# A Language for the Specification and Efficient Implementation of Type Systems

## Motivating Example

### Intuitive formalization of the simply typed lambda calculus with subtyping

```
============== T-int
$C |- &i : int

(%x : ~T1 ; $C) |- ~e : ~T2
=================================== T-abs
$C |- \ %x : ~T1 . ~e : ~T1 -> ~T2

$C |- ~e1 : ~T11 -> ~T12
$C |- ~e2 : ~T2
======================== T-app
$C |- ~e1 ~e2 : ~T12

 %x : ~T in $C
 ============== T-var
 $C |- %x : ~T

   $C |- ~t : ~S
   ~S <: ~T
   ============== T-sub
   $C |- ~t : ~T

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

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

Typing rules `T-sub` and `S-refl` are not syntax directed and thus require backtracking

### Unfolding of not syntax directed typing rules

We unfold not syntax directed rules by substituting the variables for all possible substructures to create syntax directed instances of those rules.

The unfolding of rule `S-refl` with the types `Int` and `Type -> Type`:

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

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

Int = ~C -> ~D
============== R3
Int <: ~C -> ~D

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

### Removal of subsumed typing rules

Unfolding rules may create ambiguities between typing rules. We attempt to eliminate those ambiguities by proving that one typing rule is subsumed by another. Subsumed typing rules are redundant and can be removed safely.

In the unfolding of `S-refl` the conclusions of typing rules `R4` and `S-arrow` match exactly the same terms. We prove the following lemma by structural induction `a -> b = c -> d ⇒ c <: a ∧ b <: d` to show that `R4` is subsumed by `S-arrow`.

Remaining typing rules of the unfolding:

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

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

Int = ~C -> ~D
============== R3
Int <: ~C -> ~D
```

### Removal of typing rules with unsatisfiable premises

To apply a typing rule all premises have to be satisfied, therefore typing rules with unsatisfiable premises cannot be applied and safely removed.

Remaining typing rules of the unfolding:

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

### Removal of valid premises

A valid premise is satisfied for all possible configurations, therefore we do not have to check if it is satisfiable during type checking and can safely remove the valid premise from the typing rule.

Remaining typing rules of the unfolding:

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

### Algorithmic version of the type system

```
============== T-int
$C |- &i : int

(%x : ~T1 ; $C) |- ~e : ~T2
=================================== T-abs
$C |- \ %x : ~T1 . ~e : ~T1 -> ~T2

%x : ~T in $C
============== T-var
$C |- %x : ~T

========== S-refl
Int <: Int

$C |- ~e1 : ~T11 -> ~T12
$C |- ~e2 : ~T2
~T2 <: ~T11
======================== T-app
$C |- ~e1 ~e2 : ~T12

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

No backtracking needed

Next step: Automate optimization of `T-sub`

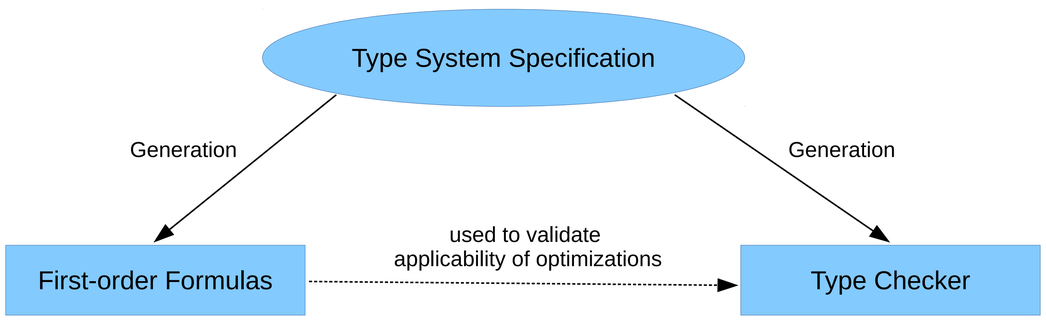
## The Approach

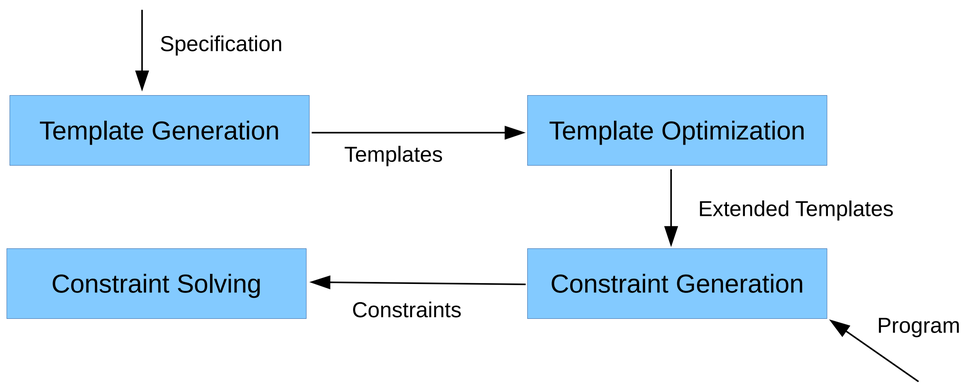
We have developed a **modular**, **declarative** and **high-level specification language** for type systems. It facilitates the specification of type systems close to text books by custom **context** and **judgment definitions**.
Typing rules can be annotated with **custom error messages** to produce user friendly messages in case a program is ill-typed.
Type systems are specified based on **SDF** syntax defintions of programming languages.

Type system specifications are used to create a **first-order formula representation** and a **type checker**. The first-order formula representation is used to validate the applicability of optimizations for typing rules.
The applicability is validated by proving corresponding theorems using automated theorem provers. The goal is to automatically transform type system specifications so that **no backtracking** is needed in the type checker.

We generate from the typing rules of a type system specification a set of normalized **templates**. Templates are an intermediate representation of typing rules, which have no implicit equalities and declare dependencies between premises explicitly.
In the next step the generated templates are **optimized** and ambiguities between templates are made explicit.

The optimized templates are the input for a **generic type checker**. The generic type checker is constraint based and has clearly separated constraint generation and constraint solving phases.
Custom errors that are thrown during constraint generation and constraint solving are collected and shown after type checking, in case of ill-typed programs.





Pascal Wittmann
TU Darmstadt
pascal.wittmann@stud.tu-darmstadt.de

Sebastian Erdweg
TU Darmstadt
erdweg@informatik.tu-darmstadt.de


TECHNISCHE UNIVERSITÄT DARMSTADT