Formal Objects in Type Theory

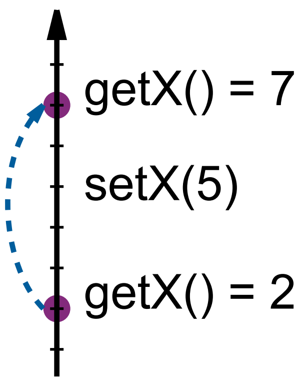
Jason Hickey

Outline

- **Object Oriented Programming**
- **Object Calculi**
- **Examples**
- **Interpretation in Nuprl**

Object Oriented Programming

Example class




```
class Point {
private:
    int x;
public:
    Point(int i) { x = i; }
    int getX() { return x; }
    Point setX(int i) {
        return new Point(i);
    }
};
```

Field

Constructor

Methods

Inheritance

What type is setX()?

```
class ColorPoint : inherits Point {
private:
    Color color;
public:
    ColorPoint(int i, Color c) : Point(i) { color = c; }
    int getColor() { return color; }
    ColorPoint setColor(Color c) {
        return new ColorPoint(getX(), c);
    }
    // int getX() inherited from Point()
    ColorPoint setX(int i) {
        return (ColorPoint) Point::setX(i);
    }
};
```

Object calculus (Cardelli, Abadi)

- **Three primitives: objects, method selection, method override**

For an object $o = [l_i = \lambda x_i.b_i]^{\,i \in \{1 \ldots n\}}$:

$$
\begin{array}{rcl}
o.l_j & \rightarrow & b_j[o/x_j] \\
o.l_j \Leftarrow \lambda x.b & \rightarrow & [l_j = \lambda x.b, l_i = \lambda x_i.b_i^{\,i \in \{1 \ldots n\} - \{j\}}]
\end{array}
$$

- **Programs are untyped**

$$Obj(X.[l_i{:}T_i[X]]^{\,i \in \{1 \ldots n\}}$$

Point example

$$p_5 = [\quad getX = \lambda o.5;\quad setX = \lambda o.\lambda i.\ \ o.getX \Leftarrow \lambda o.i\ ]$$

$$p_5 = [\quad x = 5;\quad getX = \lambda o.\ \ o.x;\quad setX = \lambda o.\lambda i.\ \ o.x \Leftarrow i\ ]$$

$$Point = Obj(X.[getX : \mathbb{N};\, setX : \mathbb{N} \rightarrow X])$$

λ-calculus

$$
\begin{aligned}
[\![\lambda x.b]\!] &= [arg = \varsigma x.x; body = \varsigma x.[\![b]\!]] \\
[\![f\ a]\!] &= (f.arg \Leftarrow a).body \\
[\![x]\!] &= x.arg
\end{aligned}
$$

$$
\begin{aligned}
[\![(\lambda x.x + x)\ 1]\!] \ = & \\
& ([arg = \varsigma x.x; body = \varsigma x.x.arg + x.arg].arg \Leftarrow 1).body \\
\longrightarrow\ & [arg = 1; body = \varsigma x.x.arg + x.arg].body \\
\longrightarrow\ & [arg = 1; body = \cdots].arg + [arg = 1; body = \cdots].arg \\
\longrightarrow\ & 1 + 1 \\
\longrightarrow\ & 2
\end{aligned}
$$

Classes

- **Implemented as traits**
- ***new* method**
- **Method traits**

$$
\begin{array}{lllll}
c & \equiv & [& new & = & \varsigma z.\ [getX = \varsigma s.\ z.getX(s); setX = \varsigma s.\ z.setX(s)]; \\
 & & & getX & = & \lambda s.\ 5; \\
 & & & setX & = & \lambda s.\lambda i.\ s.getX \Leftarrow \lambda s.i
\end{array}
$$

Interpretations

- **Primitive, axiomatic (Cardelli, Abadi)**
  - Fully functional
  - Syntactic
  - F ω, <:, μ
- **Existential (Hoffman, Pierce, Turner)**
  - Updates on fields only
  - Type-specific interpretation
  - F ω, <:

Type Theory

- **Advantages**
  - Many models:

    Set theoretic [Howe]

    PER [Allen, Mendler]

    Denotational [Rezus]

    Recursive Realizibility [Aczel]

    Categorical [Palmgren]

    •••
  - Predicativity
  - Large, expressive, formal foundation
- **Disadvantages**
  - Predicativity

Formal Objects in Type Theory

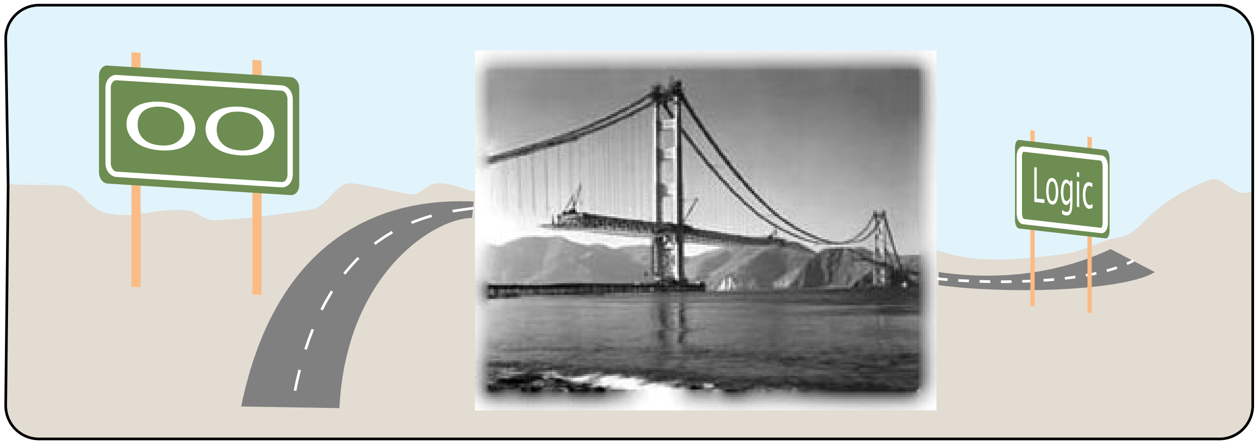
- **Scale in formal systems**
  - Jackson, large formalization of abstract algebra

    difficulty with: subtyping, modularity, inheritance
  - Large software verifications

    Horus group communications system in Nuprl

    SML in HOL, strong normalization of *F* in LEGO

    AAMP5 Microprocessor in PVS
- **Join two communities**

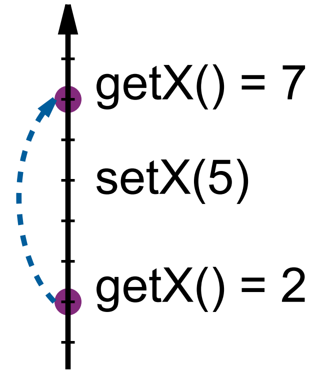


Results

- **One new type constructor for defining object types**
  - Formal interpretation of objects
- **New theory structure for verification systems**
  - Correspondences (propositions-as-types)
    - Object types and formal theories
    - Objects and proofs

Object Types

- **Existential interpretation [Hoffman, Pierce, Turner]**
  - "state-application" semantics
- **Primitive calculus is possible [Hickey]**
  - complex interpretation
- **Canonical example: movable point**



$$\begin{aligned} Point \quad \equiv \quad \exists Rep\colon \mathbf{Type}.\{\ & state\colon Rep; \\ & methods\colon \{\ getX : Rep \to \mathbb{Z}; \\ & \qquad\qquad\quad setX : Rep \to \mathbb{Z} \to Rep\}\} \end{aligned}$$

Weak existential

$$\exists a\colon A.B_a \equiv \bigcap_{T:\mathbf{Type}} \left(\bigcap_{a:A} (B_a \to T) \right) \to T$$

- **Hides first component**
- **Two points *p*$_{1,}$ *p*$_{2}$ ∈ *Point* <u>may</u> have different representation types**

$$\begin{aligned} \texttt{pack}(x) &\equiv \lambda f.f(x) \\ \texttt{open}\, o \,\texttt{as}\, r \,\texttt{in}\, b_r &\equiv o\,(\lambda r.b_r) \end{aligned}$$

$\texttt{pack}(x) \in \exists a\colon A.B_a$ if $\exists a \in A$ such that $x \in B_a$,

$\texttt{open}\, o \,\texttt{as}\, r \,\texttt{in}\, b_r \in T$ if $o \in \exists a\colon A.B_a$ and $\lambda y.b_y \in \bigcap a\colon A.(B_a \to T)$

Objects

- **Object types specified with method descriptions**

  $PointM \equiv \lambda Rep.\{getX : Rep \to \mathbb{Z}; setX : Rep \to \mathbb{Z} \to Rep\}$

- **Generic type constructor**

  $Object \equiv \lambda M.\exists Rep : \mathbf{Type}.\{state : Rep; methods : M(Rep)\}$

- **Point example:**

$$\begin{aligned} Point &\equiv Object(PointM) \\ &= \exists Rep : \mathbf{Type}.\{\ state : Rep; \\ &\qquad methods : \{\ getX : Rep \to \mathbb{Z}; \\ &\qquad\qquad setX : Rep \to \mathbb{Z} \to Rep\}\} \end{aligned}$$

Record Encoding

- **Functions on labels**

$$\begin{aligned} \{l_i = x_i{}^{i\in\{1\ldots n\}}\} &\equiv \lambda l.\,\textbf{case}\; l \;\textbf{of}\; l_1{:}\,x_1|\cdots|l_n{:}\,x_n|\_{:}\,\textbf{Top} \\ r\cdot l_j &\equiv r(l_j) \\ r\cdot l_j := x &\equiv \lambda l.\,\textbf{if}\; l = l_j \;\textbf{then}\; x \;\textbf{else}\; r\cdot l \end{aligned}$$

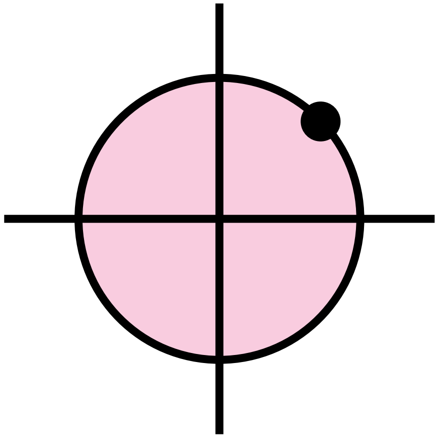
- **Type: dependent function type**

$$\{l_i{:}\,T_i{}^{i\in\{1\ldots n\}}\} \equiv l{:}\,\textbf{Atom} \rightarrow \textbf{case}\; l \;\textbf{of}\; l_1{:}\,T_1|\cdots|l_n{:}\,T_n|\_{:}\,\textbf{Top}$$

Dependent Records

- More expressive specifications
- Binary methods
- Points on the unit circle:

$$\begin{aligned} P^{\circ} \equiv [\, x, y\colon \mathbb{R};\ & \\ & spec\colon x^2 + y^2 \approx 1; \\ & rot\colon \mathbb{R} \to P^{\circ}] \end{aligned}$$



Very-dependent function encoding

- **Dependent function** $a\colon A \rightarrow B_a \qquad (\Pi a\colon A.B_a)$
- **Very-dependent function allows calls in the range:**

$$\{f \mid a\colon A \rightarrow B_{f,a}\}$$

- **Circle points:**

$$\begin{aligned}\{f \mid l\colon \mathbf{Atom} \rightarrow \ & \mathbf{case}\ l\ \mathbf{of}\\ & \text{“}getX\text{”} \rightarrow \mathbb{R}\\ & \text{“}getY\text{”} \rightarrow \mathbb{R}\\ & \text{“}spec\text{”} \rightarrow f(\text{“}getX\text{”})^2 + f(\text{“}getY\text{”})^2 \approx 1\\ & \text{“}rot\text{”} \rightarrow \cdots\\ & \_ \rightarrow \mathbf{Top}\}\end{aligned}$$

Dependent Record Encoding

- **Map function *f* over occurrences of labels**

$$\{l_i : M_i \ ^{i \in \{1 \ldots n\}}\} \quad = \quad \{f \mid l : \mathbf{Atom} \rightarrow \ \mathbf{case}\ l\ \mathbf{of}$$

$$\vdots$$

$$l_i \rightarrow M_i[f(l_j)/l_j \ ^{i \in \{1 \ldots i-1\}}]$$

$$\vdots$$

$$\_ \rightarrow \mathbf{Top}\}$$

Point object

- **Point is untyped**

$$
\begin{array}{lll}
p_0 & \equiv & \texttt{pack}\{ \; state = 0; \\
 & & \quad methods\!:\!\{ \; getX = \lambda x.x; \\
 & & \quad \quad setX = \lambda x, i.x + i\}\}
\end{array}
$$

Interfaces

- **unpack object, call method, repack object**

$$
\begin{aligned}
Point\text{`}setX = \lambda p, i.\ & \texttt{open}\, p\, \texttt{as}\ r\ \texttt{in} \\
& \texttt{pack}(r \cdot state := (r \cdot methods \cdot setX)\ (r \cdot state)\ i)
\end{aligned}
$$

- ***Point`setX*** Υ ***Point*** $\Delta \subseteq \Delta$ ***Point***

Subsumption

- **Untyped interfaces are polymorphic**

$$Point\text{`}setX \in \bigcap_{A \preceq Point} (A \to \mathbb{Z} \to A)$$

- **Interface typing**
- **Define subobject relation on methods**

$$\begin{aligned} A \preceq B \quad \text{iff} \quad & \exists AM, BM : \mathbf{Type} \to \mathbf{Type}. \\ & A = Object(AM) \wedge B = Object(BM) \\ & \wedge \forall T : \mathbf{Type}. AM\ T \subseteq BM\ T \end{aligned}$$

Propositions-as-types

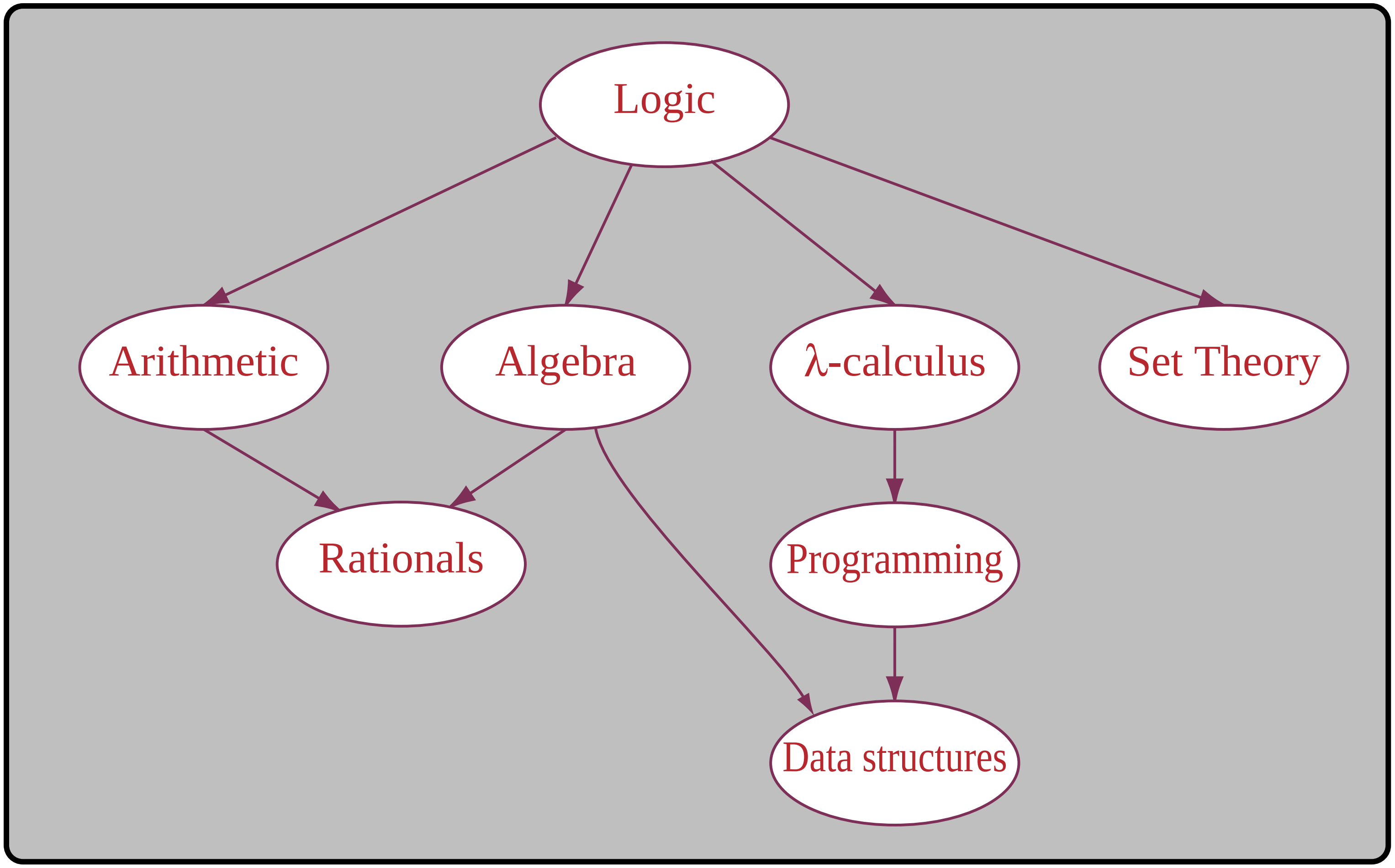
- **Proof of an object type is an object**
- **Theory construction:**
  - state axioms
  - derive theorems
- **Relationships of theories are formalizable**
- **Add rules to object types**

II. Theories as Objects

- **Expressive, dependent, object signatures**
- **Binary operators**
- **Expected subtyping properties GroupM 9 MonoidM**

$$
\begin{aligned}
\mathit{MonoidM} \equiv \{\ & car\colon \mathbf{Type};\\
& \oplus\colon car \rightarrow car \rightarrow car;\\
& \approx\colon car \rightarrow car \rightarrow \mathbf{Prop};\\
& e\colon car;\\
& assoc\colon \forall x, y, z\colon car.(x \oplus y) \oplus z \approx x \oplus (y \oplus z);\\
& left\_unit\colon \forall x\colon car.x \oplus e \approx x;\\
& right\_unit\colon \forall x\colon car.e \oplus x \approx x\}
\end{aligned}
$$

Modular Type Theory



Conclusion

- **Type theoretic interpretation provides mathematical understanding**
- **New contributions:**
  - Dependent object types
  - New form of “binary” methods
- **Feed principles of object-oriented programming back into formal systems**
  - Modularity (multiple, possibly inconsistent, domains)
  - Formal re-use
  - Enables collaboration