Typechecker generation via equational reasoning

Cameron Wong

CS-252R final project

2022-12-01

Driving question

- Can we systematically generate a typechecker from a collection of inference rules?

Typechecking, the problem statement

- Let Exp be an arbitrary expression language with typing judgment

$$\Gamma \vdash e : \tau$$

- Two obvious ways to define a "typechecking" function:

$$\texttt{check} : \texttt{Ctx} \to \texttt{Exp} \to \texttt{Ty} \to \texttt{Bool} \tag{1}$$

$$\texttt{synth} : \texttt{Ctx} \to \texttt{Exp} \to \texttt{Option Ty} \tag{2}$$

Typechecking, the problem statement

- What's our correctness theorem?
- "$\Gamma \vdash e : \tau$ if $\texttt{synth}\ \Gamma\ e \equiv \texttt{Some}\ \tau$"

Theorem Prover Background

- How to represent inference rules in the host language?
- In a proof assistant, typically done with an inductive family:

$$
\begin{array}{l}
\textbf{data}\ \_ \vdash \_ : \_\ :\ \texttt{Ctx} \to \texttt{Exp} \to \texttt{Ty} \to\ \texttt{Set} \\
\quad \texttt{typ-nat}\ :\ \forall\{\Gamma\ \ \texttt{n}\} \to \Gamma \vdash \texttt{Val}\ \ n : \texttt{Nat} \\
\quad \texttt{typ-add}\ :\ \forall\{\Gamma\ \ e_1\ \ e_2\} \to \\
\qquad \Gamma \vdash e_1 : \texttt{Nat} \to \Gamma \vdash e_1 : \texttt{Nat} \to\ \ \text{-- } \textit{premises} \\
\qquad \Gamma \vdash e_1 + e_2 : \texttt{Nat}\ \ \text{-- } \textit{conclusion}
\end{array}
$$

Typechecking, the problem statement, in Agda

- Correctness theorem, in Agda (some noise elided):

  $$\texttt{synth-correct : } \Gamma \vdash e : \tau \rightarrow \texttt{synth } \Gamma \; e \equiv \texttt{ Some } \tau$$

The Plan

- Use equational reasoning to derive `synth-correct` and `synth` simultaneously
- Advantage is that we can perform induction on the judgment $\Gamma \vdash e : \tau$ instead of just structurally on e

Source Language

$$\tau := \texttt{Nat} \mid \texttt{Bool} \mid \tau \to \tau$$
$$e := \mathit{true} \mid \mathit{false} \mid \overline{n} \mid e + e \mid \lambda(x : \tau).e$$

$$\frac{}{\Gamma \vdash \mathit{true} : \texttt{Bool}}\ \textsc{Typ-true} \qquad \frac{}{\Gamma \vdash \mathit{false} : \texttt{Bool}}\ \textsc{Typ-false}$$

$$\frac{}{\Gamma \vdash \overline{n} : \texttt{Nat}}\ \textsc{Typ-nat} \qquad \frac{\Gamma \vdash e_1 : \texttt{Nat} \qquad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 + e_2 : \texttt{Nat}}\ \textsc{Typ-add}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda(x : \tau).e : \tau \to \tau'}\ \textsc{Typ-abs}$$

The Derivation

- Proceed by induction on the derivation of $\Gamma \vdash e : \tau$

The Derivation

Case $\textsc{Typ-true}$:

- $e = \mathit{true}$, $\tau = \texttt{Bool}$
- Need proof of $\texttt{synth}\ \Gamma\ \mathit{true} \equiv \texttt{Some Bool}$
- Currently, no clause of $\texttt{synth}$ for *true*!
  - So define $\texttt{synth}\ \Gamma\ \mathit{true} = \texttt{Some Bool}$

The Derivation

Case $\textsc{Typ-true}$:

```
synth-correct Typ-true = begin
    synth Γ true
  ≡ ⟨Define clause: synth Γ true = Some Bool⟩
    Some Bool
  □
```

The Derivation

Case $\textsc{Typ-add}$:

- $e = e_1 + e_2$, $\tau = \texttt{Nat}$
- Have $\Gamma \vdash e_1 : \texttt{Nat}$, $\Gamma \vdash e_2 : \texttt{Nat}$
- Need a proof of $\texttt{synth}\ \Gamma\ (e_1 + e_2) \equiv \texttt{Nat}$
- Currently, no clause of `synth` for $e_1 + e_2$, so define $\texttt{synth}\ \Gamma\ (e_1 + e_2) = \texttt{Nat}$

The Derivation

Case TYP-ADD:

- Wait, what?

Now what?

- Need to forbid degenerate typecheckers!

Now what?

- Idea 1: Force `synth` to use all arguments
- This works, but not very principled and probably doesn't scale to a more complex system

Problem Statement, revisited

- Correctness theorem is insufficiently precise
- Need to talk about failure conditions explicitly
  - "$\Gamma \vdash e : \tau$ if <u>and only if</u> $\texttt{synth}\ \Gamma\ e \equiv \texttt{Some}\ \tau$"

Problem Statement, revisited

- How to state the "only if"?
- A few different formulations:

$$\texttt{synth-correct'} \ : \ \texttt{synth} \ \Gamma \ e \equiv \texttt{Some} \ \tau \to \Gamma \vdash e : \tau$$
$$\texttt{synth-correct'} \ : \ \neg(\Gamma \vdash e : \tau) \to \texttt{synth} \ \Gamma \ e \equiv \texttt{None}$$

- Neither actually work; instead need to explicitly define a judgment for "ill-typed":

$$\texttt{synth-correct'} \ : \ \Gamma \vdash e : \times \to \texttt{synth} \ \Gamma \ e \equiv \texttt{None}$$

Explicitly ill-typed expressions

$$\frac{\Gamma \vdash e_1 : \texttt{Bool}}{\Gamma \vdash e_1 + e_2 : \times} \qquad \frac{\Gamma \vdash e_2 : \texttt{Bool}}{\Gamma \vdash e_1 + e_2 : \times}$$

$$\frac{\Gamma \vdash e_1 : \times}{\Gamma \vdash e_1 + e_2 : \times} \qquad \frac{\Gamma \vdash e_2 : \times}{\Gamma \vdash e_1 + e_2 : \times} \qquad \frac{\Gamma, x : \tau \vdash e : \times}{\Gamma \vdash \lambda(x : \tau).e : \times}$$

Explicitly ill-typed expressions

- Terrible to construct by hand, but easy to do via automation (for decidable type systems)
- Rough algorithm to negate a single rule:
  - For each premise $\Gamma \vdash e : \tau$, generate a bad-type rule with premise $\Gamma \vdash e : \tau'$ for all $\tau' \neq \tau$
  - For each premise $\Gamma \vdash e : \tau$, generate a bad-type rule with premise $\Gamma \vdash e : \times$
  - Remove redundant rules

Back to the derivation

Case TYP-ADD:

$$\frac{\Gamma \vdash e_1 : \texttt{Nat} \qquad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 + e_2 : \texttt{Nat}}\ \textsc{Typ-add}$$

Back to the derivation

Case $\textsc{Typ-Add}$:

- Log premises $\Gamma \vdash e_1 : \texttt{Nat}$ and $\Gamma \vdash e_2 : \texttt{Nat}$ as $(\texttt{Nat}, \texttt{Nat}) \mapsto \texttt{Nat}$
- Collect all other rules for the $+$ syntax form

Back to the derivation

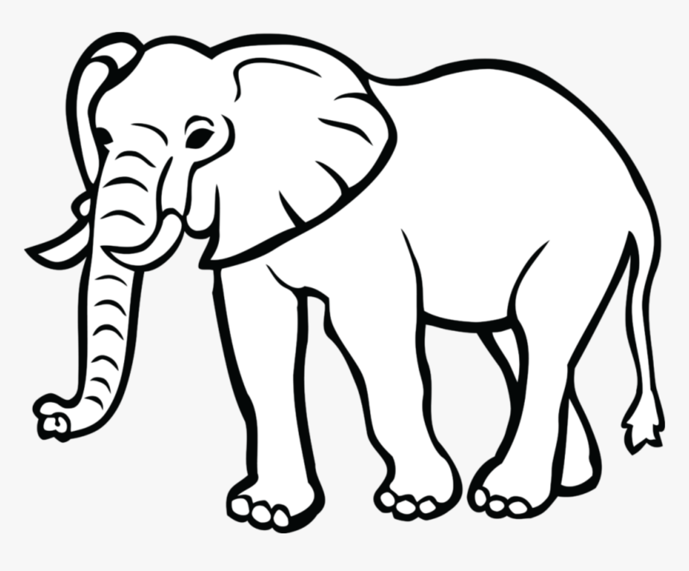
Case $\textsc{Typ-add}$:

$$
\begin{array}{l|c}
(\texttt{Nat}, \texttt{Nat}) & \texttt{Nat} \\
(\texttt{Bool}, \_) & \times \\
(\_, \texttt{Bool}) & \times \\
(\times, \_) & \times \\
(\_, \times) & \times
\end{array}
$$

Back to the derivation

Case TYP-ADD:

```
match (synth e₁, e₂) with
| (Some Nat, Some Nat) => Some Nat
| (Some Bool, _) => None
| (_, Some Bool) => None
| (None, _) => None
| (_, None) => None
```



Reflection

- Core idea is not particularly interesting
- Main challenges are engineering-based, not conceptual
  - How to represent generic syntax for rule input?
- Limitations are pretty severe:
  - Typing derivations must be unique
  - All variables in premises must appear in conclusion

Status

- Generated `synth` and proofs for STLC and other toy languages
  - (doesn't work well enough for a demo)
- TODO:
  - Make a frontend (rules are currently hand-entered)
  - Generalize to System F (two different type judgments)