Curry, Howard, Coq

$\psi(\varepsilon_{\Omega+1})$

Two Kinds O' Proofs

- Informal Proof
  - Convincing natural language argument
- Formal Proof
  - Built from a strict set of rules
  - Syntactic manipulation
  - "Proof Theory" proofs
  - Machine checkable / manipulatable

A Proof Theory

- Axioms:

$$\overline{\mathbf{T}}$$

- Inference Rules:

$$\frac{\begin{matrix}[A]^u \\ \vdots \\ B\end{matrix}}{A \to B}\,u \qquad \mathbf{I}{\to}$$

$$\frac{\begin{matrix}\vdots \\ A \to B\end{matrix} \qquad \begin{matrix}\vdots \\ A\end{matrix}}{B} \ \mathbf{E}{\to}$$

- Build Derivations

An Example Proof

Want to prove:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

An Example Proof

Want to prove:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \to (B \to C))^u \quad A^w}{B \to C} \qquad \dfrac{(A \to B)^v \quad A^w}{B}}{C}}{A \to C}\,w}{(A \to B) \to (A \to C)}\,v}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\,u$$

An Example Proof

Want to prove:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \to (B \to C))^u \quad A^w}{B \to C}\,\mathbf{E}{\to} \qquad \dfrac{(A \to B)^v \quad A^w}{B}}{C}}{A \to C}\,w}{(A \to B) \to (A \to C)}\,v}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\,u$$

An Example Proof

Want to prove:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \to (B \to C))^u \qquad A^w}{B \to C} \qquad \dfrac{(A \to B)^v \qquad A^w}{B}\,\mathbf{E}{\to}}{C}}{A \to C}\,w}{(A \to B) \to (A \to C)}\,v}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\,u$$

An Example Proof

Want to prove:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \to (B \to C))^{u} \qquad A^{w}}{B \to C} \qquad \dfrac{(A \to B)^{v} \qquad A^{w}}{B}}{C}\ \mathbf{E}{\to}}{A \to C}\ w}{(A \to B) \to (A \to C)}\ v}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\ u$$

An Example Proof

Want to prove:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

$$\dfrac{(A \to (B \to C))^u \qquad A^w}{B \to C} \qquad \dfrac{(A \to B)^v \qquad A^w}{B}$$

$$\dfrac{B \to C \qquad B}{C}$$

$$\dfrac{C}{A \to C}\; {}^{w}\mathbf{I}{\to}$$

$$\dfrac{A \to C}{(A \to B) \to (A \to C)}\; v$$

$$\dfrac{(A \to B) \to (A \to C)}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\; u$$

An Example Proof

Want to prove:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \to (B \to C))^u \quad A^w}{B \to C} \qquad \dfrac{(A \to B)^v \quad A^w}{B}}{C}}{A \to C}\, w}{(A \to B) \to (A \to C)}\, v \ \ \mathbf{I\!\to}}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\, u$$

An Example Proof

Want to prove:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \to (B \to C))^u \quad A^w}{B \to C} \qquad \dfrac{(A \to B)^v \quad A^w}{B}}{C}}{A \to C}\,w}{(A \to B) \to (A \to C)}\,v}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\,u\ \mathbf{I}{\to}$$

Something Completely Different

- Lambda Calculus:
  - Core Functional Language
- Two typing rules:

Function Abstraction	Function Application
$$\frac{(x:A) \vdash y:B}{\vdash \lambda x:A.y:A \to B}$$	$$\frac{x:A \to B \quad y:A}{x\ y: B}$$

Something Completely Different ?

- Lambda Calculus:
  - Core Functional Language
- Two typing rules:

Function Abstraction	Function Application
$$\frac{(x:A) \vdash y:B}{\vdash \lambda x:A.y : A \rightarrow B}$$	$$\frac{x:A \rightarrow B \quad y:A}{x\ y : B}$$

Thursday, February 24, 2011

Something Completely Different ?

- Lambda Calculus:
  - Core Functional Language
- Two typing rules:

Function Abstraction	Function Application
$\dfrac{(x:A) \vdash y:B}{\vdash \lambda x:A.y:A \to B}$ **I→**	$\dfrac{x:A \to B \quad y:A}{x\ y: B}$ **E→**

An Example Redux

$$\dfrac{\dfrac{\dfrac{\dfrac{u:(A \to (B \to C)) \quad w:A}{uw : B \to C} \qquad \dfrac{v:(A \to B) \quad w:A}{vw : B}}{(uw)(vw):C}}{\lambda w.(uw)(vw):A \to C}}{\lambda vw.(uw)(vw):(A \to B) \to (A \to C)}$$

$$\lambda uvw.(uw)(vw):(A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

Two Coins

λ-Calculus

$$\frac{f{:}A\to B,\ y{:}A \vdash f\ y : B}{\text{Application takes A's to B's}}$$

Propositional Logic

$$\frac{\dfrac{A \quad A\to B}{B}}{\text{Modus Ponens derives B}}$$

Two Coins?

λ-Calculus

$$\frac{f{:}A \to B,\ y{:}A \vdash f\ y : B}{\text{Application takes A's to B's}}$$

Propositional Logic

$$\frac{\dfrac{A \quad A \to B}{B}}{\text{Modus Ponens derives B}}$$

Two Coins?

Two Sides !

$\psi(\varepsilon_{\Omega+1})$

Curry–Howard Isomorphism:

Any derivation in intuitionistic propositional logic corresponds to a typeable λ-term.

Two Sides !

$\psi(\varepsilon_{\Omega+1})$

Curry–Howard Isomorphism:

Any derivation in intuitionistic propositional logic corresponds to a typeable λ-term.

We can show a formula is derivable if we can build a term with the corresponding type!

Two Sides !

Two Sides !

λ-Calculus

Propositional Logic

Two Sides!

λ-Calculus	**Propositional Logic**
Type Variable	Propositional variable

Two Sides !

λ-Calculus	**Propositional Logic**
Type Variable	Propositional variable
Type	Formula

Two Sides !

λ-Calculus	**Propositional Logic**
Type Variable	Propositional variable
Type	Formula
Inhabitation	Proof

Two Sides!

λ-Calculus	**Propositional Logic**
Type Variable	Propositional variable
Type	Formula
Inhabitation	Proof
Type Constructor Left (x:A) (x:A, y:B)	Connective A∨B (A+B) A∧B (A×B)

L'Coq Proof Assistant

- Built on Calculus of (Co)-Inductive Constructions
  - Dependently-Type Lambda Calculus + Inductive Definitions
  - OCaml Implementation
    - Extraction to ML
- Goal: Build a term with the desired type
  - Small, trusted type checker
  - DeBruijn Criterion

L'Example

L'Example

- Goal : $\forall$(A:Type) (a b c : list A), a++(b++c) = (a++b)++c.

L'Example

- Goal : ∀(A:Type) (a b c : list A), a++(b++c) = (a++b)++c.

```
Definition app_assoc :=
list_ind
  (fun a0 : list A => forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c)
  (fun b c : list A => refl_equal (b ++ c))
  (fun (a0 : A) (a1 : list A)
     (IHa : forall b c : list A, a1 ++ b ++ c = (a1 ++ b) ++ c)
     (b c : list A) =>
   let H :=
     eq_ind_r (fun l : list A => a0 :: (a1 ++ b) ++ c = a0 :: l)
       (refl_equal (a0 :: (a1 ++ b) ++ c)) (IHa b c) in
   eq_ind_r (fun l : list A => a0 :: a1 ++ b ++ c = l)
     (eq_ind_r (fun l : list A => a0 :: l = a0 :: l)
        (refl_equal (a0 :: (a1 ++ b) ++ c)) (IHa b c)) H) a
```

L'Example

- Goal : ∀(A:Type) (a b c : list A), a++(b++c) = (a++b)++c.

```
Definition app_assoc :=
list_ind
  (fun a0 : list A => … c : li … + b ++ c = (a0 ++ b) ++ c)
  (fun b c : list A => … l (b …
  (fun (a0 : A) (a1 : li …
     (IHa : forall b c : l … + c = (a1 ++ b) ++ c)
     (b c : list A) =>
   let H :=
     eq_ind_r (fun l : li … ++ b) ++ c = a0 :: l)
       (refl_equal (a0 … b) … IHa b c) in
   eq_ind_r (fun l : l … a0 :: … ++ c = l)
     (eq_ind_r (fun … => a0 … :: l)
        (refl_equa… a1 ++ b) ++ … b c)) H) a
```

Agda **Epigram**

Tactics

- Recall: Want to build functions
  - Use program-generating functions called *tactics*
- Backward reasoning:

$A \wedge B$

- Combined into **Proof Scripts**
- 3 kinds Tactics:
  - Basic Inference Rules
  - Derived Rules
  - Decision Procedures

Tactics

- Recall: Want to build functions
  - Use program-generating functions called *tactics*
- Backward reasoning:

$$\frac{A}{A \wedge B}$$

- Combined into **Proof Scripts**
- 3 kinds Tactics:
  - Basic Inference Rules
  - Derived Rules
  - Decision Procedures

Tactics

- Recall: Want to build functions
  - Use program-generating functions called *tactics*
- Backward reasoning:

$$\frac{A \quad B}{A \land B}$$

- Combined into **Proof Scripts**
- 3 kinds Tactics:
  - Basic Inference Rules
  - Derived Rules
  - Decision Procedures

L'Example Redux

- Goal : ∀(A:Type) (a b c : list A), a++(b++c) = (a++b)++c.

```
Definition app_assoc :=
list_ind
  (fun a0 : list A => forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c)
  (fun b c : list A => refl_equal (b ++ c))
  (fun (a0 : A) (a1 : list A)
     (IHa : forall b c : list A, a1 ++ b ++ c = (a1 ++ b) ++ c)
     (b c : list A) =>
   let H :=
     eq_ind_r (fun l : list A => a0 :: (a1 ++ b) ++ c = a0 :: l)
       (refl_equal (a0 :: (a1 ++ b) ++ c)) (IHa b c) in
   eq_ind_r (fun l : list A => a0 :: a1 ++ b ++ c = l)
     (eq_ind_r (fun l : list A => a0 :: l = a0 :: l)
        (refl_equal (a0 :: (a1 ++ b) ++ c)) (IHa b c)) H) a
```

L'Example Redux

- Goal : ∀(A:Type) (a b c : list A), a++(b++c) = (a++b)++c.

```
Lemma app_assoc : forall A (a b c : list A), a ++ b ++ c = (a ++ b) ++ c.
  induction a; simpl; intros.
  reflexivity.
  cut (a :: (a0 ++ b) ++ c = a :: (a0 ++ b ++ c)).
  intros; rewrite H; rewrite IHa; reflexivity.
  rewrite IHa; reflexivity.
Qed.
```

L'Example Redux

- Goal : $\forall$(A:Type) (a b c : list A), a++(b++c) = (a++b)++c.

```
Lemma app_assoc : forall A (a b c : list A), a ++ b ++ c = (a ++ b) ++ c.
  induction a; simpl; intros.
  reflexivity.
  cut (a :: (a0 ++ b) ++ c = a :: (a0 ++ b ++ c)).
  intros; rewrite H; rewrite IHa; reflexivity.
  rewrite IHa; reflexivity.
Qed.
```

```
Lemma Double_Even : forall n, Even (n + n).
induction n; simpl; try rewrite plus_comm; simpl; constructor.
exact IHn.
Qed.
```

Proof Buffer

```
2 subgoals

  A : Type
  a : A
  a0 : list A
  IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
  b : list A
  c : list A
  ============================
   a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
   a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c

subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Proof Buffer

```
2 subgoals

A : Type
a : A
a0 : list A
IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
b : list A
c : list A
============================
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
 a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c

subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Context

Proof Buffer

```
2 subgoals

  A : Type
  a : A
  a0 : list A
  IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
  b : list A
  c : list A
  ============================
   a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
   a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c

subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Proof Buffer

```
2 subgoals

  A : Type
  a : A
  a0 : list A
  IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
  b : list A
  c : list A
  ============================
   a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
   a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c

subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Proof Buffer

```
2 subgoals

  A : Type
  a : A
  a0 : list A
  IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
  b : list A
  c : list A
  ============================
   a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
   a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c
```

Current Goal

```
subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Proof Buffer

```
2 subgoals

  A : Type
  a : A
  a0 : list A
  IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
  b : list A
  c : list A
  ============================
   a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
   a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c

subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Proof Buffer

```
2 subgoals

  A : Type
  a : A
  a0 : list A
  IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
  b : list A
  c : list A
  ============================
   a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
   a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c

subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Proof Buffer

```
2 subgoals

  A : Type
  a : A
  a0 : list A
  IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
  b : list A
  c : list A
  ============================
   a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
   a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c

subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Remaining Goals

Proof Buffer

```
2 subgoals

  A : Type
  a : A
  a0 : list A
  IHa : forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c
  b : list A
  c : list A
  ============================
   a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c ->
   a :: a0 ++ b ++ c = a :: (a0 ++ b) ++ c

subgoal 2 is:
 a :: (a0 ++ b) ++ c = a :: a0 ++ b ++ c
```

Modular Mechanized Metatheory

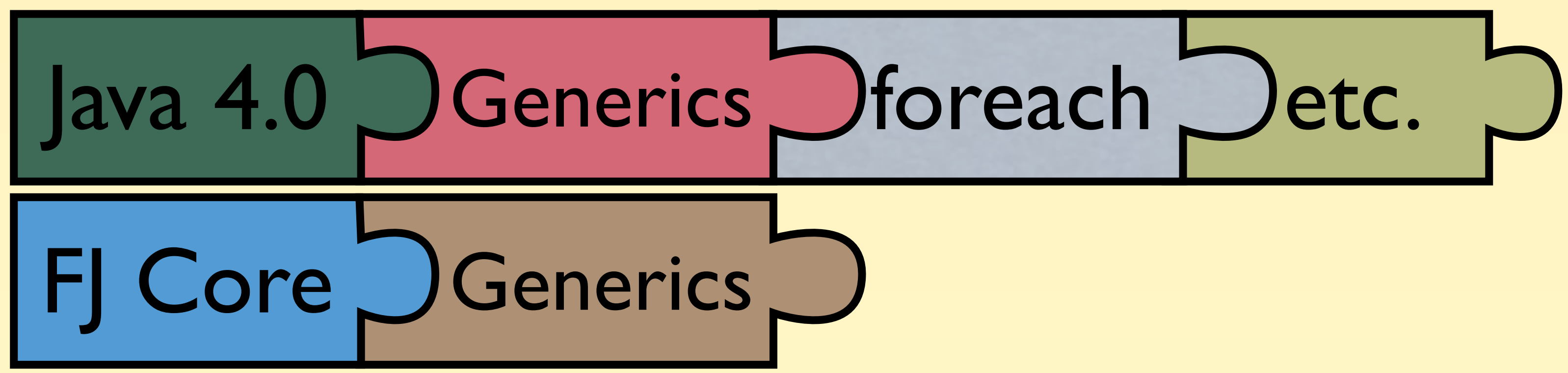
Ben Delaware

Today's Problem

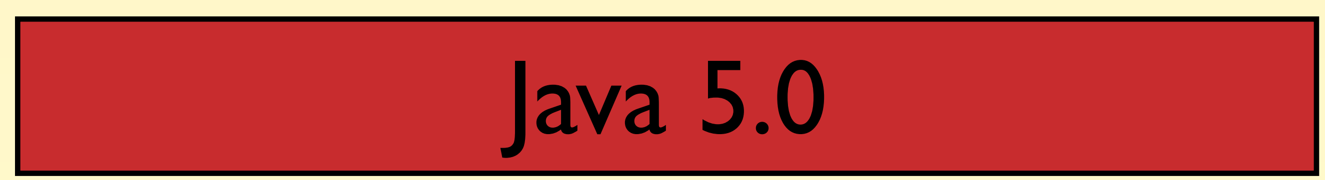
- Programming Languages change:
  - Java 5.0
  - GFJ
- New features added
- Standard practice: Java 5.0
- Our goal: Extensible Definitions

Today's Problem

- Programming Languages change:
  - Java 5.0 =
  - GFJ =



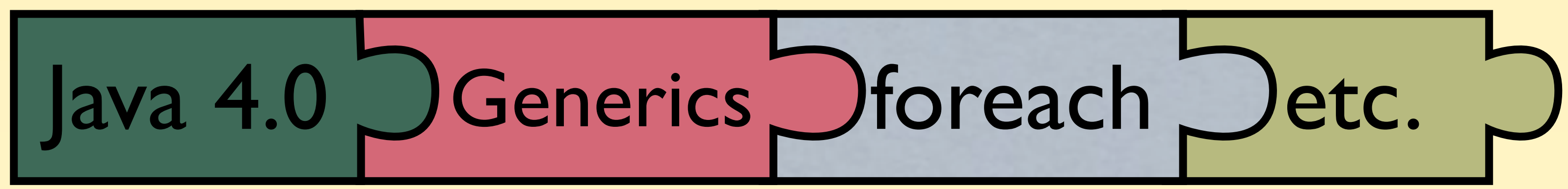
- New features added
- Standard practice:



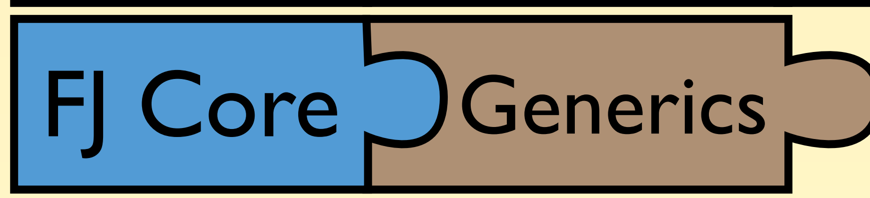
- Our goal: Extensible Definitions

Today's Problem

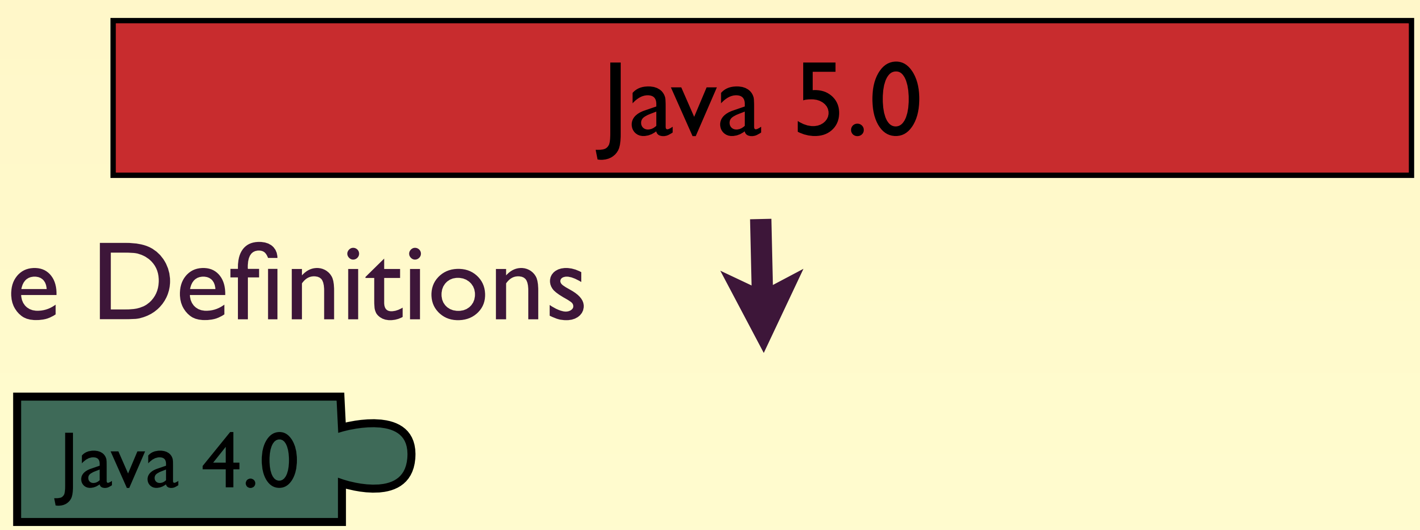
- Programming Languages change:
  - Java 5.0 =



  - GFJ =

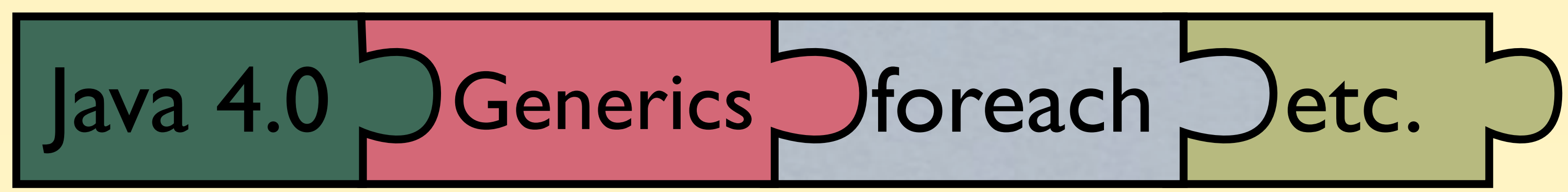


- New features added
- Standard practice:
- Our goal: Extensible Definitions

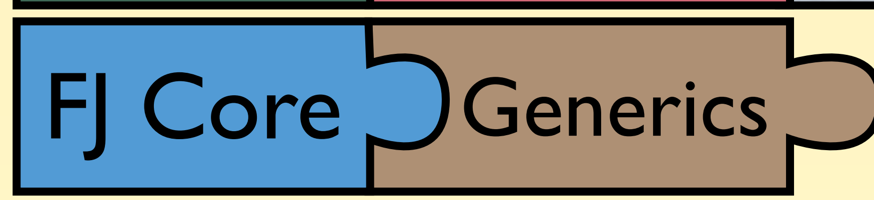


Today's Problem

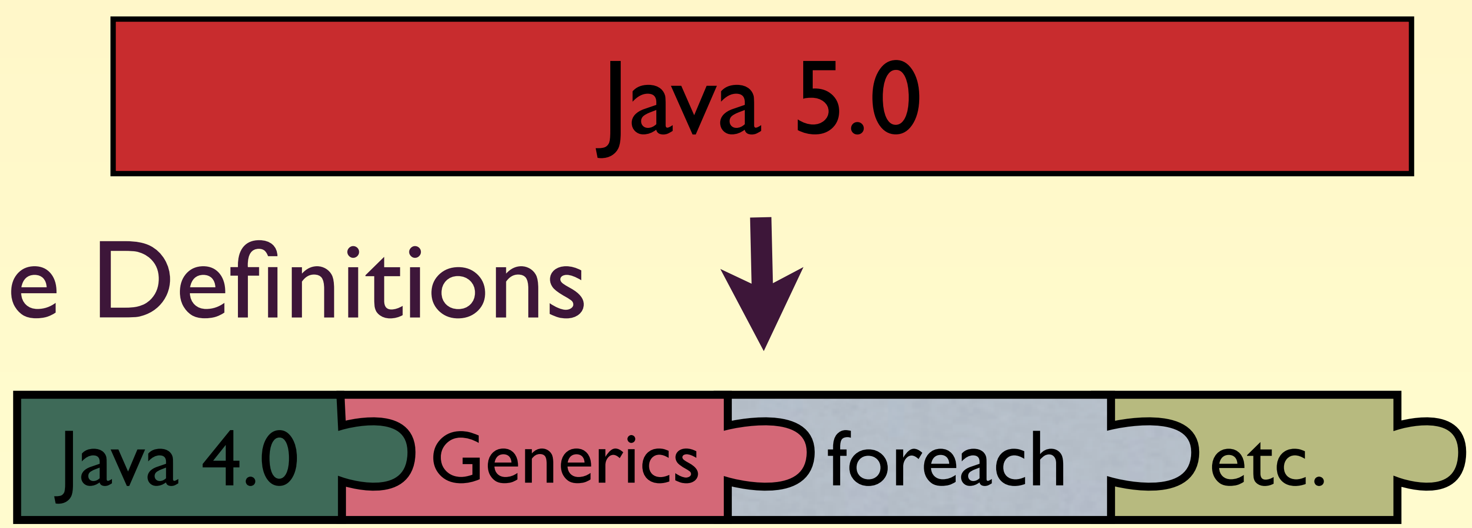
- Programming Languages change:
  - Java 5.0 =



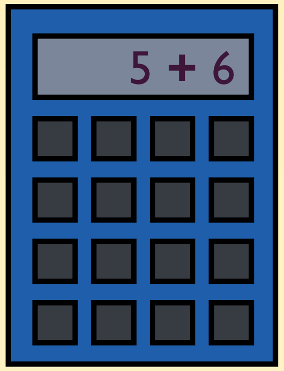
  - GFJ =



- New features added
- Standard practice:
- Our goal: Extensible Definitions



Defining a Language



- Arithmetic Expression Language (**AL**)
- Syntax:

$$E ::= E + E \mid \mathbb{N}$$

- Semantics:
  - Assign meaning to expressions
  - Interpreter:

eval (“5 + 6”) = 11

Operational Semantics

- Small-Step Operational Semantics
  - Set of transitions:
  - Presented as judgements

$$\frac{n_1 + n_2 = n_3 \qquad n_1, n_2 \in \mathbb{N}}{n_1 \mathbf{+} n_2 \rightsquigarrow n_3}$$

$$\frac{s_1 \rightsquigarrow s_1'}{s_1 \mathbf{+} s_2 \rightsquigarrow s_1' \mathbf{+} s_2} \qquad \frac{s_2 \rightsquigarrow s_2'}{s_1 \mathbf{+} s_2 \rightsquigarrow s_1 \mathbf{+} s_2'}$$

- Interpreters conform to these rules

Type Systems

- Approximation of run-time semantics
  - Typing Rules:

$$\frac{n \in \mathbb{N}}{\vdash n : \text{nat}} \qquad \frac{\vdash s_1 : \text{nat} \quad \vdash s_2 : \text{nat}}{\vdash s_1 + s_2 : \text{nat}}$$

  - Disallow 'misbehaving' programs

Type Safety Proofs

- Want to prove approximation is correct
- Two key lemmas for **AL:**

Progress

$$\frac{\vdash e : \text{nat}}{e \rightsquigarrow e' \lor e \in \mathbb{N}}$$

⋮

Proof

⋮

Qed.

Preservation

$$\frac{\vdash e : \text{nat} \qquad e \rightsquigarrow e'}{\vdash e' : \text{nat}}$$

⋮

Proof

⋮

Qed.

- These are our metatheory proofs

Complete Language

- Full language definition:

Syntax	Operational Semantics	Type System	Proofs

- Proof assistants manage complexity
    - Coq, ACL2, Isabelle/HOL
- Reuse today:

Complete Language

- Full language definition:

Syntax	Operational Semantics	Type System	Proofs

- Proof assistants manage complexity
  - Coq, ACL2, Isabelle/HOL
- Reuse today:

Syntax	Operational Semantics	Static Semantics	Proofs

Complete Language

- Full language definition:

Syntax	Operational Semantics	Type System	Proofs

- Proof assistants manage complexity
    - Coq, ACL2, Isabelle/HOL
- Reuse today:

Syntax	Operational Semantics	Static Semantics	Proofs
Syntax	Semantics	System	Proofs

Complete Language

- Full language definition:

Syntax	Operational Semantics	Type System	Proofs

- Proof assistants manage complexity
  - Coq, ACL2, Isabelle/HOL
- Reuse today:

Syntax	Operational Semantics	Static Semantics	Proofs

Complete Language

- Full language definition:

Syntax	Operational Semantics	Type System	Proofs

- Proof assistants manage complexity
  - Coq, ACL2, Isabelle/HOL
- Reuse today:

Syntax	Operational Semantics	Static Semantics	Proofs
Syntax Updates	Operational Semantics Updates	Static Semantics Updates	Proof Updates

Extending AL

- **BAL** = Booleans + **AL**
- New features make changes throughout

Syntax	Dynamic Semantics	Static Semantics	Proofs
E ::= | $\mathbb{N}$ | E + E	$\dfrac{s_1 \rightarrow s_1'}{s_1 + s_2 \rightarrow s_1' + s_2}$ $\dfrac{s_2 \rightarrow s_2'}{s_1 + s_2 \rightarrow s_1 + s_2'}$ $\dfrac{n_1 + n_2 = n_3 \quad n_1, n_2 \in \mathbb{N}}{n_1 + n_2 \rightarrow n_3}$	$\dfrac{n \in \mathbb{N}}{\vdash n : nat}$ $\dfrac{\vdash s_1 : nat \quad \vdash s_2 : nat}{\vdash s_1 + s_2 : nat}$	**Progress:** $\forall s : S, \vdash s : A \rightarrow \exists s', s \rightarrow s' \vee$ **Value** s. ⋮ Proof ⋮ **Qed**. **Preservation:** $\forall s\, s' : S, \vdash s : A \rightarrow s \rightarrow s' \rightarrow \vdash s' : A$ ⋮ Proof ⋮ **Qed**.

Building BAL

- **BAL** = Booleans + **AL**
- New features make changes throughout

Syntax	Dynamic Semantics	Static Semantics	Proofs
E ::= | $\mathbb{N}$ | E + E | $\mathbb{B}$ | **if** E **then** E **else** E | E = E	$\frac{s_1 \to s_1'}{s_1 + s_2 \to s_1' + s_2}$ $\frac{s_2 \to s_2'}{s_1 + s_2 \to s_1 + s_2'}$ $\frac{n_1 + n_2 = n_3 \quad n_1, n_2 \in \mathbb{N}}{n_1 + n_2 \to n_3}$ $\frac{}{\textbf{if } T \textbf{ then } s_2 \textbf{ else } s_3 \to s_2}$ $\frac{}{\textbf{if } F \textbf{ then } s_2 \textbf{ else } s_3 \to s_3}$ $\frac{s_1 \to s_1'}{\textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 \to \textbf{if } s_1' \textbf{ then } s_2 \textbf{ else } s_3}$ $\frac{}{T = T \to T}$ $\frac{}{F = F \to T}$ $\frac{}{T = T \to T}$ $\frac{}{F = F \to T}$ $\frac{n1 = n2}{n_1 = n_2 \to T}$ $\frac{n1 \neq n2}{n_1 = n_2 \to F}$	$\frac{n \in \mathbb{N}}{\vdash n : nat}$ $\frac{\vdash s_1 : nat \quad \vdash s_2 : nat}{\vdash s_1 + s_2 : nat}$ $\frac{b \in \mathbb{B}}{\vdash b : bool}$ $\frac{\vdash s_1 : bool \quad \vdash s_2 : A \quad \vdash s_3 : A}{\vdash \textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 : A}$ $\frac{\vdash s_1 : A \quad \vdash s_2 : A}{\vdash s1 = s2 : bool}$	**Progress:** $\forall$ s : S, $\vdash$ s : A $\to$ $\exists$ s', s $\to$ s' $\lor$ **Value** s. ⋮ Updated Proof ⋮ **Qed**. **Preservation:** $\forall$ s s': S, $\vdash$ s : A $\to$ s $\to$ s' $\to$ $\vdash$ s' : A ⋮ Updated Proof ⋮ **Qed**.

Language Modules

- Feature = Module with updates

Syntax	Dynamic Semantics	Static Semantics	Proofs
$E' ::=$ $\mid E$ $\mid \mathbb{B}$ $\mid$ **if** E' **then** E' **else** E' $\mid E' = E'$	$\dfrac{s \to s'}{s \to \square s'}$ $\dfrac{}{\textbf{if } T \textbf{ then } s_2 \textbf{ else } s_3 \to s_2}$ $\dfrac{}{\textbf{if } F \textbf{ then } s_2 \textbf{ else } s_3 \to s_3}$ $\dfrac{s_1 \to s_1'}{\textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 \to \textbf{if } s_1' \textbf{ then } s_2 \textbf{ else } s_3}$ $\dfrac{}{T = T \to T}$ $\dfrac{}{F = F \to T}$ $\dfrac{}{T = T \to T}$ $\dfrac{}{F = F \to T}$ $\dfrac{n1 = n2}{n_1 = n_2 \to T}$ $\dfrac{n1 \neq n2}{n_1 = n_2 \to F}$	$\dfrac{\vdash s : A}{\vdash' s : A}$ $\dfrac{b \in \mathbb{B}}{\vdash b : bool}$ $\dfrac{\vdash s_1 : bool \quad \vdash s_2 : A \quad \vdash s_3 : A}{\vdash \textbf{if } s_1 \textbf{ then } s_2 \textbf{ else } s_3 : A}$ $\dfrac{\vdash s_1 : A \quad \vdash s_2 : A}{\vdash s1 = s2 : bool}$	**Progress:** Proof Updates **Preservation:** Proof Updates

- Language = Composition of Modules

BAL = **Boolean** • **AL**

Extensible Syntax

Boolean Syntax

$E' ::=$
$\mid E$
$\mid \mathbb{B}$
$\mid \textbf{if}\ E'\ \textbf{then}\ E'\ \textbf{else}\ E'$
$\mid E' \mathbin{\textbf{=}} E'$

•

AL Syntax

$E ::=$
$\mid \mathbb{N}$
$\mid E \mathbin{\textbf{+}} E$

=

BAL Syntax

$E ::=$
$\mid \mathbb{N}$
$\mid E \mathbin{\textbf{+}} E$

$E' ::=$
$\mid E$
$\mid \mathbb{B}$
$\mid \textbf{if}\ E'\ \textbf{then}\ E'\ \textbf{else}\ E'$
$\mid E' \mathbin{\textbf{=}} E'$

Extensible Syntax

- First stab: "Wrap" Syntax

Boolean Syntax		AL Syntax		BAL Syntax
$E' ::=$ $\mid E$ $\mid \mathbb{B}$ $\mid$ **if** E' **then** E' **else** E' $\mid E' = E'$	•	$E ::=$ $\mid \mathbb{N}$ $\mid E + E$	=	$E ::=$ $\mid \mathbb{N}$ $\mid E + E$ $E' ::=$ $\mid E$ $\mid \mathbb{B}$ $\mid$ **if** E' **then** E' **else** E' $\mid E' = E'$

Extensible Syntax

- First stab: "Wrap" Syntax

Boolean Syntax		AL Syntax		BAL Syntax ?
$E' ::=$ $\mid E$ $\mid \mathbb{B}$ $\mid \textbf{if } E' \textbf{ then } E' \textbf{ else } E'$ $\mid E' = E'$	$\bullet$	$E ::=$ $\mid \mathbb{N}$ $\mid E + E$	$=$	$E ::=$ $\mid \mathbb{N}$ $\mid E + E$ $E' ::=$ $\mid E$ $\mid \mathbb{B}$ $\mid \textbf{if } E' \textbf{ then } E' \textbf{ else } E'$ $\mid E' = E'$

Extensible Syntax

- First stab: "Wrap" Syntax

Boolean Syntax		AL Syntax		BAL Syntax ?
$E' ::=$ $\mid E$ $\mid \mathbb{B}$ $\mid \textbf{if } E' \textbf{ then } E' \textbf{ else } E'$ $\mid E' = E'$	•	$E ::=$ $\mid \mathbb{N}$ $\mid E + E$	=	$E ::=$ $\mid \mathbb{N}$ $\mid E + E$ $E' ::=$ $\mid E$ $\mid \mathbb{B}$ $\mid \textbf{if } E' \textbf{ then } E' \textbf{ else } E'$ $\mid E' = E'$

- Need inductive updates!
  - Can't build (**if** T **then** 2 **else** 3) **+** 4

Extensible Syntax

- First stab: "Wrap" Syntax

Boolean Syntax		AL Syntax		BAL Syntax?
$E' ::= \mid E \mid \mathbb{B} \mid \textbf{if } E' \textbf{ then } E' \textbf{ else } E' \mid E' = E'$	•	$E ::= \mid \mathbb{N} \mid E + E$	=	$E ::= \mid \mathbb{N} \mid E + E$ $E' ::= \mid E \mid \mathbb{B} \mid \textbf{if } E' \textbf{ then } E' \textbf{ else } E' \mid E' = E'$

- Need inductive updates!
  - Can't build (**if** T **then** 2 **else** 3) + 4
- Nonterminals reference Final Language

Extensible Syntax

- Solution: Leave definitions open:

Boolean Syntax

$E_B(\mathbf{S})$ ::=
| $\mathbb{B}$
| **if S then S else S**
| **S = S**

AL Syntax

$E_A(\mathbf{S})$ ::=
| $\mathbb{N}$
| **S + S**

- Final language closes the induction:

Final Syntax

$\mathbf{S} ::= E_A(\mathbf{S}) \mid E_B(\mathbf{S})$

Extensible Judgements

- Operational Semantics: Abstract transitions $\rightsquigarrow$

$$\frac{s_1 \rightsquigarrow s_1'}{s_1 + s_2 \rightsquigarrow_A s_1' + s_2} \qquad \frac{s_2 \rightsquigarrow s_2'}{s_1 + s_2 \rightsquigarrow_A s_1 + s_2'}$$

- Final judgement closes the induction:

$$\frac{s \rightsquigarrow_A s'}{s \rightsquigarrow s'} \qquad \frac{s \rightsquigarrow_B s'}{s \rightsquigarrow s'}$$

- Typing Rules : Abstract typing judgement $\vdash$

$$\frac{n \in \mathbb{N}}{\vdash_A n : nat} \qquad \frac{\vdash s_1 : nat \quad \vdash s_2 : nat}{\vdash_A s_1 + s_2 : nat}$$

"Open" Proofs

- Extensible definitions need Extensible Proofs
  - Proofs over final language
- Module has proofs for its definitions
  - Subterms from abstract language
  - Progress uses **ReduceEqual**

ReduceEqual

$$\frac{s_1 = s_2 \qquad \textbf{Value}\ s_1 \qquad \textbf{Value}\ s_2}{s_1 = s_2 \rightsquigarrow s_3}$$

Boolean Progress

Progress$_\mathbf{B}$ ($\mathbf{S}$, $\rightsquigarrow$, $\vdash$):

$$\frac{\vdash e : A}{e \rightsquigarrow e' \lor \textbf{Value}\ e}$$

Induction on s.

Case $\mathbb{B}$:

⋮

Case **if** s_1 **then** s_2 **else** s_3:

⋮

Case $s_1 = s_2$:

⋮

Use **ReduceEqual**

⋮

Qed.

Externalizing Assumptions

- Properties of S become assumptions:

Boolean Progress

Progress$_B$ (**S**, $\rightarrow$, $\vdash$) **:**

$$\frac{\vdash e : A \qquad \textbf{ReduceEqual}}{e \rightarrow e' \vee \textbf{Value}\ e}$$

⋮

Proof using **ReduceEqual**

⋮

Qed.

- To use proof of Boolean **Progress**,
  - Build proof of **ReduceEqual** as separate Lemma
  - Pass to Boolean **Progress**

Modular Inductive Proofs

- **Progress** can't be externalized

Boolean Progress

Progress$_B$ (**S**, ⇝ ,⊢) **:**

$$\frac{\vdash e : A \qquad \textbf{ReduceEqual} \qquad \textbf{Progress}}{e \rightsquigarrow e' \lor \textbf{Value}\ e}$$

⋮

Qed.

- Inductive Hypothesis fills the hole
  - Only use on subterms

Building Inductive Proofs

Building Inductive Proofs

- To build the final inductive proof,

Building Inductive Proofs

- To build the final inductive proof,
  1. Build external lemmas

ReduceEqual

Case $s \in E_A(\mathbf{S})$:
  Proof of $\mathbf{ReduceEqual_A}$ $(\mathbf{S}, \rightarrow)$
Case $s \in E_B(\mathbf{S})$:
  Proof of $\mathbf{ReduceEqual_B}$ $(\mathbf{S}, \rightarrow)$

Building Inductive Proofs

- To build the final inductive proof,
  1. Build external lemmas
  2. Proceed by induction

ReduceEqual

Case s ∈ E_A(**S**) :
  Proof of **ReduceEqual$_A$** (**S**,→)
Case s ∈ E_B(**S**) :
  Proof of **ReduceEqual$_B$** (**S**, →)

Boolean Progress

Progress : ∀ s : E, ⊢ s : E → ∃ s′, s → s′ ∨ Value s.

Induction on s
 Case s ∈ E_A(**S**) :
  Proof of **Progress$_A$**(**S**, **ReduceEqual**, **Progress**)
 Case s ∈ E_B(**S**) :
  Proof of **Progress$_B$**(**S**, **ReduceEqual**, **Progress**)

Qed.

Building Inductive Proofs

- To build the final inductive proof,
  1. Build external lemmas
  2. Proceed by induction
  3. Pass IH to "close" the loop

ReduceEqual

Case $s \in E_A(\mathbf{S})$:
  Proof of $\mathbf{ReduceEqual_A}$ $(\mathbf{S}, \rightarrow)$
Case $s \in E_B(\mathbf{S})$:
  Proof of $\mathbf{ReduceEqual_B}$ $(\mathbf{S}, \rightarrow)$

Boolean Progress

$\mathbf{Progress : \forall\, s : E, \vdash s : E \rightarrow \exists\, s', s \rightarrow s' \vee Value\ s.}$

Induction on s
 Case $s \in E_A(\mathbf{S})$:
  Proof of $\mathbf{Progress_A}(\mathbf{S}, \mathbf{ReduceEqual}, \mathbf{Progress})$
 Case $s \in E_B(\mathbf{S})$:
  Proof of $\mathbf{Progress_B}(\mathbf{S}, \mathbf{ReduceEqual}, \mathbf{Progress})$

Qed.

Building Inductive Proofs

- To build the final inductive proof,
  1. Build external lemmas
  2. Proceed by induction
  3. Pass IH to "close" the loop

ReduceEqual

Case s ∈ $E_A(\mathbf{S})$:
 Proof of $\mathbf{ReduceEqual_A}$ $(\mathbf{S}, \rightsquigarrow)$
Case s ∈ $E_B(\mathbf{S})$:
 Proof of $\mathbf{ReduceEqual_B}$ $(\mathbf{S}, \rightsquigarrow)$

Boolean Progress

Progress : ∀ s : E, ⊢ s : E → ∃ s′, s ⇝ s′ ∨ Value s.

Induction on s
 Case s ∈ $E_A(\mathbf{S})$:
  Proof of $\mathbf{Progress_A}$(**S**, **ReduceEqual**, **Progress**)
 Case s ∈ $E_B(\mathbf{S})$:
  Proof of $\mathbf{Progress_B}$(**S**, **ReduceEqual**, **Progress**)
Qed.

- Coq checks proper IH use

Language Variations

- Can build 3 languages:

AL | Boolean

- Features can interact:

$$\frac{n_1 = n_2 \quad n_1, n_2 \in \mathbb{N}}{n_1 \mathbf{=} n_2 \rightsquigarrow \mathbf{T}}$$

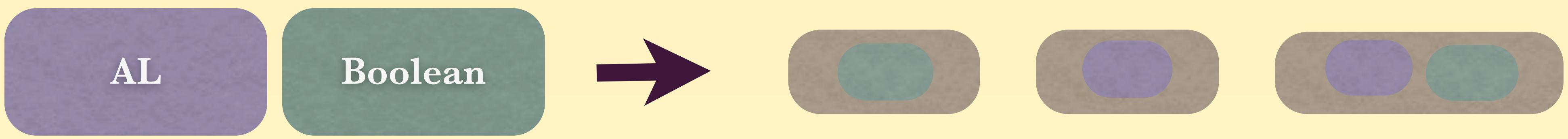
$$\frac{n_1 \neq n_2 \quad n_1, n_2 \in \mathbb{N}}{n_1 \mathbf{=} n_2 \rightsquigarrow \mathbf{F}}$$

- Interactions are also features:

AL | Boolean | BAL Interactions

Language Variations

- Can build 3 languages:



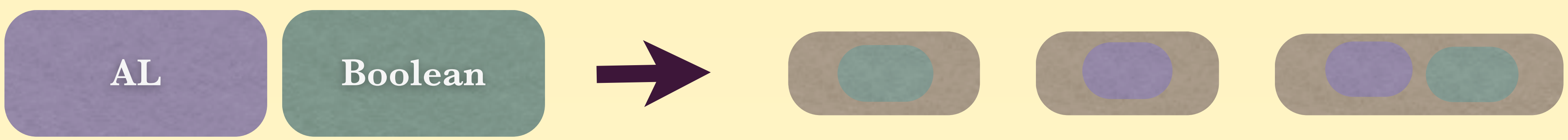
- Features can interact:

$$\frac{n_1 = n_2 \quad n_1, n_2 \in \mathbb{N}}{n_1 \mathbf{=} n_2 \rightsquigarrow \mathbf{T}} \qquad \frac{n_1 \neq n_2 \quad n_1, n_2 \in \mathbb{N}}{n_1 \mathbf{=} n_2 \rightsquigarrow \mathbf{F}}$$

- Interactions are also features:

Language Variations

- Can build 3 languages:



- Features can interact:

$$\frac{n_1 = n_2 \quad n_1, n_2 \in \mathbb{N}}{n_1 = n_2 \rightsquigarrow \mathbf{T}} \qquad \frac{n_1 \neq n_2 \quad n_1, n_2 \in \mathbb{N}}{n_1 = n_2 \rightsquigarrow \mathbf{F}}$$

- Interactions are also features:

More Updates

FJ Expression Syntax		FJ • Generic Expression Syntax
$\texttt{e} ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}\ (\bar{\texttt{e}}) \mid \texttt{new C}(\bar{\texttt{e}}) \mid (\texttt{C})\ \texttt{e}$	$\Longrightarrow$	$\texttt{e} ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}\langle\bar{\texttt{T}}\rangle^{\beta}\ (\bar{\texttt{e}}) \mid \texttt{new C}\ \langle\bar{\texttt{T}}\rangle^{\beta}\ (\bar{\texttt{e}}) \mid (\texttt{C}\ \langle\bar{\texttt{T}}\rangle^{\beta})\ \texttt{e}$
FJ Subtyping $\quad \texttt{T} <: \texttt{T}$		**GFJ Subtyping** $\quad \Delta^{\delta} \vdash \texttt{T} <: \texttt{T}$
		$\Delta \vdash \texttt{X} <: \Delta(\texttt{X})$ (GS-VAR)$^{\alpha}$
$\dfrac{\texttt{S}<:\texttt{T} \qquad \texttt{T}<:\texttt{V}}{\texttt{S}<:\texttt{V}}$ (S-TRANS)	$\Longrightarrow$	$\dfrac{\Delta^{\delta} \vdash \texttt{S}<:\texttt{T} \qquad \Delta^{\delta} \vdash \texttt{T}<:\texttt{V}}{\Delta^{\delta} \vdash \texttt{S}<:\texttt{V}}$ (GS-TRANS)
$\texttt{T}<:\texttt{T}$ (S-REFL)		$\Delta^{\delta} \vdash \texttt{T}<:\texttt{T}$ (GS-REFL)
$\dfrac{\texttt{class C extends D}\ \{\ldots\}}{\texttt{C}<:\texttt{D}}$ (S-DIR)		$\dfrac{\texttt{class C}\ \langle\overline{\texttt{X} \triangleright \texttt{N}}\rangle^{\beta}\ \texttt{extends D}\ \langle\bar{\texttt{V}}\rangle^{\beta}\ \{\ldots\}}{\Delta^{\delta} \vdash \texttt{C}\ \langle\bar{\texttt{T}}\rangle^{\beta} <: [\bar{\texttt{T}}/\bar{\texttt{X}}]^{\eta}\texttt{D}\ \langle\bar{\texttt{V}}\rangle^{\beta}}$ (GS-DIR)
FJ New Typing $\quad \Gamma \vdash \texttt{e} : \texttt{T}$		**GFJ New Typing** $\quad \Delta;^{\delta} \Gamma \vdash \texttt{e} : \texttt{T}$
$\dfrac{\texttt{fields}(\texttt{C}) = \bar{\texttt{D}}\ \bar{\texttt{f}} \qquad \Gamma \vdash \bar{\texttt{e}} : \bar{\texttt{C}} \qquad \bar{\texttt{C}}<:\bar{\texttt{D}}}{\Gamma \vdash \texttt{new C}(\bar{\texttt{e}}) : \texttt{C}}$ (T-NEW)	$\Longrightarrow$	$\dfrac{\Delta \vdash \texttt{C}\langle\bar{\texttt{T}}\rangle^{\gamma} \quad \texttt{fields}(\texttt{C}\ \langle\bar{\texttt{T}}\rangle^{\beta}) = \bar{\texttt{V}}\ \bar{\texttt{f}} \quad \Delta;^{\delta}\ \Gamma \vdash \bar{\texttt{e}} : \bar{\texttt{U}} \quad \Delta^{\delta} \vdash \bar{\texttt{U}}<:\bar{\texttt{V}}}{\Delta;^{\delta}\ \Gamma \vdash \texttt{new C}\ \langle\bar{\texttt{T}}\rangle^{\beta}\ (\bar{\texttt{e}}) : \texttt{C}}$ (GT-NEW)

Contributions

- Developed technique for extensible language design
  - Update syntax, semantics, and proofs
    - Add new definitions
    - Update existing definitions
    - Reuse existing Proofs
- Modules independently mechanically verifiable
- ECOOP paper under construction
  - Builds GFJ + Interfaces w/ our techniques

Questions?

Related Work

- R. Stärk, J. Schmid, and E. Börger. Java and the java virtual machine - definition, verification, validation.
- P. D. Mosses. Modular structural operational semantics.
- A. Chlipala. A verified compiler for an impure functional language.