Exploring a Type-Theoretic Environment for Python

Ryan Brill.

Research in Industrial Projects for Students REU: UCLA iPAM & National University of Singapore.

Set Theory vs. Type Theory

- Russell's paradox, 1901: Let $R = \{x | x \notin x\}$. $R \in R \implies R \notin R$ and $R \notin R \implies R \in R$, contradiction
- Need to put mathematics on a strong logical/axiomatic foundation
- Competing candidates: set theory & type theory
- The expressions of Type Theory are **terms**; all terms have a **type**
- The expression "x has type T" is written $x : T$
- $0 : \mathbb{N}$ means 0 has type "natural number"
- Every object in Type Theory has a type: there are types for functions, types for proofs, types for types themselves, etc.
- **Theorems are types, proofs are terms**
- Set Theory is built on top of propositional and predicate logic, and elements can belong to multiple sets; In Type Theory, propositional and predicate logic are encoded as types, and terms can only belong to one type
- **It is easier for a machine to check type-theoretic proofs**

Inductive & Record Types

- An **Inductive Type** is a type equipped with rules (called **constructors**) that explain how the terms of a type are built
- Defining $\mathbb{N}$ as the inductive type `nat` in Coq:

```
Inductive nat :  Set := 0 :  nat, S : nat -> nat
```

- Using the constructors `0` and `S`, which have type `nat` and `nat -> nat`, we construct terms of type `nat`
- Represent $0, 1, 2, ... \in \mathbb{N}$ by `0, S(0), S(S(0)),...  :  nat`
- A **Record Type** is a type composed of fields of different types
- Defining $\mathbb{Q}$ as the record type `rat` in Coq:

```
Record rat :  { num :  nat, denom :  nat, sign :  bool
denom_cond :  bottom_neq_0, irred_cond :  irreducible}
```

- Can represent a wide variety of mathematical structures as inductive/record types
- Easier to prove statements about inductive/record types because terms of these types are **constructive** (we can build them), and their types are made explicit

Syntactic Rules

- Formally, a **proof** is a sequence of applications of **syntactic rules**
- Γ is a set of formulas, φ, ψ, and θ are formulas, and $\vdash$ means "proves"
- $\frac{A}{B}$ means if A holds, then we may deduce B
- Examples:

$\Gamma \vdash \varphi$ if $\varphi \in \Gamma$ (Assume)
$\Gamma \vdash t = t$ for all terms t (Reflexivity)

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}(\wedge EL) \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}(\wedge ER) \qquad \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}(\wedge I)$$

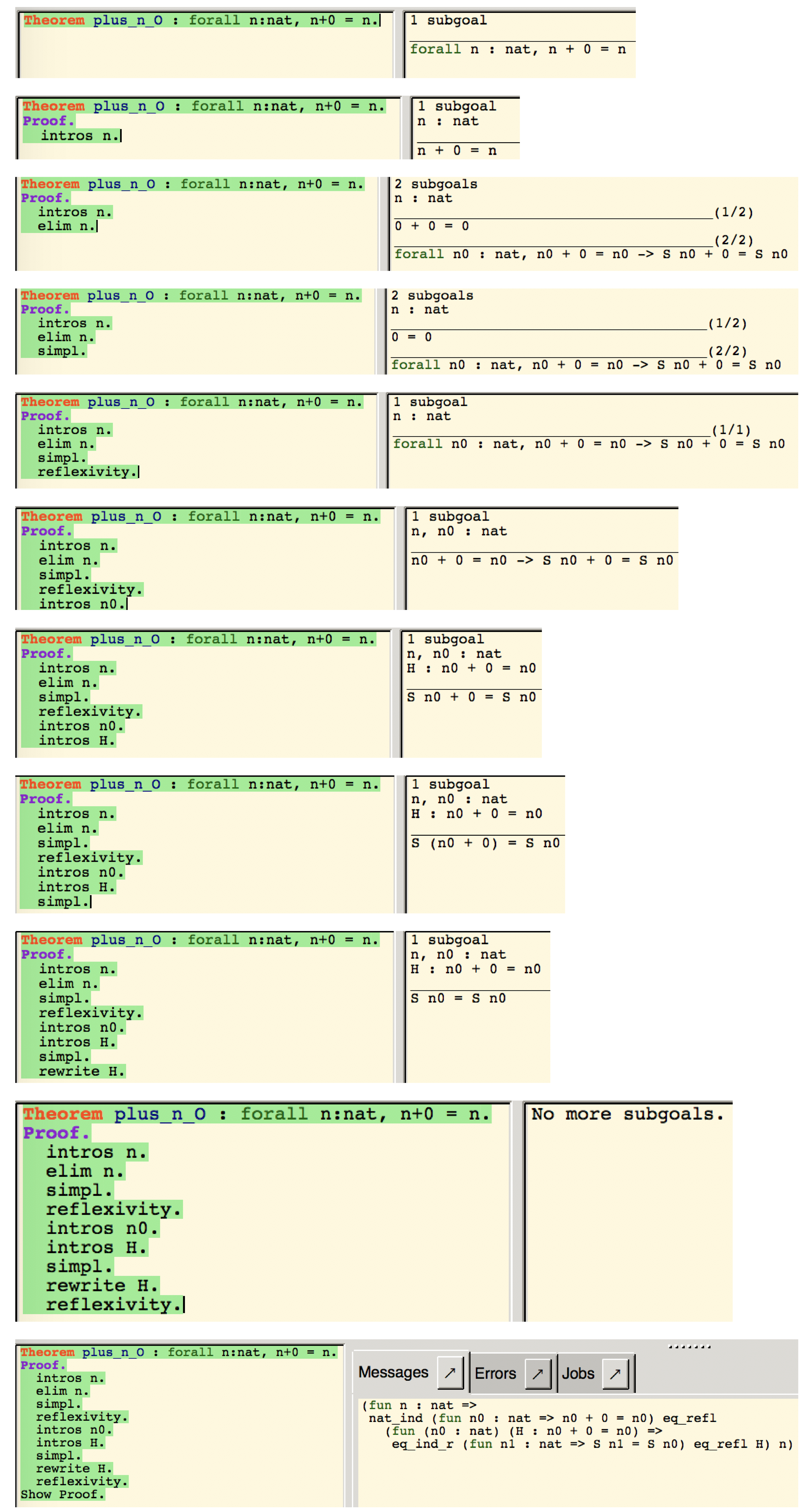
$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}(\vee IL) \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}(\vee IR)$$

$$\frac{\Gamma \vdash \varphi \to \psi}{\Gamma \cup \{\varphi\} \vdash \psi}(\to E) \qquad \frac{\Gamma \cup \{\varphi\} \vdash \psi}{\Gamma \vdash \varphi \to \psi}(\to I)$$

$$\frac{\Gamma \cup \{\varphi\} \vdash \theta \quad \Gamma \cup \{\psi\} \vdash \theta}{\Gamma \cup \{\varphi \wedge \psi\} \vdash \theta}(\vee PC) \qquad \frac{\Gamma \cup \{\psi\} \vdash \varphi \quad \Gamma \cup \{\neg\psi\} \vdash \varphi}{\Gamma \vdash \varphi}(\neg PC)$$

Interactive Theorem Proving in Coq

- Theorems are types, proofs are terms
- Interactive Theorem Provers are **Type-Checkers**: build a proof term, and check that its type matches your desired theorem
- Build proof terms by applying syntactic rules, using backwards-chaining logic
- Example: prove "`forall n :  nat, n + 0 = n`" in Coq:



Interactive Theorem Proving in Python?

- **Motivation:** Nvidia, our project's sponsor, wants a type-theoretic environment in Python, because Python is ubiquitous these days, and because Nvidia wants machine learning and "machine reasoning" on the same platform (Python)
- **Main Challenge #1:** A Coq expert told us it would take a team of experts > 3 years to write a robust interactive theorem prover in Python
- **Main Challenge #2:** Successful interactive theorem provers (Coq, Isabelle) written in functional programming languages (ML, OCAML), whereas Python allows functional programming, imperative programming, object-oriented programming
- We think an **object-oriented** layout is best for a type-theoretic library in Python
- Classes Type and `Term`
- `Term` has subclasses `Variable, Constant, Application, Abstraction, OrderedPair, RecordTerm, ...`
- `Type` has subclasses `Inductive, Record, Implication, Conjunction, Disjunction, ...`
- Using Nat (i.e. $\mathbb{N}$) in Python:

```
nat = Inductive("nat")
O = Const("O")
S = Const("S")
InductiveTypeIntro(O, nat)
InductiveTypeIntro(S, Implication(nat, nat))

one = Application(S, O)
two = Application(S, Application(S, O))
print(type_check(two))
>>> nat
```

- Implement syntactic rules as methods of the `Thm` class:

0. $A \to B \vdash A \to B$ by assume $A \to B$
1. $A \vdash A$ by assume A
2. $A, A \to B \vdash B$ by implication elimination on 0,1
3. $A \vdash (A \to B) \to B$ by implication introduction on 2
4. $\vdash A \to (A \to B) \to B$ by implication introduction on 3

```
th0 = Thm.assume(Term.mk_implies(A, B))
th1 = Thm.assume(A)
th2 = Thm.implies_elim(th0, th1)
th3 = Thm.implies_intr(Term.mk_implies(A, B), th2)
th4 = Thm.implies_intr(A, th3)
print(printer.print_thm(thy, th4, unicode=True))

⊢ A ⟶ (A ⟶ B) ⟶ B
```

Acknowledgements

Thank you Zhangsheng Lai at Nvidia for sponsoring the project. Thank you David Chew and Ziyuan Gao at NUS for mentoring us. Thank you Susana Serna for coordinating the RIPS program. Thank you IMS at NUS and IPAM at UCLA for hosting the project.

Contact Information

- **Email**: ryansbrill@gmail.com