A Language for the Specification and Efficient Implementation of Type Systems

Pascal Wittmann

TU Darmstadt

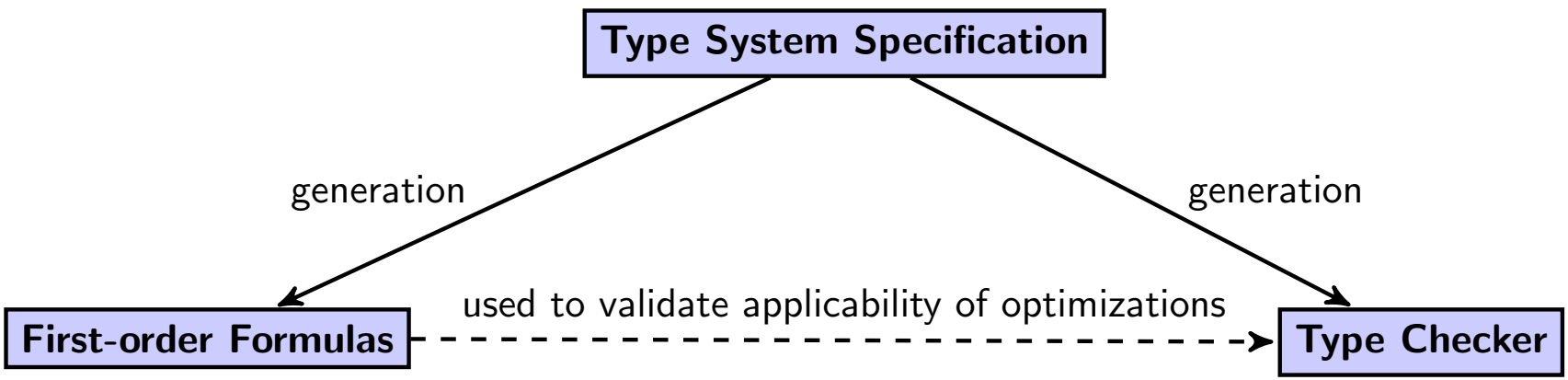
September 11, 2014

Motivation

- Type systems provide
  - static approximation of programs semantics
  - means to establish and enforce abstraction barriers
  - documentation that is always correct
- Domains specific languages benefit from specialized type systems
- Gap between formal definitions of type systems and their implementations

Research Problem

- Design of a declarative specification language that
  - is close to text-book formalisms
  - makes it easy to use existing programming language definitions
- Generate first-order formula representations of specifications
- Develop a type checker generator which
  - is constraint-based
  - can cope with non-syntax directed rules
  - takes advantage of facts proven by automated theorem provers



Specification Language I

```
module Typesystem

language specifications/SystemF/SystemF

contexts
TermBinding := ID{I} x Type{O}
TypeBinding := ID{I}

meta-variables  Term "~" { Type Exp }
                Ctx "$" { TermBinding TypeBinding }
                Id "%" { ID }
                Num "&" { Int }
```

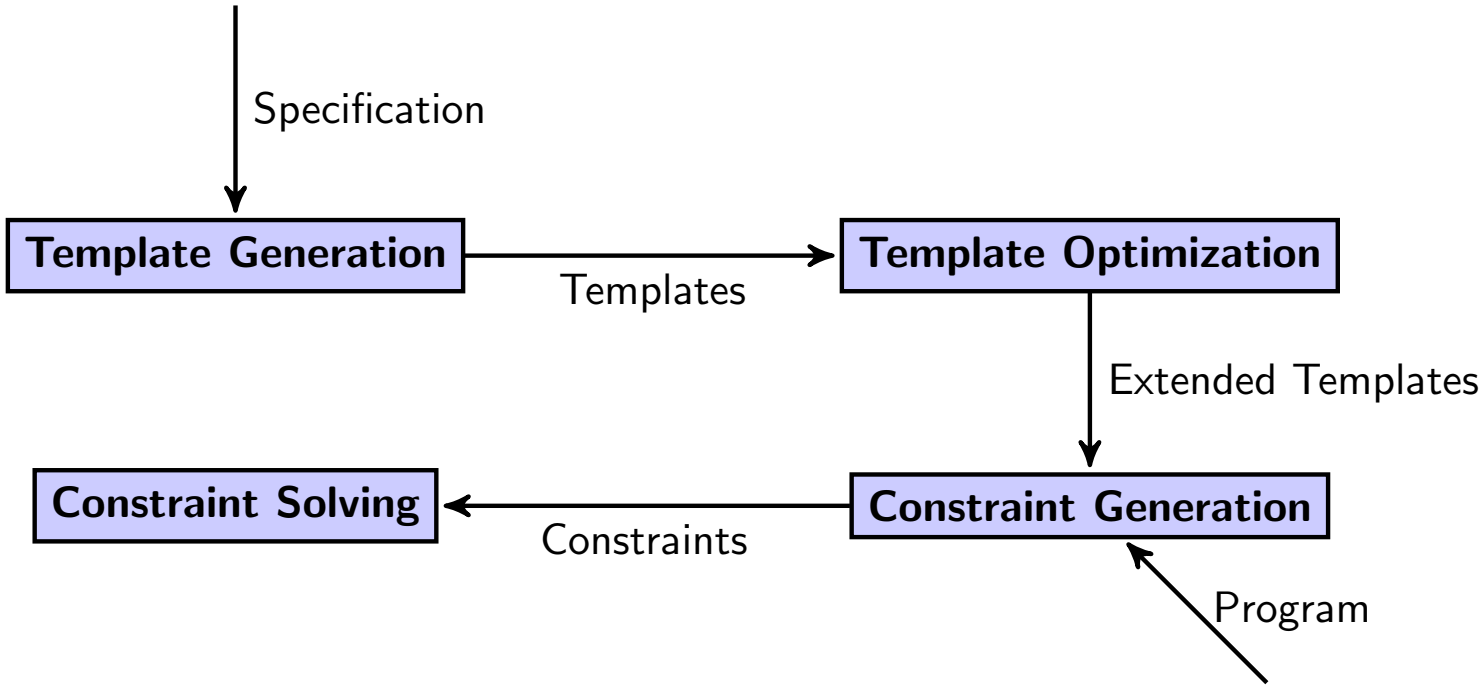
Specification Language II

```
judgments
TermBinding{I} "|" TypeBinding{I} "|-" Exp{I} ":" Type{O}.
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}.
ID{I} "fresh in" TypeBinding{I}.
ID{I} "!=" ID{I} is Neq.
```

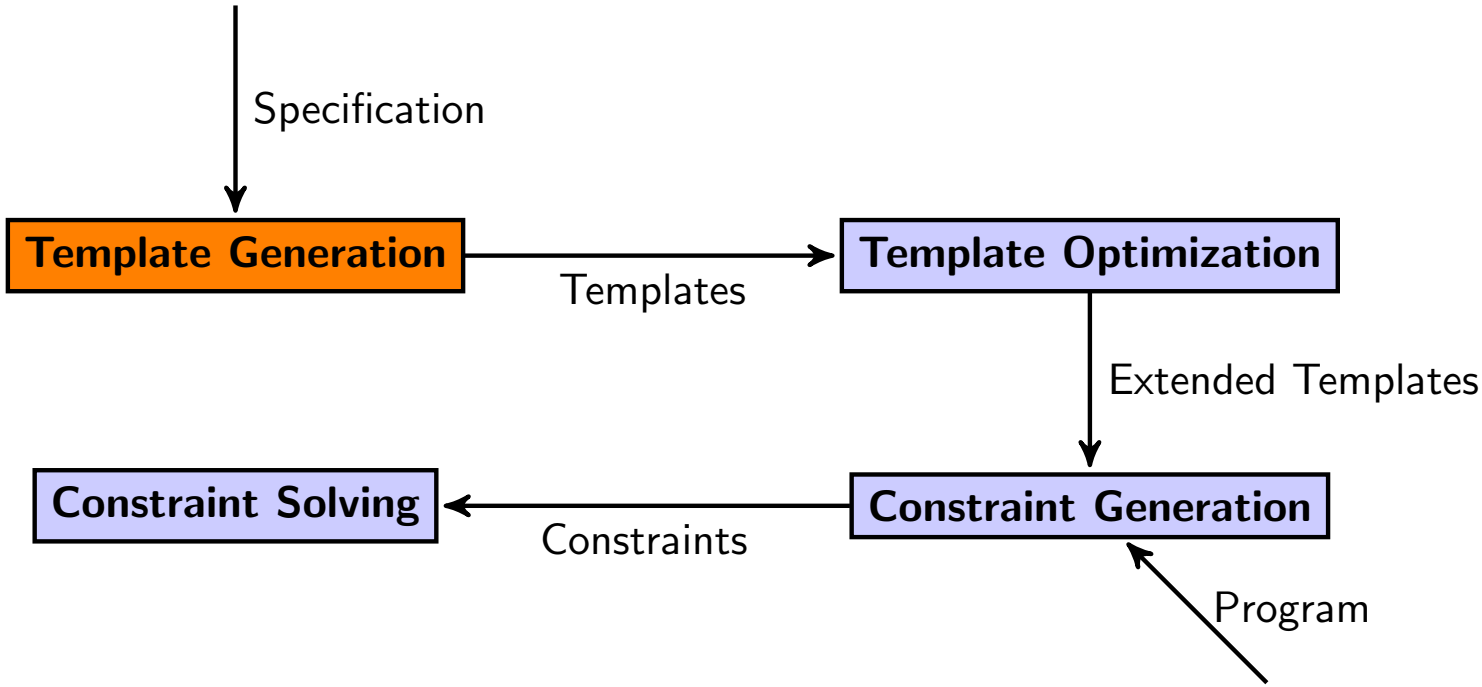
Specification Language III

```
rules

%x : ~T in $C1          @error %x "should have type" ~T
                               "but has type" {}.
=============== T-Var
$C1 | $C2 |- %x : ~T


~U = [ %x -> ~S ] ~T              @error ~U "is not" ~T "where"
                                         %x "is replaced by" ~S.
$C1 | $C2 |- ~e : all %x . ~T
================================ T-Tapp
$C1 | $C2 |- ~e [ ~S ] : ~U
```

Phases of the type checker generator



Phases of the type checker generator



Template Generation I

- Templates are an intermediate representation of the rules suitable for constraint generation
- An excerpt of the syntax definition of a template:

⟨*Template*⟩ ::= '`Template`' ⟨Premisses⟩ ⟨Conjecture⟩

⟨*Conjecture*⟩ ::= '`Conjecture`' ⟨Judg⟩ ⟨Name⟩ ⟨Pattern⟩
⟨Outputs⟩

⟨*Premisses*⟩ ::= ϵ | ⟨Premise⟩ ⟨Dependencies⟩ ⟨Premisses⟩

⟨*Premise*⟩ ::= '`Lookup`' ⟨Ctx⟩ ⟨Inputs⟩ ⟨Outputs⟩ ⟨Error⟩
| '`Judgment`' ⟨Judg⟩ ⟨Inputs⟩ ⟨Binding⟩ ⟨Outputs⟩ ⟨Error⟩
| '`Eq`' ⟨Term⟩ ⟨Term⟩ ⟨Error⟩
| '`Neq`' ⟨Term⟩ ⟨Term⟩ ⟨Error⟩

Template Generation II

- Resolve Dependencies between premisses
  $\langle \textit{Dependencies} \rangle ::= \epsilon \mid \langle \text{Judg} \rangle \langle \text{Outputs} \rangle \langle \text{Dependencies} \rangle$
- Resolve implicit equalities

```
TermBinding{I} "|" TypeBinding{I} "|-" Exp{I} ":" Type{O}.
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}.
```

```
~U = [ %x -> ~S ] ~T
$C1 | $C2 |- ~e : all %x . ~T
================================ T-Tapp
$C1 | $C2 |- ~e [ ~S ] : ~U


======================== Subst-Eq
~S = [ %x -> ~S ] %x
```

Figure: Premisses depend on each other in T-Tapp and Subst-Eq has an implicit equality

Template Generation II

- Resolve Dependencies between premisses
  $\langle \textit{Dependencies} \rangle ::= \epsilon \mid \langle \text{Judg} \rangle \ \langle \text{Outputs} \rangle \ \langle \text{Dependencies} \rangle$
- Resolve implicit equalities

```
TermBinding{I} "|" TypeBinding{I} "|-" Exp{I} ":" Type{O}.
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}.
```

```
$C1 | $C2 |- ~e : all %x . ~T
~U = [ %x -> ~S ] ~T
------------------------------ T-Tapp
$C1 | $C2 |- ~e [ ~S ] : ~U
```

```
---------------------- Subst-Eq
~S = [ %x -> ~S ] %x
```

Figure: Premisses depend on each other in T-Tapp and Subst-Eq has an implicit equality

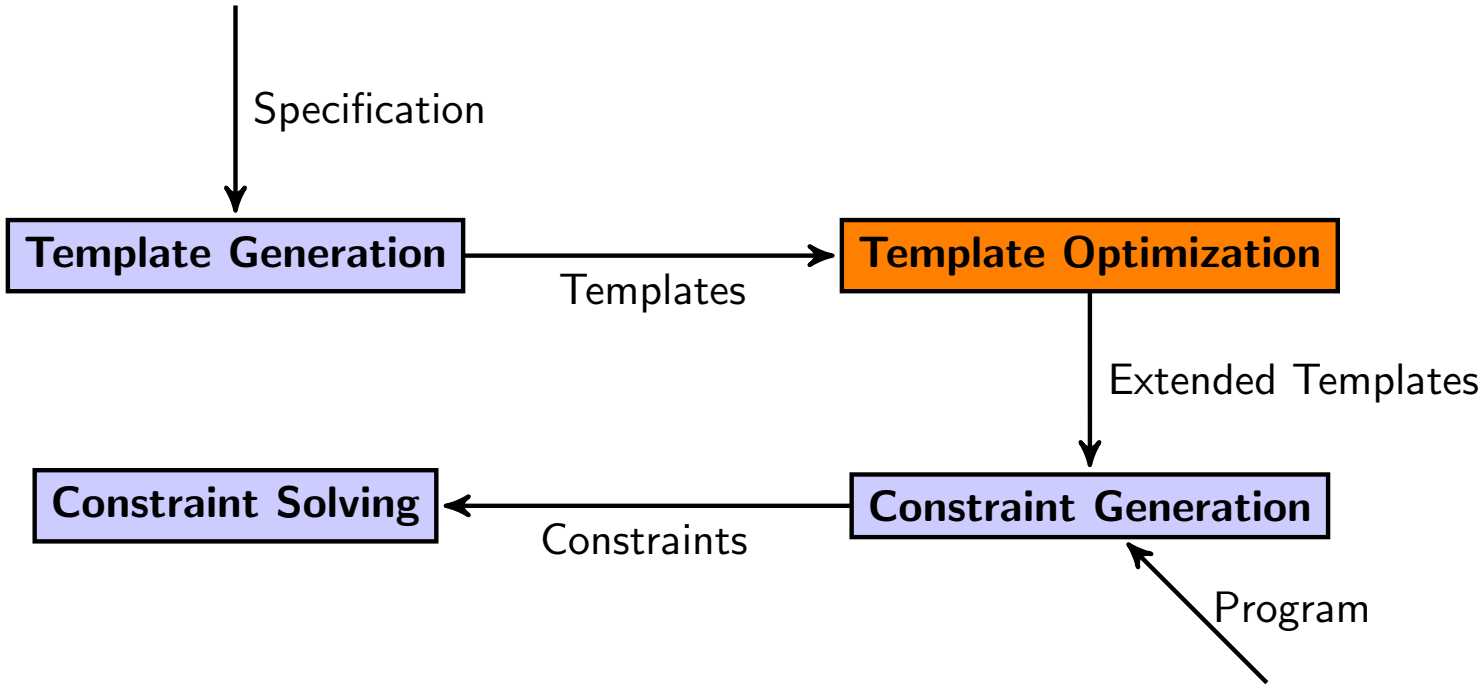
Template Generation II

- Resolve Dependencies between premisses
  $\langle \textit{Dependencies} \rangle ::= \epsilon \mid \langle \text{Judg} \rangle \langle \text{Outputs} \rangle \langle \text{Dependencies} \rangle$
- Resolve implicit equalities

```
TermBinding{I} "|" TypeBinding{I} "|-" Exp{I} ":" Type{O}.
Type{O} "= [" ID{I} "->" Type{I} "]" Type{I}.

$C1 | $C2 |- ~e : all %x . ~T
~U = [ %x -> ~S ] ~T
================================ T-Tapp
$C1 | $C2 |- ~e [ ~S ] : ~U

%y = %z
====================== SubstEq
~S = [ %y -> ~S ] %z
```

Figure: Premisses depend on each other in T-Tapp and Subst-Eq has an implicit equality

Phases of the type checker generator



Template Optimization I

- In contrast to the rules templates are normalized
- Templates can still contain redundancies and non-syntax-directed rules
- We use a first-order formula representation of the templates to prove properties that allow to remove redundancies and to make templates syntax directed
- The first-order formula representation of a template looks roughly like this

$$\forall FV(p_1, \ldots, p_n, c).p_1 \wedge \cdots \wedge p_n \implies c \tag{1}$$

where p_i and c are propositions representing the premisses respectively the conclusion

Template Optimization II

- A template is *when-ambiguous* if there is an other template such that there is at least one term that matches the conclusion of both templates
- If the set of terms is equal, they are *which-ambiguous*

```
~T <: ~S
{ $R } <: { $U }
================================ S-depth
{ %l : ~T $R } <: { %l : ~S $U }

{ $R } <: { $S }
================================ S-width
{ %l : ~T $R } <: { %l : ~T $S}
```

Figure: Which-ambiguity between depth and width subtyping rules of records

Template Optimization III

- We also try to solve a certain class of when-ambiguities, namely relations that do not involve contexts
- Look at the subtyping relation `Type{I} "<:" Type{I}`, where `Type` contains arrow types and the base type `int`

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


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

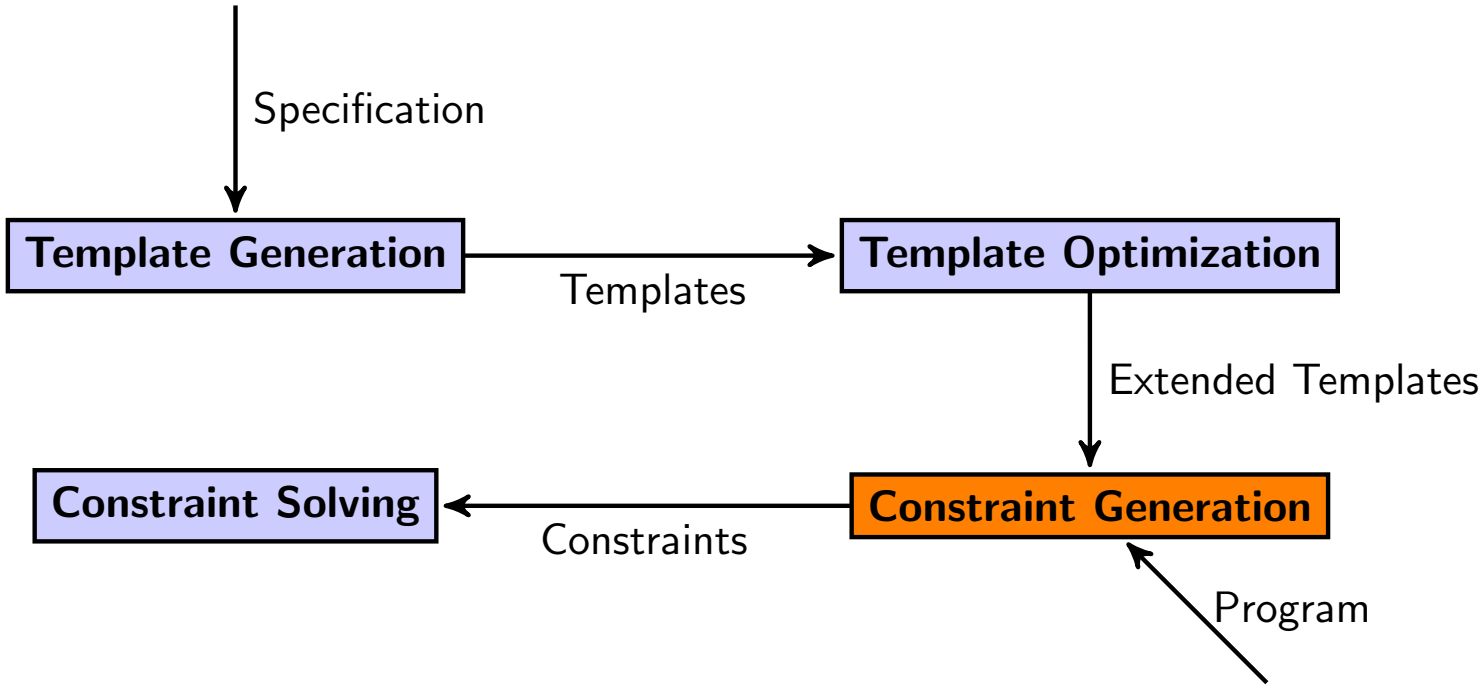
Template Optimization IV

- Ambiguities are captured in the template language by `Fork`
- `Fork` groups templates, such that they match at least on one common term
- After the last optimization step forks are sorted by $<$

$$t_1 < t_2 \iff m(t_1) \subseteq m(t_2) \tag{2}$$

where $m(t)$ computes the set of terms that match the conclusion of t

Phases of the type checker generator



Constraint Generation

Definition (The constraint language)

⟨*Constraint*⟩ ::= '`CFail`' ⟨Error⟩
| '`CEq`' ⟨Term⟩ ⟨Term⟩ ⟨Error⟩
| '`CNeq`' ⟨Term⟩ ⟨Term⟩ ⟨Error⟩

Definition (Rules for variables and abstraction for PCF)

```
%X : ~T in $C
============== T-var
$C |- %X : ~T
```

```
$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
==================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
(%X : ~T1 ; $C) |- ~E : ~T2
=================================== T-abs
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2
```

Constraint Generation II

```
%X : ~T in $C
=============
$C |- %X : ~T
```

```
(%X : ~T1 ; $C) |- ~E : ~T2
=======================================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2
```

```
$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
==================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
=================================================
() |- fun x : int (fun y : int (x)) : int -> int -> int
```

Constraint Generation II

```
%X : ~T in $C
=============
$C |- %X : ~T
```

```
(%X : ~T1 ; $C) |- ~E : ~T2
=======================================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2
```

```
$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
==================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
=================================================
() |- fun x : int (fun y : int (x x)) : int -> int -> int
```

Constraint Generation II

```
%X : ~T in $C
=============
$C |- %X : ~T
```

```
(%X : ~T1 ; $C) |- ~E : ~T2
=======================================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2
```

```
$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
==================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
          x:int |- fun y : int (x x) : int -> int
=================================================================
() |- fun x : int (fun y : int (x x)) : int -> int -> int
```

Constraint Generation II

```
%X : ~T in $C
=============
$C |- %X : ~T
```

```
(%X : ~T1 ; $C) |- ~E : ~T2
=======================================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2
```

```
$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
==================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
                    y:int; x:int |- x x : int
                  ===========================
                  x:int |- fun y : int (x x) : int -> int
  ================================================================
  () |- fun x : int (fun y : int (x x)) : int -> int -> int
```

Constraint Generation II

```
%X : ~T in $C
=============
$C |- %X : ~T

(%X : ~T1 ; $C) |- ~E : ~T2
=======================================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2

$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
=================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
y:int; x:int |- x : int          y:int; x:int |- x : int
=============================================================
                  y:int; x:int |- x x : int
          ========================================
          x:int |- fun y : int (x x) : int -> int
=============================================================
() |- fun x : int (fun y : int (x x)) : int -> int -> int
```

Constraint Generation II

```
%X : ~T in $C
============
$C |- %X : ~T

(%X : ~T1 ; $C) |- ~E : ~T2
=====================================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2

$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
=================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
x : int in (y:int; x:int)
=========================

y:int; x:int |- x : int          y:int; x:int |- x : int
=======================================================

                  y:int; x:int |- x x : int
          ========================================

          x:int |- fun y : int (x x) : int -> int
=================================================================

() |- fun x : int (fun y : int (x x)) : int -> int -> int
```

```
CEq(IntType,Var(Y13),Error("x should have type Var(Y13)
                            but has IntType")
```

Constraint Generation II

```
%X : ~T in $C
============
$C |- %X : ~T

(%X : ~T1 ; $C) |- ~E : ~T2
=====================================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2

$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
=================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
x : int in (y:int; x:int)          x : int in (y:int; x:int)
==========================         ==========================
y:int; x:int |- x : int             y:int; x:int |- x : int
=============================================================
                  y:int; x:int |- x x : int
          ======================================
          x:int |- fun y : int (x x) : int -> int
=============================================================
() |- fun x : int (fun y : int (x x)) : int -> int -> int
```

```
CEq(IntType,Var(Y13),Error("x should have type Var(Y13)
                            but has IntType"))
CEq(IntType,Var(Y15),Error("x should have type Var(Y15)
                            but has IntType"))
```

Constraint Generation II

```
%X : ~T in $C
=============
$C |- %X : ~T
```

```
(%X : ~T1 ; $C) |- ~E : ~T2
===========================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2
```

```
$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
=================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
y:int; x:int |- x : int          y:int; x:int |- x : int
=========================================================
                  y:int; x:int |- x x : int
              ======================================
              x:int |- fun y : int (x x) : int -> int
=========================================================
() |- fun x : int (fun y : int (x x)) : int -> int -> int
```

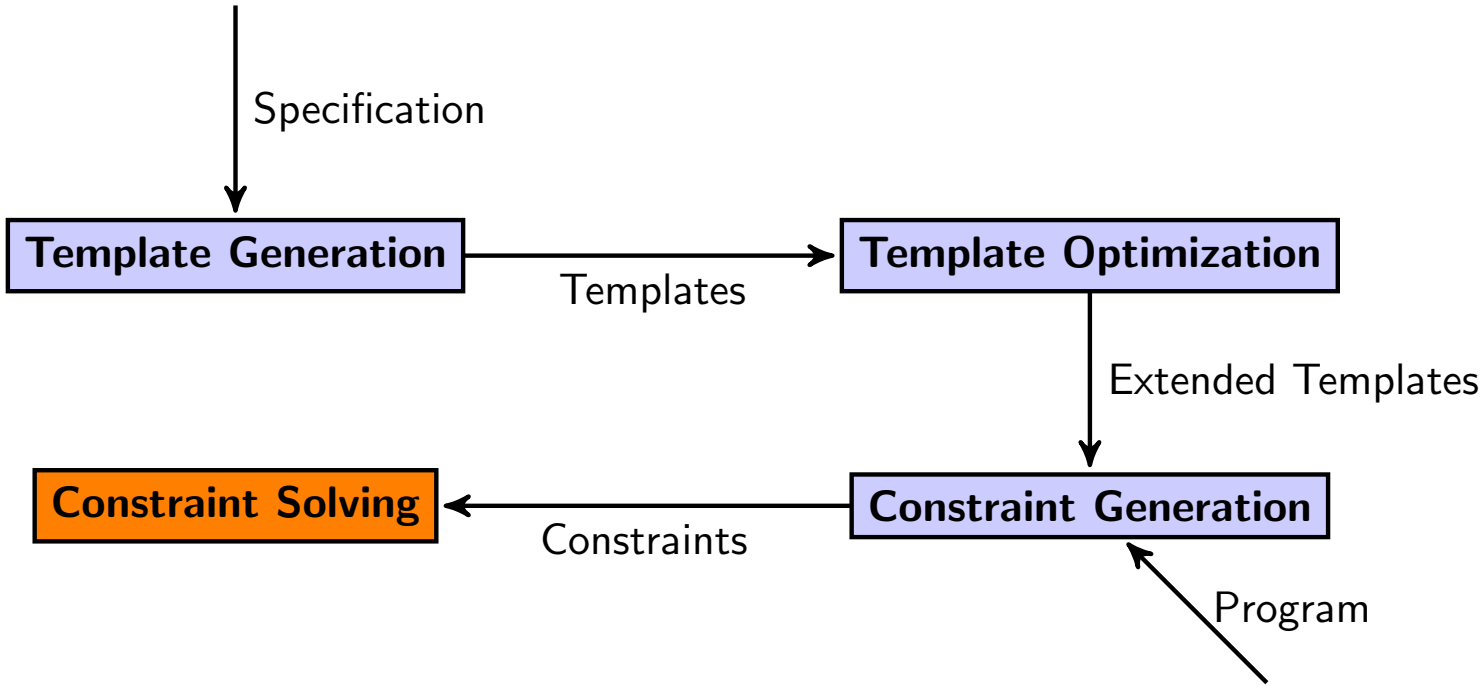
```
CEq(IntType,Var(Y13),Error("x should have type Var(Y13)
                            but has IntType"))
CEq(IntType,Var(Y15),Error("x should have type Var(Y15)
                            but has IntType"))
CEq(Var(Y13),FunType(Var(Y11),Var(Y9)),None)
CEq(Var(Y15),Var(Y11),None)
```

Constraint Generation II

```
%X : ~T in $C
=============
$C |- %X : ~T

(%X : ~T1 ; $C) |- ~E : ~T2
=======================================
$C |- (fun %X : ~T1 (~E)) : ~T1 -> ~T2

$C |- ~E1 : ~T1 -> ~T2
$C |- ~E2 : ~T1
==================== T-app
$C |- ~E1 ~E2 : ~T2
```

```
Error(
      [ "x"
      , "\"should have type\""
      , FunType(IntType(), IntType())
      , "\"but has\""
      , [IntType()]
      ]
     )
```

Phases of the type checker generator



Constraint Solving

- Constraints are solved by Robinson unification
- If a constraint cannot be unified the error message is recorded
- During unification a most general unifier (mgu) is computed
- On a successful unification the mgu is applied to the output of the constraint generation
- Otherwise the mgu is applied to the collected error messages

Conclusion

- We presented
  - a high-level specification language for type systems in SDF
  - a template language that normalizes typing rules
  - methods to optimize templates
  - a generic constraint-based type checker that uses templates as input
  - a first-order formula representation for type systems
- We applied the results to
  - programming languages like SystemF and variants of the lambda calculus with subtyping
  - security type systems from information flow security [VIS96]

References


Dennis Volpano, Cynthia Irvine, and Geoffrey Smith, *A sound type system for secure flow analysis*, J. Comput. Secur. **4** (1996), no. 2-3, 167–187.