Lecture Notes on Inference Rules and Careful Definition of λ-Calculus

15-814: Types and Programming Languages
Jan Hoffmann and Frank Pfenning

Lecture 4
Thursday, September 5, 2024

1 Introduction

We formalize the definitions that we introduced informally using the concept of *rule induction*.

2 Inference Rules and Rule Induction

We now revisit the definition of the λ-calculus to formalize the concepts like β-equality, which have introduced informally. Throughout this course, we use *inference rules* to write down inductive definitions. Inference rules usually define a *judgment*, which can be thought of as a relation between certain objects. An inference rule has the form

$$\frac{J_1 \cdots J_n}{J}$$

where J_i and J are judgments. We call $J_1, \ldots, J_n$ the premises of the rule and J the conclusion. The meaning of the rule is that J_1 and J_2 and $\ldots J_n$ imply J.

Inference rules can be best understood by example. For instance, we can inductively define a judgment e *exp*, which states that e is an expression. We define this judgment with 3 inference rules.

$$\frac{x\ \mathit{var}}{x\ \mathit{exp}}\,E_1 \qquad \frac{x\ \mathit{var} \qquad e\ \mathit{exp}}{\lambda x.e\ \mathit{exp}}\,E_2 \qquad \frac{e_1\ \mathit{exp} \qquad e_2\ \mathit{exp}}{(e_1\ e_2)\ \mathit{exp}}\,E_3$$

Rules are often annotated with names, here E_1, E_2, and E_3, so that we can easily reference them. In the rule E_1, we assume that we already have a judgment $x\ var$ for variables available. The rule states that each variable is an expression. The other rules reflect the remainder of the inductive definition of expressions.

We can take two views on the meaning of such inductive definitions. The first one is that inference rules define the smallest set of judgments that is closed under the rules. In the example, that is the smallest set that contains x *exp* for each variable, $\lambda x.e$ *exp* if e *exp* is in the set and x is a variable, and $(e_1\ e_2)$ *exp* if e_1 *exp* and e_2 *exp* are in the set.

Equivalently, we can say that the rules define the set of judgments that are *derivable* using the rules. A judgment is derivable if it has a *derivation tree*.

For example, we can derive $((\lambda x.\, x)\ y)$ *exp* with the following derivation tree.

$$\dfrac{\dfrac{x\ \mathit{var} \qquad \dfrac{x\ \mathit{var}}{x\ \mathit{exp}}\,E_1}{(\lambda x.\, x)\ \mathit{exp}}\,E_2 \qquad \dfrac{y\ \mathit{var}}{y\ \mathit{exp}}\,E_1}{((\lambda x.\, x)\, y)\ \mathit{exp}}\,E_3$$

As we have seen, the expressions of the λ-calculus can be described by a grammar that can be seen as an abbreviation of the rules. So it is in general not necessary to write down such inference rules when we define the syntax of a programming language. However, most inductive definitions cannot be defined by a simple grammar. For example, let us define a judgment *size*(e, n) that states that an expression e has size n.

$$\dfrac{x\ \mathit{var}}{\mathit{size}(x, 1)}\,S_1 \qquad \dfrac{x\ \mathit{var} \qquad \mathit{size}(e, n)}{\mathit{size}(\lambda x.e, n+1)}\,S_2 \qquad \dfrac{\mathit{size}(e_1, n_1) \qquad \mathit{size}(e_2, n_2)}{\mathit{size}(e_1\ e_2, n_1 + n_2 + 1)}\,S_3$$

We prove a statement about judgments by *rule induction*. This means that for each rule, we assume the statement holds for the premises of the rule (induction hypothesis) and show that it holds for the conclusion. For example, we can prove the following lemma.

Lemma 1 *If e exp then there exist an n such that size(e, n).*

Proof: By induction on the judgment e *exp*.

Case E_1: Then $e = x$ and x *var*. But then *size*$(x, 1)$ by rule S_1.

Case E_2 **:** Then $e = \lambda x.e'$ for some e', x *var*, and e' *exp*. We can apply the induction hypothesis to the premise e' *exp* and conclude that there exists n' such that *size*(e', n'). Then we can apply rule S_2 to derive *size*$(\lambda x.e', n' + 1)$.

Case E_3**:** Then $e = (e_1\, e_2)$ for some e_i, and we have e_1 *exp* and e_2 *exp*. So we can apply the induction hypothesis twice and conclude that there exists n_1 and n_2 such that *size*(e_1, n_1) and *size*(e_2, n_2). But then we can apply the rule S_3 to conclude *size*$(e_1\ e_2, n_1 + n_2 + 1)$.

□

Again, there are two views we can take to justify rule induction. The first view is that we show that the judgments for which the statement holds are closed under the inference rules. But since the inductively-defined judgments are the smallest such set, we have shown that the judgments are included in the set for which the statement holds. The second view is to think of rule induction as an induction on derivation trees. We make a case distinction on the rule that has been used at the root and assume that the statement holds for the subtrees (induction hypothesis). If you examine the proof of Lemma 1 again, you can even few it as a program that converts a derivation tree of e *exp* into a derivation tree of *size*(e, n).

Another statement we could prove is that the size of an expression is unique. Given that we proved Lemma 1 already, we only need to show that, for a given expression e, there is at most one n such that *size*(e, n). A good way to state formulate the needed lemma is as follows.

Lemma 2 *If size*(e, n) *and size*(e, m) *then* $n = m$*.*

Again, we prove the lemma by rule induction. We have two options: We can induct on the judgment *size*(e, n) or the judgment *size*(e, m), which will not make a difference in the proof because the statement we aim to prove is symmetric.

Let's proceed by induction on *size*(e, n). When we come to the case of the Rule S_1 we know $e = x$, x *var*, and $n = 1$. But how do we proceed now? We would like to show that $m = 1$. This is intuitively clear because the only rule that applies to variables is Rule S_1. We can prove $m = 1$ by an inner rule induction on *size*(x, m) (recall that $e = x$). In the case of Rule S_1, we immediately have $m = 1$. Rule S_2 does not apply because $x \neq (e_1\, e_2)$. Similarly, Rule S_3 also does not apply. So it follows that $m = 1$.

The other cases are similar and both require an inner induction on *size*(e, m) as well.

In the future, you do not have to write out the inner induction on *size*(e, m) but instead simply apply *inversion*, which is stated by the following lemma.

Lemma 3 (Inversion) *Let size*(e, n).

1. *If* $e = var\ x$ *then* $n = 1$.
2. *If* $e = (e_1\, e_2)$ *then there exists* n_1 *and* n_2 *such size*(e_1, n_1), *size*(e_2, n_2), *and* $n = n_1 + n_2 + 1$.
3. *If* $e = \lambda x.e'$ *then there exists* n *such that size*(e', n') *and* $n = n' + 1$.

The prove proceeds by induction on *size*(e, n) as previously sketched for the case of Rule S_1.

3 Binding, Scope, and Substitution

We will first discuss abstract binding trees and then return to the λ-calculus to formally define substitution.

3.1 Abstract Binding Trees

In the following, we rigorously define substitution and related concepts like α-equivalence for the λ-calculus. However, these definitions directly generalize to all programming languages we discuss in this course. An economical way to formalize these syntax-related concepts is to abstract from a single programming language and to define them for so-called *abstract binding trees (ABTs)* [Har16]. Then the apply to every language whose syntax is defined using ABTs.

In this course, we are not going to define ABTs but we will use the ABT notation to define the syntax of languages. For example, the syntax of the λ-calculus can be defined as follows.

			Concrete	Abstract
Expressions	e	::=	$\lambda x.\, e$	$\mathrm{lam}(x.e)$
			$e_1\, e_2$	$\mathrm{app}(e_1, e_2)$

ABTs are a generalization of abastract syntax trees that add the concepts of variables and binding. In the previous grammar, we do not mention variables explicitly. ABTs automatically come with a notion of variables that range over the different *sorts* of the language. For the λ-calculus, we only have the sort of expressions and variables always stand for expressions. However, other languages have multiple sorts such as expressions and types. Then we have variables that range over expressions and other variables that range over types.

The definition of the syntactic objects (here expressions) is given by a set of *operators*. In the previous example, the operators are lam and app. Each operator comes with a fixed number of arguments and a binding structure. The binding is indicated by writing $x.e$ like in the lambda abstraction. It means that one variable (x) is bound here and the scope of the binding is e. If multiple variables are bound then we write $x_1, \dots, x_n.e$.

ABTs are useful not only for formally defining general syntactic concepts but also when specifying the syntax semantics of language. The advantage is that the scope of bindings and operators is unambiguous. Later in the semester, you will likely prefer specifying languages in ABT notation.

3.2 Free and Bound Variables

The free variables of an expression of the λ-calculus are defined by the judgment $x \in e$, which reads *variable x is free in expression e.*

$$\frac{x\ \mathit{var}}{x \in x}\,V_1 \qquad \frac{y \in e \qquad y \neq x}{y \in \mathrm{lam}(x.e)}\,V_2 \qquad \frac{x \in e_1}{x \in \mathrm{app}(e_1, e_2)}\,V_3 \qquad \frac{x \in e_2}{x \in \mathrm{app}(e_1, e_2)}\,V_4$$

Free variables are placeholders for expressions. Changing the name of a free variable changes the meaning of an expression. This can be seen if the expression is a sub-expression of a larger expression. One the other hand, we can change the name of bound variables without changing the meaning of an expression in any context (as long as we don't intrigued name clashes). Since the name of bound variables does not matter, it does generally not make sense to define something like *the bound variables of an expression*.

3.3 Substitution and Alpha-Equivalence

Through substitution we give meaning to variables. Maybe surprisingly, substitution is subtle to formalize. The main difficulty is an issue that is called *variable capture*. To understand the issue, consider for example the expression $\lambda x.y\ x$, which is a function that applies the variable y to its argument. If we substitute x for y then we get $\lambda x.x\ x$. The result we expected is a function that applies the variable x to its argument but what we got is a function that applies its argument to itself.

One way to solve the problem is to define substitution only if such a variable capture does not happen. We define the judgment $\mathrm{sub}(e_x, x, e, e')$ through the rules below. The meaning is that the expression e' is the result of replacing variable x with expression e_x in expression e.

$$\frac{}{\mathrm{sub}(e_x, x, x, e_x)}\ S_1 \qquad \frac{y \neq x}{\mathrm{sub}(e_x, x, y, y)}\ S_2$$

$$\frac{\mathrm{sub}(e_x, x, e_1, e_1') \qquad \mathrm{sub}(e_x, x, e_2, e_2')}{\mathrm{sub}(e_x, x, \mathrm{app}(e_1, e_2), \mathrm{app}(e_1', e_2'))}\ S_3$$

$$\frac{}{\mathrm{sub}(e_x, x, \mathrm{lam}(x.e), \mathrm{lam}(x.e))}\ S_4 \qquad \frac{x \neq y \qquad y \notin e_x \qquad \mathrm{sub}(e_x, x, e, e')}{\mathrm{sub}(e_x, x, \mathrm{lam}(y.e), \mathrm{lam}(y.e'))}\ S_5$$

With this approach, substitution is undefined some expressions: There does not exist an e such that $\mathrm{sub}(x, y, \lambda x.y\ x, e)$. This is however very inconvenient. For example, it is difficult to define β-reduction since we now have to address the cases in which substitution is not defined. Our goal is therefore to define a total version of the judgment sub.

The idea of the total substitution judgment is to use α-equivalence. In future lectures, we will implicitly identify α-equivalent expressions (and types). So if we talk about an expression such as $\lambda x.x$, we mean the α-equivalence class of $\lambda x.x$.

The judgment for α-equivalence is defined by the following rules. Note that the judgment $e_1 =_\alpha e_2$ is indeed an equivalence relation.

$$\frac{}{e =_\alpha e}\ A_1 \qquad \frac{e_1 =_\alpha e_1' \qquad e_2 =_\alpha e_2'}{\mathrm{app}(e_1, e_2) =_\alpha \mathrm{app}(e_1', e_2')}\ A_2 \qquad \frac{\mathrm{sub}(y, x, e, e') \qquad y \notin e}{\mathrm{lam}(x.e) =_\alpha \mathrm{lam}(y.e')}\ A_3$$

$$\frac{e =_\alpha e'}{e' =_\alpha e}\ A_4 \qquad \frac{e_1 =_\alpha e_2 \qquad e_2 =_\alpha e_3}{e_1 =_\alpha e_3}\ A_5$$

Equipped with α-equivalence we can now define the result of substitution to rename bound variables to avoid name capturing. The following rule defines a judgment $\mathrm{sub}_\alpha(e_x, x, e, e')$ that is total in the sense that for every e_x, x and e there exists an e' such that $\mathrm{sub}_\alpha(e_x, x, e, e')$.

$$\frac{\mathrm{sub}(e_x, x, e_0, e') \qquad e_0 =_\alpha e}{\mathrm{sub}_\alpha(e_x, x, e, e')}\ \mathrm{Sub}_\alpha$$

We write $[e_x/x]e$ for (the equivalence class of) an expression e' such that $\mathrm{sub}_\alpha(e_x, x, e, e')$. The substitution $[e_x/x]e$ is well-defined since all such e' are α-equivalent.

4 Beta-Reduction

Finally, we can use inference rules to define β-reduction and β-equality. The rules for β-equality are as follows.

$$\frac{}{\mathrm{app}(\mathrm{lam}(x.e), e') =_\beta [e'/x]e}\ B_1 \qquad \frac{e_1 =_\beta e_1' \qquad e_2 =_\beta e_2'}{\mathrm{app}(e_1, e_2) =_\beta \mathrm{app}(e_1', e_2')}\ B_2$$

$$\frac{e =_\beta e'}{\mathrm{lam}(x.e) =_\beta \mathrm{lam}(x.e')}\ B_3 \qquad \frac{}{e =_\beta e}\ B_4 \qquad \frac{e_1 =_\beta e_2 \qquad e_2 =_\beta e_3}{e_1 =_\beta e_3}\ B_5$$

$$\frac{e =_\beta e'}{e' =_\beta e}\ B_6$$

To define β-reduction we would only use rules $B_1, \ldots, B_3$. To define the multi-step β-reduction $e \longmapsto_\beta^* e'$ we would use only rules $B_1, \ldots, B_5$.

5 Limitations of the λ-Calculus

The λ-Calculus is arguably the most elegant programming language. If we believe Church's law then it can also express all commutable functions. So why should we even consider other programming languages? There are multiple reasons to study other languages and in particular typed languages.

Expressivity One could say that we got more than we bargained for with the λ-Calculus. We defined a seemingly simple programming language (which only provides functions) and discovered that it is expressive enough to not only express non-termination but all commutable functions and abstractions such as pairs and natural numbers. However, we set out to study such abstraction in isolation. We also want to study languages that are *normalizing*, that is, languages in which all programs terminate. Both is difficult in the λ-Calculus.

Weak Abstractions While we can express abstractions such as pairs and natural numbers in the λ-Calculus, these abstractions are not enforced. For example, we can call the function *plus* with arguments that are not Church numerals and the resulting expression can either be evaluated to a normal form or will diverge. There will be no warning or error. In this way, the λ-Calculus behaves similarly to an assembly language. On the one hand, this makes it difficult to use the λ-Calculus to implement complex software. It would simple be hard to debug large λ expressions when we only see some nonsensical result but do not get hints for finding the point at which the computation went wrong. On the other hand, it makes it difficult to reason about programs in the λ-Calculus. For example, we cannot prove by induction on the natural numbers that *plus* will always terminate (has a normal form) if its arguments terminate.

Strong Reduction β-reduction and normal (or leftmost-outermost) evaluation differ from evaluation in most programming languages. So far we have considered only evaluation strategies that are based on *strong reduction*. In strong reduction, we allow evaluation of a function without an argument as defined by Rule B_3.

$$\frac{e =_\beta e'}{\mathrm{lam}(x.e) =_\beta \mathrm{lam}(x.e')}\ B_3$$

Evaluation strategies based on *weak reduction* (like call-by-name or call-by-value, which we will discuss in the next lecture) do not allow such evaluations. As a result, all lambda abstractions $\mathrm{lam}(x.e)$ are normal forms.

It would be perfectly possible, to define the λ-Calculus using a weak reduction strategy. We would still be able to express all commutable functions. However, we would have to make some modifications. For instance, the Church numerals would not work in such a setting since, for example, the normal form a the successor of a Church numeral would not be a Church numeral. So we would have to use a different encoding like Barendregt numerals. Moreover, the Church-Rosser theorem would not hold anymore since we the order of β-reductions would matter [BLM05] and we would not have the confluence property. Finally, we would only define *a* λ-Calculus instead of *the* λ-Calculus, which was defined by Church using β-equivalence. For these reasons, the λ-Calculus with a weak reduction strategy looses some of its appeal.

Exercises

Exercise 1 *Prove by rule induction: For all e and n, if size(e, n) then e exp.*

Exercise 2 *Provide an expression e such that $\mathrm{sub}_\alpha(x, y, \lambda x.y\ x, e)$. Give a derivation tree for the judgment $\mathrm{sub}_\alpha(x, y, \lambda x.y\ x, e)$.*

References

[BLM05] Tomasz Blanc, Jean-Jacques Lévy, and Luc Maranget. Sharing in the weak lambda-calculus. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel C. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2005.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.