A Language for the Specification and Efficient Implementation of Type Systems

Pascal Wittmann

December 3, 2014

Motivation

Goal: **Automatically** generate **efficient** type checkers from **high-level** specifications.

- Type systems provide
  - static approximation of program semantics
  - means to establish and enforce abstraction barriers
  - documentation in sync with source code
- Domain Specific Languages (DSL) benefit from specialized type systems
- Gap between formal definitions of type systems and their implementations

Research Problem

- Type system specifications may be have rules that overlap in non-trivial ways
- Those overlaps require the type checker to backtrack
- Currently, type systems are transformed hand into algorithmic type systems
- Ensuring preservation of semantics requires non-trivial proofs

How to remove overlap automatically while preserving the semantics?

Optimization Strategies

Example 1

Consider a subtyping relation on types `Int` *and* `Type -> Type`

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

Goal: Remove the overlap between rules `refl` *and* `arrow`*.*

General idea:

- Identify problematic rules
- Derive more specific versions of problematic rules
- Remove unnecessary rules

Strategy I: Unfolding

- The problematic rule in this example is `refl`
  - It is applicable to strictly more terms than `trans`
  - It is applicable to all instances of the subtyping judgment
- Idea: Unfold the structure of the variables in the conclusion

```
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
```

Strategy II/III: Unsatisfiable & Valid Premises

- The unfolding of rules is purely syntactic
- Exploit semantics to unnecessary rules
  - Remove valid premises from rules

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

  - Remove rules with unsatisfiable premises

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

Strategy IV: Subsumption

Definition 1

Rule r_1 subsumes r_2 if they have the same conclusion (modulo variable renaming) and the premises of rule r_2 imply the conjunction of all premisses of r_1.

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

Is one of these rules subsumed by the other?
Conjecture: The left rule is subsumed by the right rule.

Strategy IV: Subsumption

- We can remove subsumed typing rules while preserving the semantics of the type system
  - Both applicable to the same terms
  - The subsuming rule is applicable whenever the subsumed rule is
- Therefore we identify all subsumed rules and remove them
- Conjecture is proved by structural induction

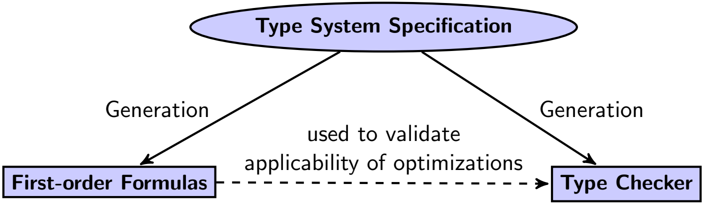
$$\forall a, b, c, d\,.\,(a \texttt{->} b = c \texttt{->} d) \implies (c <: a \wedge b <: d)$$

- One case of the induction proof

$$\begin{aligned}
(a_1\texttt{->}a_2)\texttt{->Int} = (c_1\texttt{->}c_2)\texttt{->Int} &\implies (c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \wedge \texttt{Int} <: \texttt{Int}) \\
((a_1\texttt{->}a_2) = (c_1\texttt{->}c_2) \wedge \texttt{Int} = \texttt{Int}) &\implies (c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \wedge \texttt{Int} <: \texttt{Int}) \\
(a_1\texttt{->}a_2) = (c_1\texttt{->}c_2) &\implies c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \\
(c_1\texttt{->}c_2) = (a_1\texttt{->}a_2) &\implies c_1\texttt{->}c_2 <: a_1\texttt{->}a_2 \\
((a_1 <: c_1) \wedge (c_2 <: a_2)) &\implies c_1\texttt{->}c_2 <: a_1\texttt{->}a_2
\end{aligned}$$

Implementation

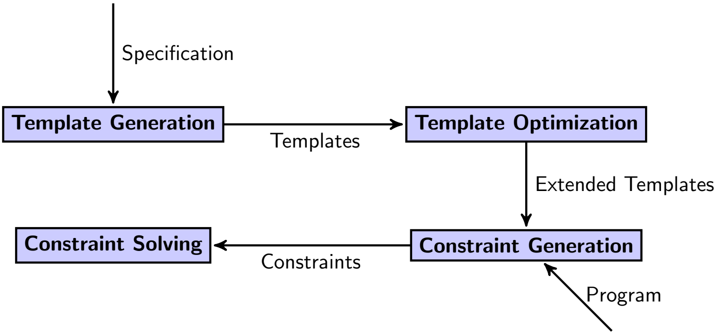
- From a type system specification we generate
  - first-order formulas
  - a type checker
- We use automated theorem proving to prove the conjectures seen in the optimization strategies
- Generation of a type checker form optimized specifications
- First-order formulas serve in combination with automated theorem proving as a reference implementation



Type Checker

The type checker has four phases

1. Translation of typing rules into normalized templates
2. Optimization of templates ✓
3. Generation of constraints according to an expression
4. Solving of the generated constraints



Templates

- Templates are an intermediate representation of the rules suitable for constraint generation
  - Resolved dependencies between premises
  - Resolved implicit equlities
  - Uniform structure
- Ambiguous rules are group into `Fork` constructors
- Templates are ordered such that the rule with the most general conclusion is applied last

Constraint Generation

- Simple constraint language consisting of
  - Equality
  - Inequality
  - Bottom / Fail
- Algorithm
  1. Find template whose conclusion matches the program fragment
  2. Update contexts
  3. Check if premises are satisfiable (Call 1. with correct terms)

     true Collect constraints

     false Use the next matching template, otherwise fail
  4. return collected constraints

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 & Future Work

- We contribute
  - a declarative, high-level specification language for type systems
  - a translation of specifications into first-order formulas
  - optimization strategies to reduce the need of backtracking
  - 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 specialized heuristics and proof strategies
  - apply our work to more realistic programming languages