Very short lecture notes:

Mathematical Foundations of Programming

University of Nottingham, Computer Science, module code G54FOP, Spring 2018

Nicolai Kraus

Draft of February 15, 2018

What is this? This document is written for students of the module G54FOP (Nottingham, Spring 2018, two times one hour per week). After each lecture, I add some notes on what we have done. Thus, this document summarises the lectures but not more. It mixes what I write on the board with what I say. **It cannot serve as a replacement for an actual book or for the lecture**, but it can supplement students' own notes.
The lecture is based on a couple of sources:

- Venanzio Capretta's lecture notes for this module from previous years: `http://www.cs.nott.ac.uk/~pszvc/g54fop/`
- My own studies of this topic (many different sources, in particular the lecture *Semantics of Programming Languages* by Andreas Abel and Ulrich Schöpp, LMU Munich 2010).
- Andrew Pitts' lecture notes: `http://www.cl.cam.ac.uk/teaching/2001/Semantics/`
- The Agda course by Thorsten Altenkirch: `http://www.cs.nott.ac.uk/~psztxa/g53cfr/`
- (more may be added later)

LECTURE 1 (31 JANUARY 2018)

Aspects of Programming Languages. Consider the following three lines:

(1) x=2, print(x);

(2) x=2; print(x);

(3) x=1; x=x+1; print(x);

Are these lines programs? Are they the same program? A programming language has several aspects:

(I) **Syntax:** Which strings (or trees) are programs? Which symbols are allowed and which are reserved? For example, the syntax of an (imaginary) language could disallow the usage of a comma as in (1), and thus (1) would not be a program in the language given by that syntax. (2) and (3) could be valid programs. From the point of view of the syntax, they are different (they are different strings!). We will see how we can specify the syntax of a language via a *Backus-Naur Form* (BNF) or via *derivation rules* (needless to say, there are other ways as well).

(II) **Semantics:** What is the meaning of a program? What should it do? A priori, a program is just a string (or a tree) of symbols. The strings (1), (2), (3) only "make sense" to us because the (non-specified) syntax on which they are based is intuitive and familiar from programming languages that we know. We could define a syntax which has no obvious "meaning" (if one sees a program written in the language Brainfuck for the first time, one will find it hard to make sense of seemingly random strings of symbols). There are various ways to give semantics to a language. One could just give a reference implementation and say that the meaning of a program is just what this reference implementation does with it. One could also describe the semantics informally, which is what one will find if you open a tutorial for Java, for example. What we are interested in this lecture is *formal sematics*, e.g. mathematically precise ways of assigning meanings to a program. The possibilities that we will look at are *denotational sematics* and *operational semantics*.

(III) Other aspects are: How should the language be used ("software engineering")? For which problems is it suitable? What about commenting and should one use indentation (if the latter is not part of the syntax)? These questions are not of mathematical nature, and they are not part of this lecture.

Example: A simple language of arithmetic expressions (see the lecture notes of Venanzio Capretta). We can specify the syntax of this language with a Backus-Naur Form:

$$\textit{Expr} \quad ::= \quad \texttt{t} \mid \texttt{f} \mid \texttt{z} \mid \texttt{s}\ \textit{Expr} \mid \texttt{p}\ \textit{Expr} \mid \texttt{iz}\ \textit{Expr} \mid \texttt{if}\ \textit{Expr}\ \texttt{then}\ \textit{Expr}\ \texttt{else}\ \textit{Expr}$$

Here, *Expr* stands for the set of all allowed arithmetic expression, and they are inductively defined through the different possibilities above (the pipe | separates possibilities). This means that an element of *Expr* is either `t` or it is `f` or it is ... or it is `if` *Expr* `then` *Expr* `else` *Expr*

where every instance of *Expr* is replaced by an element of the set *Expr* . Thus, we should think of an expression as a tree (`t` is a node with no children, `ifthenelse` is a node with three children). In order to write them as strings, we use parentheses to encode the tree structure; we do not go into the details of this, but we regard `s ( z )` and `s z` as syntactically equal. Examples of expressions are:

(i) `z`
(ii) `p t`
(iii) `if ( s f ) then z else ( if z then t else f )`

If at this point it is unclear what the above expressions are supposed to mean, then this is at least partially intended. Remember that we are currently only discussing the syntax, but not yet the semantics.
If we use *inference rules*, we can present the language *Expr* as follows:

$$\frac{}{\texttt{t} : \mathit{Expr}} \qquad \frac{}{\texttt{f} : \mathit{Expr}} \qquad \frac{}{\texttt{z} : \mathit{Expr}} \qquad \frac{e : \mathit{Expr}}{\texttt{s}\ e : \mathit{Expr}} \qquad \frac{e : \mathit{Expr}}{\texttt{p}\ e : \mathit{Expr}} \qquad \frac{e : \mathit{Expr}}{\texttt{iz}\ e : \mathit{Expr}}$$

$$\frac{e_1 : \mathit{Expr} \qquad e_2 : \mathit{Expr} \qquad e_3 : \mathit{Expr}}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mathit{Expr}}$$

This should be read as follows. Each horizontal line stands for one rule. The colon ":" means "is", we could as well write "$\in$" instead of ":". The statements above each horizontal line are the *assumptions*, the statement below each line is the *conclusion* of the rule. In general, each rule as an arbitrary (non-negative but, at least for us, finite) number of assumptions, and exactly one conclusion. Thus, the fourth rule can be read as "if e is an arithmetic expression, then so is `s` e". The last rule has three separate assumptions. The first three rules do not have any assumptions.
If we want to derive the expression (iii) from above with these rules, we need to draw a derivation tree as follows:

$$\dfrac{\dfrac{\dfrac{}{\texttt{f} : \mathit{Expr}}}{\texttt{s f} : \mathit{Expr}} \qquad \dfrac{}{\texttt{z} : \mathit{Expr}} \qquad \dfrac{\dfrac{}{\texttt{z} : \mathit{Expr}} \qquad \dfrac{}{\texttt{t} : \mathit{Expr}} \qquad \dfrac{}{\texttt{f} : \mathit{Expr}}}{\texttt{if z then t else f} : \mathit{Expr}}}{\texttt{if (s f) then z else (if z then t else f)} : \mathit{Expr}}$$

Remark 1. So far, it probably seems as if the derivation rules are just a long and tedious way to write down the same information that we get from the Backus-Naur form. We will later see that derivation rules offer more flexibility and are somewhat more powerful.

Now that we have specified the syntax of the language *Expr* , let us try to specify possible semantics. We look at *denotational semantics* first. In this approach, we choose a mathematical structure (ideally one which we understand well). Then, we say what each expression arithmetic expression "means", by mapping the set *Expr* to this structure.

In our example, as the structure we simply choose the set

$$\begin{aligned} S &:= \{\text{True}, \text{False}\} \cup \mathbb{N} \cup \{\bot\} \\ &= \{\text{True}, \text{False}, 0, 1, 2, 3, \ldots, \bot\} \end{aligned}$$

That is, each element of the set S is a boolean value, or a natural number, or the symbol $\bot$. The latter can be read as "bottom" or "undefined". We define a function $[\![-]\!] : \mathit{Expr} \to S$; the name of this function ("semantic brackets") probably looks strange, but the idea is that we write $[\![e]\!]$ instead of $[\![-]\!](e)$. This function is defined by recursion (sometimes called *structural recursion*, or just *induction*) on how elements of *Expr* are generated, much like you have defined functions in Haskell in earlier courses using pattern matching (e, e_1 and so on are variables which replace expressions):

$$[\![\,\texttt{t}\,]\!] = \text{True}$$

$$[\![\,\texttt{f}\,]\!] = \text{False}$$

$$[\![\,\texttt{z}\,]\!] = 0$$

$$[\![\,\texttt{s}\ e]\!] = \begin{cases} [\![e]\!] + 1 & \text{if } [\![e]\!] \text{ is a number} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\,\texttt{p}\ e]\!] = \begin{cases} 0 & \text{if } [\![e]\!] = 0 \\ [\![e]\!] - 1 & \text{if } [\![e]\!] \text{ is a number larger than } 0 \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\,\texttt{iz}\ e]\!] = \begin{cases} \text{True} & \text{if } [\![e]\!] = 0 \\ \text{False} & \text{if } [\![e]\!] \text{ is a number other than } 0 \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\,\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = \begin{cases} [\![e_2]\!] & \text{if } [\![e_1]\!] = \text{True and } [\![e_2]\!], [\![e_3]\!] \text{ are both} \\ & \text{numbers or both boolean values} \\ [\![e_3]\!] & \text{if } [\![e_1]\!] = \text{False and } [\![e_2]\!], [\![e_3]\!] \text{ are both} \\ & \text{numbers or both boolean values} \\ \bot & \text{otherwise} \end{cases}$$

Remark 2. Now, the *meaning* of expressions in *Expr* is suddenly clear! `t` stands for True, `f` stands for False, `z` for 0, `s` for "+1" (successor), `p` for "−1" (predecessor), `iz` for "is the argument 0?", and `ifthenelse` for a case distinction. Expressions which "don't make sense", such as "True+1", are simply undefined.
However, it is important to keep in mind that this is **only one possible meaning**! **We** have defined this meaning, and we could understand the expressions completely differently! For example, we could say that `f` stands for 0 and `t` for 5, but we could also do much crazier things and assign the name of a lecturer to every expression, or whatever we want.
But even if we already had in mind that `t` should mean True, and that we don't want negative numbers, and so on, we still have made some choices in the definition above. For example, we have chosen that `if t then t else z` is undefined, but it could as well be True.

Exercises. Note: The module catalogue determines that the module G54FOP does not come with problem sets/homework. To pass the module, you do not need to do these exercises. However, they may help you to deepen your understanding.

1. Calculate:
   - $[\![\texttt{s(s(s z))}]\!]$
   - $[\![\texttt{p(s z)}]\!]$
   - $[\![\texttt{s(p z)}]\!]$
   - $[\![\texttt{s(iz z)}]\!]$
   - $[\![\texttt{if t then z else t}]\!]$
2. Define new semantics, by giving a new definition of $[\![-]\!]$, which evaluates an expression to True if the expression "makes sense" and to False if the expression "makes no sense". Here, you need to choose yourself what "making sense" means since I have not given a definition. It could just mean that in our semantics above the expression is not undefined, but there are other possibilities.
3. Extend the language *Expr* in some reasonable way, e.g. by adding symbols for "or", "and" (for booleans) or "plus", "times" (for numbers). Extend the denotational semantics with these new constructs.

LECTURE 2 (1 FEBRUARY 2018)

Let us continue with the language *Expr* of simple arithmetic expressions. We want to look at *operational semantics*. In this case, we specify the meaning of an expression (or a program) by saying how it computes. For example, we could take the following set of *reduction* or *simplification* rules:

$$\begin{aligned}
&\texttt{iz z} \rightsquigarrow \texttt{t} \\
&\texttt{iz(s}\,e\texttt{)} \rightsquigarrow \texttt{f} \\
&\texttt{if t then}\,e_2\,\texttt{else}\,e_3 \rightsquigarrow e_2 \\
&\texttt{if f then}\,e_2\,\texttt{else}\,e_3 \rightsquigarrow e_3 \\
&\texttt{p(s}\,e\texttt{)} \rightsquigarrow e \\
&\texttt{p z} \rightsquigarrow \texttt{z}
\end{aligned}$$

If we have the above rules, we cannot reduce a term such as `if(iz z)then t else f`. To do this, we would first have to reduce a *subterm* (or we have to reduce "in a context"), namely `iz z`. We can make this precise by giving the following *structural rules* (we have not stated

them all explicitly in the lecture, but here we go):

$$\frac{e \rightsquigarrow e'}{\mathtt{s}\, e \rightsquigarrow \mathtt{s}\, e'} \qquad \frac{e \rightsquigarrow e'}{\mathtt{p}\, e \rightsquigarrow \mathtt{p}\, e'} \qquad \frac{e \rightsquigarrow e'}{\mathtt{iz}\, e_1 \rightsquigarrow \mathtt{iz}\, e_2}$$

$$\frac{e_1 \rightsquigarrow e'}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightsquigarrow \mathtt{if}\ e'\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3}$$

$$\frac{e_2 \rightsquigarrow e'}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightsquigarrow \mathtt{if}\ e_1\ \mathtt{then}\ e'\ \mathtt{else}\ e_3}$$

$$\frac{e_3 \rightsquigarrow e'}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightsquigarrow \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e'}$$

Let us write $e \rightsquigarrow^n e'$ for "e can be reduced to e' in n steps", i.e. there are $e_1, e_2, \ldots, e_n$ with

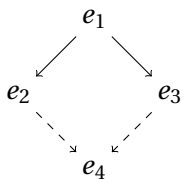
$$e = e_1 \rightsquigarrow e_2 \rightsquigarrow e_3 \rightsquigarrow \ldots \rightsquigarrow e_n = e'.$$

Let us further write $e \rightsquigarrow^* e'$ ("e reduces to e' in any number of steps") if there is a number n such that $e \rightsquigarrow^n e'$. We also allow ourselves to write $e \rightsquigarrow^{\leq n} e'$, meaning that e reduces to e' in n or fewer steps.

Remark 3. It is important to note that we have **not** defined a reduction algorithm. We have not specified in which order an expression should be reduced; we have only said what allowed reductions are.

We certainly *could* specify a reduction strategy, but we might not *want* to. Maybe we do not know what the most efficient way of reducing is, and want to leave it to other people to figure this out. However, we have brought ourselves in a potentially tricky situation: What if an expression reduces to `f` with one reduction sequence, but it also to `t` with another reduction sequence? A reduction system is called *confluent* if this sort of situation cannot occur. We want to show it for our language:

Theorem 4 (Confluence)**.** *If we have expressions e_1, e_2, e_3 such that $e_1 \rightsquigarrow^* e_2$ and $e_1 \rightsquigarrow^* e_3$, then there is a fourth expression e_4 such that $e_2 \rightsquigarrow^* e_4$ and $e_3 \rightsquigarrow^* e_4$. This can also be expressed by saying that whenever we have the solid part of the following diagram, we can find e_4 and the dashed part:*



Before we give a proof, we show a slightly simpler property.

Lemma 5. *Assume we have e_1, e_2, e_3 such that $e_1 \rightsquigarrow^{\leq 1} e_2$ (recall that this notation means that e_1 reduces to e_2 in at most one step) and $e_1 \rightsquigarrow^{\leq 1} e_3$. Then, there is an e_4 such that $e_2 \rightsquigarrow^{\leq 1} e_4$ and $e_3 \rightsquigarrow^{\leq 1} e_4$.*

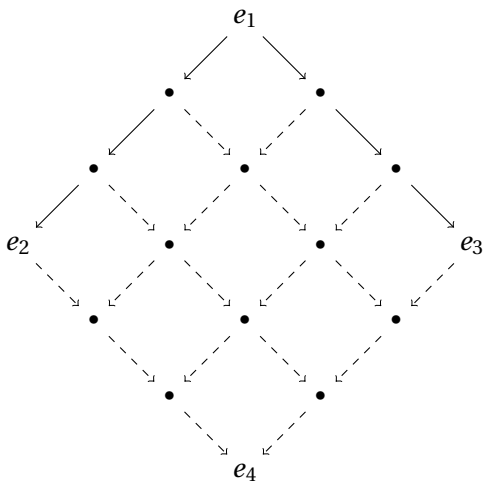
Proof. If we have $e_1 \rightsquigarrow^0 e_2$, we have $e_1 = e_2$ and can choose $e_4 := e_3$, and similarly, the case $e_1 \rightsquigarrow^0 e_3$ is trivial (we will later see that these cases are only included in the lemma to make it directly applicable to the proof of the confluence theorem). Thus, what is left is the case where $e_1 \rightsquigarrow e_2$ and $e_1 \rightsquigarrow e_3$ (recall that this means "exactly one step").
Recall that a tree s is a *full subtree* of a tree t if every node of s is a node of t, and every node of s has the same children that it has in t. Consider the syntax tree of e_1. The first reduction determines a full subtree of the syntax tree of e_1; it is the tree where the reduction $e_1 \rightsquigarrow r_2$ happens at the root, and we call it r_2. The corresponsing expression must be one of the left sides of the six reduction rules. Similarly, the subtree of e_1 determined by the second reduction is called r_3.
We distinguish three cases:

- If $r_2 = r_3$, the statement is trivial since no two reduction rules have identical left sides (i.e. if we know where we reduce, we do not have more than one choice), implying $e_2 = e_3$ and we can chose $e_4 := e_2$.
- If these two subtrees are disjoint, then the two reductions happen in two different branches of the tree e_1. In this case, if we do r_1 first, we can perform r_2 afterwards, and vice versa, reaching the same expression e_4 in both cases.
- The last remaining case is that r_3 is a subtree of r_2 or vice versa, but we can without loss of generality assume that the first is the case. The expression corresponding to r_2 (which we also call r_2) is of the form of a left side of one of the reduction rules. If r_2 is `(s` x), then r_3 is contained in x, since no reduction rule starts with `s`. In this case, the two reductions of e_1 can be done independently of the other, so if we do one first, we can do the other afterwards, and vice versa; in each case, we arrive at the same expression which we call e_4. If r_2 is `if t then` x_2 `else` x_3, then r_3 is either contained in x_2 (the same argument as before applies) or in x_3 (in this case, it is easy to see that we can choose $e_4 := x_2$). The other cases are very similar. □

The above lemma makes it simple to show the confluence theorem:

Proof of Theorem 4. By assumption, we have numbers k, m such that $e_1 \rightsquigarrow^k e_2$ and $e_1 \rightsquigarrow^m e_3$. We can apply Lemma 5 $k \cdot m$ times, which formally amounts to applying induction first on one, then the other. The easiest way of seeing what happens is to look at the following

diagram, which illustrates the case $k = m = 3$:



□

Remark 6. Confluence is a very important property, but not every reduction system enjoys it. In fact, there are very subtle ways in which the proof of Lemma 5 could go wrong, and where confluence could ultimately fail. For example, we could consider the language *Expr* with the six simplification rules given above, and the innocent-looking seventh rule

$$\texttt{s}\,(\texttt{p}\,e) \rightsquigarrow e.$$

This rule seems to make sense, since it says that "subtracting 1" is canceled out by adding 1", similar to how the rule `p (s` e`)` $\rightsquigarrow e$ works. But, with these seventh rule, confluence fails: take the expression `iz (s (p z))`. It can be reduced to `t` and, with another strategy, to `f`. But neither `t` nor `f` can be reduced further.

Exercises.

1. Find e such that $[\![e]\!] = \bot$ and $e \rightsquigarrow^* \texttt{t}$.
2. Give appropriate reduction rules for the extensions of *Expr* that you have considered in one of the previous exercises.
3. Explain in which step the proof of confluence fails for the reduction system with the seventh rule considered in Remark 6.

LECTURE 3 (7 FEBRUARY 2018)

This lecture consisted of an introduction to Agda. Please see the website for details. You can find the Agda file there, and you can complete the definition of $[\![-]\!]$ as a useful exercise.

LECTURE 4 (8 FEBRUARY 2018)

In the first half of the lecture, we discussed possible projects for the G54FPP module. Please see the website for the list of projects that we had. Afterwards, we continued with properties of the language *Expr*.
The following is some useful terminology which we should keep in mind:

Definition 7. A *redex* is an expression which can be reduced without using the structural rules, and an expression is in *normal form* if it does not contain a redex (i.e. if it cannot be reduced at all). Further (for our language *Expr*), an expression e is a *value* if it is in one of the following languages:

$$\begin{aligned} Bv &::= \texttt{t} \mid \texttt{f} \\ Nv &::= \texttt{z} \mid \texttt{s}\ (Nv) \end{aligned}$$

In other words, a value is either `t` or `f` or a sequence of `s` applied on `z`.

Lemma 8. *There is no infinite sequence of reduction steps* $e_1 \leadsto e_2 \leadsto e_3 \leadsto \ldots$.

Proof. Most of the time, the easiest way to prove this sorts of statement is to find a function $f : Expr \to \mathbb{N}$ such that $(e \leadsto e') \Rightarrow (f(e) > f(e'))$. Assuming that we have an infinite sequence $e_1 \leadsto e_2 \leadsto e_3 \leadsto \ldots$, we then get an infinite sequence of natural numbers, decreasing in each step, which is impossible. Here, as $f(e)$ we can take the function which counts the number of nodes in the syntax tree of e, and it's easy to see that this number decreases with every reduction step. □

Remark 9. The statement that each sequence of reductions is finite is very simple for our language *Expr*. For other reduction systems (e.g. the simply typed λ-calculus), this is less trivial.

What does it mean to "fully evaluate" an expression using the reduction rules that we have discussed? Given an expression, we want to reduce it and eventually reach a normal form. But what if we never reach a normal form, or there are multiple normal forms that we could reach? Fortunately, for our language *Expr*, this cannot happen. We have all ingredients necessary to prove this:

Theorem 10 (unique normal forms)**.** *Let e be an expression in the language* Expr*. Then, there is exactly one (i.e. a* unique*) normal form d* : Expr *such that* $e \leadsto^* d$.

Proof. Given e, we can start reducing randomly. By Lemma 8, we eventually reach some d which cannot be reduced anymore. Thus, d is a normal form by definition. We need to show that d is unique. Assume that there is a second normal form c such that $e \leadsto^* c$. By the confluence theorem (Theorem 4), we get an expression b together with reduction sequences $c \leadsto^* b$ and $d \leadsto^* b$. But, since c and d cannot be reduced, these reduction sequences must both consist of zero steps. This means that d, b and c are all equal. □

We have looked at denotational and operational semantics for *Expr*. How do they compare? We have already seen in the lecture (see the first exercise of Lecture 2) that expressions e where $[\![e]\!]$ is undefined may still reduce to a value ($[\![-]\!]$ is strict, while $\leadsto$ is not). However, we can say something for the other direction:

Theorem 11. *If we have an expression e such that $[\![e]\!]$ is a boolean value or a number, then there is a value v such that $e \rightsquigarrow^* v$. More precisely,*

- *if* $[\![e]\!] = \mathit{True}$, *then* $e \rightsquigarrow^* \mathtt{t}$.
- *if* $[\![e]\!] = \mathit{False}$, *then* $e \rightsquigarrow^* \mathtt{f}$.
- *if* $[\![e]\!] = n$ *(where n is a number), then* $e \rightsquigarrow^* \mathtt{s}\,(\mathtt{s}\,(\ldots(\mathtt{s}\,\mathtt{z})\ldots))$, *where the number of* $\mathtt{s}$ *coincides with n.*

The proof will be given in Lecture 5.

Exercises.

1. Find a new reduction rule such that, if you add this reduction rule to the rules given in the lecture, Lemma 8 becomes wrong.
2. Find a new additional reduction rule such that Theorem 10 becomes wrong. Find a rule which makes both Theorem 10 and Lemma 8 wrong.
3. Find a new additional reduction rule such that Theorem 10 still holds, but Lemma 8 becomes wrong.
4. Find a new additional reduction such that Lemma 8 still holds, but Theorem 10 becomes wrong.
5. Is it possible to add a new reduction rule such that Theorem 11 becomes wrong?

LECTURE 5 (14 FEBRUARY 2018)

Expr is an *inductively generated set*, like the natural numbers, lists, trees, and many other datatypes that we regularly use. Therefore, we can prove properties *by induction*. Let P be a property of expressions; we write $P(e)$ for "the property P holds for expression e". The induction principle says: If we can prove $P(e)$ under the assumption that $P(e')$ holds for all subexpressions e' of e, then $P(e)$ holds for all expressions e. (A subexpression is a subtree in the syntax tree.) The concrete form in which it is usually used is the following:

Principle 12 (Induction for *Expr*). Assume P is a property of expressions. Assume all of the following hold:

- $P(\mathtt{t})$
- $P(\mathtt{f})$
- $P(\mathtt{z})$
- $\forall e.P(e) \Rightarrow P(\mathtt{s}\ e)$
- $\forall e.P(e) \Rightarrow P(\mathtt{p}\ e)$
- $\forall e.P(e) \Rightarrow P(\mathtt{iz}\ e)$
- $\forall e_1, e_2, e_3.(P(e_1) \wedge P(e_2) \wedge P(e_3)) \Rightarrow P(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3)$

Then, we can conclude $\forall e.P(e)$.

Remark 13. The above principle looks a bit weaker than what we said informally directly before, since in Principle 12, not *all* subexpressions can be assumed to satisfy P. Nevertheless, Principle 12 is what is used nearly always, and the two formulations can be shown to be equivalent.

Let us use this induction principle to prove the theorem from last lecture:

Theorem 11. *If we have an expression e such that $[\![e]\!]$ is a boolean value or a number, then there is a value v such that $e \rightsquigarrow^* v$. More precisely,*

- *if $[\![e]\!] = \mathit{True}$, then $e \rightsquigarrow^* \mathtt{t}$.*
- *if $[\![e]\!] = \mathit{False}$, then $e \rightsquigarrow^* \mathtt{f}$.*
- *if $[\![e]\!] = n$ (where n is a number), then $e \rightsquigarrow^* \mathtt{s}\,(\mathtt{s}\,(\ldots(\mathtt{s}\,\mathtt{z})\ldots))$, where the number of $\mathtt{s}$ coincides with n.*

Proof. Let us define four properties A, B, C, and P:

- $A(e) := ([\![e]\!] = \text{True}) \Rightarrow (e \rightsquigarrow^* \mathtt{t})$
- $B(e) := ([\![e]\!] = \text{False}) \Rightarrow (e \rightsquigarrow^* \mathtt{f})$
- $C(e) := ([\![e]\!] = n) \Rightarrow (e \rightsquigarrow^* \mathtt{s}\,(\mathtt{s}\,(\ldots(\mathtt{s}\,\mathtt{z})\ldots)))$, where n is a number and $\mathtt{s}$ occurs n times
- $P(e) := A(e) \wedge B(e) \wedge C(e)$

We apply induction (Principle 12). Thus, we need to show all the points in the list of assumptions of this principle.

- SHOW $P(\mathtt{t})$:

  Since this is a conjunction, we show A, B, C separately. We start with $A(\mathtt{t})$. We can assume $[\![\,\mathtt{t}\,]\!] = \text{True}$; unfortunately, this does not give us any new information. We need to show $\mathtt{t} \rightsquigarrow^* \mathtt{t}$; fortunately, this is clearly the case (in 0 steps).

  Next, we need to show $B(\mathtt{t})$. We can assume $[\![\,\mathtt{t}\,]\!] = \text{False}$. If we look at the definition of $[\![-]\!]$, we see that this case cannot occur; hence, there is nothing to do (from a contradictory assumption, we can conclude whatever we want: *ex falso quodlibet*).

  We also need to show $C(\mathtt{t})$. In this case, the assumption is $[\![\,\mathtt{t}\,]\!] = n$. Again, this cannot happen and there is nothing to do.

- SHOW $P(\mathtt{f})$:

  Very similar to the above. This time, $A(\mathtt{f})$ and $C(\mathtt{f})$ have assumptions which are not satisfied, and the only case in which we have to do something is $B(\mathtt{f})$, where the conclusion is trivial ($\mathtt{f} \rightsquigarrow^* \mathtt{f}$).

- SHOW $P(\mathtt{z})$:

  Again, very similar.

- SHOW $\forall e.P(e) \Rightarrow P(\texttt{s}\ e)$:

  Let e be an expression. We can assume $P(e)$. We need to show $P(\texttt{s}\ e)$; again, we show $A(\texttt{s}\ e)$ and $B(\texttt{s}\ e)$ and $C(\texttt{s}\ e)$ separately.

  We start with $A(\texttt{s}\ e)$. In this case, we can assume $[\![\texttt{s}\ e]\!] = \text{True}$. Looking at how $[\![-]\!]$ is defined, we see that this is impossible.

  Similarly, $B(\texttt{s}\ e)$ requires us to show something under an impossible assumption.

  Finally, if we want to show $C(\texttt{s}\ e)$, we can assume $[\![\texttt{s}\ e]\!] = n$, for a number n. Looking at the definition of $[\![-]\!]$, we see that this can only happen if $[\![e]\!] = (n-1)$ and $n \geq 1$. Remember that we have assumed $P(e)$; thus, in particular, we know $C(e)$. Together, $C(e)$ and $[\![e]\!] = (n-1)$ show that we have $e \rightsquigarrow^* \texttt{s}\,(\texttt{s}\,(\ldots(\texttt{s}\ \texttt{z})\ldots))$, with $(n-1)$ occurrences of $\texttt{s}$. Using one of the structural rules, we get that $\texttt{e} \rightsquigarrow^* \texttt{s}\,(\texttt{s}\,(\ldots(\texttt{s}\ \texttt{z})\ldots))$ with n occurrences of $\texttt{s}$, as required.

- SHOW $P(\texttt{p}\ e)$:

  This is similar to the case $\texttt{s}\ e$ that we have just discussed.

- SHOW $P(\texttt{iz}\ e)$:

  Again, similar to the discussed case.

- SHOW $P(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$:

  We can assume $P(e_1)$ and $P(e_2)$ and $P(e_3)$. We need to show $A(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$ and $B(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$ and $C(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$.

  Let us discuss the goal $A(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$. We can assume $[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = \text{True}$. If we look at the definition of $[\![-]\!]$, we see that this is possible in two cases:

  1. first case: $[\![e_1]\!] = \text{True}$ and $[\![e_2]\!] = \text{True}$ and $[\![e_3]\!]$ is a boolean (either True or False). Recall that we can assume $P(e_1)$, thus in particular $A(e_1)$. From this, we get $e_1 \rightsquigarrow^* \texttt{t}$. Thus, by repeating the fourth structural rule (as given in Lecture 2),[1] we have

     $$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow^* \texttt{if}\ \texttt{t}\ \texttt{then}\ e_2\ \texttt{else}\ e_3.$$

     The first simplification rule of $\texttt{ifthenelse}$ gives us

     $$\texttt{if}\ \texttt{t}\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow e_2.$$

     Since we know $[\![e_2]\!] = \text{True}$, we get from $A(e_2)$ that $e_2 \rightsquigarrow^* \texttt{t}$. By combining these reductions, we have

     $$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \texttt{t}$$

     as required

  2. second case: $[\![e_1]\!] = \text{False}$ and $[\![e_3]\!] = \text{True}$ and $[\![e_2]\!]$ is a boolean (either True or False). Following the same strategy as in the first case, we get the desired result.

[1] "Repeating this rule" is, to be precise, another argument by induction – this is how one would formalise it in a proof assistant.

We now have to show $B(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$ and $C(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$. These are essentially copies of our argument for $A(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$. □

Remark 14. The above proof demonstrates how this sort of proof by induction can lead to many cases. Although most cases are easy, it is easy to miss one, or overlook a hidden difficulty. This is a situation in which a proof assistant can shine.

Remark 15. As someone has remarked, instead of *Expr* we can consider the language

```
E := B|N
B := t|f|iz N|if B then B else B
N := z|s N|p N|if B then N else N
```

The language `E` has the same "meaningful" expressions as *Expr* , but the typing ensures that "meaningless" (in the sense of $[\![e]\!] = \bot$) expressions are impossible to form. This is the general idea of types.

Exercises. 1. Can you reduce a single reduction rule such that the proof of this lecture fails and Theorem 11 becomes wrong? Which reduction rules can do the job?

LECTURE 6 (15 FEBRUARY 2018)

We have implemented more parts of these notes (operational semantics) in Agda. Please see the Agda files on the website. I have added an easy, a medium, and a very hard exercise to the main file.