Rhys Braginton Pettee Olsen
<rho19958@colorado.edu>

We express basic statements through *judgments*.

I have colored the syntax to clarify features of these statements. Note that it's possible to define more structures in a grammar, which would necessitate more colors! I am keeping this language small.

The meta-language is colored blue. For our purposes, we assume statements in the meta-language are well-formed and coherent (already make sense either to a human or to a computer). These define the semantics of our language (what the language means in terms of something we know). We have a choice in defining our meta-language, though a mixture of Backus–Naur-like syntax and plain old arithmetic with a suitable set theory are common choices.

Non-terminal expressions inside our language are colored green. Terminals (discrete syntactic units that cannot be reduced further) are colored gold.

Here are some examples of grammatical statements. Note that all pieces of blue text are idiomatic expressions.

$e : \tau$	The expression e is of type τ (concrete example: "Hello" : `String`)
$e ::= n \mid e * n$	The expression e is defined to be either as an n (which we'd have to define elsewhere) or as $(e * n)$: a left-recursive expression.
$n ::= 0 \mid \text{succ}(n)$	The expression n is defined to be either the value 0 in our language or a successor of some other n.
$n \geq 0$	The expression n is greater than equal to 0.
$n \Downarrow 1$	The number n evaluates to 1.

With these judgments, it's now possible to define *rules* that show assertions as consequences of other assertions. In the traditions of most logics, rules are written (Premise $\Rightarrow$ Conclusion). In type theory, a rule assumes the following quotient-like form:

$$\frac{\text{Premise}}{\text{Conclusion}}$$

Note that a premise can have multiple terms, like this:

$$\frac{\text{Premise}_1 \quad \text{Premise}_2 \quad \ldots \quad \text{Premise}_n}{\text{Conclusion}}$$

This would be more cumbersome to write in (Premise $\Rightarrow$ Conclusion) notation. If the premise is empty, the rule is called an *axiom* (something we know to be true); otherwise, it's called a *proper rule*.

As an exercise, let's set up the natural numbers.

Rather than assume the naturals as part of the meta-language, let's define a non-terminal natural inductively:

$$n \ ::= \ 0 \mid \mathrm{succ}(n)$$

Above: the type n is defined to be either 0 (the "base natural") or a successor element $\mathrm{succ}(n)$. This inductive definition is also known as the "Peano axioms". It's one way to set up natural numbers.

$$n ::= 0 \mid \text{succ}(n)$$

In terms of the language we're defining, our grammar for naturals will suffice to express any natural in the language. But suppose we want to represent the correspondence between our inductive definition and meta-language arithmetic. We need operational semantic judgments for this, namely:

$$\textit{Base Case}: \frac{}{0 \Downarrow 0} \qquad \textit{Induction Hypothesis}: \frac{n \Downarrow x}{\text{succ}(n) \Downarrow x + 1}$$

Our base case says that the terminal 0 will assume the value the arithmetic value 0. Note that these aren't the same! The 0 before our arrow is the token "0" in our defined language, while the 0 after is the number in our meta-language.

Our induction hypothesis says that if token n assumes arithmetic value x, then its successor element, $\text{succ}(n)$, will assume arithmetic value $x + 1$.

Our arithmetic semantics for natural numbers work by induction and are valid for all $n \in \mathbb{N}$.

$$n ::= 0 \mid \mathrm{succ}(n)$$

$$\frac{}{0 \Downarrow 0}$$

$$\frac{n \Downarrow x}{\mathrm{succ}(n) \Downarrow x + 1}$$

Consider the following derivation of the value of $\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0))))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \Downarrow 0}}{\mathrm{succ}(0) \Downarrow 0 + 1 = 1}}{\mathrm{succ}(\mathrm{succ}(0)) \Downarrow 0 + 1 + 1 = 2}}{\mathrm{succ}\left(\mathrm{succ}(\mathrm{succ}(0))\right) \Downarrow 0 + 1 + 1 + 1 = 3}}{\mathrm{succ}\left(\mathrm{succ}\left(\mathrm{succ}(\mathrm{succ}(0))\right)\right) \Downarrow 0 + 1 + 1 + 1 + 1 = 4}$$

Thus we find that $\mathrm{succ}\left(\mathrm{succ}\left(\mathrm{succ}(\mathrm{succ}(0))\right)\right) \Downarrow 4$.

Note that our derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \Downarrow 0}}{\mathrm{succ}(0) \Downarrow 0+1=1}}{\mathrm{succ}(\mathrm{succ}(0)) \Downarrow 0+1+1=2}}{\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0))) \Downarrow 0+1+1+1=3}}{\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0)))) \Downarrow 0+1+1+1+1=4}$$

$$n ::= 0 \mid \mathrm{succ}(n)$$

$$\dfrac{}{0 \Downarrow 0}$$

$$\dfrac{n \Downarrow x}{\mathrm{succ}(n) \Downarrow x+1}$$

can also be written this (more traditional) way:

$$0 \Downarrow 0 \Rightarrow \mathrm{succ}(0) \Downarrow 0+1=1 \Rightarrow \mathrm{succ}(\mathrm{succ}(0)) \Downarrow 0+1+1=2 \Rightarrow \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0))) \Downarrow 0+1+1+$$

$$1=3 \Rightarrow \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0)))) \Downarrow 0+1+1+1+1=4$$

Hopefully, the benefits of the quotient-like notation for rules is steadily becoming more obvious.

Our Peano axiomization of the natural numbers allow us to construct inductive proofs on naturals.

Suppose that we wish to show the property $P(n)$ holds for all $n \in \mathbb{N}$. It then suffices to produce the following two rules:

$$\textit{Base Case}: \frac{}{P(0)} \qquad \textit{Induction Hypothesis}: \frac{P(n)}{P(\text{succ}(n))} \textit{ or } \frac{P(n)}{P(n+1)}$$

(*Your choice of notation*)

As you'll see soon, however, this rule form is *much* more powerful than this and generalizes to induction over many kinds of structures.

A simple grammar for a list of integers through type judgments:

n Integer $\in \mathbb{Z}$ n is the type of integers.

List ::= Empty
| Node(elem : n, rest : List)

Above: the type List is defined to be either Empty (the empty tree) or a Node that contains an element elem of type n, and a child list rest, which is also a List (this type is recursive). Equivalently, we can write this:

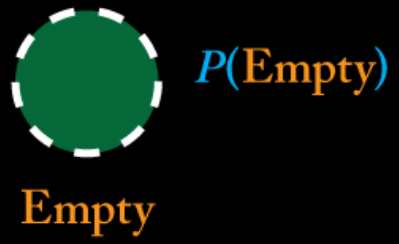
Empty : List

Node(elem : n, rest : List) : List

We can define induction over our lists with the following induction rules: suppose that we wish to show the property $P(\text{List})$ holds for several lists. It then suffices to produce the following two rules:

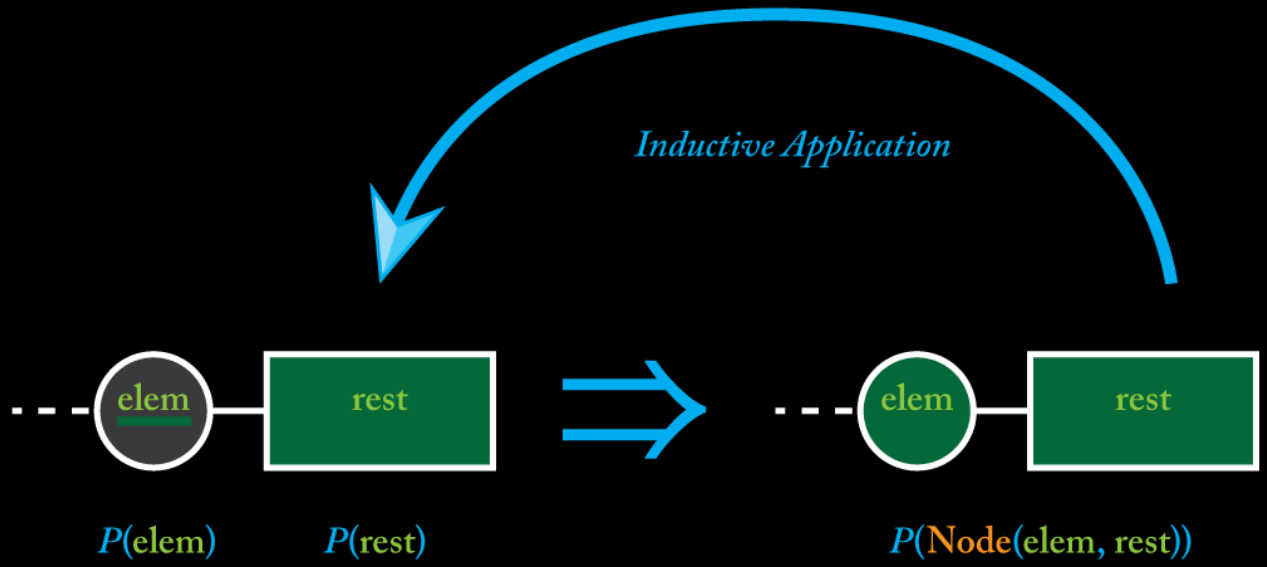
Base Case:

$$\frac{}{P(\text{Empty})}$$



Induction Hypothesis:

$$\frac{P(\text{elem}) \;\&\; P(\text{rest})}{P(\text{Node}(\text{elem}, \text{rest}))}$$



A simple grammar for a tree of integers through type judgments:

n Integer $\in \mathbb{Z}$ n is the type of integers.

$$\begin{aligned}\text{Tree} ::= &\ \text{Empty} \\ &\mid \text{Node}(\text{elem} : n, l : \text{Tree}, r : \text{Tree})\end{aligned}$$

Above: the type Tree is defined to be either Empty (the empty tree) or a Node that contains an element elem of type n, and two children l and r, both of which are also Trees (this type is recursive). Equivalently, we can write this:

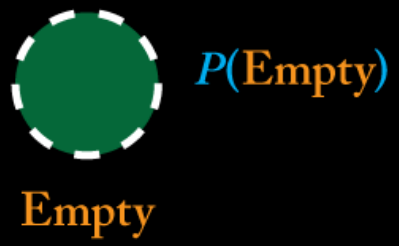
$$\text{Empty} : \text{Tree}$$

$$\text{Node}(\text{elem}: n, l: \text{Tree}, r: \text{Tree}) : \text{Tree}$$

We can define induction over our trees with the following induction rules: suppose that we wish to show the property $P(\text{Tree})$ holds for several tress. It then suffices to produce the following two rules:

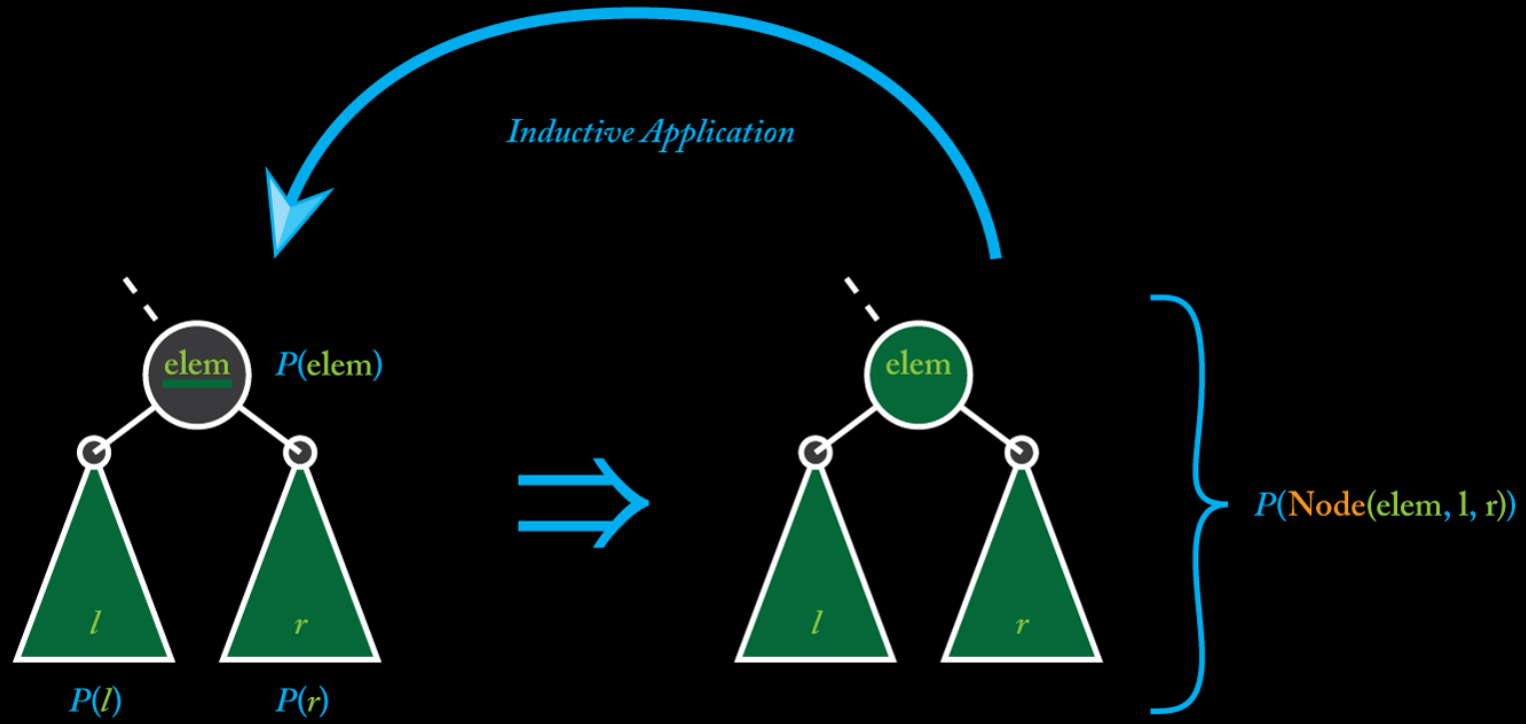
Base Case:

$$\frac{}{P(\text{Empty})}$$



Induction Hypothesis:

$$\frac{P(\text{elem}) \;\&\; P(l) \;\&\; P(r)}{P(\text{Node}(\text{elem}, l, r))}$$



Here's an example: let's set up some rules for our simple tree language to see if our tree is a *natural tree*: that is, a tree whose nodes are all natural numbers.

Base Rule:

$$\frac{}{\text{isNatural}(\text{Empty})}$$

Inductive Rule:

$$\frac{\text{isNatural}(a_1 : \text{Tree}) \quad \text{isNatural}(a_2 : \text{Tree}) \quad \text{elem} \geq 0}{\text{isNatural}(\text{Node}(\text{elem}, a_1, a_2))}$$

The form of induction generalizing all of these inductive forms is called *structural induction*.

Inductive Rule:

$$\frac{P(a_1)\ \&\ P(a_1)\ \&\ \cdots\ \&\ P(a_k)}{P(a)}$$

The rule says that whenever elements $a_1, a_2, \ldots, a_k$ of some larger structure (for your choice of "larger structure") a_k satisfy some property P, then the entire structure a_k satisfies the same rule.

This form of induction can be pushed still further to generalize over *judgments* or statements in the meta-language.

Rule induction lets you show that

Inductive Rule:

$$\frac{P(a_1)\ \&\ P(a_1)\ \&\ \cdots\ \&\ P(a_k)}{P(a)}$$

The rule says that whenever elements $a_1, a_2, \ldots, a_k$ of some larger structure (for your choice of "larger structure") a_k satisfy some rule P, then the entire structure a_k satisfies the same rule.

One final bit of foundation:

An environment E sets up a series of rules that we assume and can refer to in judgments. E can be anything from the binding of expressions to variables to additional typing constraints in the language. To say that some judgment J is true in an environment E, we write $E \vdash J$. For instance, to write that in a value environment E, the expression e assumes a value v, we write $E \vdash e \Downarrow v$.

We have enough background to ask the question of ultimate relevance:

Why the hell would I use this?

Computer scientists want to show many different kinds of features about the programs they write, including (but not limited to):

- Bounds on runtime performance
  - Their programs are well-typed (your compiler tells you if your programs make sense; *all* state changes are valid and don't get the program *stuck* in any catastrophic sense).
- Their programs don't leak memory or resources (a performance *and* security problem).
- Their concurrent programs are free of *race conditions*, which are damned near impossible to find with test cases and hard to debug (anyone still have PTSD from Computer Systems?).

Remember that in some cases, knowing these things can be the difference between **life and death**.

Chang's Syntactic Grammar for JavaScripty (note a slightly different choice of notation to my examples but vast similarities):

expressions e ::= x | n | b | undefined | *uop e1* | *e1 bop e2* |
| e_1 ? e_2 : e_3 | const $x = e_1$; e_2 | console.log(e_1)

values e ::= n | b | undefined

unary operators uop ::= − | !

unary operators bop ::= , | + | − | * | / | < | <= | > | >= | ===
| ! == | && | ||

booleans b ::= true | false

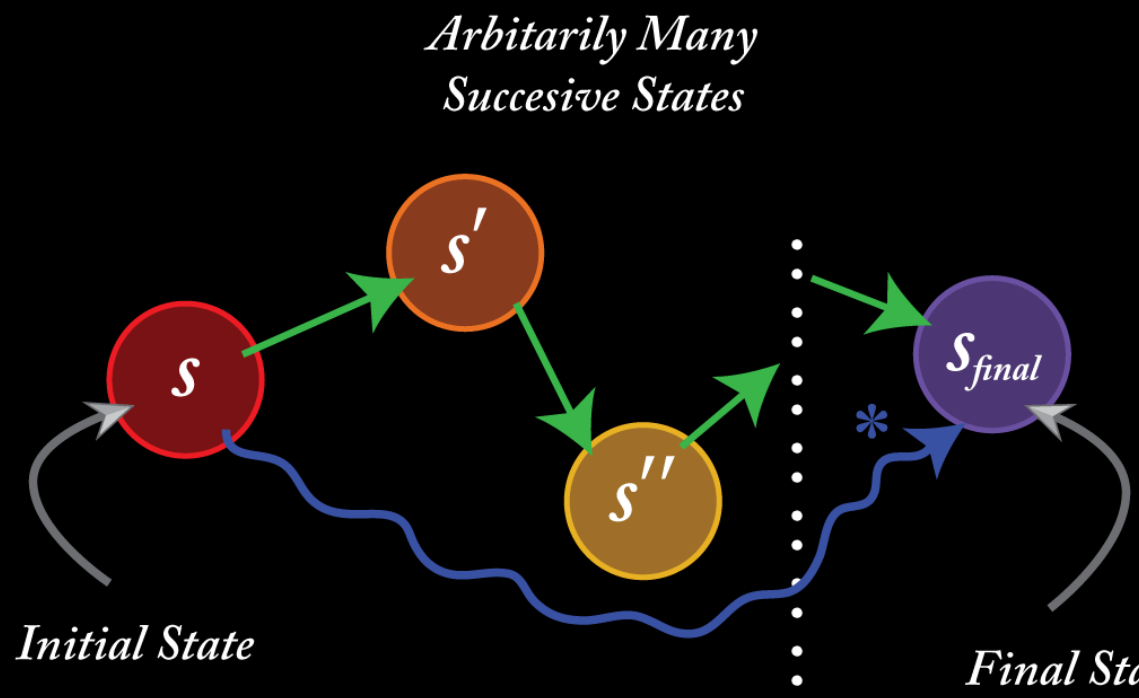
variables x; *numbers (doubles) n*

But as we saw in the Peano arithmetic example, the *syntax* of the language usually isn't enough. We need *semantics* for what terms in our language do and mean.

To reason about how programming languages are evaluated, we specify *dynamics*, or operational semantics.

Prof. Chang defines two kinds of operational semantics: big-step semantics, or natural semantics, which define what an expression will eventually become, and small-step semantics, or structural semantics, which define how a single reduction, like left-associative addition, is carried out. Big-step semantics correspond to how the language is typically interpreted (several "steps" or state-changes at once), while small-step semantics correspond to doing a single reduction.

Big- and small-step semantics are easy to characterize by each step as a transition from one evaluation state to another. The initial state is the program (or sub-program) as received by the interpreter, while the final state corresponds to what an expression eventually becomes. While it's impossible to describe every final value of every program in a language like JavaScripty in one rule, you *can* describe several collections of JavaScripty expressions this way.



Big-Step Semantics *vs* Small-Step Semantics

Advantages of big-step semantics:

- Usually corresponds with interpreter/compiler evaluation.
- Less work: you only have to show where a term is headed instead of how to get there step by step.

Advantages of small-step semantics:

- Effectively captures concurrency, non-determinacy and runtime errors.
- More explicit about certain evaluation dynamics (associativity, order of operations, etc.).
- More can be proven about programs that fail to halt (loop infinitely, get stuck in a state, etc.).

Evaluations in small-step semantics are usually denoted $\longrightarrow$. Evaluations in big-step semantics are usually denoted $\longrightarrow^*$ or $\Downarrow$ (as we've seen).

Important note: for the same language, big- and small-step semantics must *always* agree on what a program will do! If they don't, they're specifying two different languages.

Prof. Chang defines a host of big- and small-step rules for JavaScripty—to many for me to list here. Here's an example of a big-step rule:

$$\text{EvalIfTrue: } \frac{E \vdash e_1 \Downarrow v_1 \qquad \text{true} = \text{toBoolean}(v_1) \qquad E \vdash e_2 \Downarrow v_2}{E \vdash e_1 ? e_2 : e_3 \Downarrow v_2}$$

This rule says that if in an environment E eventually evaluates expression e_1 to v_1 and expression e_2 to v_2, if v_1 coerced to a Boolean (note that toBoolean is a function defined in the meta-language for type coercion) is false, then in the same environment E, the expression $e_1 ? e_2 : e_3$ assumes the value v_2.

Here's an expression of the same rule in a JavaScripty-like meta-language:

```
environment E {
  if ((e1 = v1) && (e1 = v1) && (v1 == true)) {e1 ? e2 : e3 = v2 ;}
}
```

Here's an example of a small-step rule:

$$\text{DoOrTrue: } \frac{\text{true} = \text{toBoolean}(v_1)}{v_1 \;||\; e_2 \longrightarrow v_1}$$

This rule says that if the value v_1, when coerced to a Boolean, is true, then the expression $v_1 \;||\; e_2$ reduces in one step to v_1. This rule defines short-circuiting on the JavaScripty "or" operator.

Note that in contrast to the big-step rule:

- This rule isn't bound to any environment: it makes no such environmental assumptions and is true in any context if the premise is satisfied.
- Because only the left term of the operator is a value, this rule is left-associative; the rule does not introduce any ambiguity. In general, all small-step rules intrinsically capture evaluation order.
- The premise of the rule can always be checked and the conclusion of the rule always evaluated in constant [$O(1)$] time. Thus this rule directly captures a cost semantics that will tell us how expensive reduction through this rule is *in general*.