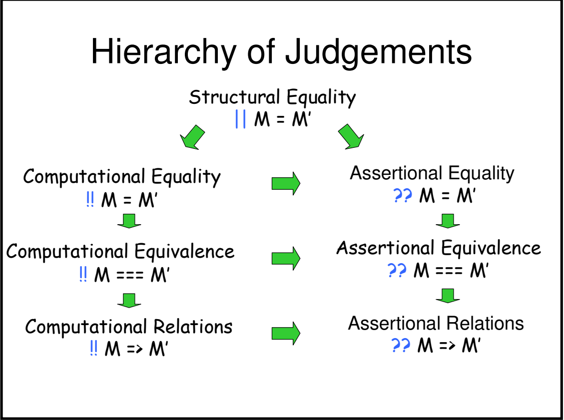
sort stack .	stack : Type .
bop top : stack -> nat .	top : stack -> nat . beq-1 : !! {s,s' : stack} (s === s') -> ((top s) = (top s')) .
bop pop : stack -> stack .	pop : stack -> stack . beq-2 : !! {s,s' : stack} (s === s') -> ((pop s) === (pop s')) .
op empty : -> stack op push : stack -> stack	empty : stack . push : stack -> stack .
var s : stack beq pop(empty) = empty . beq pop(push(s)) = s .	pop-eq-1 : !! (pop empty === empty) . pop-eq-2 : !! {s : stack} (pop (push s)) === s .

Hidden Algebra

bred (pop (push empty)) .
empty

bred (push (pop (push empty))) .
(push (pop (push empty)))

red (top (pop (push empty))) .
(top empty)



Algebras and Categories

Algebras

```
car : Type .
id : car .
op : car -> car -> car .
right_id : {x : car} (op x id) = x .
left_id : {x : car} (op id x) = x .
assoc : {x,y,z : car} (op (op x y) z) = (op x (op y z)) .

Monoid : Type .
monoid :
    {car : Type}
    {id : car}
    {op : car -> car -> car}
    {right_id : {x : car} (op x id) = x}
    {left_id : {x : car} (op id x) = x}
    {assoc : {x,y,z : car} (op (op x y) z) = (op x (op y z))}
    Monoid .
```

Algebras

```
.car : Monoid -> Type .

.id : {M : Monoid} (.car M) .
.op : {M : Monoid} (.car M) -> (.car M) -> (.car M) .

.right_id : {M : Monoid}
    {x : (.car M)} (.op M x (.id M)) = x .
.left_id : {M : Monoid}
    {x : (.car M)} (.op M (.id M) x) = x .
.assoc : {M : Monoid}
    {x,y,z : (.car M)} (.op M (.op M x y) z) = (.op M x (.op M y z)) .
```

Algebras

```
.car_eq : !! {car : Type}{id : car}{op : car -> car -> car}
    {right_id : {x : car} (op x id) = x}
    {left_id : {x : car} (op id x) = x}
    {assoc : {x,y,z : car} (op (op x y) z) = (op x (op y z))}
    (.car (monoid car id op right_id left_id assoc)) = car .

.id_eq : !! {car : Type}{id : car}{op : car -> car -> car}
    {right_id : {x : car} (op x id) = x}
    {left_id : {x : car} (op id x) = x}
    {assoc : {x,y,z : car} (op (op x y) z) = (op x (op y z))}
    (.id (monoid car id op right_id left_id assoc)) = id .
...
```

Algebras

```
nat_monoid =  (monoid
                 nat
                 0
                 plus
                 plus_id
                 plus_id
                 plus_assoc) : Monoid .
```

Categories

```
Obj : Type .
Mor : Obj -> Obj -> Type .

id : {A | Obj} (Mor A A) .
comp : {A,B,C | Obj}(Mor B C) -> (Mor A B) -> (Mor A C) .

left_id : {A,B : Obj}{f : (Mor A B)} (comp id f) = f .

right_id : {A,B : Obj}{f : (Mor A B)} (comp f id) = f .

assoc : {A,B,C,D : Obj} {f : (Mor A B)}{g : (Mor B C)}{h : (Mor C D)}
    (comp h (comp g f)) = (comp (comp h g) f) .
```

Categories

```
initial := [I : Obj] {X : Obj}
    (And (Ex ? ([h : (Mor I X)] True))
         ({f,g : (Mor I X)}(f = g))) : Obj -> Prop .

final := [F : Obj] {X : Obj}
    (And (Ex ? ([h : (Mor X F)] True))
         ({f,g : (Mor X F)}(f = g))) : Obj -> Prop .
```

Deriving Elimination Principles

Category of Nat Algebras

```
Nat : Type .

MkNat : {car : Type}
          {zero : car}
          {suc : car -> car}
          Nat .

NatMor : {nat1, nat2 : Nat} Type .

MkNatMor : {nat1, nat2 : Nat}{h : (.car nat1) -> (.car nat2)}
    {mor_eq_1 : (h (.zero nat1)) = (.zero nat2)}
    {mor_eq_2 : {n : (.car nat1)} (h ((.suc nat1) n)) = ((.suc nat2) (h n))}
    (NatMor nat1 nat2) .
```

Initiality Axiom

```
nat : Nat .

nat_initial_mor : {nat' : Nat} (NatMor nat nat') .

nat_initial_mor_unique : {nat' : Nat}{h1,h2 : (NatMor nat nat')} (h1 = h2) .
```

Derived Elimination Principle

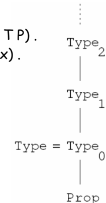
```
...
nat_elim := ... : {T : (.car nat) -> Type}
    {z : (T (.zero nat))}
    {s : {n : (.car nat)}(T n) -> (T (.suc nat n))}
    {n : (.car nat)} (T n) .

nat_elim_eq_1 := ? : !! {T : (.car nat) -> Type}
    {z : (T (.zero nat))}{f : {n : (.car nat)}(T n) -> (T (.suc nat n))}
    (nat_elim T z f (.zero nat)) = z .

nat_elim_eq_2 := ? : !! {T : (.car nat) -> Type}
    {z : (T (.zero nat))}{f : {n : (.car nat)}(T n) -> (T (.suc nat n))}
    {n : (.car nat)}
    (nat_elim T z f (.suc nat n)) = (f n (nat_elim T z f n)) .
```

Impredicative Universes needed ?

```
All : {T : Type} (T -> Prop) -> Prop .
All_intro : {T : Type}{P : (T -> Prop)} ({x : T} (P x)) -> (All T P) .
All_elim : {T : Type}{P : (T -> Prop)} (All T P) -> {x : T} (P x) .
```



Assume Prop is not impredicative.

Then {x : T} (P x) is in Type but not in Prop.

Fortunately, the embedded higher-order logic is
still impredicative, because (All T P) remains a type in Prop.

⇒ Better use (All T P) instead of {x : T} (P x) for logical purposes
⇒ Use interpretation $[\![\text{Prop}]\!] = \mathbb{B} = \{\mathbb{F}, \mathbb{T}\} = \{ \emptyset, \{\bullet\} \}$
⇒ Prop cannot be a (standard) predicative universe either !

Advantage of this approach: Straightforward set-theoretic semantics
(modification not needed)

Conclusions

- integration of functional programming, executable specifications, and interactive theorem proving
- hierarchy of computational concepts of increasing strength
- higher-order logic via propositions-as-types interpretation
- operational semantics is context-dependent
- uniform operational semantics for terms and types
- constraints, parameterization, reasoning principles, and proofs expressed internally, rather than outside of the logic
- computational openness: non-conservative theories typical case
- not a foundational system, three layers instead: sets, types, and categories
- prototypes available in Maude and Prolog, both based on a reflective architecture

New Prototype Features

- product and universe types
- dynamic typechecking
- implicit argument synthesis
- matching conditions, rewrite conditions, quantifiers in conditions
- labelled and unlabelled rewriting
- generalized behavioral specifications (experimental)
- context matching (experimental)
- various syntactic extensions
- support for Maude/OBJ-style syntax
- various switches and modes of tracing
- optimized Maude code generation for a fragment (experimental)

Bad News: Most of these features are only available in OCC/Prolog,
which is much less efficient than OCC/Maude

Website: formal.cs.uiuc.edu/stehr/occ.html