Program Design with Typing Rules

Norman Ramsey

Spring 2019

1 Introduction

This handout explains how to turn a type system (or really any proof system) into code. You will use it for two assignments: type checking and type inference.

2 Overall program design

When we code up a type system or other proof system, every judgment form is implemented by a function. A judgment form is a logical statement, amounting to "a claim is provable." A function implementing a judgment form follows one of two models:

- Every metavariable in the judgment form is an input to the function, and the function returns a Boolean that answers the question, "is this claim provable?"

  Example: judgment $x \in \mathrm{dom}\,\Gamma$. Both x and Γ are inputs, so the function takes a name and an environment and returns a Boolean. This is the function `isBound` from the ML homework.

  Example: judgment $\tau_1 \equiv \tau_2$. Both τ_1 and τ_2 are inputs, so the function takes two types and returns a Boolean. This is the function `eqType` from the type-systems chapter.

- Some metavariables are inputs and some are outputs. The function tries to compute values for the output metavariables such that the whole judgment is provable. If it succeeds, it returns those values. If it fails, it raises an exception.

  Example: judgment $\Delta, \Gamma \vdash e : \tau$. The inputs are Δ, Γ, and e, and the output is τ. For example, using the environments from the initial basis, if the input expression is `(+ 2 2)`, the function will succeed and output type `int`; if the input expression is `(+ 2 #t)`, the function will fail by raising an exception.

  To identify functions like this, I frequently write a judgment form with boxes around the outputs, as in "$\Delta, \Gamma \vdash e : \boxed{\tau}$."

To implement a type checker, a type inferencer, another static analyzer, or even an interpreter, you implement all the judgment forms. Here are some key questions:

- What function will each judgment form be implemented by? What are the inputs and the outputs?
- What are the proof rules for the judgment forms I implement? What judgments do those proof rules use?
- Which judgment forms are already implemented for me? Perhaps in the book?
- Which judgment forms do I have to implement?

The answers are different for each different type system, but many of the answers are found in the book:

Typed Impcore	Table 6.1 on page 406
Typed μScheme	Table 6.3 on page 441
nano-ML	Table 7.3 on page 516

3 Design steps for one function

To design a function that implements a judgment form, we follow the usual design steps:

1. *Forms of data.* Key data types found in type systems are

Γ	Type environment
Δ	Kind environment
C	Equality constraint (inference only)
e	Abstract syntax
τ	Type
σ	Type scheme (Hindley-Milner only)
α	Type variable

   Like the judgment forms, these forms of data and their ML representations are shown in the tables on pages 406, 441, and 516.

   Not all of these data are broken down by cases:

   - Environments are never broken down by cases.
   - Types are not usually broken down by cases, *except* when implementing a constraint solver for type inference.

- Type schemes have only one form, but when instantiating polymorphic values, type schemes are deconstructed. (Instantiation is implemented in the book.)
- Equality constraints and abstract syntax, when consumed, are broken down by cases.

2. *Example inputs.* To write examples of abstract syntax, we use concrete syntax. (Writing abstract syntax is what concrete syntax is for!) When possible, we do the same with types, which also have a concrete syntax.

3. *Function name.* To form names, we usually use nouns like "type" or verbs like "elaborate," "evaluate," "substitute," "conjoin," or "solve." Some name connected with the proof system. Function names are often given by one of the tables on pages 406, 441, and 516.

4. *Function contract.* This is a key step. At an abstract level, all the contracts are the same: "implement a judgment." But it helps to be concrete. Example: judgment form $C, \Gamma \vdash e : \tau$ from nano-ML. Here's the function and its contract:

   ```
   val typeof : exp * type_env -> ty * con
   ```
   Calling `typeof` (e, Γ) returns (τ, C) such that $C, \Gamma \vdash e : \tau$.[1] Constraint C is not guaranteed to be solvable.

   To help us remember a function's contract, we can write the corresponding judgment form with boxes around the outputs: $\boxed{C}, \Gamma \vdash e : \boxed{\tau}$.

5. *Example results.* While it is possible to write example inputs and results directly as ML values, this level of work can usually be avoided. For most type-checking tasks, try this method:
   - Whenever possible, use the environments in initial basis.
   - When necessary, extend the initial environment with just one or two definitions.
   - Write example inputs (expressions and definitions) using the *concrete* syntax of Typed Impcore, Typed μScheme, or nano-ML. Write example outputs likewise.
   - Turn the examples into unit tests using the `check-type`, `check-principal-type`, and `check-type-error` forms.

   When implementing the proof system for the constraint solver, there is no concrete syntax for constraints. Fortunately, the ML syntax for writing constraints is not too painful. So to design the constraint solver, write example inputs and results as you learned to do when coding the ML warmup assignment.

6, 7, 8. *Algebraic laws and code.* Experienced type-system hackers can code a type system from inference rules alone. But while you are learning, you are better off following the step-by-step procedure outlined below.

9. *Revisit unit tests.* To test a type checker, use concrete syntax with unit-test forms in source code, as described in step 5 above. To test the nano-ML constraint solver, use `Unit` functions to embed unit tests inside your interpreter.

4 Translating rules to code

To implement a type system or other proof system, you collect all the rules with the same form of judgment in the conclusion, and you turn that collection into a function. So for example, if you consult the table 6.3 on page 441, you'll see that the Typed μScheme type system needs you to write two functions: `typeof` and `elabdef`. The corresponding forms of judgment are $\Delta, \Gamma \vdash e : \boxed{\tau}$ and $\langle d, \Delta, \Gamma \rangle \to \boxed{\Gamma'}$. You'll find the corresponding collections of rules summarized in Figures 6.9 and 6.10 on pages 471 and 472. (Not everything is in the summaries; in particular, the rules for literal values are found only at the beginning of section 6.6.5, which starts on page 436.)

Now you know the function you are implementing, its inputs and outputs, and the collection of rules that specifies the implementation. Your next step is to break the rules down by forms of data, which is to say the forms of the *abstract syntax* in the conclusions of the rules. For `typeof`, this is expression syntax; for `elabdef`, this is definition syntax. For each form of syntax, expect one or more rules. In our case, we're expecting *exactly* one rule per form of syntax—systems with more than rule are not hard to implement, but they are beyond the scope of this handout.

To finish the job, you translate each rule into a law, or possibly code. For generality, I recommend that *every* right-hand side (whether a law or direct to ML code) take the form of a `let` expression, with bindings and a body. The `let` expression should be

[1] The inputs and outputs to type inference are a frequent source of confusion. Examples of other type systems, as well as operational semantics, suggest the heuristic, "inputs on the left, outputs on the right." But that's not how logic works—the heuristic is wrong. The logical structure of a sequent is

$$\textit{context} \vdash \textit{claim}.$$

In type inference, one of the outputs of the algorithm is the constraint C. This constraint captures the assumptions that have to be made about the context in order for term e to be typable.

developed using the step-by-step procedure given below. I list all the steps, then repeat them in a full example.

(A) In your chosen rule, identify all the judgments above the line. Put them on a list of *unproved judgments*.

(B) In each unproved judgment, identify (1) what function implements the judgment and (2) which parts of the judgment are inputs and which parts are outputs. Draw a box around each output.

(C) Look at all the *variables* that appear in *output* positions in *unproved* judgments. If any variable appears in more than one such position, all those outputs must be equivalent. In the original rule and the list of unproved judgments, rename outputs until all output variables have unique names, and introduce equality or equivalence constraints.

If you are writing type inference, return the new constraints from the function. If you are writing a type checker, add equivalence judgments to your list of unproved judgments.

(D) Look at all the *literals* or *expressions* that appear in output positions in unproved judgments. In a type checker, these are likely to be types like `bool` or $\tau_1 \to \tau_2$. These also must be renamed, and equality or equivalence constraints must be introduced.

(E) Once every judgment above the line has only variables in output positions, and no variable appears in more than one output position, you are ready to write code to discharge the unproved judgments. You will need to keep track of the *available metavariables*. These include any inputs to the judgment below the line (and their parts), plus any metavariables introduced in the `let` bindings (of which you don't have any yet).

As long as there is an unproved judgment, repeat the following step: find an unproved judgment whose *inputs* are all available. Now implement that judgment in one of two ways:

- If the judgment has any outputs, add a `let` binding that calls the function:

  `val` *output-pattern* `=` *f inputs*

- If the judgment has *no* outputs, add a trivial binding that confirms the judgment is provable, and if not, raises an exception.[2]

```
val () =
  if f inputs then
    ()
  else
    raise TypeError message
```

Once the judgment is implemented:

- Cross it off the list of unproved judgments.
- Note that its outputs, if any, are now available.

Continue repeating this step until there are no more unproved judgments.

(F) Once all the unproved judgments have been dealt with, return to the judgment below the line. Look at the outputs. Every output should either be an available variable or should be formed from available variables. Use these variables to make the output, and place it in the body of the `let`.

Complete example

Let's demonstrate the process with the IF rule:

$$\frac{\Delta, \Gamma \vdash e_1 : \texttt{bool} \qquad \Delta, \Gamma \vdash e_2 : \tau \qquad \Delta, \Gamma \vdash e_3 : \tau}{\Delta, \Gamma \vdash \textsc{If}(e_1, e_2, e_3) : \tau}$$

(A) We have these unproved judgments:

$$\Delta, \Gamma \vdash e_1 : \boxed{\texttt{bool}}$$
$$\Delta, \Gamma \vdash e_2 : \boxed{\tau}$$
$$\Delta, \Gamma \vdash e_3 : \boxed{\tau}$$

(B) Each of the unproved judgments is implemented by `typeof`. The inputs are Δ, Γ, e_1, e_2, and e_3, and the outputs are `bool` and τ.

(C) The variable τ appears in two output positions in unproved judgments. I rename the second position τ_3, and I introduce the type-equivalence judgement $\tau \equiv \tau_3$.

My original rule now looks like this:

$$\frac{\Delta, \Gamma \vdash e_1 : \texttt{bool} \qquad \begin{array}{c}\Delta, \Gamma \vdash e_2 : \tau \qquad \Delta, \Gamma \vdash e_3 : \tau_3 \\ \tau \equiv \tau_3\end{array}}{\Delta, \Gamma \vdash \textsc{If}(e_1, e_2, e_3) : \tau}$$

and the unproved judgements look like this:

$$\Delta, \Gamma \vdash e_1 : \boxed{\texttt{bool}}$$
$$\Delta, \Gamma \vdash e_2 : \boxed{\tau}$$
$$\Delta, \Gamma \vdash e_3 : \boxed{\tau_3}$$
$$\tau \equiv \tau_3$$

[2] If your proof system has more than one rule per syntactic form, as in an interpreter for an operational semantics, for example, you would not raise an exception. Instead, you would try the next rule. Once you've tried all possible rules, *then* you raise the exception.

(D) Next, I spot `bool` in an output position. I rewrite it to τ_1. My original rule now looks like this:

$$\frac{\Delta,\Gamma \vdash e_1 : \tau_1 \qquad \Delta,\Gamma \vdash e_2 : \tau \qquad \Delta,\Gamma \vdash e_3 : \tau_3 \qquad \tau \equiv \tau_3 \qquad \tau_1 \equiv \texttt{bool}}{\Delta,\Gamma \vdash \textsc{If}(e_1, e_2, e_3) : \tau}$$

and the unproved judgements look like this:

$$\Delta,\Gamma \vdash e_1 : \boxed{\tau_1}$$
$$\Delta,\Gamma \vdash e_2 : \boxed{\tau}$$
$$\Delta,\Gamma \vdash e_3 : \boxed{\tau_3}$$
$$\tau \equiv \tau_3$$
$$\tau_1 \equiv \texttt{bool}$$

(E) All of my unproved judgments have only variables in output positions (τ_1, τ, and τ_3), and no variable appears in more than one output position. I'm ready to start discharging them.

The type system has only one rule for If, so I'm going to code it directly in one clause of a clausal definition for `typeof`. That clause begins something like this:

```
fun typeof (IFX (e1, e2, e3), Delta, Gamma) = ...
```

And my available variables are Δ, Γ, e_1, e_2, and e_3. Time to start proving judgments. Variables are available for any of the first three unproved judgments. I start with the first.

- Judgment $\Delta,\Gamma \vdash e_1 : \boxed{\tau_1}$ has output τ_1, so I add this definition form to my `let` bindings:

  ```
  val tau_1 = typeof (e1, Delta, Gamma)
  ```

  This binding adds τ_1 to my list of available variables.

- Now that τ_1 is available, I can discharge the unproved judgment $\tau_1 \equiv$ `bool`. This judgment doesn't have any outputs, so I have to test it to see if it is provable. From Table 6.3, the corresponding function is `eqType`. I add this definition form to my `let` bindings:

  ```
  val () =
    if eqType (tau_1, booltype) then
      ()
    else
      raise TypeError "...message..."
  ```

- I now discharge the unproved judgment $\Delta,\Gamma \vdash e_2 : \boxed{\tau}$ with this binding:

  ```
  val tau = typeof (e2, Delta, Gamma)
  ```

- And I discharge the unproved judgment $\Delta,\Gamma \vdash e_3 : \boxed{\tau_3}$ with this binding:

  ```
  val tau_3 = typeof (e3, Delta, Gamma)
  ```

- With variables τ and τ_3 finally available, I can discharge the last unproved judgment, $\tau \equiv \tau_3$:

  ```
  val () =
    if eqType (tau, tau_3) then
      ()
    else
      raise TypeError "...message..."
  ```

- Finally, I return to the judgment below the line, $\Delta,\Gamma \vdash \textsc{If}(e_1, e_2, e_3) : \boxed{\tau}$. The output is τ, so `tau` is what I return in the body of the `let`.

With all steps complete, here's my full implementation of the If rule:

```
fun typeof (IFX (e1, e2, e3), Delta, Gamma) =
  let val tau_1 = typeof (e1, Delta, Gamma)
      val () =
        if eqType (tau_1, booltype) then
          ()
        else
          raise TypeError "...message..."
      val tau   = typeof (e2, Delta, Gamma)
      val tau_3 = typeof (e3, Delta, Gamma)
      val () =
        if eqType (tau, tau_3) then
          ()
        else
          raise TypeError "...message..."
  in  tau
  end
```

There are more compact and more idiomatic ways to write this code, which I encourage you to try, but the step-by-step procedure works every time.

Summary of the steps

You'll repeat these steps for every rule, so it's worth having a short summary:

(A) List unproved judgments.
(B) Know function names; box the outputs.
(C) Rename duplicated output variables.
(D) Introduce variables for output literals.
(E) Discharge each unproved judgment:
  - Find a judgment whose inputs are available.
  - If it has outputs, `val` bind them.
  - It if has no outputs, confirm it returns `true` or raise an exception.
(F) Return the outputs from the rule's conclusion (the judgment below the line).