Contributions in Programming Languages Theory

Logical Relations and Type Theory

Amin Timany

May 29$^{\text{th}}$, 2018

Leuven, Belgium

Introduction

- Computer systems are ubiquitous
- Crucial to **formally** verify correctness of safety- and security-critical systems
- **Types systems** play an important role
  - Foundation of (a class of) proof assistants
    - Formalization of mathematics, including the theory and practice of program verification
  - Compilers use types to ensure certain aspects of correctness of programs, e.g., type safety (well-typed terms do not crash)
- In this thesis we contribute to the theory of programming languages and type theory

In this talk

- A short historical account of set theory and type theory
- Part I: Type theory and formalization of mathematics
- Part II: Studying programs & programming languages through types

Cantorian Set theory

Set theory was introduced by **Georg Cantor** in 1870s to study infinities

In the first paper on the subject
"**On a Property of the Collection of All Real Algebraic Numbers**":

- $|\mathbb{N}| = |\mathsf{Alg}|$
- $|\mathbb{N}| < |\mathbb{R}|$

Rusell's paradox

In Cantorian set theory any collection is set!

In 1901 Bertrand Russell asked "**How about the set S of all sets that do not include themselves!?**"

$$S = \{X | X \notin X\}$$

This leads to contradictions

$$S \in S \quad \text{if and only if} \quad S \notin S$$

Saving set theory from paradoxes

- Theory of types by Russell and Whitehead
  - A hierarchy of types $T_0, T_1, \ldots$
  - Each set has a type
  - Elements of a set have **strictly** smaller type

- Axiomatic set theory by (amongst others) Zermelo, Fraenkel
  - Axioms stating properties and construction of sets
  - Zermelo-Fraenkel set theory with axiom of Choice (ZFC) is best known and most used set theory among mathematicians

Lambda calculus for logic and computation

In 1932 Church introduced λ-calculus in "**A Set of Postulates for the Foundation of Logic**" as the computational part of a logical system

Kleene and Rosser showed this system to by logically inconsistent

Church introduced

- **Simply typed** λ-calculus as a logically consistent system
  - Later extended with dependent types, universes, etc., e.g., the Coq proof assistant
- **Untyped** λ-calculus as a model computation (along Turing machines and recursion theory)
  - Later extended with other primitives and type systems
  - Forms the basis of (functional) programming languages, e.g., Haskell and ML family

An overview of the contributions in this thesis

- Part I: Type theory and formalization of mathematics
  - Formalization of category theory in Coq (Chapter 3)
  - Extend the predicative calculus of inductive constructions (pCIC), the underlying type system of Coq (Chapter 4)

- Part II: Studying programs & programming languages through types
  - Logical relations models (a versatile proof technique based on types)
  - Prove type safety and equivalence of programs
    - Formalized (in Coq) logical relations model for an advanced programming language (Chapter 5)
    - Establish proper encapsulation of state by a Haskell-style ST monad (Chapter 6)
    - Study continuations in the presence of concurrency (Chapter 7)

Part I

Dependent type theory, universes and cumulativity

- Typing judgement: $\Gamma \vdash t : T$ (e.g., $\Gamma \vdash 1 : \mathbb{N}$)
- Dependent type theory: every type is also a term
- $\Gamma \vdash T : T$ is paradoxical (similar to Russell's paradox)
- Solution: a hierarchy of universes (types of types), e.g., in Coq:

$$\mathtt{Type}_0 : \mathtt{Type}_1 : \ldots$$

- Cumulative type theory (e.g., Coq): $\mathtt{Type}_i \preceq \mathtt{Type}_j$ for $i \leq j$
- For cumulativity (subtyping) relation $T \preceq T'$

$$t : T \quad \text{then} \quad t : T'$$

Dependent type theory, universes and cumulativity

- Universe polymorphism:

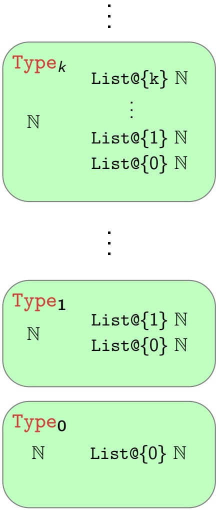
```
Inductive List@{i} (A : Type@{i}) : Type@{i}
| nil : List@{i} A
| cons : A → List@{i} A → List@{i} A.
```

Example:

$\texttt{nil} : \texttt{List}\ \mathbb{N}$,

$\texttt{cons}\ 1\ \texttt{nil} : \texttt{List}\ \mathbb{N}$,

$\texttt{cons}\ 2\ (\texttt{cons}\ 1\ \texttt{nil}) : \texttt{List}\ \mathbb{N}$



Category Theory in Coq (Chapter 3)

- Formalized a category theory library in Coq
- The most complete formalization of category theory in a proof assistant when considering basic category theory (not enriched or higher category theory)
- Defines categories in a universe polymorphic way

```
Record Category@{i j} :=
{
  Obj : Type@{i};
  Hom : Obj → Obj → Type@{j};
  ...
}.
```

Category Theory in Coq (Chapter 3)

- Universes to represent smallness/largeness
- Category of categories:

```
Definition Cat@{i j j l} : Category@{i j} :=
{
  Obj : Category@{k l};
  ...
}.
```

- Coq infers constraints, e.g., for `Cat`: $k < i, l < i, k \leq j, l \leq j$

Cumulative inductive types in Coq (Chapter 4)

- In category theory, every small category is also large
- No cumulativity in pCIC:

⋮

Type

`Category@{i j}`

$\preceq$

`Category@{k l}`

$\mathtt{i} = \mathtt{k}$ and $\mathtt{j} = \mathtt{l}$

⋮

Type

`Category@{i j}`

⋮

Cumulative inductive types in Coq (Chapter 4)

- We introduce the predicative calculus of cumulative inductive constructions (pCuIC)
- Extend the cumulativity to inductive types, e.g., lists, categories, etc.

Cumulative inductive types in Coq (Chapter 4)

- We introduce the predicative calculus of cumulative inductive constructions (pCuIC)
- Extend the cumulativity to inductive types, e.g., lists, categories, etc.
- This means:

Small categories are large as expected!

$\vdots$

Type

`Category@{i j}`

$\preceq$

`Category@{k l}`

i $\leq$ k and j $\leq$ l

$\vdots$

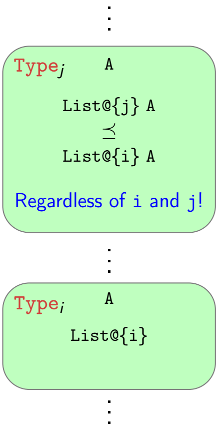
Type

`Category@{i j}`

$\vdots$

Cumulative inductive types in Coq (Chapter 4)

- We introduce the predicative calculus of cumulative inductive constructions (pCuIC)
- Extend the cumulativity to inductive types, e.g., lists, categories, etc.
- This means:



Lists with elements of type `A` are just lists independent of the universe!

Cumulative inductive types in Coq (Chapter 4)

- We introduce the predicative calculus of cumulative inductive constructions (pCuIC)
- Extend the cumulativity to inductive types, e.g., lists, categories, etc.
- This means:

Lists with elements of type `A` are just lists independent of the universe!

We extend Coq's judgemental equality:

$$\texttt{List@\{i\} A} \simeq \texttt{List@\{j\} A}$$

$\vdots$

$\texttt{Type}_j$ `A`

`List@{j} A`

$\simeq$

`List@{i} A`

Regardless of `i` and `j`!

$\vdots$

$\texttt{Type}_i$ `A`

`List@{i}`

$\vdots$

Cumulative inductive types in Coq (Chapter 4)

- A set theoretic model in ZFC based on the model of Werner and Lee
- Axiom: a hierarchy of uncountable strongly inaccessible cardinals to model universes

$$\kappa_0, \kappa_1, \ldots$$

- This extension is available in Coq as Coq 8.7

Part II

Logical relations

A semantic approach to type safety and contextual equivalence

- A versatile tool to study programs and programming languages through their types
- Versatility: (strong) normalization, **type safety**, **contextual equivalence**, non-interference, etc.

Type safety

- Type safety:

$$\cdot \vdash e : T \quad \text{then} \quad \mathit{Safe}(e)$$

$$\mathit{Safe}(e) \triangleq e \ \ \textbf{will not crash}$$

- Example of unsafe program:

$$\texttt{10 - ``abc''}$$

- Idea: define logical relations $\Gamma \models e : T$
  - We show congruence w.r.t. typing

$$\frac{\Gamma \vdash f : T_1 \to T_2 \qquad \Gamma \vdash e : T_1}{\Gamma \vdash f\ e : T_2} \quad \Rightarrow \quad \frac{\Gamma \models f : T_1 \to T_2 \qquad \Gamma \models e : T_1}{\Gamma \models f\ e : T_2}$$

  Example: $\Gamma \vdash \mathit{fact} : \mathbb{N} \to \mathbb{N}$ and $\Gamma \vdash 5 : \mathbb{N}$. Hence $\Gamma \vdash \mathit{fact}\ 5 : \mathbb{N}$

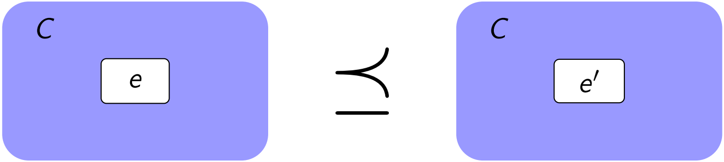
  - Fundamental theorem: $\Gamma \vdash e : T$ then $\Gamma \models e : T$
  - Adequacy: $\cdot \models e : T$ then $\mathit{Safe}(e)$
  - Soundness: $\cdot \vdash e : T$ then $\mathit{Safe}(e)$

Contextual refinement and equivalence

- Contextual refinement (the gold standard of comparison of programs):

  $\Gamma \vdash e \preceq_{\mathrm{ctx}} e' : T \triangleq$ **No program can distinguish replacing** e' **with** e

  That is, for any context C (a program with a hole)



$$\Gamma \vdash e \simeq e' : T \triangleq \Gamma \vdash e \preceq e' : T \wedge \Gamma \vdash e' \preceq_{\mathrm{ctx}} e : T$$

Contextual refinement and equivalence

- Idea: define logical relations $\Gamma \models e \preceq e' : T$
  - We show congruence w.r.t. typing

$$\frac{\Gamma \vdash f : T_1 \to T_2 \qquad \Gamma \vdash e : T_1}{\Gamma \vdash f\ e : T_2} \quad \Rightarrow \quad \frac{\Gamma \models f \preceq f' : T_1 \to T_2 \qquad \Gamma \models e \preceq e' : T_1}{\Gamma \models f\ e \preceq f'\ e' : T_2}$$

  - Fundamental theorem: $\Gamma \vdash e : T$ then $\Gamma \models e \preceq e : T$
  - Soundness: $\Gamma \models e \preceq e' : T$ then $\Gamma \vdash e \preceq_{\mathsf{ctx}} e' : T$

Logical relations for advanced type systems

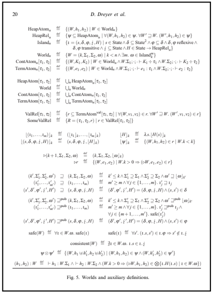
- Constructing LR models for advanced features, e.g., higher-order references, is **complicated**
- Requires advanced techniques: **step-indexing and recursive Kripke worlds**
- These complicate the model
- We use Iris featuring high-level reasoning principles for these techniques

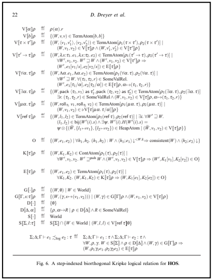
TS and CR via LR in Iris (chapter 5)

- Unary and binary LR models for $\mathsf{F}_{\mu,ref,conc}$ (ML-like) featuring:
  - **polymorphic types**
  - **higher-order references**
  - **recursive types**
  - **concurrency**

$$
\begin{aligned}
[\![\Xi \vdash X]\!]_\Delta(v, v') &\triangleq \Delta(X)(v, v') \\
[\![\Xi \vdash \mathbb{N}]\!]_\Delta(v, v') &\triangleq v = v' \in \mathbb{N} \\
[\![\Xi \vdash \tau_1 \times \tau_2]\!]_\Delta(v, v') &\triangleq \exists v_1, v_2, v_1', v_2'.\ v = (v_1, v_2) \land v' = (v_1', v_2') \land \\
&\qquad [\![\Xi \vdash \tau_1]\!]_\Delta(v_1, v') \land [\![\Xi \vdash \tau_2]\!]_\Delta(v_2, v_2') \\
[\![\Xi \vdash \tau_1 \to \tau_2]\!]_\Delta(v, v') &\triangleq \forall w, w'.\ \square([\![\Xi \vdash \tau_1]\!]_\Delta(w, w') \twoheadrightarrow [\![\Xi \vdash \tau_2]\!]^{\mathcal{E}}_\Delta(v\ w, v'\ w')) \\
[\![\Xi \vdash \forall X.\, \tau]\!]_\Delta(v, v') &\triangleq \forall f.\ \square([\![X, \Xi \vdash \tau]\!]_{\Delta[X \mapsto f]}(v\ \_, v'\ \_)) \\
[\![\Xi \vdash \mu X.\, \tau]\!]_\Delta(v, v') &\triangleq \mu f.\, \exists w, w'.\ v = \mathtt{fold}\ w \land v' = \mathtt{fold}\ w' \land \\
&\qquad \triangleright [\![X, \Xi \vdash \tau]\!]_{\Delta[X \mapsto f]}(w, w') \\
[\![\Xi \vdash \mathtt{ref}(\tau)]\!]_\Delta(v, v') &\triangleq \exists \ell.\ v = \ell \land v' = \ell' \land \\
&\qquad \boxed{\exists w, w'.\ \ell \mapsto w * \ell' \mapsto w' * [\![\Xi \vdash \tau]\!]_\Delta(w, w')}^{\mathcal{N}.\ell}
\end{aligned}
$$

$$
[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta(e, e') \triangleq \forall \rho, j, K,\quad
\begin{array}{c}
\{spec\_inv(\rho) * j \Rrightarrow e'\} \\
e \\
\{v.\ \exists v'.\ j \Rrightarrow v' * [\![\Xi \vdash \tau]\!]_\Delta(v, v')\}
\end{array}
$$





- Prove refinement of pairs of fine-/coarse-grained concurrent modules: counters and stacks
- All results are formalized in Coq

Equivalences in the presence of the ST monad (chapter 6)

- Prove equivalences for STLang, A PL featuring a Haskell-style STmonad
- Idea of STmonad (by Launchbury and Peyton Jones): encapsulate state
- That is, memory is used but programs remain pure (as though they do not use memory)!
- Type system ensures effects are **restricted**
- ST monad marks computations with their memory region

Equivalences in the presence of the ST monad (chapter 6)

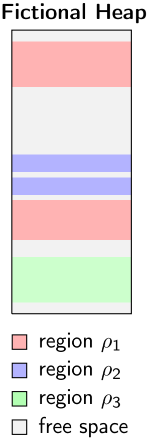
- ST monad marks computations with their memory region

$$\mathtt{ST}\ \rho\ \tau$$

$$\textsf{Tderef}\ \frac{\Xi \mid \Gamma \vdash e : \mathtt{STRef}\ \rho\ \tau}{\Xi \mid \Gamma \vdash !\,e : \mathtt{ST}\ \rho\ \tau}$$

- `runST` runs a suspended computation that **can be run in any region**, i.e., region-independent programs

- These programs an run in any region and thus also in the empty region!



State-independence theorem

- We formally prove the explained intuitive reasoning why programs are pure

- State-independence theorem:

  Consider the program

  $$\cdot \mid x : \texttt{STRef}\ \rho\ \tau' \vdash e : \tau$$

  If e can run in one memory state then it can run in any memory state!

Contextual equations we prove

Justifies proper encapsulation of state

$$e \preceq_{\mathrm{ctx}} () : 1 \qquad \text{(Neutrality)}$$

$$\mathtt{let}\, x = e_2 \,\mathtt{in}\, (e_1, x) \approx_{ctx} (e_1, e_2) : \tau_1 \times \tau_2 \qquad \text{(Commutativity)}$$

$$\mathtt{let}\, x = e \,\mathtt{in}\, (x, x) \approx_{ctx} (e, e) : \tau \times \tau \qquad \text{(Idempotency)}$$

$$\mathtt{let}\, y = e_1 \,\mathtt{in}\, \mathtt{rec}\, f(x) = e_2 \preceq_{\mathrm{ctx}} \mathtt{rec}\, f(x) = \mathtt{let}\, y = e_1 \,\mathtt{in}\, e_2 : \tau_1 \to \tau_2 \qquad \text{(Rec hoisting)}$$

$$\mathtt{let}\, y = e_1 \,\mathtt{in}\, \Lambda\, e_2 \preceq_{\mathrm{ctx}} \Lambda\, (\mathtt{let}\, y = e_1 \,\mathtt{in}\, e_2) : \forall X.\, \tau \qquad \text{(Λ hoisting)}$$

$$e \preceq_{\mathrm{ctx}} \mathtt{rec}\, f(x) = (e\ x) : \tau_1 \to \tau_2 \qquad \text{(η expansion for rec)}$$

$$e \preceq_{\mathrm{ctx}} \Lambda\, (e\ \_) : \forall X.\, \tau \qquad \text{(η expansion for Λ)}$$

$$(\mathtt{rec}\, f(x) = e_1)\ e_2 \preceq_{\mathrm{ctx}} e_1[e_2, (\mathtt{rec}\, f(x) = e_1)/x, f] : \tau \qquad \text{(β reduction for rec)}$$

$$(\Lambda\, e)\ \_ \approx_{ctx} e : \tau[\tau'/X] \qquad \text{(β reduction for Λ)}$$

$$\mathtt{bind}\, e \,\mathtt{in}\, (\lambda\, x.\, \mathtt{return}\, x) \approx_{ctx} e : \mathrm{ST}\ \rho\ \tau \qquad \text{(Left Identity)}$$

$$e_2\ e_1 \preceq_{\mathrm{ctx}} \mathtt{bind}\, (\mathtt{return}\, e_1) \,\mathtt{in}\, e_2 : \mathrm{ST}\ \rho\ \tau \qquad \text{(Right Identity)}$$

$$\mathtt{bind}\, (\mathtt{bind}\, e_1 \,\mathtt{in}\, e_2) \,\mathtt{in}\, e_3 \preceq_{\mathrm{ctx}} \mathtt{bind}\, e_1 \,\mathtt{in}\, (\lambda\, x.\, \mathtt{bind}\, (e_2\ x) \,\mathtt{in}\, e_3) : \mathrm{ST}\ \rho\ \tau' \qquad \text{(Associativity)}$$

Rel. verification of programs with continuations (chapter 7)

- We study $\mathsf{F}^{\mu,ref}_{conc,cc}$: $\mathsf{F}_{\mu,ref,conc}$ with continuations
- Programs can be suspended into continuations
- Continuations can be resumed
- We use weakest preconditions to prove correctness of programs

$$\mathsf{wp}\ e\ \{\Phi\}$$

Example:

$$\mathsf{wp}\ \mathtt{let}\ x = 3\ \mathtt{in}\ x * 2\ \{v.\ v = 6\}$$

- Continuations make the bind rule inadmissible
- The bind rule is essential for modular (context-local) reasoning

$$\text{inadmissible-bind}\quad \frac{\mathsf{wp}\ e\ \{v.\ \mathsf{wp}\ K[v]\ \{\Phi\}\}}{\mathsf{wp}\ K[e]\ \{\Phi\}}$$

Rel. verification of programs with continuations (chapter 7)

- Introduce context-local weakest preconditions

$$\textsf{bind}\quad \frac{\mathsf{clwp}\ e\left\{v.\ \mathsf{clwp}\ K[v]\ \{\Phi\}\right\}}{\mathsf{clwp}\ K[e]\ \{\Phi\}}$$

- Same proof rules as weakest preconditions (except for continuations themselves)
- We can mix and match weakest preconditions with context local ones

$$\textsf{clwp-wp}\quad \frac{\mathsf{clwp}\ e\left\{\Psi\right\} \qquad \forall v.\ \Psi(v) \twoheadrightarrow \mathsf{wp}\ K[v]\ \{\Phi\}}{\mathsf{wp}\ K[e]\ \{\Phi\}}$$

Rel. verification of programs with continuations (chapter 7)

- Use context-local weakest preconditions together with our LR model
- Prove equivalence of continuation-based web servers and state-storing web servers
  - Prove equivalence of context-local parts using context-local reasoning principles

```
let fname =
  read_client ()
in
let lname =
  read_client ()
in
...
```

```
if sessions[sessid].fname = "" then
  sessions[sessid].fname := input;
  exit ()
else
if sessions[sessid].lname = "" then
  sessions[sessid].lname := input;
  exit ()
else if ...
```

Rel. verification of programs with continuations (chapter 7)

- Formalize the proof that continuations can be simulated using one-shot continuations in the presence of **concurrency**
  - In the presence of concurrency this holds subtly
  - Requires a more involved proof than the sequential version

Thanks!