A Language for the Specification and Efficient Implementation of Type Systems

Pascal Wittmann

TU Darmstadt

October 23, 2014

Motivation

Example

Simply typed lambda calculus + Subtyping

```
~S = ~T
======== S-refl
~S <: ~T

~T1 <: ~S1
~S2 <: ~T2
========================= S-arrow
~S1 -> ~S2 <: ~T1 -> ~T2
```

Optimization Strategies

- Unfold typing rules

```
Int = Int                ~A -> ~B = Int
========== R1            =============== R2
Int <: Int               ~A -> ~B <: Int

Int = ~C -> ~D           ~A -> ~B = ~C -> ~D
=============== R3      ==================== R4
Int <: ~C -> ~D          ~A -> ~B <: ~C -> ~D
```

Optimization Strategies

- Unfold typing rules

```
Int = Int                ~A -> ~B = Int
========== R1            =============== R2
Int <: Int               ~A -> ~B <: Int

Int = ~C -> ~D           ~A -> ~B = ~C -> ~D
=============== R3      ==================== R4
Int <: ~C -> ~D          ~A -> ~B <: ~C -> ~D
```

- Remove typing rules that are subsumed by other typing rules

$$\texttt{~A -> ~B = ~C -> ~D} \implies (\texttt{~T1 <:~S1} \wedge \texttt{~S2 <:~T2})$$

$$(\texttt{~T1 <:~S1} \wedge \texttt{~S2 <:~T2}) \implies \texttt{~A -> ~B = ~C -> ~D}$$

Optimization Strategies

- Removal of typing rules with unsatisfiable premises

```
~A -> ~B = Int                Int = ~C -> ~D
=============== R2           =============== R3
~A -> ~B <: Int               Int <: ~C -> ~D
```

Optimization Strategies

- Removal of typing rules with unsatisfiable premises

```
~A -> ~B = Int                Int = ~C -> ~D
=============== R2           =============== R3
~A -> ~B <: Int               Int <: ~C -> ~D
```

- Removal of valid premises

```
Int = Int
========== R1
Int <: Int
```

Optimization Strategies

- Removal of typing rules with unsatisfiable premises

```
~A -> ~B = Int                Int = ~C -> ~D
=============== R2           =============== R3
~A -> ~B <: Int               Int <: ~C -> ~D
```

- Removal of valid premises

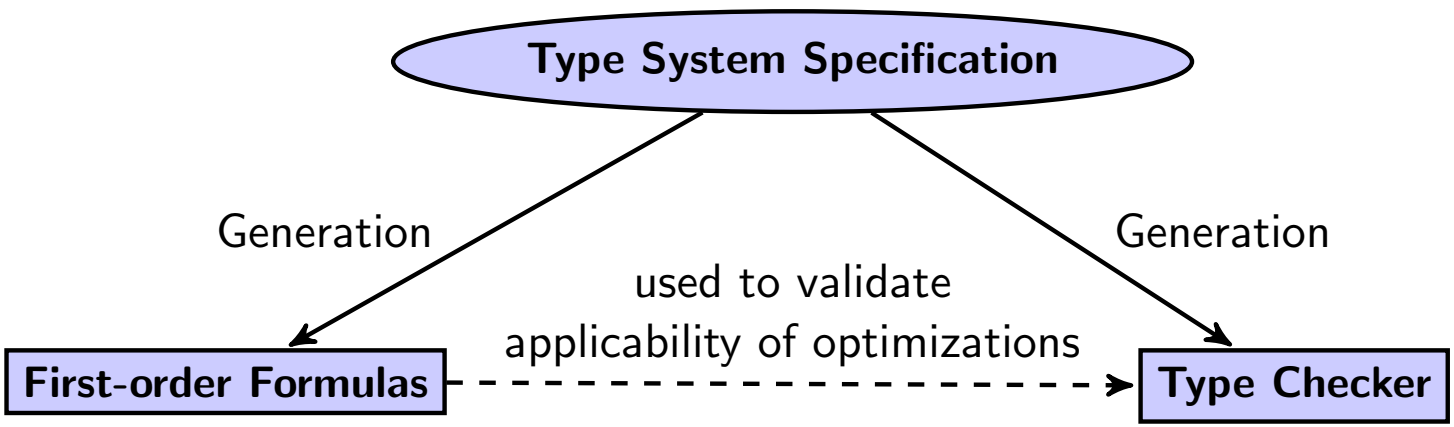
```
Int = Int
========== R1
Int <: Int
```

- After optimizations we have the following typing rules

```
                    ~T1 <: ~S1
                    ~S2 <: ~T2
========== R1       ========================== S-arrow
Int <: Int          ~S1 -> ~S2 <: ~T1 -> ~T2
```

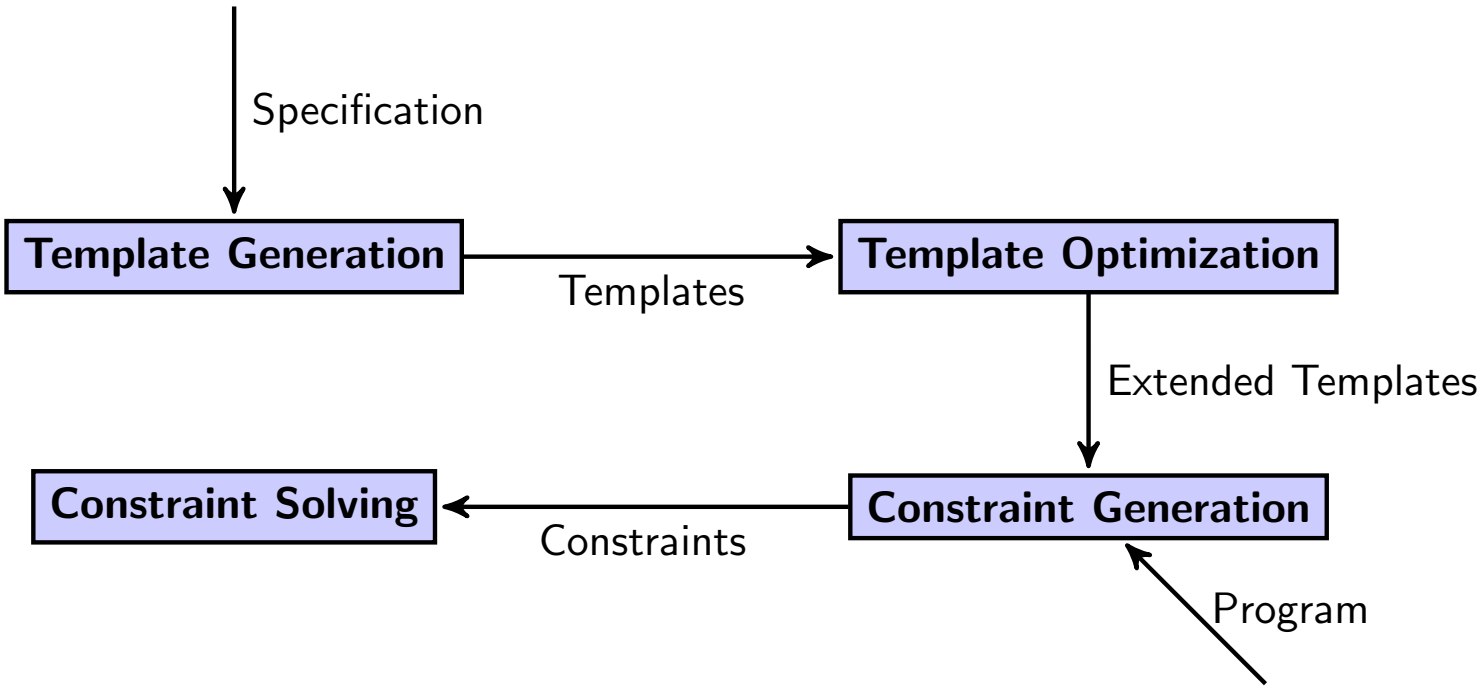
Architecture

- Automatic translation of specifications into first-order formulas
- Automated theorem provers are used to validate applicability of optimizations
- Generation of a type checker from optimized specifications



Type Checker

- Translation of typing rules in normalized templates
- Optimization of templates
- Type checking according to templates using a generic type checker



Conclusion & Future Work

- We contribute
  - a declarative, high-level specification language for type system
  - a translation of specifications into first-order formulas
  - optimization strategies to reduce the need of backtracking in the type checker
  - a type checker generator, which generates constraint-based type checkers
- We plan to
  - develop more optimization strategies (e.g. to optimize subsumption-like rules)
  - develop heuristics and proof strategies to validate the applicability of optimizations
  - apply our work to more realistic programming languages