Chapter 5

Inductive Definitions

By now, you've hopefully got a taste of some of the kinds of models that we will build (of language syntax and semantics) in the context of a very very simple language—featuring a finite number of programs *and* results. Now let's start to develop the machinery for building more complex languages, starting with a language that at least features an *infinite* number of programs (still finite results though!). The tools we introduce here, called *inductive definitions* will be critical throughout the course, and we'll find later that they connect directly to what you may have learned about *proof by induction* in prior courses.

5.1 Inductive Definitions

Now we are going to look at a somewhat more interesting language. The language will have an infinite number of programs. This means that we can't simply enumerate all of the programs: we need a new strategy.

The way that we define our set of programs follows a pattern similar to what happens in a real language implementation. Consider how a parser for the C programming language works. It takes as input any finite *strings* (i.e., sequences) of *bytes* (of which there are $2^8 = 256$), and filters out the strings that count as legal C programs.[1]

$$\text{CP}\textsc{gms} = \{\, s \in \text{BYTE}^* \mid s \text{ is a valid C program} \,\}$$

Here BYTE is the set of bytes, and in general, we say that if S is some set, then S^* is the set of all finite sequences of elements of S (including the empty sequence, which for visibility's sake we denote by ε). So essentially, the set of C programs is defined by filtering all the valid C programs out of the set of all byte strings.

The key idea here is that we start with some basic, easily defined pre-existing set and filter it down to the subset that we want. Most mathematical definitions are based around this general idea.

We do not want to concern ourselves with as low level a representation as strings of bytes, so we start more abstractly. First, we assume some infinite set of *atomic* elements:

$$a \in \textsc{Atom}$$

Our base assumption is that there are an infinite number of Atoms in the set (not just 256), so we can always find another one if we need one.[2] Our only requirement of Atoms is that we can *tell them apart*: we don't care what they really look like, i.e. their internal structure, just whether we can tell if two of them are the same Atom or not. This is why they we call them atoms.[3] We can state this formally as a (seemingly obvious) property:

Proposition 16. $\forall a_1, a_2 \in \textit{Atom}.\, a_1 = a_2 \vee a_1 \neq a_2$.

[1]You wouldn't want your C compiler to segfault if you gave it a file of random noise, right? It should reject the program and exit gracefully.

[2]To make this concrete, it's as though we had a computer that could work with arbitrary natural numbers instead of bytes.

[3]It helps to know that early physicists thought that atoms were the most primitive non-decomposable elements in the universe. Then along came neutrons and electrons and protons...and then quarks!

For our purposes, this should be interpreted as saying "given atoms a_1 and a_2, we can determine whether they are identical or not." That reading may be a bit different than you are used to, in that it might seem like a content-less tautology. In classical logic it is, but for us it's not! These Atoms will be used to represent the primitive constructs of our languages. If you've ever studied parsing, this is analogous to your tokens. For purposes of reasoning, we'll give each atom an abstract name, written in blue, like car, or avocado-toast, and whenever we use two different abstract names, we are referring to two different atoms, e.g. car $\neq$ avocado-toast, by convention. Note that this is *very* different than referring to atoms using *metavariables* like a_1 or a_2. We cannot up-front assume that $a_1 \neq a_2$: to do so we must have that fact as an immediate assumption, or be able to prove it from other assumptions.

Then, building on top of the Atoms, we assume a set of all possible trees of Atoms.

$$r \in \textsc{Tree}[\textsc{Atom}]$$

In set theory, we can create pairs of elements of sets, subsets of sets, and so forth. Using the basic tools of set theory, we can design a representation for trees, just like we can build tree data structures in the programming language of your choice. We're not going to build up a particular set-based representation of trees in any detail, but here's a sketch of a representation that gets the job done:

1. Pick some set $\mathcal{X}$;
2. Use sets to define some representation for sequences of $\mathcal{X}$, i.e. $\mathcal{X}^*$.
3. Define a tree of $\mathcal{X}$, i.e. $\textsc{Tree}[\mathcal{X}]$ as a set of non-empty sequences of $\mathcal{X}$, where each sequence describes a path from the root node of the tree to some subtree. This implies that:
   (a) The empty tree is represented by the empty set;
   (b) Every sequence starts with the same element of $\mathcal{X}$ (the root node of the tree);
   (c) If the sequence $\langle x_1, \ldots, x_n\rangle$ is an element of the set, then so is every non-empty prefix $\langle x_1, \ldots, x_k\rangle$ for $k \geq 1$: those are the paths to the ancestors of the node x_n;
   (d) If subtrees need to be ordered, or if multiple immediate subtrees can have the same payload, then that can be represented by ornamenting the "payload" $\mathcal{X}$ with indices.

In this case $\textsc{Tree}[\textsc{Atom}]$ is the name we use for the set of all trees that have atoms at their nodes. In general, we write $\textsc{Tree}[X]$ for "trees of elements of X." Note, that this name is just a convention: we define each set of trees manually, though we could use notational definitions to make the process schematic.

For now, since we are only concerned with trees of atoms, we will just write Tree as an abbreviation.

These trees in $\textsc{Tree}[\textsc{Atom}]$ have Atoms for nodes. For convenience, we can write them in parenthesized notation, where a node is juxtaposed with a parenthesized list of subtrees. For example, $a_1(a_2(), a_3())$ is the tree with parent node a_1 and two subtrees with root nodes a_2 and a_3 respectively. We can draw this as a diagrammatic tree:

$$\begin{tikzcd}
 & \mathrm{a}_1 \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{a}_2 & & \mathrm{a}_3
\end{tikzcd}$$

For succinctness, we elide the empty parentheses after root nodes, writing instead $a_1(a_2, a_3)$. We will disambiguate where needed between the Atom a_1 and the $\textsc{Tree}[\textsc{Atom}]$ a_1, and similar for other kinds of trees.

[RG: End Maybe move to sets chapter, or abstraction chapter later]

Now, armed with a set of Trees, we define the *abstract syntax* of our language. The general definition strategy is to isolate a set of *terms*, which will be some subset of $\textsc{Tree}[\textsc{Atom}]$:

$$t \in \textsc{Term}, \text{ where } \textsc{Term} := \{\, r \in \textsc{Tree} \mid \Phi(r) \,\} \tag{5.1}$$

and Φ is some property of $\textsc{Tree}[\textsc{Atom}]$ that chooses the ones that we consider to be terms. It's our job, then, to specify that property.

Our language is called B, a language of Boolean expressions. Our intention is that the terms of our language $t \in \textsc{Term}$ are trees like true, false, and $\mathsf{if}\ (t_1, t_2, t_3)$ where t_1, t_2, t_3 are also Terms. In practice, such a language is typically described using a notation called Backus Naur Form (BNF). We're going to define our language explicitly as an inductive definition. In essence, we are going to treat BNF as concise notation for what ends up being an involved set-theoretic encoding. The unpleasant thing about this will be that the translation is involved and somewhat tedious. The nice thing about it is that there are *many* things that we can do with inductive definitions, and knowing that all of the reasoning principles that they provide will apply to syntax means that we have quite uniform reasoning principles: to define and work with syntax we use the same exact tools as we use to define semantics.

First we give convenient names to some Atoms that we'll use in our language definition:

$$\text{Assume } \{\, \mathsf{true}, \mathsf{false}, \mathsf{if} \,\} \subseteq \textsc{Atom}.$$

By our convention, these three atoms are distinct from one another. And we know by our definition of Tree[Atom] that if $(r_1, \ldots, r_n)$ is a sequence of Tree[Atom]s, then $a(r_1, \ldots, r_n)$ is a single Tree[Atom] for any $a \in \textsc{Atom}$. We focus on building new trees using the three select atoms from above.

Introduce Inductive Rules of Trees of Atoms (more generally Rules of X)

Then we carve out our set of Terms using what are called *inductive rules*.

$\boxed{\textsc{Term} \subseteq \textsc{Tree}}$

$$\frac{}{\mathsf{true}}\ (\text{rtrue}) \qquad\qquad \frac{}{\mathsf{false}}\ (\text{rfalse}) \qquad\qquad \frac{r_1 \quad r_2 \quad r_3}{\mathsf{if}(r_1, r_2, r_3)}\ (\text{rif})$$

What you see above are three *rules* which together will help us define the set of Terms. Each rule has zero or more *premises* above the horizontal bar, and one *conclusion* below it. Next to each rule is a *name* in parentheses. In essence, each of these rules can be informally read as saying "if all of the premises are members of the set, then the conclusion is as well."

Technically, a rule stands for the set of all of its instances. For example:

$$(\text{rif}) := \left\{ \frac{r_1 \quad r_2 \quad r_3}{\mathsf{if}(r_1, r_2, r_3)} \ \textbf{for } r_1 \in \textsc{Tree}, r_2 \in \textsc{Tree}, r_2 \in \textsc{Tree} \right\} \subseteq \textsc{Rule}[\textsc{Tree}[\textsc{Atom}]]$$

Really, pushing down even further, each rule instance is just a pair of a set of trees and a single tree: the premises (in no particular order) and the conclusion.

$$\textsc{Rule}[\textsc{Tree}[\textsc{Atom}]] := \mathcal{P}(\textsc{Tree}[\textsc{Atom}]) \times \textsc{Tree}[\textsc{Atom}]$$
$$\langle X, x \rangle \in \textsc{Rule}[\textsc{Tree}[\textsc{Atom}]]$$

So *really* the rule above is interpreted as follows:

$$(\text{rif}) = \{\, \langle \{\, r_1, r_2, r_3 \,\}, \mathsf{if}(r_1, r_2, r_3) \rangle \in \textsc{Rule}[\textsc{Tree}[\textsc{Atom}]] \mid r_1 \in \textsc{Tree}, r_2 \in \textsc{Tree}, r_2 \in \textsc{Tree} \,\}.$$

Each *judgment* (i.e. each premise and each conclusion) in a rule represents a Tree, and each rule stands for a number of *instances* of rules. An instance of a rule is the result of replacing the black italic *metavariables* (e.g., r_1) with a concrete Tree. Here is an example of an unsurprising instance of the (rif) rule:

$$\frac{\mathsf{true} \quad \mathsf{false} \quad \mathsf{true}}{\mathsf{if}(\mathsf{true}, \mathsf{false}, \mathsf{true})}$$

However, the rules also imply "nonsensical" rule instances, since *any* concrete Tree can be placed where there is a metavariable r. For example, if we assume that there exist Atoms named under, and while, then the following is a perfectly fine instance of the (rif) rule:

$$\frac{\mathsf{under} \quad \mathsf{while} \quad \mathsf{while}}{\mathsf{if}(\mathsf{under}, \mathsf{while}, \mathsf{while})}$$

Notice that even the nonsensical rule instance we wrote above makes sense under this intuitive interpretation: if under and while *were* Terms, then the Tree in the conclusion would also be a term; but they are not, so it is not.

We now introduce two forms of *syntactic sugar* that are common in books and papers that use this technique to define sets. Both of these notational devices will make our representation look a little prettier, but otherwise add no real content. To make the syntax of our language look a bit more like a real programming language, we will write the Term

$$\mathsf{if}(t_1, t_2, t_3)$$

using the notation

$$\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3.$$

In general we will often introduce expressions where the Atom "name" is split between the different arguments. This is often called *mixed-fix* notation (as opposed to the *prefix* notation we were using above).

The second syntactic nicety we will add is that we'll write the rules so that they refer to the name of the set that we are defining. For example, we'll rewrite the rules as follows:

$$\frac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue}) \qquad \frac{}{\mathsf{false} \in \textsc{Term}}\ (\text{rfalse}) \qquad \frac{r_1 \in \textsc{Term} \quad r_2 \in \textsc{Term} \quad r_3 \in \textsc{Term}}{\mathsf{if}\ r_1\ \mathsf{then}\ r_2\ \mathsf{else}\ r_3 \in \textsc{Term}}\ (\text{rif})$$

Going forward, it helps to remember that the "$\in$ Term" part is really just a decoration for our human purposes. Think of it like a comment in a programming language (i.e., in C you would write `/* in Term */`).

With the syntactic sugar out of the way, we still have to discuss how these rules define our set of Terms. For some insight, consider the (rtrue) rule. According to our interpretation of rules, this rule says that with no premises whatsoever, true is a Term. The (rfalse) rule is similar. Generalizing this observation, we can show in multiple steps that some Tree is a Term under no premises by building up a *derivation tree* of rule instances that as a whole has no premises, meaning that every rule instance at the top of the derivation tree is closed with a horizontal bar that has nothing above it. Here is an example of a derivation tree $\mathcal{D}$, built from rule instances, that shows that if true then false else true $\in$ Term:

$$\mathcal{D} = \frac{\dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue}) \quad \dfrac{}{\mathsf{false} \in \textsc{Term}}\ (\text{rfalse}) \quad \dfrac{}{\mathsf{true} \in \textsc{Term}}\ (\text{rtrue})}{\mathsf{if\ true\ then\ false\ else\ true} \in \textsc{Term}}\ (\text{rif})$$

What we did was instantiate the (rif) rule such that $r_1 = \mathsf{true}$, $r_2 = \mathsf{false}$, and $r_3 = \mathsf{true}$, giving:

$$\frac{\mathsf{true} \in \textsc{Term} \quad \mathsf{false} \in \textsc{Term} \quad \mathsf{true} \in \textsc{Term}}{\mathsf{if\ true\ then\ false\ else\ true} \in \textsc{Term}}\ (\text{rif}).$$

Since a derivation can't have open premises, we had to consider each of the three premises in turn, but we were able to find closed derivations for each of them using the (rtrue) and (rfalse) rules. As above, we'll use uppercase calligraphic letters like $\mathcal{D}, \mathcal{E}, \mathcal{F}$ as metavariables that stand for derivation trees, and we write both

$$\overset{\mathcal{D}}{r \in \textsc{Term}}$$

and

$$\mathcal{D} :: r \in \textsc{Term}$$

to mean that $\mathcal{D}$ is a derivation of $r \in$ Term.

Technically, we collect all of the rule instances into a *rule set*

$$\mathcal{R}_{\textsc{Term}} := (\text{rtrue}) \cup (\text{rfalse}) \cup (\text{rif}),$$

and use it to induce the set of derivations $\mathcal{D} \in \textsc{Deriv}[\mathcal{R}] \subseteq$ Tree that can be constructed using the rules $\mathcal{R}$.

In this case we can refer the set of derivations of $t \in$ Term by the name $\textsc{Deriv}[\mathcal{R}]$, to mean the set of all derivations that can be built using the rules listed in square brackets.

These derivations are the piece we've been building toward to define the set of Terms. Notice that no matter what Trees r_i we instantiate the (rif) rule with, the conclusion will be a Tree. Technically the (rtrue) and (rfalse) rules each have only one instantiation (because they have no Tree metavariables), each of which is a Tree. We can thus define the set of Terms to be the subset of Trees that have derivations according to our rules, or in formal terms:

$$\textsc{Term} = \{\, r \in \textsc{Tree}[\textsc{Atom}] \mid \exists \mathcal{D} \in \textsc{Deriv}[\mathcal{R}].\, \mathcal{D} :: r \in \textsc{Term} \,\}.$$

i.e., $\Phi(r) := \exists \mathcal{D} \in \textsc{Deriv}[\mathcal{R}].\, \mathcal{D} :: r \in \textsc{Term}$ in Equation (5.1) above.

In the particular case of defining the abstract syntax of a language, we have at our disposal the well-known Backus-Naur Form (BNF), to succinctly capture inductive definitions of syntax.

$$\begin{array}{lcl} t \in \textsc{Term}, & & \\ t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \end{array}$$

We treat grammar above as shorthand for the inductive rules that we gave above. In the future, we'll use BNF to specify our syntax, but we'll end up using inductive rules to define other sets and relations. This kind of set definition is in general called an *inductive definition*.

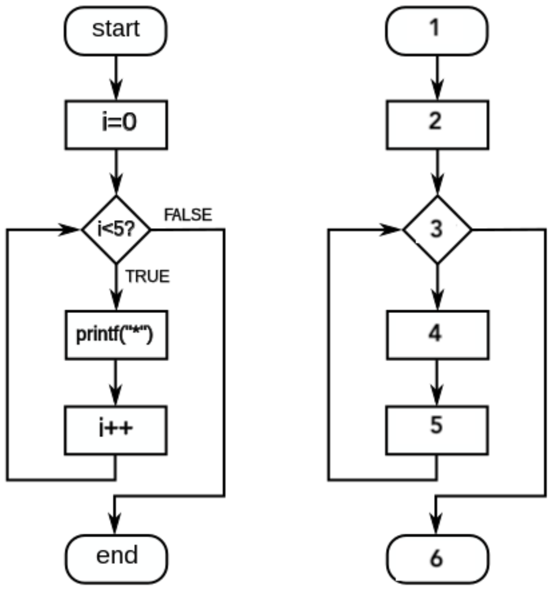
5.2 A Different-Feeling Example: Control Flow Analysis

Inductive definition is not just for syntax: it's a quite general-purpose tool. Inductive rules and their resulting derivation trees can serve as a quite precise tool for describing a set of interest, including finite sets!

A classic example of using an inductive definition to define a finite set from the world of programming languages is *control-flow analysis*. Given an imperative program that lets you assign to variables, read from variables, and test and branch, we may want to know about which instructions can eventually be reached from which other functions.

To keep things simple, we'll go old-school and write our program as a *flowchart*, a graphical representation of programs, where instructions are indicated by nodes and possibly-next instructions are indicated by edges.

Here is an example program written as a flowchart, and a corresponding *overapproximate* control-flow graph:[4]



The flowchart on the left is a *precise* program description. We can use it to reason about exactly how the program runs with specific inputs. For instance, the third bubble says specifically which direction execution will go depending on the specific value of `i` during a particular traversal of the instructions.

The control-flow graph on the left, on the other hand, is approximate. It blurs out the details of the program, simply giving each node a unique numeric identifier, and making no distinctions between outgoing

[4]From `https://en.wikipedia.org/wiki/Flowchart`

edges. So this graph answers the question "which nodes *might* be *immediately* reached from some given node." If a node no outgoing edges, than the program ends at it, as per the node marked 6. If a node has one outgoing edge, then the program will definitely follow that edge after the instruction at the node completes. But if a node has more than one edge, then execution will take one of these edges, but we do not know which. We can map the program on the left onto the control-flow graph on the right. But consider a variation of the original program, where we reverse the `TRUE` and `FALSE` branches. This new program maps onto this control-flow graph as well, since the outgoing edges do not commit to which is the `TRUE` branch and which is the `FALSE` branch. Alternatively we could replace `i < 5` with `i >= 5` and that program would also match the flowchart. Even replacing the condition with `i == i` works. This one is particularly interesting because in the concrete program we know exactly which edge will always be followed, but our control graph correctly *overapproximates* the program's behaviour, in the sense that if one node can directly reach another node in practice, then that will be evident in the control flow graph, but the control-flow graph may have edges that are never actually followed in practice. If we ignore the details of the instructions, then this is the most precise information that remains.

We can represent the control-flow graph in set theory, and use that to reason approximately about program execution. Since the control-flow graph numbers its nodes from 1 to 6 from top to bottom, we can represent the nodes and edges of the graph with finite sets:

$$n \in \textsc{Node} := \{\, 1,2,3,4,5,6 \,\}$$
$$e \in \textsc{Edge} := \{\, \langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle, \langle 3,6\rangle, \langle 4,5\rangle, \langle 5,3\rangle \,\}$$

We represent nodes as numbers and edges as some subset of $\textsc{Node} \times \textsc{Node}$, the set of all pairs of nodes. So these two sets represent the basic control structure of a particular program. We model $\textsc{Node}$ specifically because we might write a program with an unreachable instruction, which would not appear in any edges.

The above sets were defined extensionally, and we can do this manually for any program (or write a program to do it), but it would be nice to *use* these definitions to define a *possible reachability* relation $(\bullet \langle\mathrm{R}\rangle \bullet) \subseteq \textsc{Node} \times \textsc{Node}$, such that $n_1 \langle\mathrm{R}\rangle n_2$ means that if program execution arrives at node n_1, then future program execution *may* arrive at n_2, in 1 or more steps.[5]

Here is an inductive definition of possible reachability:

$\boxed{(\bullet \langle\mathrm{R}\rangle \bullet) \subseteq \textsc{Node} \times \textsc{Node}}$ **Possible Reachability**

$$\frac{}{n_1 \langle\mathrm{R}\rangle n_2}\ (\text{incl}) \quad \langle n_1, n_2\rangle \in \textsc{Edge} \qquad\qquad \frac{n_1 \langle\mathrm{R}\rangle n_2 \quad n_2 \langle\mathrm{R}\rangle n_3}{n_1 \langle\mathrm{R}\rangle n_3}\ (\text{trans})$$

The (incl) rule says that if n_1 may *immediately* reach n_2 then n_1 may reach n_2. The latter is a generalization of the former. The (trans) rule says that if n_1 may reach n_2, then n_1 may also reach any node that n_2 may reach. So n_2 can be a stopover.

Formally, the (trans) rule represents a ruleset that is similar to previously-seen ones:

$$(\mathit{trans}) := \{\, \langle \{\, \langle n_1, n_2\rangle, \langle n_2, n_3\rangle \,\}, \langle n_1, n_3\rangle\rangle \ \mathbf{for}\ n_1, n_2, n_3 \in \textsc{Node} \,\}$$

The (incl) rule, though, adds a new feature, called a *side-condition.* a side-condition is a proposition, written to the side of a rule, which imposes a restriction on which instances of the rule are accepted. This particular rule can be written as a set expression as follows:

$$(\mathit{incl}) := \{\, R \in \mathcal{R} \mid \exists X \in \mathcal{P}(\textsc{Node} \times \textsc{Node}).\, R = \langle X, \langle n_1, n_2\rangle\rangle \wedge \langle n_1, n_2\rangle \in \textsc{Edge} \,\}$$
$$\text{where}$$
$$\mathcal{R} := \{\, \langle \emptyset, \langle n_1, n_1\rangle\rangle \ \mathbf{for}\ n_1, n_2 \in \textsc{Node} \,\}$$

As described above, the side-condition in a rule is a different mechanism from a premise. A premise must be an element of the universe/superset $\textsc{Node} \times \textsc{Node}$, whereas a side-condition can be *any* proposition, possibly (in fact probably) referring to components of the rule, but also pre-existing sets, like $\textsc{Edge}$.

This distinction is often misunderstood for two reasons:

[5]The notation for possible reachability is inspired by the notation $\diamond\Phi$ from modal logic, which means "Φ possibly holds".

1. Our intuitive logical reading of side-conditions is often similar to the reading of premises: "if the premises and the side-conditions hold then the conclusion holds"; so side-conditions and premises *of the rule* are both read as premises *of the implied logical implication.*

2. Side-conditions are often typeset above the horizontal bar, rather than to the side, just like premises.

However, premises and side-conditions play distinct roles in inductive definitions, and should be treated separately. We'll discuss this more especially when we get to reasoning principles.

Since $\textsc{Node} \times \textsc{Node}$ is a finite set, we *could have* defined the relation using an extensional set expression. So why bother with induction? One particularly compelling reason is that we can modify our program, or replace it entirely, by redefining $\textsc{Node}$ and $\textsc{Edge}$, and this definition will yield a new updated relation. So these rules provide a transparent and precise specification, a *description*, of what we mean by "possible reachability", independently of the details of the particular program about which we are reasoning.

Computationally speaking, we can use this definition as guidance to implement an algorithm for computing possible reachability for *any* program. In fact, we could transliterate these rules nearly exactly as a Datalog[6] program, and run it to compute the reachability relation.

5.2.1 Necessary Reachability

Now that we have "possible reachability", what about "necessary reachability"? Intuitively speaking, that should be a relation $(\bullet\ [\mathrm{R}]\ \bullet) \subseteq \textsc{Node} \times \textsc{Node}$ that describes that if execution arrives at one node, then execution *will necessarily* arrive at some other node in the future: execution cannot avoid the second node.[7]

As it happens, we cannot *directly* define this relation inductively, but we can use induction to help us define it.[8] The strategy is:

1. Define a "possibly won't reach" relation $(\bullet\ \langle !\mathrm{R}\rangle\ \bullet) \subseteq \textsc{Node} \times \textsc{Node}$

2. "must reach" is the inverse of "possibly won't reach": $(\bullet\ [\mathrm{R}]\ \bullet) := (\textsc{Node} \times \textsc{Node}) \setminus (\bullet\ \langle !\mathrm{R}\rangle\ \bullet)$. The intuition is that given some node n_1, any node n_2 that is impossible to avoid is necessarily reachable.

Here the inductive definition promised in step 1:

$\boxed{(\bullet\ \langle !\mathrm{R}\rangle\ \bullet) \subseteq \textsc{Node} \times \textsc{Node}}$ **Possible Non-Reachability**

$$\frac{}{n_1\ \langle !\mathrm{R}\rangle\ n_2}\ (\text{term}) \quad \forall n \in \textsc{Node}.\ \langle n_1, n\rangle \notin \textsc{Edge} \qquad\qquad \frac{n_2\ \langle !\mathrm{R}\rangle\ n_3}{n_1\ \langle !\mathrm{R}\rangle\ n_3}\ (\text{extend}) \quad \begin{array}{l}\langle n_1, n_2\rangle \in \textsc{Edge}\\ n_2 \neq n_3\end{array}$$

The (term) rule tells us that any terminal node may not reach any node whatsoever, because it will surely never reach any other node (a stronger property). Then, (extend) says that one node that transitions some other node may not reach any node that the latter node may not reach (except for itself). This rule is written with two side-conditions, which we could have equivalently written as their conjunction.

[6] `https://en.wikipedia.org/wiki/Datalog`

[7] The notation for necessary reachability is inspired by the notation from modal logic, which means "Φ necessarily holds".

[8] It *is* possible to directly define the relation, using a technique we study later.

Bibliography

P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C.7, pages 739–782. North-Holland, 1977.

J. Avigad. Reliability of mathematical inference. *Synthese*, 2020. doi: 10.1007/s11229-019-02524-y. `https://doi.org/10.1007/s11229-019-02524-y`.

F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

J. Bagaria. Set theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, winter 2014 edition, 2014. URL `http://plato.stanford.edu/archives/win2014/entries/set-theory/`.

R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL `http://doi.acm.org/10.1145/3182657`.

H. P. Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

A. Bauer. Proof of negation and proof by contradiction. Mathematics and Computation Blog, March 2010. `http://math.andrej.com/2010/03/29/proof-of-negation-and-proof-by-contradiction/`.

L. Crosilla. Set Theory: Constructive and Intuitionistic ZF. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2020 edition, 2020.

M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(89)90069-8. URL `http://www.sciencedirect.com/science/article/pii/0304397589900698`.

M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 1st edition, 2009.

J. Ferreirós. The early development of set theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2019 edition, 2019. URL `https://plato.stanford.edu/archives/sum2019/entries/settheory-early/`.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563. URL `http://doi.acm.org/10.1145/512529.512563`.

P. R. Halmos. *Naive Set Theory*. Springer-Verlag, first edition, Jan. 1960. ISBN 0387900926.
A classic introductory textbook on set theory.

P. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. URL `http://www.cs.cmu.edu/%7Erwh/plbook/1sted-revised.pdf`.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124 (2):103–112, Feb. 1996. ISSN 0890-5401.

G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, UK, 1987. Springer-Verlag.

R. Krebbers. *The C standard formalized in Coq.* PhD thesis, Radboud University Nijmegen, December 2015. URL `https://robbertkrebbers.nl/thesis.html`.

C. Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1): 161–171, 1921. doi: 10.4064/fm-2-1-161-171. `https://web.archive.org/web/20190429103938/http://matwbn.icm.edu.pl/ksiazki/fm/fm2/fm2122.pdf`.

P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964. doi: 10.1093/comjnl/6.4.308. URL `https://doi.org/10.1093/comjnl/6.4.308`.

X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, Feb. 2009. ISSN 0890-5401.

P. Maddy. Believing the axioms. I. *The Journal of Symbolic Logic*, 53(02):481–511, 1988.
An interesting (though complicated) analysis of why set theorists believe in their axioms.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

J. H. J. Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, Feb. 1969. URL `http://hdl.handle.net/1721.1/64850`.

A. M. Pitts. *Nominal sets: Names and symmetry in computer science.* Cambridge University Press, 2013.

G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004a. doi: 10.1016/j.jlap.2004.03.009. URL `http://dx.doi.org/10.1016/j.jlap.2004.03.009`.

G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004b.

B. Popik. "pull yourself up by your bootstraps". Weblog entry, September 2012. `https://www.barrypopik.com/index.php/new_york_city/entry/pull_yourself_up_by_your_bootstraps/`.

E. Reck and G. Schiemer. Structuralism in the Philosophy of Mathematics. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Spring 2020 edition, 2020.

D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4): 15:1–15:41, May 2009. ISSN 0164-0925.

W. Sieg and D. Schlimm. Dedekind's analysis of number: Systems and axioms. *Synthese*, 147(1):121–170, Oct 2005.

J. Spolsky. The law of leaky abstractions. Joel on Software Blog, November 2002. `https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/`.

G. L. Steele. Debunking the "expensive procedure call"" myth or, procedure call implementations considered harmful or, lamdba: The ultimate goto. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977. URL `http://dspace.mit.edu/handle/1721.1/5753`.

S. Stenlund. Descriptions in intuitionistic logic. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundations of Mathematics*, pages 197 – 212. Elsevier, 1975. URL `http://www.sciencedirect.com/science/article/pii/S0049237X08707328`.

M. Tiles. Book Review: Stephen Pollard. Philosophical Introduction to Set Theory. *Notre Dame Journal of Formal Logic*, 32(1):161–166, 1990.
A brief introduction to the philosophical issues underlying set theory as a foundation for mathematics.

C. Urban and M. Norrish. A formal treatment of the Barendregt variable convention in rule inductions. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding*, MERLIN '05, page 25–32, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930728. doi: 10.1145/1088454.1088458. URL `https://doi.org/10.1145/1088454.1088458`.

C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, CADE-21, page 35–50, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540735946. doi: 10.1007/978-3-540-73595-3_4. URL `https://doi.org/10.1007/978-3-540-73595-3_4`.

D. van Dalen. *Logic and structure (3. ed.).* Universitext. Springer, 1994. ISBN 978-3-540-57839-0.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, Nov. 1994.

B. Zimmer. figurative "bootstraps". email to linguistlist mailing list, August 2005. `http://listserv.linguistlist.org/pipermail/ads-l/2005-August/052756.html`.