Lecture Notes on
Rules as Algorithms

15-317: Constructive Logic
Frank Pfenning

Lecture 6
Thursday, February 2, 2023

1 Introduction

So far, we have used inference rules mainly to *define* fundamental notions in logic, such as the notion of (intuitionistic) truth A *true* or the notion of verification $A\uparrow$. There is another very important role for inference rules, namely to communicate *algorithms* at a very high level of abstraction. There turns out to be more than one way to interpret inference rules as defining certain algorithms, something we will study this further later in the course. One particular way is to think of the rules as describing how one might construct a deduction bottom-up.

An important application of this point of view is bidirectional type-checking, which arises in an entirely principled way from the notion of verification. But how does it actually relate to an algorithm for type-checking? This is the subject of today's lecture. But let's first look at our basic judgments and determine what kind of questions one might ask about them.

- $M : A$ where A is given. This is *theorem proving*. In other words, we have to find a constructive proof of
$$\forall A.\, (\exists M.\, M : A) \vee \neg(\exists M.\, M : A)$$
Here, we use the judgment $M : A$ actually as a proposition in our mathematical metalanguage in which we talk about properties of judgments such as local soundness or completeness. In classical mathematics, this statement is trivial; in intuitionistic mathematics a proof would contain an algorithm that decides whether an arbitrary proposition A has a proof (and returns that) or whether there is no proof (and gives evidence for that). An extracted function might have type

```
decide_true : prop -> term option
```

where `prop` is the type of propositions and `term` is the type of proof terms. We use the τ `option` type to return `SOME(M)` for M represents a proof of A, and `NONE` if there is no such proof. Recall that in SML, we define `datatype 'a option = NONE | SOME of 'a`.

- $M : A$ where M and A are given. This is *proof checking*. In other words, we have to find a constructive proof of

  $$\forall M.\, \forall A.\, (M : A) \vee \neg(M : A)$$

  A function corresponding to such a constructive proof might have type

  ```
  type_check : term -> prop -> bool
  ```

  Here we imagine we return `true` if $M : A$ and `false` otherwise. The type `bool` just stands in for a type with two possible values and is a convenient choice, just as the option type for the theorem proving problem.

- $M : A$ where M is given. This is *type inference*. In other words, we have to find a constructive proof of

  $$\forall M.\, (\exists A.\, M : A) \vee \neg(\exists A.\, M : A)$$

  An extracted function might have type

  ```
  type_infer : term -> prop option
  ```

  which given a term returns `SOME(A)` for a proposition A (viewed here as a type) or `NONE` if no proposition with $M : A$ exists.

In all these examples, the metatheoretic proofs are quite difficult. Having inference rules for the judgment $M : A$ is of course critical, but they don't immediately give rise to any such algorithms.

2 An Algorithm for Bidirectional Typing

The rules for bidirectional typing have a particular goal-directed form that allows us to read them "algorithmically". Please refer to the rules in Figure 1.

We see that there are two judgments, $M \Leftarrow A$ and $M \Rightarrow A$. The notation is suggestive of the following interpretation:

1. $M \Leftarrow A$: both M and A are given, and we *check* M against the type A. This is one direction.

2. $M \Rightarrow A$: only M is given and we synthesize a type A (if such a type exists). This is the other direction.

Note that we move freely between the notion of a type and a proposition, since we already know that they correspond to each other.

Whether this interpretation of the judgments is correct will emerge during the process of turning the rules of the judgment into an implementation.

Given the interpretation, we are trying to construct the computational interpretation of a constructive proof of the following statements

1. $\forall M.\, \forall A.\, (M \Leftarrow A) \vee \neg(M \Leftarrow A)$

$$\frac{M \Rightarrow P' \quad P' = P}{M \Leftarrow P}\ {\downarrow}{\uparrow}^*$$

$$\frac{M \Leftarrow A \quad N \Leftarrow B}{\langle M, N\rangle \Leftarrow A \wedge B}\ \wedge I \qquad \frac{M \Rightarrow A \wedge B}{\mathbf{fst}\, M \Rightarrow A}\ \wedge E_1 \quad \frac{M \Rightarrow A \wedge B}{\mathbf{snd}\, M \Rightarrow B}\ \wedge E_2$$

$$\frac{\begin{array}{c}\overline{x \Rightarrow A}\ x \\ \vdots \\ M \Leftarrow B\end{array}}{\lambda x.\, M \Leftarrow A \supset B}\ \supset I^x \qquad \frac{M \Rightarrow A \supset B \quad N \Leftarrow A}{M\, N \Rightarrow B}\ \supset E$$

$$\frac{M \Leftarrow A}{\mathbf{inl}\, M \Leftarrow A \vee B}\ \vee I_1 \quad \frac{M \Leftarrow B}{\mathbf{inr}\, M \Leftarrow A \vee B}\ \vee I_2 \qquad \frac{M \Rightarrow A \vee B \quad \begin{array}{c}\overline{u \Rightarrow A}\ u \\ \vdots \\ N \Leftarrow C\end{array} \quad \begin{array}{c}\overline{w \Rightarrow B}\ w \\ \vdots \\ P \Leftarrow C\end{array}}{\mathbf{case}(M, u.\, N, w.\, P) \Leftarrow C}\ \vee E^{u,w}$$

$$\frac{}{\langle\,\rangle \Leftarrow \top}\ \top I \qquad (\text{no } \top E)$$

$$(\text{no } \bot I) \qquad \frac{M \Rightarrow \bot}{\mathbf{abort}\, M \Leftarrow C}\ \bot E$$

Figure 1: Rules for Bidirectional Typing

2. $\forall M.\, (\exists A.\, M \Rightarrow A) \vee \neg\exists A.\, M \Rightarrow A$

It turns out that these are not quite sufficient for our purpose, but pretty close. We'll come back to it in a future lecture.

From this we can extract two functions:

```
check : term -> prop -> bool
synth : term -> prop option
```

We write them directly, interpreting the rules, rather than writing out a (metatheoretic) proof.

As a general development methodology, I have found it best to get a fragment fully working and then extend it. Here, a natural way to break it down is connective by connective, which is especially appropriate since we constructed the introduction and elimination rules for each connective independently.

Conjunction. Let's define the datatypes of propositions and terms for conjunction first.

```
1  datatype prop =
2           And of prop * prop
3
4  datatype term =
5           Pair of term * term (* <M,N> *)
6         | Fst of term         (* fst M *)
7         | Snd of term         (* snd M *)
```

There is only one rule for checking a pair, and one rule each for synthesizing the type of a projection.

$$\frac{M \Leftarrow A \quad N \Leftarrow B}{\langle M, N\rangle \Leftarrow A \wedge B}\ \wedge I$$

$$\frac{M \Leftarrow A \quad N \Leftarrow B}{\langle M, N\rangle \Leftarrow A \wedge B}\ \wedge I \qquad \frac{M \Rightarrow A \wedge B}{\mathbf{fst}\, M \Rightarrow A}\ \wedge E_1 \qquad \frac{M \Rightarrow A \wedge B}{\mathbf{snd}\, M \Rightarrow B}\ \wedge E_2$$

These translate directly into the following code fragments

```
1  (* check : term -> prop -> bool *)
2  (* synth : term -> prop option *)
3  fun check (Pair(M,N)) (And(A,B)) = check M A andalso check N B
4    | check M A = false
5
6  and synth (Fst(M)) =
7      (case synth M
8        of SOME(And(A,B)) => SOME(A)
9         | _ => NONE)
10   | synth (Snd(M)) =
11     (case synth M
12       of SOME(And(A,B)) => SOME(B)
13        | _ => NONE)
14   | synth M = NONE
```

In the code we take advantage of the fact that pattern matching in ML proceeds sequentially so that, for example, the case `check M` applies when either the first argument is not a pair, or the second is not a conjunction, or both. Similarly, on the current language fragment we can synthesize only types for the first and second projection.

We cannot yet try this out, because there are no base cases, for example, in the definition of `prop`. Let's fix this situation so we can try a simple example.

Truth. There is only an introduction rule and no elimination rule.

$$\frac{}{\langle\rangle \Leftarrow \top}\ \top I$$

This means we add just a new proposition `True`, a new term `Unit`, and one case in the `check` function.

```
1  (* check : term -> prop -> bool *)
2  (* synth : term -> prop option *)
3  fun check (Pair(M,N)) (And(A,B)) = check M A andalso check N B
4    | check (Unit) (True) = true
5    | check M A = false
6
7  and synth (Fst(M)) =
8      (case synth M
9        of SOME(And(A,B)) => SOME(A)
10        | _ => NONE)
11   | synth (Snd(M)) =
12     (case synth M
13       of SOME(And(A,B)) => SOME(B)
14        | _ => NONE)
15   | synth M = NONE
```

Now we can ask if $\langle\langle\rangle, \langle\rangle\rangle \Leftarrow \top \wedge \top$, which should certain be the case. We should also have negative examples, lest a type-checker which always succeeds passes all of our tests. Here, we want to check that $\langle\rangle \not\Leftarrow \top \wedge \top$. We arrange it so an exception is raised if any of our examples fail.

```
1  fun assert true = ()
2    | assert false = raise Fail "assert"
3  fun deny true = raise Fail "deny"
4    | deny false = ()
5
6  val () = assert (check (Pair(Unit,Unit)) (And(True,True)))
7  val () = deny (check Unit (And(True,True)))
```

Fortunately, they passed!

Bridge from Synthesis to Verification. The rule

$$\frac{M \Rightarrow P' \quad P' = P}{M \Leftarrow P}\ \downarrow\uparrow^*$$

provides the bridge from synthesis to verification, where we restrict P to be a propositional variable rather than an arbitrary proposition. In order to capture this, we need variables in our syntax, and a case for checking against variables, which we identify by their names (as a string). The language of terms does not change here, nor does the synthesis function.

```
1  datatype prop =
2           And of prop * prop   (* A /\ B *)
3         | True                 (* T *)
4         | Var of string        (* P *)
5
6  datatype term =
7           Pair of term * term (* <M,N> *)
8         | Fst of term          (* fst M *)
9         | Snd of term          (* snd M *)
10        | Unit                 (* <> *)
11
12 (* check : term -> prop -> bool *)
13 (* synth : term -> prop option *)
14 fun check (Pair(M,N)) (And(A,B)) = check M A andalso check N B
15   | check (Unit) (True) = true
16   | check M (Var(P)) =
17     (case synth M
18       of SOME(Var(P')) => P' = P
19        | _ => false)
20   | check M A = false
21
22 and synth (Fst(M)) =
23     (case synth M
24       of SOME(And(A,B)) => SOME(A)
25        | _ => NONE)
26   | synth (Snd(M)) =
27     (case synth M
28       of SOME(And(A,B)) => SOME(B)
29        | _ => NONE)
30   | synth M = NONE
```

3 Implication.

Now we come to a somewhat mind-boggling part. The proof term for implication introduction is λ-abstraction, which introduces a new hypothesis.

$$\dfrac{\begin{array}{c}\overline{x \Rightarrow A}\ x \\ \vdots \\ M \Leftarrow B\end{array}}{\lambda x.\, M \Leftarrow A \supset B}\ \supset I^x \qquad \dfrac{M \Rightarrow A \supset B \quad N \Leftarrow A}{M\ N \Rightarrow B}\ \supset E$$

The question is how do we represent the new variable x and its scope (which is the term M)? Furthermore, how do we represent the hypothesis $x \Rightarrow A$ that is introduced into the deduction?

The key idea is to represent the scope of the λ constructor as a function in ML, an idea that has been called *higher-order abstract syntax*. Generally, whenever a new variable (= new hypothesis) is introduced into a derivation, we use an ML function to represent this scope. This also means that the bound variable x is represented by an ML variable of the same name. Application $M\ N$ is simpler because no binding or scope is involved.

```
datatype prop =
         And of prop * prop
       | Imp of prop * prop
       | True
       | Var of string

datatype term =
         Pair of term * term (* <M,N> *)
       | Fst of term         (* fst M *)
       | Snd of term         (* snd M *)
       | Lam of term -> term (* \x. M *)
       | App of term * term  (* M N *)
       | Unit                (* <> *)
```

Notice the type of the constructor `Lam : (term -> term) -> term`. This is unlikely to be familiar, let's go through some examples of what we might expect for the representation.

Blackboard	ML value of type `term`
$\lambda x.\, x$	`Lam(fn x => x)`
$\lambda x.\, \lambda y.\, x$	`Lam(fn x => Lam(fn y => x))`
$\lambda x.\, \langle \mathbf{snd}\, x, \mathbf{fst}\, x \rangle$	`Lam(fn x => Pair(Snd(x),Fst(x)))`
$\lambda x.\, (\mathbf{snd}\, x)\, (\mathbf{fst}\, x)$	`Lam(fn x => App(Snd(x),Fst(x)))`

Now that we understand the representation of terms, how do we proceed with the type-checking? We get the following situation:

```
(* check : term -> prop -> bool *)
(* synth : term -> prop option *)
fun check (Pair(M,N)) (And(A,B)) = check M A andalso check N B
  | check (Lam(F)) (Imp(A,B)) = check (F ???) B
  ...
```

In the case for `Lam`, we could like to check the body of the λ-abstraction against proposition B. In order to access the body, we need to apply `F : term -> term` with some term of type A, but we don't have any such term. Intuitively, this term should represent the hypothesis $x \Rightarrow A$, although the name of the variable x is irrelevant for this purpose. So we create a new kind of term `Hyp(A)` representing a hypothesis of type A (eliding the name since it doesn't matter). We then plug in `Hyp(A)` for the bound variable x, which we accomplish by function application in ML.

```
(* check : term -> prop -> bool *)
(* synth : term -> prop option *)
fun check (Pair(M,N)) (And(A,B)) = check M A andalso check N B
  | check (Lam(F)) (Imp(A,B)) = check (F (Hyp(A))) B
  ...
```

To incorporate the fact that a hypothesis $x \Rightarrow A$ should synthesize type A, we add a case for hypothesis into the `synth` function (eliding the cases already present for the first and second projections.

```
1  and synth (Hyp(A)) = SOME(A)
2    | synth (Fst(M)) = ...
3    | synth (Snd(M)) = ...
4    | synth M = NONE
```

It remains to add a case for implication elimination, that is, application of proof terms. Recall:

$$\frac{M \Rightarrow A \supset B \quad N \Leftarrow A}{M\,N \Rightarrow B}\,\supset E$$

Taking heed of the direction of checking and synthesis we proceed in the following manner: first we synthesize the type of M, which must be of the form $A \supset B$. From this we extract A and now check the argument N against A. If that returns true, we can return B as the synthesized type. If either of the steps fails, we return `NONE`. We transcribe this into the following code:

```
1  and synth (Hyp(A)) = SOME(A)
2    | synth (Fst(M)) = ...
3    | synth (Snd(M)) = ...
4    | synth (App(M,N)) =
5      (case synth M
6        of SOME(Imp(A,B)) => if check N A
7                             then SOME(B)
8                             else NONE
9         | _ => NONE)
10   | synth M = NONE
```

Now we can try some slightly more challenging example, such as that the proof terms for $A \supset A$, $(A \wedge B) \supset A$, and $A \supset (B \supset A)$, and also that $\lambda x.\, x \not\Leftarrow A \supset B$.

```
1  val A = Var("A")
2  val B = Var("B")
3  val C = Var("C")
4
5  val () = assert (check (Lam(fn x => x)) (Imp(A,A)))
6  val () = deny (check (Lam(fn x => x)) (Imp(A,B)))
7  val () = assert (check (Lam(fn x => Fst(x))) (Imp(And(A,B),A)))
8  val () = assert (check (Lam(fn x => Lam(fn y => x))) (Imp(A,Imp(B,A))))
```

4 Disjunction

For disjunction, we just have to remember that the elimination form (a) introduces two hypotheses so that the proof term has two bound variables, and (b) the conclusion is a verification. We also throw in falsehood (which is simpler) so we reach our complete language

more quickly.

$$\frac{M \Leftarrow A}{\mathbf{inl}\, M \Leftarrow A \vee B}\ \vee I_1 \qquad \frac{M \Leftarrow B}{\mathbf{inr}\, M \Leftarrow A \vee B}\ \vee I_2 \qquad \frac{M \Rightarrow A \vee B \quad \begin{array}{c}\overline{u \Rightarrow A}^{\,u} \\ \vdots \\ N \Leftarrow C\end{array} \quad \begin{array}{c}\overline{w \Rightarrow B}^{\,w} \\ \vdots \\ P \Leftarrow C\end{array}}{\mathbf{case}(M, u.\, N, w.\, P) \Leftarrow C}\ \vee E^{u,w}$$

$$\text{(no } \bot I\text{)} \qquad \frac{M \Rightarrow \bot}{\mathbf{abort}\, M \Leftarrow C}\ \bot E$$

We extend the propositions and proof terms accordingly

```
1  datatype prop =
2           And of prop * prop
3         | Imp of prop * prop
4         | Or of prop * prop
5         | True
6         | False
7         | Var of string
8
9  fun Not A = Imp(A,False)
10
11 datatype term =
12          Pair of term * term (* <M,N> *)
13        | Fst of term         (* fst M *)
14        | Snd of term         (* snd M *)
15        | Lam of term -> term (* \x. M *)
16        | App of term * term  (* M N *)
17        | Inl of term         (* inl M *)
18        | Inr of term         (* inr M *)
19        | Case of term * (term -> term) * (term -> term)
20                              (* case (M, u.N, w.P) *)
21        | Unit                (* <> *)
22        | Abort of term       (* abort M *)
23        | Hyp of prop         (* _ : A *)
```

We notice two things: (1) we throw in negation as a *notational definition*, which is just a definition in ML, and (2) we see how the two branches in a case expressions are represented by ML functions. Fortunately, we do not need more than that. Note that the `Case` constructor appears in the `check` function because the rule requires a verification of C. Also, in the two branches we instantiate it with the hypothesis A and B respectively.

```
1  (* check : term -> prop -> bool *)
2  (* synth : term -> prop option *)
3  fun check (Pair(M,N)) (And(A,B)) = check M A andalso check N B
4    | check (Lam(F)) (Imp(A,B)) = check (F (Hyp A)) B
5    | check (Inl(M)) (Or(A,B)) = check M A
6    | check (Inr(M)) (Or(A,B)) = check M B
7    | check (Case(M,F,G)) C =
8      (case synth M
9        of SOME(Or(A,B)) => check (F (Hyp A)) C
10                           andalso check (G (Hyp B)) C
11       | _ => false)
12   | check (Abort(M)) C =
13     (case synth M
14       of SOME(False) => true
15        | _ => false)
16   | check (Unit) (True) = true
17   | check M (Var(P)) =
18     (case synth M
19       of SOME(Var(P')) => P' = P
20        | _ => false)
21   | check M A = false
22
23 and synth (Hyp(A)) = SOME(A)
24   | synth (Fst(M)) =
25     (case synth M
26       of SOME(And(A,B)) => SOME(A)
27        | _ => NONE)
28   | synth (Snd(M)) =
29     (case synth M
30       of SOME(And(A,B)) => SOME(B)
31        | _ => NONE)
32   | synth (App(M,N)) =
33     (case synth M
34       of SOME(Imp(A,B)) => if check N A
35                            then SOME(B)
36                            else NONE
37        | _ => NONE)
38   | synth M = NONE
```

The final set of examples points out something interesting: When moving from $M \Rightarrow \bot$ (representing $\bot\downarrow$) to $M \Leftarrow \bot$ (representing $\bot\uparrow$) we have to use the $\bot E$ rule, rather than silently moving between the judgments. That's because $\bot$ is not a propositional variable, even though it may appear atomic in the sense that it cannot be broken down.

```
1  fun assert true = ()
2    | assert false = raise Fail "assert"
3  fun deny true = raise Fail "deny"
4    | deny false = ()
5
6  val A = Var("A")
7  val B = Var("B")
8  val C = Var("C")
9
10 val () = assert (check (Pair(Unit,Unit)) (And(True,True)))
11 val () = deny (check Unit (And(True,True)))
12 val () = assert (check (Lam(fn x => x)) (Imp(A,A)))
13 val () = deny (check (Lam(fn x => x)) (Imp(A,B)))
14 val () = assert (check (Lam(fn x => Fst(x))) (Imp(And(A,B),A)))
15 val () = assert (check (Lam(fn x => Lam(fn y => x))) (Imp(A,Imp(B,A))))
16 val () = assert (check (Lam(fn x => Case(x, fn u => Inr u,
17                                             fn w => Inl w)))
18                       (Imp(Or(A,B),Or(B,A))))
19 val () = deny (check (Lam(fn x => Case(x, fn u => Inr u,
20                                           fn w => Inl w)))
21                      (Imp(Or(A,B),Or(A,B))))
22 val () = assert (check (Lam(fn x => Abort(App(Snd(x),Fst(x)))))
23                       (Not(And(A,Not(A)))))
24 val () = deny (check (Lam(fn x => Abort(App(Fst(x),Snd(x)))))
25                      (Not(And(A,Not(A)))))
```

The last positive example is a verification that $\neg(A \wedge \neg A)$, written mathematically as $\lambda x.\, \mathbf{abort}\,((\mathbf{snd}\,x)\,(\mathbf{fst}\,x))$.

References