Preliminaries

Gabriele Keller

July 13, 2017

1 Introduction

As we are going to discuss and reason about properties of various programming languages and language features, we need a formal meta-language which allows us to make statements about these properties. We need to specify the grammar of a language, the static semantics (often in form of typing and scoping rules) and dynamic semantics. Fortunately, it turns out that a single formalism, inductive definitions built on inference rules, is sufficient.

2 Judgements and Inference Rules

A judgement is simply a statement that a certain property holds for a specific object,[1] or alternatively

- $3 + 4 * 5$ is a valid arithmetic expression
- the string "aba" is a palindrome
- 0.43423 is a floating point value

Judgements are not unlike predicates you might know from Predicate Logic. We write

$$a\ S$$

for a judgement of the form *The property S holds for object a.* In predicate logic, this is usually written differently, as *S(a).* However, we will see later that for our purpose, it is much more convenient to write it in the above given post-fix notation. Alternatively, we can interpret S as a set of objects with a certain property, and read the judgement aS as: a is an element of the set S. Some examples of judgements and how to read them are:

- 5 *even*
  - 5 is even, or
  - 5 is an element of the set of even numbers
- $3 + 4 * 5$ *expr*
  - $3 + 4 * 5$ is a syntactically correct expression, or
  - $3 + 4 * 5$ an element of the set containing all syntactically correct expressions
- 0.43423 *float*
  - 0.43423 is a floating point value
  - 0.43423 is an element of the set of floating points values

[1] More generally, a relationship between a number of objects holds. For now, we just look at statements about a single object.

Judgements by themselves would be boring and fairly useless. Most interesting sets have an infinite number of elements, and to define such a set it would obviously be impossible to explicitly list them all using simple judgements. Luckily, the sets we are interested in are also no random collections of objects, but the sets and properties can be systematically defined by so-called inference rules.

Inference rules allow us to combine judgements to obtain new judgements and have the following general form:

> If judgements J_1, and J_2, and $\ldots$ and J_n are inferable, then the judgement J is inferable

and are usually given in the following standard form:

$$\frac{J_1 \quad J_2 \quad \ldots \quad J_n}{J}$$

where the judgements J_1 to J_n are called premises, and J is called a conclusion. An inference rule does not have to have premises, it can consist of a single conclusion. Such inference rules are called axioms. But let us have a look at a concrete example now.

We start by defining some simple properties over the set of natural numbers (*Nat*). For simplicity reasons, we represent them as 0, $(s\ 0)$, $(s\ ((s\ 0)))$, $(s(s(s\ 0)))$ for $0, 1, 2, 3$, and so on (s here stands for *successor*). So, first of all, how can we define *Nat* itself using inference rules? Listing all the element of *Nat* would be equivalent to including an axiom for each number:

$$\frac{}{0\ \mathit{Nat}}$$

$$\frac{}{(s\ 0)\ \mathit{Nat}}$$

$$\frac{}{(s\ (s\ 0))\ \mathit{Nat}}$$

$$\vdots$$

Apart from the first rule, all the rules have the form

$$\frac{}{(s\ x)\ \mathit{Nat}}$$

where x *Nat* has been established by the previously listed axiom. In words, we have

1. 0 is in *Nat*, and
2. for all x, if x is in *Nat*, then $(s\ x)$ in *Nat*

which can be translated directly into the following two inference rules:

$$\frac{}{0\ \mathit{Nat}} \qquad \frac{x\ \mathit{Nat}}{(s\ x)\ \mathit{Nat}}$$

where x can be instantiated to any In the same way, we can define the sets *Even* and *Odd*:

$$\frac{}{0\ \mathit{Even}} \qquad \frac{x\ \mathit{Even}}{(s(s\ x))\ \mathit{Even}} \qquad \frac{}{(s\ 0)\ \mathit{Odd}} \qquad \frac{x\ \mathit{Odd}}{(s(s\ x))\ \mathit{Odd}}$$

Rules do work in two ways: we can use them to define a property, but we can also use them to show that a judgement is valid. How does it work with inference rules? Assume we want to show that some judgement J is valid. We have to look for a rule which has J as a conclusion. If

this rule is an axiom, we are already done. If not, we have to show that all of its premises are valid by recursively applying the same strategy to all of them. For example, we can show that $s(s(0))$ *Even*, since

$$\frac{0\ \mathit{Even}}{(s(s\ 0))\ \mathit{Even}}$$

and

$$\frac{}{0\ \mathit{Even}}$$

As the last rule is an axiom, there are no premises left to prove, and we are done.

Similarly, we can show that $0+1+1+1+1$ *Even*. An alternative and often quite convenient way to write inference proofs is to stack the rules we apply together and draw a "proof tree" — in our example, more a proof stack, since each rule has just a single premise.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{(s\ 0)\ \mathit{Odd}}}{(s(s\ 0))\ \mathit{Odd}}}{(s(s(s\ 0)))\ \mathit{Odd}}}{(s(s(s(s\ 0))))\ \mathit{Odd}}$$

Note that inference works on a purely syntactic basis. Given the rules above, we are not able to prove 2 *Even*, even though we can show that $s(s(0))$ *Even*, and we know that $s(s(0))$ is equal to 2, we cannot apply that knowledge, since we have no rule which tells us it is ok to do so. We just mechanically manipulate terms according to the given rules.

Let us look at a slightly more interesting example: we want to define the language M which contains all expressions of properly matched parenthesis (and no other characters):[2]

$$M = \{\epsilon, \texttt{()}, \texttt{()()}, \texttt{()()()}, \ldots, \texttt{(())}, \texttt{((()))}, \ldots, \texttt{()(())}, \texttt{()()(())}, \ldots\}$$

Again, let us start by describing the rules in (semi-)natural language. There are basically two ways to "legally" combine parenthesis: we can either nest them, or concatenate them:

1. The empty string (denoted by ϵ) is in M
2. If s_1 and s_2 are in M, then $s_1 s_2$ is in M
3. If s is in M, then `(`s`)` is in M

Again, these rules can be directly translated into inference rules:

$$(1)\qquad \frac{}{\epsilon\ M}$$

$$(2)\qquad \frac{s_1\ M \qquad s_2\ M}{s_1 s_2\ M}$$

$$(3)\qquad \frac{s\ M}{\texttt{(}s\texttt{)}\ \ M}$$

How can we show that `()(())` is in M? As we did previously, we check if there is a rule whose conclusion matches the judgement we want to infer. If we apply Rule (2), we have to show that both () and (()) are in M. Since () = (ϵ), we can apply Rule (3), and only have to show that ϵ is in M (Rule (1)). By applying Rule (3) in the same way, we can show that (()) is in M as well, and we are done:

The problem is, however, not as straight forward as it seems, because instead of applying Rule (2), we could as well apply Rule (3). It has only a single premise, so we only have to prove

[2] ϵ represents the empty string

that)(() is in M. The only rule that's applicable now is Rule (2), and we could apply it in several different ways resulting in different premises:

$$\frac{)\ M \quad (()\ M}{)(()\ M}$$

or

$$\frac{)(\ M \quad ()\ M}{)(()\ M}$$

or

$$\frac{)((\ M \quad)\ M}{)(()\ M}$$

In the first application, we end up with the premise:) is in M, but there is no rule which we can apply to get rid of it. This is not that surprising, since M should only contain expressions of properly matched parenthesis, and), as well as)(() are not properly matched. So, by choosing the wrong rule, or applying the right rule in a wrong way – for example, splitting ()() up into ()(and) – it is easily possible to end up with premises which are not actually valid and reach a dead end. In our example, this was not hard to see. It can be extremely difficult to decide which rule to apply and how, without some background knowledge about the objects and properties, as there might be an infinite number of possibilities. This is one reason why it is not possible to write a program which automatically infers judgements, and which guarantees to find such a derivation if it exists. It is, however, possible to write semi-automatic theorem provers, which come up with proofs in cases where it is fairly standard, and rely on user input otherwise.

2.1 Derivable and Admissible Rules

What would happen if we added the following rule:

$$(4) \qquad \frac{s\ M}{((s))\ M}$$

Does this change the set M in any way, that is, is there a string s for which we can infer $s\ M$ if we use 4, but not otherwise? This doesn't seem very likely: the rule just says that, if a string s is in M it is ok to add two pairs of matching parenthesis. Since we already had a rule which allows us to add one pair of parenthesis, we can apply this rules twice and achieve the same effect:

$$\frac{\dfrac{s\ M}{(s)\ M}}{((s))\ M}$$

This means that Rule (4) is derivable from the existing rules.

In all the previous rules, the strings in the premises were simpler than the string in the conclusion. The following rule if different in this respect:

$$(5) \qquad \frac{()s\ M}{s\ M}$$

Interestingly, although Rule (4) does not introduce any new strings to M, the rule is also not derivable from any of the existing rules. Such a rule is called admissible.

2.2 Inductive Definitions

A set of inference rules R defining a set A is called an inductive definition of A, if $s\ A$ holds if and only if $s\ A$ is derivable using R. All the examples we discussed are inductive definitions.

Not all sets can be defined using inductive definitions. For example, while natural numbers are one of the standard examples for such sets, floating point numbers cannot be defined in such a way.

2.3 Judgements and Relations

So far, we defined a judgement to be a statement about a property of an object. We can generalise this definition to relation between a number of objects. Consider the following inductive definition of the relation "a < b" on natural numbers. For convenience reasons, we choose an infix notation here:

$$\frac{n\ Nat}{0\ <\ (s\ n)}$$

$$\frac{n\ <\ m}{(s\ n)\ <\ (s\ m)}$$

As before, we can also view this as an inductive definition of a set. In this case, a set of pairs, where (a, b) in $<$ if and only if a is less than b.

3 Rule Induction

Natural deduction by itself is sometimes not powerful enough. For example, although we can see that the Rule (5) in Section 2.1 is valid for every string s in M, we cannot show this by simply combining the existing rules. We will therefore introduce another proof technique here, called induction. You will probably know induction over natural numbers and structural induction from mathematics and previous courses. Both are special cases of a more general induction principle called rule induction.

Let us go back to our previous example set M of properly matched parenthesis. The rules 1 to 3 provide an inductive definition of M:

$$(1)\qquad \frac{}{\epsilon\ M}$$

$$(2)\qquad \frac{s_1\ M \qquad s_2\ M}{s_1 s_2\ M}$$

$$(3)\qquad \frac{s\ M}{(s)\ M}$$

Now, let us assume we want to prove some property P of the strings in M, that is: show that if $s\ M$ then $s\ P$. Since we know that there is a derivation for each $s\ M$, we only need to show that:

- $\epsilon\ P$
- if $s_1\ P$ and $s_2\ P$, then $s_1 s_2\ P$
- if $s\ P$, then $(s)\ P$

Which in essence correspond to the original rules only with M replaced by P. For example, if we want to show that all s in M have the same number of opening and closing brackets, we need to prove the following statements (where *open* denotes the number of opening, *close* those of closing brackets):

1. $open(\epsilon) = close\ (\epsilon)$

   Proof: $open(\epsilon) = 0 = close\ (\epsilon)$

2. if $open(s_1) = close\ (s_1)$ and $open(s_2) = close\ (s_2)$ then $open(s_1\ s_2) = close\ (s_1\ s_2)$

   For the proof, we assume that the statements corresponding to the judgements in the premises of the rules hold. These assumptions are called the induction hypothesis.

- Induction Hypothesis 1: $open(s_1) = close\,(s_1)$
- Induction Hypothesis 2: $open(s_2) = close\,(s_2)$

Proof: $open(s_1\ s_2) = open(s_1) + open\,(s_2) = close(s_1) + close\,(s_2) = close\,(s_1 + s_2)$

3. if $open(s) = close\,(s)$ then $open(\texttt{(}s\texttt{)})$

   - Induction Hypothesis 1: $open(s) = close\,(s)$

   Proof:

$$
\begin{aligned}
& open(\texttt{(}s\texttt{)}) \\
= \quad & \{\text{property of } open\} \\
& open(\texttt{(}) + open(s) + open(\texttt{)}) \\
= \quad & \{\text{property of } open\} \\
& 1 + open(s) + 0 \\
= \quad & \{\text{Induction Hypothesis,Arithmetic}\} \\
& 1 + close(s) + 0 \\
= \quad & \{\text{property of } close\} \\
& close(\texttt{(}) + close(s) + close(\texttt{)}) \\
= \quad & \{\text{property of } close\} \\
& close(\texttt{(}s\texttt{)})
\end{aligned}
$$

In the proof, we used the rules of arithmetic, and that properties of *close* and *open*, such as $open(\texttt{(}) = close\,(\texttt{)}) = 1$, and $open(\texttt{)}) = close(\texttt{(}) = 0$. In a fully formal proof, we would also need a formal definition of these two functions.

3.1 Ambiguity

The definition of M, although correct, has a undesirable property: for any string in M, we do not have just one derivation, but an infinite number of possible derivations, since any string s can be split into ϵ and s by applying Rule (2), and then Rule (1) to get rid of ϵ. However, this derivation step is completely unnecessary.

Fortunately, if we take a more structured view on the elements of this language, we can come up with an alternative set of rules, where we have exactly one derivation for each string in the set. We can interpret each string as a possibly empty list (L) of non-empty parenthesis expressions (N) according to the following inference rules:

$$
\begin{aligned}
&(1) \quad \frac{}{\epsilon\ L} \\
&(2) \quad \frac{s_1\ N \quad s_2\ L}{\epsilon\ L} \\
&(2) \quad \frac{s\ L}{\texttt{(}s\texttt{)}\ N}
\end{aligned}
$$

The interesting point here is that L and N are defined in terms of each other: we have a mutually recursive definition.

Let us look at one more example of an ambiguous definition: simple arithmetic expressions, given here both in EBNF form and as inductive definition using inference rules:

The EBNF

Expr $\rightarrow$ *int* | `(`*Expr*`)` | *Expr* `+` *Expr* | *Expr* `*` *Expr*

describes the same language as the following set of inference rules (*int* represents an integer constant):

$$\frac{e\ Expr}{(e)\ Expr} \qquad \frac{e_1\ Expr \quad e_2\ Expr}{e_1\ \texttt{+}\ e_2\ Expr} \qquad \frac{e_1\ Expr \quad e_2\ Expr}{e_1\ \texttt{*}\ e_2\ Expr}$$

Although in this case, there are not an infinite number of possible derivations for each expression, every expression which contains more than a single arithmetic operation still can be derived in more than one way:

$$\dfrac{\dfrac{\dfrac{}{1\ Expr} \quad \dfrac{}{2\ Expr}}{1+2\ Expr} \quad \dfrac{}{3\ Expr}}{1+2*3\ Expr}$$

$$\dfrac{\dfrac{}{1\ Expr} \quad \dfrac{\dfrac{}{2\ Expr} \quad \dfrac{}{3\ Expr}}{2*3\ Expr}}{1+2*3\ Expr}$$

Although both derivations are correct with respect to the rules given, the second derivation is more appropriate for an arithmetic expression, as it decomposes the expression first into two summands. We give an alternative definition, which takes precedence and associativity of the operators into account:

$$\frac{e_1\ SExpr \quad e_2\ PExpr}{e_1\ \texttt{+}\ e_2\ SExpr} \qquad \frac{e\ PExpr}{e\ SExpr}$$

$$\frac{e_1\ PExpr \quad e_2\ FExpr}{e_1\ \texttt{*}\ e_2\ PExpr} \qquad \frac{e\ FExpr}{e\ PExpr}$$

$$\frac{e\ SExpr}{\texttt{(}e\texttt{)}\ FExpr} \qquad \frac{}{int\ FExpr}$$

The unambiguous grammar is, again, much more complicated than the original grammar, even for such a simple language. This is not surprising, as it contains additional structural information. For programming languages, ambiguous grammars are problematic, as they may allow different interpretations of programs, and are therefore usually avoided.

3.2 Simultaneous Induction

How can we apply the principle of rule induction to mutually recursive definitions like those of L and *SExpr*? In most cases, we have to generalise the proof goal. For example, if we want to prove a property P for all e in *SExpr*, that is e *SExpr* implies e *PExpr*. By the principle of rule induction we have to show that

- under the assumption that
  - $e = e_1 + e_2$
  - e_1 *SExpr*
  - e_2 *PExpr*

  and the Induction Hypothesis
  - e_1 P

show that $e_1 + e_2\ P$

- under the assumption that

  - e *PExpr*

  show that $e\ P$

The problem is that, since we only try to show something about *SExpr*, the induction hypothesis does not say anything about e_2 (we only know that e_2 *PExpr*, and nothing at all about e in for the second rule. In most cases, this is not enough to prove anything. The solution is often to try and prove a more general statement instead which leads to stronger induction hypothesis. If we try to show, for instance, that e *SExpr* or e *PExpr* or e *FExpr* implies $e\ P$, we have more cases to cover on one hand (one for each inference rule which has *PExpr*, *SExpr* or *FExpr* in the conclusion), on the other hand the induction hypothesis cover all the premises in the rules:

- under the assumption that

  - $e = e_1 + e_2$
  - e_1 *SExpr*
  - e_2 *PExpr*

  and the Induction Hypothesis

  - $e_1\ P$
  - $e_2\ P$

  show that $e_1 + e_2\ P$

- under the assumption that

  - e *PExpr*

  and the Induction Hypothesis

  - $e\ P$

  show that $e\ P$ (trivial)

- under the assumption that

  - $e = e_1 * e_2$
  - e_1 *SExpr*
  - e_2 *PExpr*

  and the Induction Hypothesis

  - $e_1\ P$
  - $e_2\ P$

  show that $e_1 * e_2\ P$

- ...

4 Examples

4.1 Boolean Expressions

As another example, consider boolean expressions. For simplicity, we only include three operators for now: $\wedge$, $\vee$, and $\neg$, the constants *True* and *False*, and parentheses. Our first attempt at defining a set of inference rules to characterise boolean expressions might look as follows:

$$\frac{}{\textbf{True}\ BExpr} \qquad \frac{}{\textbf{False}\ BExpr} \qquad \frac{e\ BExpr}{\neg e\ BExpr}$$

$$\frac{e\ BExpr}{(e)\ BExpr} \qquad \frac{e_1\ BExpr \quad e_2\ BExpr}{e_1 \wedge e_2\ BExpr} \qquad \frac{e_1\ BExpr \quad e_2\ BExpr}{e_1 \vee e_2\ BExpr}$$

Unfortunately, with this set of rules, we have the same problem we had with our rules for arithmetic expressions. Even though they inductively define the set of boolean expressions, they are ambiguous and do not reflect associativity and precedence of the operators. So, we need to come up with an alternative definition. The operator $\neg$ has the highest precedence, $\vee$ the lowest, and both $\wedge$ and $\vee$ are left associative. The solution is also similar to the solution for arithmetic expressions. First, we need rules to define the subset of boolean expressions which can be arguments of the operator with the highest precedence, negation. These can only be atomic expressions (constants), any expression in parentheses, or such an expression preceded by negation. Let's call this subset *NbExpr*. We call the boolean expressions we generate with the new rules *Bexpr*.

$$\frac{}{\textbf{True}\ Nbexpr} \qquad \frac{}{\textbf{False}\ Nbexpr} \qquad \frac{e\ Bexpr}{\neg(e)\ Nbexpr}$$

The rules for the operators $\wedge$ and $\vee$ correspond to those for addition and multiplication. Since the operators are left associative, the expression on the left hand side can only be an expression with stronger cohesion than the one on the right hand side. For the $\wedge$ operator, it has to be a *Nbexpr*.

$$\frac{e_1\ NbExpr \quad e_2\ Abexpr}{e_1 \wedge e_2\ Abexpr} \qquad \frac{e_1\ Abexpr \quad e_2\ Bexpr}{e_1 \vee e_2\ Bexpr}$$

And finally we need rules to express the fact the $Nbexpr \subseteq Abexpr \subseteq Bexpr$

$$\frac{e\ Nbexpr}{e\ Abexpr} \qquad \frac{e\ Abexpr}{e\ Bexpr}$$

4.2 More Arithmetic

We can define addition on natural numbers inductively as relation between three numbers n,m, and k, where $n + m = k$.

$$(Add-1) \qquad \frac{n\ Nat}{0 + n = n}$$

$$(Add-2) \qquad \frac{n + m = k}{(s\ m) + n = (s\ k)}$$

The rule

$$\frac{n\ Nat}{n + 0 = n}$$

is not derivable from $Add - 1$ and $Add - 2$, but it is admissible. You can show that this is the case using induction over natural numbers. Similarly, for the following rule:

$$\frac{n + m = k}{m + n = k}$$

5 Exercises

1. Define a set of booleand expressions *TBExpr* which only contains boolean expressions which would evaluate to *True*.
2. Using induction over natural numbers, show that the two admissible rules of the arithmetic expression example are indeed admissible.
3. Define a Haskell data type which models boolean expressions.

Acknowledgements These lecture draw on material from the draft book on *Programming Languages: Theory and Practice*, Robert Harper, and contain examples taken from Frank Pfenning's lecture notes for the CMU course *Foundations of Programming Languages*.