Contributions in Programming Languages Theory
Logical Relations and Type Theory

Amin Timany

May 29th, 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”:

- $|\mathbb{N}| = |\text{Alg}|$
- $|\mathbb{N}| < |\mathbb{R}|$
Russell’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

$$S \in S \text{ if and only if } S \notin S$$
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
In 1932 Church introduced $\lambda$-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 be logically inconsistent.
Church introduced

- **Simply typed** $\lambda$-calculus as a logically consistent system
  - Later extended with dependent types, universes, etc., e.g., the Coq proof assistant

- **Untyped** $\lambda$-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:
  \[
  \text{Type}_0 : \text{Type}_1 : \ldots
  \]
- Cumulative type theory (e.g., Coq): $\text{Type}_i \preceq \text{Type}_j$ for $i \leq j$
- For cumulativity (subtyping) relation $T \preceq T'$
  \[
  t : T \text{ then } t : T'
  \]
Dependent type theory, universes and cumulativity

- **Universe polymorphism:**

  Inductive $\text{List}_{\{i\}} (A : \text{Type}_{\{i\}}) : \text{Type}_{\{i\}}$
  
  | nil : $\text{List}_{\{i\}} A$
  | cons : $A \rightarrow \text{List}_{\{i\}} A \rightarrow \text{List}_{\{i\}} A$.

Example:

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

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

- Coq infers constraints, e.g., for Cat: $k < i, l < i, k \leq j, l \leq j$
In category theory, every small category is also large.

No cumulativity in pCIC:

\[
\begin{align*}
\text{Type} & \quad \text{Category}\{i, j\} \\
& \preccurlyeq \quad \text{Category}\{k, l\} \\
i = k \quad \text{and} \quad j = l
\end{align*}
\]
We introduce the predicative calculus of cumulative inductive constructions (pCuIC)

Extend the cumulativity to inductive types, e.g., lists, categories, etc.
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!
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 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:

$$\text{List}^@{i} A \simeq \text{List}^@{j} A$$

Regardless of $i$ and $j$!
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:

\[ \cdot \vdash e : T \text{ then } \text{Safe}(e) \]

\[ \text{Safe}(e) \triangleq e \text{ will not crash} \]

Example of unsafe program:

\[ 10 - "abc" \]

Idea: define logical relations \( \Gamma \models e : T \)

We show congruence w.r.t. typing

\[
\frac{\Gamma \vdash f : T_1 \rightarrow T_2 \quad \Gamma \vdash e : T_1}{\Gamma \vdash f \> e : T_2} \quad \Rightarrow \quad \frac{\Gamma \models f : T_1 \rightarrow T_2 \quad \Gamma \models e : T_1}{\Gamma \models f \> e : T_2}
\]

Example: \( \Gamma \vdash \text{fact} : \mathbb{N} \rightarrow \mathbb{N} \) and \( \Gamma \vdash 5 : \mathbb{N} \). Hence \( \Gamma \vdash \text{fact} \> 5 : \mathbb{N} \)

Fundamental theorem: \( \Gamma \vdash e : T \text{ then } \Gamma \models e : T \)

Adequacy: \( \cdot \models e : T \text{ then } \text{Safe}(e) \)

Soundness: \( \cdot \vdash e : T \text{ then } \text{Safe}(e) \)
Contextual refinement and equivalence

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

\[ \Gamma \vdash e \leq_{ctx} e' : T \triangleq \text{No program can distinguish replacing } e' \text{ with } e \]

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

\[ \Gamma \vdash e \simeq e' : T \triangleq \Gamma \vdash e \leq e' : T \wedge \Gamma \vdash e' \leq_{ctx} e : T \]
Contextual refinement and equivalence

- Idea: define logical relations $\Gamma \models e \preceq e' : T$

- We show congruence w.r.t. typing

\[
\begin{align*}
\Gamma \vdash f : T_1 \rightarrow T_2 & \quad \Gamma \vdash e : T_1 \\
\Gamma \vdash f \ e : T_2 & \quad \Rightarrow \\
\Gamma \vdash f \preceq f' : T_1 \rightarrow T_2 & \quad \Gamma \vdash e \preceq e' : T_1 \\
\Gamma \vdash f \ e \preceq f' \ e' : T_2 &
\end{align*}
\]

- Fundamental theorem: $\Gamma \vdash e : T$ then $\Gamma \models e \preceq e : T$

- Soundness: $\Gamma \models e \preceq e' : T$ then $\Gamma \vdash e \preceq_{ctx} e' : T$
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
Prove refinement of pairs of fine-/coarse-grained concurrent modules: counters and stacks

All results are formalized in Coq
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

  $$\text{ST } \rho \tau$$

  \[\text{Tderef} \quad \Xi | \Gamma \vdash e : \text{STRef } \rho \tau\]

  \[\Xi | \Gamma \vdash ! e : \text{ST } \rho \tau\]

- **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

\[ \cdot | x : \text{STRef} \rho \tau' \vdash e : \tau \]

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

\[ e \preceq_{\text{ctx}} () : 1 \]  \hspace{1cm} \text{(Neutrality)}

\[ \text{let } x = e_2 \text{ in } (e_1, x) \approx_{\text{ctx}} (e_1, e_2) : \tau_1 \times \tau_2 \]  \hspace{1cm} \text{(Commutativity)}

\[ \text{let } x = e \text{ in } (x, x) \approx_{\text{ctx}} (e, e) : \tau \times \tau \]  \hspace{1cm} \text{(Idempotency)}

\[ \text{let } y = e_1 \text{ in rec } f(x) = e_2 \preceq_{\text{ctx}} \text{ rec } f(x) = \text{let } y = e_1 \text{ in } e_2 : \tau_1 \to \tau_2 \]  \hspace{1cm} \text{(Rec hoisting)}

\[ \text{let } y = e_1 \text{ in } \Lambda e_2 \preceq_{\text{ctx}} \Lambda (\text{let } y = e_1 \text{ in } e_2) : \forall X. \tau \]  \hspace{1cm} \text{(\(\Lambda\) hoisting)}

\[ e \preceq_{\text{ctx}} \text{ rec } f(x) = (e \ x) : \tau_1 \to \tau_2 \]  \hspace{1cm} \text{(\(\eta\) expansion for rec)}

\[ e \preceq_{\text{ctx}} \Lambda (e _) : \forall X. \tau \]  \hspace{1cm} \text{(\(\eta\) expansion for \(\Lambda\))}

\[ (\text{rec } f(x) = e_1) \ e_2 \preceq_{\text{ctx}} e_1[e_2, (\text{rec } f(x) = e_1)/x, f] : \tau \]  \hspace{1cm} \text{(\(\beta\) reduction for rec)}

\[ (\Lambda e) _ e \approx_{\text{ctx}} e : \tau[^\tau'/X] \]  \hspace{1cm} \text{(\(\beta\) reduction for \(\Lambda\))}

\[ \text{bind } e \text{ in } (\lambda x. \text{return } x) \approx_{\text{ctx}} e : \text{ST } \rho \tau \]  \hspace{1cm} \text{(Left Identity)}

\[ e_2 \ e_1 \preceq_{\text{ctx}} \text{bind } (\text{return } e_1) \text{ in } e_2 : \text{ST } \rho \tau \]  \hspace{1cm} \text{(Right Identity)}

\[ \text{bind } (\text{bind } e_1 \text{ in } e_2) \text{ in } e_3 \preceq_{\text{ctx}} \text{bind } e_1 \text{ in } (\lambda x. \text{bind } (e_2 \ x) \text{ in } e_3) : \text{ST } \rho \tau[^\tau'/\tau] \]  \hspace{1cm} \text{(Associativity)}
We study $F_{\text{conc},\text{cc}}, F_{\mu,\text{ref},\text{conc}}$ with continuations.

Programs can be suspended into continuations.

Continuations can be resumed.

We use weakest preconditions to prove correctness of programs.

\[ wp\ e\ \{\Phi\} \]

Example:

\[ wp \ \text{let} \ x = 3 \ \text{in} \ x \times 2 \ \{v.\ v = 6\} \]

Continuations make the bind rule inadmissible.

The bind rule is essential for modular (context-local) reasoning.

\[ \text{inadmissible-bind} \]

\[ wp\ e\ \{v.\ wp\ K[v]\ \{\Phi\}\} \]

\[ \underline{wp\ K[e]\ \{\Phi\}} \]
Introduce context-local weakest preconditions

\[
\begin{align*}
\text{bind} \\
\text{clwp } e \{ v. \text{clwp } K[v] \{ \Phi \} \} \\
\text{clwp } K[e] \{ \Phi \}
\end{align*}
\]

Same proof rules as weakest preconditions (except for continuations themselves)

We can mix and match weakest preconditions with context local ones

\[
\begin{align*}
\text{clwp-wp} \\
\text{clwp } e \{ \Psi \} \quad \forall v. \Psi(v) \rightarrow \text{wp } K[v] \{ \Phi \} \\
\text{wp } K[e] \{ \Phi \}
\end{align*}
\]
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

```
1 let fname = 1 if sessions[sessid].fname = "" then
2   read_client () 2   sessions[sessid].fname := input;
3 in 3   exit ()
4 let lname = 4 else
5   read_client () 5 if sessions[sessid].lname = "" then
6 in 6   sessions[sessid].lname := input;
7 ... 7   exit ()
8 else if ...
```
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!