Machine-checked typesafety proofs Semester Project Presentation

Samuel Grütter

EPFL

June 10, 2014

Overview

- Learn Twelf and Coq
- ▶ Learn how to represent a programming language in a proof language
 - Concrete syntax: Boilerplate code explosion
 - "Reversed De Bruijn indices": Not general enough
 - Locally nameless: Promising
- "Transitivity pushing" proof for DOT
 - ▶ In "reversed De Bruijn indices" representation
 - ► Contribution: Translated into locally nameless (in Coq)
 - Confirmed that it still works
 - Good basis for future work (substitution/small step easier)

Reversed De Bruijn indices

```
type T1 = \{ z => \}
    type A <: { a => type E; val e: a.E }
    type T <: { y =>
     val a: z.A
would be represented as
  type T1 = \{ (0) = > \}
    type A <: { (1) => type E; val e: (1).E }
    type T <: { (1) =>
     val a: (0).A
```

Reversed De Bruijn indices

```
type T1 = \{ z => \}
    type A <: { a => type E; val e: a.E }
    type T <: { y =>
     val a: z.A
would be represented as
  type T1 = \{ (0) = > \}
    type A <: { (1) => type E; val e: (1).E }
    type T <: { (1) =>
      val a: (0).A
environment = list of types
e.g. \Gamma = [T1, \{(1) = \forall E; \forall E \in (1).E\}]
```

Problem: Can only compare types defined at same depth

```
type T1 = { (0) =>
  type A <: { (1) =>
    type E; val e: (1).E
  }
  type T <: { (1) =>
    val a: { (2) =>
    type E; val e: (2).E
  }
  val a: (0).A
  }
}
```

```
To check T1 <: T2, need to check
{(1) => type E; val e: (1).E} <: {(2) => type E; val e: (2).E}
i.e. that
(1).E <: (2).E
```

Fix: Static local renaming

```
type T1 = \{ (0) = > \}
                                          type T2 = \{ (0) = > \}
    type A <: { (1) =>
                                            type T <: { (1) =>
       type E; val e: (1).E
                                               val a: { (1) =>
                                                 type E; val e: (1).E
    type T <: { (1) =>
      val a: (0).A
To check T1 <: T2, need to check
\{(1) \Rightarrow \text{type E}; \text{ val e: } (1).E\} <: \{(1) \Rightarrow \text{type E}; \text{ val e: } (1).E\}
i.e. that
(1).E <: (1).E (works)
```

Problem: Unwanted partitioning of types

Problem: We partition the bind types, can only compare if in same partition

Problem: Unwanted partitioning of types

Problem: We partition the bind types, can only compare if in same partition

```
type T1 = \{ z =>
                           type T2 = \{ z = \}
 type A <: { a1 =>
                             type A <: { a2 =>
   type E = Any
                                type E = Any
   type F
                                type F
   val f: a1.F
                                val f: a2.F
   val e: Anv
                                val b: { b3 =>
   val b: z.B
                                  type F; val f: b3.F; val e: a2.E; val a: z.A
 type B <: { b1 =>
                              type B <: { b2 =>
   type E = Any
                              type E = Any
   type F
                                type F
   val f: b1.F
                                val f: b2.F
   val e: Anv
                                val a: { a3 =>
   val a: z.A
                                  type F: val f: a3.F: val e: b2.E: val b: z.B
```

Would need a1 = a2 < b3 = b1 = b2 < a3 = a1 (contradiction).

Reversed De Bruijn indices

Conclusion

- ► There are terms which typecheck on paper, but not in this representation
- Not an adequate representation

Locally nameless

Concrete syntax

- Represent all variables by an identifier
- Need to reason a lot about name capture, substitution, alpha-equivalence

Locally nameless

- ▶ Tries to be smarter. Representation:
 - ► Free variables: as identifiers (string, integer, ...)
 - Bound variables: De Bruijn index
- ► Unique representation for closed terms ⇒ we always work up to alpha-equivalence
- Still close to proofs on paper
- Substitution: no variable capture issues

Subtyping transitivity

Subtyping transitivity:

$$\Gamma \vdash T_1 <: T_2 \quad \land \quad \Gamma \vdash T_2 <: T_3 \quad \Rightarrow \quad \Gamma \vdash T_1 <: T_3$$

Needed to prove inversion lemmas.

Difficulties

Mutual dependency between transitivity and narrowing

⇒ Proof by mutual induction?

But what to use as termination measure?

- Size of derivations? Might increase through narrowing
- ► Size of types? p.L not structurally bigger than its bounds
 - Different size measure?
 - Difficult...

Why not an explicit transitivity rule?

Consider INVERT-SUBTYPE-BIND:

$$\Gamma \, \vdash \, \{z \Rightarrow \overline{D_1}\} \mathrel{<:} \{z \Rightarrow \overline{D_2}\} \quad \Rightarrow \quad \Gamma, \big(z : \{z \Rightarrow \overline{D_1}\}\big) \, \vdash \, \overline{D_1} \mathrel{<:} \overline{D_2}$$

Proof:

Case 1): bind
$$\implies$$
 trivial

Case 1): bind
$$\Gamma$$
 \Longrightarrow trivial -

$$\Gamma, (z: \{z \Rightarrow \overline{D_1}\}) \vdash \overline{D_1} <: \overline{D_2}$$

$$\Gamma \vdash \{z \Rightarrow \overline{D_1}\} <: \{z \Rightarrow \overline{D_2}\}$$

$$\frac{\Gamma \vdash \{z \Rightarrow \overline{D_1}\} <: p.L \quad \overline{\Gamma \vdash p.L} <: \{z \Rightarrow \overline{D_2}\}}{\Gamma \vdash \{z \Rightarrow \overline{D_1}\} <: \{z \Rightarrow \overline{D_2}\}}$$

. . .

"Transitivity pushing" approach

Two subtyping modes:

- oktrans mode: Explicit transitivity rule allowed at top
- ▶ notrans mode: Not allowed at top, but allowed at deeper levels.

oktrans_to_notrans theorem:

► Can "push" oktrans at top into deeper levels

Transitivity in notrans mode without p.L in the middle

Lemma trans* (easy to prove):

$$\Gamma \vdash T_1 <:_n T_2 \land T_2 \neq p.L \land \Gamma \vdash T_2 <:_n T_3 \Rightarrow \Gamma \vdash T_1 <:_n T_3$$

Why not easy if $T_2 = p.L$?

- ▶ If p.L : S..U, we can get $T_1 <: S$, S <: U, $U <: T_3$
- ▶ Problems:
 - 1. What if S and U are again path types?
 - 2. All in oktrans mode

Observation

Two situations we don't like:

- 1. Path types in the middle: \dots <: p.L <: \dots
 - \longrightarrow extract bounds S..U
 - \longrightarrow replace by ... <: S <: U <: ...
- 2. Derivations with explicit transitivity at top: $\ldots <:_t \ldots$
 - --> unwrap premises and middle man
 - \longrightarrow replace by ... <: M <: ...

Problem:

► These fixes might produce new "situations we don't like"

$\mathsf{Idea} \ \mathsf{for} \ \mathsf{an} \ \mathsf{oktrans} \ \to \mathsf{notrans} \ \mathsf{algorithm}$

$$A \leq :_t Z$$

$$A \leq :_{t} Z$$

$$A \leq :_{t} p.L \leq :_{n} Z$$

$$A \leq :_{t} Z$$

$$A \leq :_{t} \underbrace{p.L} <:_{n} Z$$

$$A \leq :_{t} \underbrace{q.M} <:_{t} T <:_{n} Z$$

$$A \leq :_{t} Z$$

$$A \leq :_{t} p.L \leq :_{n} Z$$

$$A \leq :_{t} q.M \leq :_{t} T \leq :_{n} Z$$

$$A \leq :_{t} D \leq :_{n} q.M \leq :_{t} T \leq :_{n} Z$$

$$A \leq :_{t} Z$$

$$A \leq :_{t} p.L \leq :_{n} Z$$

$$A \leq :_{t} q.M \leq :_{t} T \leq :_{n} Z$$

$$A \leq :_{t} D \leq :_{n} q.M \leq :_{t} T \leq :_{n} Z$$

$$\vdots$$

$$A \leq :_{n} B \leq :_{n} C \leq :_{n} \ldots \leq :_{n} X \leq :_{n} Y \leq :_{n} Z$$

$\mathsf{Idea} \ \mathsf{for} \ \mathsf{an} \ \mathsf{oktrans} \ \to \mathsf{notrans} \ \mathsf{algorithm}$

$$A \leq :_{t} Z$$

$$A \leq :_{t} p.L \leq :_{n} Z$$

$$A \leq :_{t} q.M \leq :_{t} T \leq :_{n} Z$$

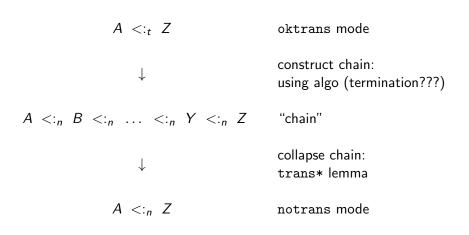
$$A \leq :_{t} D \leq :_{n} q.M \leq :_{t} T \leq :_{n} Z$$

$$\vdots$$

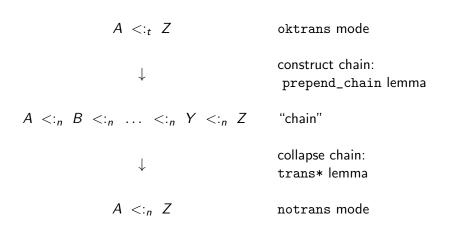
$$A \leq :_{n} B \leq :_{n} C \leq :_{n} \ldots \leq :_{n} X \leq :_{n} Y \leq :_{n} Z$$

- ▶ Collapse chain into $A <:_n Z$ by repeatedly applying trans* lemma
- But how to prove termination?

Strategy for proving oktrans_to_notrans theorem



Strategy for proving oktrans_to_notrans theorem



The prepend_chain lemma

$$A_1 <:_t A_2 \land (A_2 <:_n \ldots <:_n Z) \Rightarrow (A_1 <:_n \ldots <:_n Z)$$

Applying it with empty chain gives what we need:

$$A_1 <:_t A_2 \Rightarrow (A_1 <:_n \ldots <:_n A_2)$$

Proof:

- ▶ By induction on the size of the $(A_1 <:_t A_2)$ derivation
- ▶ If A_2 is not a path type: easy
- ▶ If A_2 is a path type: Get rid of it by following upper bound
 - → follow_ub data structure

Follow upper/lower bound

```
Follow upper bound: (follow_ub p1.X1 B) =  (p_1.X_1: \_..p_2.X_2), (p_2.X_2: \_..p_3.X_3), ..., (p_N.X_N: \_..B)
```

Follow upper/lower bound

$$(p_1.X_1: _..p_2.X_2), (p_2.X_2: _..p_3.X_3), ..., (p_N.X_N: _..B)$$

Follow lower bound (follow_lb C pN.XN) =

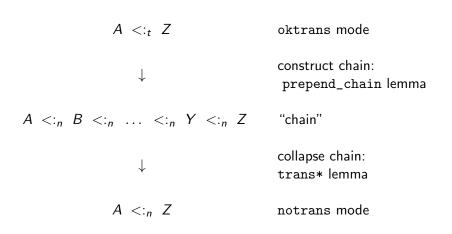
$$(p_1.X_1 : C...), (p_2.X_2 : p_1.X_1...), ..., (p_N.X_N : p_{N-1}.X_{N-1}...)$$

Final definition of chain

Final definition of chain

```
Definition st_middle (G: ctx) (B C: typ): Prop :=
  B = C \setminus /
  subtyp notrans G typ_top C \/
  (notsel B / subtyp notrans G B C).
Definition chain (G: ctx) (A D: typ): Prop :=
   (exists B C, follow_ub G A B /\
                 st_middle G B C /\
                 follow_lb G C D).
```

Strategy for proving oktrans_to_notrans theorem



"Transitivity pushing" approach: Summary

- Break mutual dependency between subtyping transitivity and narrowing
- ► Whenever bothered by explicit transitivity rule (e.g. in INVERT-SUBTYP-BIND), apply oktrans_to_notrans

Report

- Choosing the representation
 - Concrete syntax
 - Reversed De Bruijn indices
 - ► The locally nameless representation
- Choosing the tool
 - Automatic typechecking of sample terms
 - Termination checking of mutually inductive proofs
 - Expressiveness of predicates
 - The concept of total functions
 - The concept of equality
 - Automation
- DOT subtyping "transitivity pushing" proof

Future work

- Work towards full typesafety proof of DOT
- ► Locally nameless + Coq: Good basis
 - General enough to represent everything we can do on paper
 - ▶ Substitution is simple
 - Good for small-step
 - Nice library for mappings (environments/declaration sets)

Questions?

Thank you ☺

additional slides

Need subtyping transitivity for inversion lemmas

Example: INVERT-VAR:

$$\Gamma \vdash x : T \Rightarrow \exists T' (x, T') \in \Gamma \land \Gamma \vdash T' <: T$$

Proof, case subsumption:
$$\frac{\overline{\Gamma \vdash x : T'} \quad \overline{\Gamma \vdash T' <: T}}{\Gamma \vdash x : T}$$

$$\mathrm{IH}(\Gamma \vdash x : T') = \exists T'' \ (x, T'') \in \Gamma \ \land \ \Gamma \vdash T'' <: T'$$

Transitivity: T'' <: T' <: T