COMSM0067: Advanced Topics in Programming Languages

Judgements

Alex Kavvos

Reading: PFPL, §2.1–2.3

1 Judgements and evidence

A **judgement** is a statement (proposition, utterance, enunciation).

An **evident judgement** is a statement for which I have evidence (proof).

For example, I can assert that 2 is a natural number: $2\ \mathsf{nat}$.

This is a judgement. It is not *evident* yet, as we do not know how to prove it.

2 Rules

To prove a judgement we have to first elaborate what we accept as evidence for it.

For example, we can write down the following **rules** for proving that something is a natural number:

$$\textsc{Zero}\ \frac{}{\mathsf{zero}\ \mathsf{nat}} \qquad\qquad \textsc{Succ}\ \frac{n\ \mathsf{nat}}{\mathsf{succ}(n)\ \mathsf{nat}}$$

Judgements above the line are called **premises**; below the line, they are called **conclusions**.

Rules with no premises are called **axioms**.

Such rules can be assembled into **derivations** that prove judgements. For example, we can define

$$2 \stackrel{\text{def}}{=} \mathsf{succ}(\mathsf{succ}(\mathsf{zero}))$$

and then prove that it is a natural number:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{zero}\ \mathsf{nat}}}{\mathsf{succ}(\mathsf{zero})\ \mathsf{nat}}}{\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))\ \mathsf{nat}}$$

Listing such rules tactily implies that anything they do not prove is not an evident judgement.

For example, the rules for natural numbers state that

- 0 is a natural number
- if n is a natural number then so is $n + 1$
- *nothing else is a natural number*

This circumscribes the notion of natural number and what constitutes evidence for its construction.

3 Simultaneous generation of judgements

When stating proof rules, judgements may be mixed and matched to generate more complicated evidence.

For example, the following rules generate judgements on the parity of natural numbers.

$$\textsc{EvenZ}\ \frac{}{\mathsf{zero}\ \mathsf{even}} \qquad \textsc{Odd}\ \frac{n\ \mathsf{even}}{\mathsf{succ}(n)\ \mathsf{odd}} \qquad \textsc{Even}\ \frac{n\ \mathsf{odd}}{\mathsf{succ}(n)\ \mathsf{even}}$$

4 Derivable and admissible rules

Claim 1. If $\mathsf{succ}(n)\ \mathsf{nat}$ then $n\ \mathsf{nat}$.

Statements like Claim 1 are often written as rules themselves: $\dfrac{\mathsf{succ}(n)\ \mathsf{nat}}{n\ \mathsf{nat}}$

This rule is not one of the defining rules of natural numbers. Rather, it is an admissible rule.

A rule is **admissible** if whenever we have a derivation of the premises, then we know we can construct a derivation of the conclusion.

In this particular instance, given a derivation of $\mathsf{succ}(n)\ \mathsf{nat}$, we can construct a derivation of $n\ \mathsf{nat}$ by trimming the last line of the derivation. For $n \stackrel{\text{def}}{=} \mathsf{succ}(\mathsf{zero})$:

$$\frac{\dfrac{\mathsf{zero}\ \mathsf{nat}}{\mathsf{succ}(\mathsf{zero})\ \mathsf{nat}}}{\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))\ \mathsf{nat}} \quad \rightsquigarrow \quad \frac{\mathsf{zero}\ \mathsf{nat}}{\mathsf{succ}(\mathsf{zero})\ \mathsf{nat}}$$

This is what the proof of Claim 1 implicitly does.

Admissible rules imply some non-trivial reasoning. When there is no such requirement, a rule is called derivable.

A rule is **derivable** if we can use a derivation of its premise as a building block in deriving its conclusion.

For example, the following rule is derivable: $\dfrac{n\ \mathsf{nat}}{\mathsf{succ}(\mathsf{succ}(n))\ \mathsf{nat}}$

Indeed, if we have a derivation of the premise, all it takes is two uses of the rule Succ:

$$\frac{\vdots}{n\ \mathsf{nat}} \quad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{\vdots}{n\ \mathsf{nat}}}{\mathsf{succ}(n)\ \mathsf{nat}}\ \textsc{Succ}}{\mathsf{succ}(\mathsf{succ}(n))\ \mathsf{nat}}\ \textsc{Succ}$$

There is no need to perform induction to show that a rule is derivable.

5 Parallels with functional programming

Notice that the above sets of rules have a flavour akin to the following data type definitions in Haskell.

```
data Nat = Zero | Succ Nat

data Even = Zero | Succ Odd
data Odd = Succ Even
```

This correspondence is rather deep, and leads to modern **proof assistants**.

The admissible and derivable rules given above also closely correspond to the following Haskell functions.

```
data Nat = Zero | Succ Nat

admissible :: Nat → Nat
admissible Zero = <can't happen, by assumption!>
admissible (Succ n) = n

derivable :: Nat → Nat
derivable n = Succ (Succ n)
```

There is an essential difference between the two: the first one uses pattern matching ($\approx$ induction) on natural numbers, while the second one does not.