One of the major endeavours of modern programming languages research is in formalizing our understanding of how language constructs behave on their own and in interaction with each other. We are interested in formalizing the meanings of the various elements of the programming language, and ultimately the language itself. This discipline is called formal semantics. In studying formal semantics, our goal is to formulate a model capable of precisely describing the behavior of every program in a given language. Such a model provides us tools to prove program correctness, program termination, or other critical properties. Furthermore, we can also use such a model to prove certain properties of the language itself, to show equivalence of programs in different programming languages. The knowledge gained could even help build compilers and interpreters to produce more efficient implementations of the language.

In Module 2, we already said that a mathematical model for the programming language itself would provide a succinct and precise representation of the core mechanics and be able to prove certain properties. However, we have not yet given a definition of the behavior of the program in mathematical logic. For example, we described AOE strategy in plain English as “always choosing the leftmost, innermost redex that is not in an abstraction”. A prosaic definition like this will usually not suffice. We would like to (hopefully) have a small set of (usually) syntax-directed rules that describe the elements of a language’s syntax in a formal, mathematical setting.

A semantic model usually comes with a set of observables, which describes the valid outputs of the model. Such outputs could be the produced value by following a number of rules, the set of all types of the language, or simply whether a program returns an error or not. In each case, we would choose an appropriate set of observables, and then build a semantic model to match.

In this course’s formal semantics, we are primarily going to study the operational semantics of various programming languages. Operational semantics is the semantics for specifying how a program executes and possibly how to extract a result from it. More specifically, as our main goal for the course is to understand how programs interacts with data and code in various programming paradigms, we are more concerned at the small-step operational semantics of such paradigms. A small-step operational semantics builds an imaginary “machine” and succinctly describes how this machine might take individual steps, rather than describing the entire computation in one step.

As mentioned, we have already introduced the operational semantics of \(\lambda\)-calculus in Module 2 in an informal way. The goal of this module is to revisit \(\lambda\)-calculus with formal, small-step, operational semantics, demonstrate that it can be used to prove some properties, and show ways of extending it with added primitives.

1 Semantics and Category Theory

We will be describing the reduction steps in our programming languages by formally describing an arrow \((\rightarrow)\) operator, which maps a program state to the “next” program state. This is described within the context of category theory, in which our language is a category, and \(\rightarrow\) is a morphism over that category. You are not expected to have seen these terms before, so we will briefly introduce category theory here. This course will not look deeply into category theory, but, since programming language semantics are described as categories in category theory, knowing some of the language from category theory will help to contextualize formal semantics.
In CS courses, you have undoubtedly seen \textit{sets} and \textit{set theory}. \textit{Group theory} extends set theory by describing groups, which are sets that correspond to, and are described by, certain \textit{axioms} (defined for given groups), and generalizes the language of functions between and within groups. It is from group theory that words such as \textit{isomorphic} and \textit{homomorphic} arise, to describe certain properties of these functions. Category theory abstracts beyond this by describing categories which may not obviously be describable as groups or sets; in particular, one can describe entire mathematical calculi as categories. For instance, one can describe set theory itself as a category, with expressions in set theory as the \textit{objects} described, and the functions being equivalences (or reductions, or expansions, etc) between them; for instance, the resolution of the expression \{1, 2\} ∪ \{3, 4\} to \{1, 2, 3, 4\} is a function (probably a function that more generally describes the resolution of all expressions of the form \(X \cup Y\)). We call these functions between objects \textit{morphisms}.

Where category theory becomes particularly relevant is its abstraction over itself. In the language of category theory, we can describe an entire category—which, recall, can be a mathematical calculi, a language—as an object within the category of categories, and describe a morphism mapping that category to another category. For instance, it is possible to reversibly map the language of sets to the language of predicate logic. By doing so, whole bodies of mathematical literature and proofs can be mapped into other contexts, allowing for a sort of generalization of proofs that was not possible before category theory. Mappings between categories like this are called \textit{functors}, but they’re really just morphisms given a funny name because mathematicians aren’t as accustomed to this kind of abstraction as we are.

\begin{quote}
\textbf{Aside:} We introduced categories as “not obviously be describable as groups or sets”. In fact, since category theory describes how categories can be mapped to other categories, and category theory is itself a category, it is perfectly possible to map any category to \textit{some} kind of set: for instance, we describe the set of all valid program states. Hence “not obviously”, rather than “not”.
\end{quote}

We describe our own languages in terms of a morphism, which maps program states to program states. At this point we will describe program states in purely the same syntax as the language itself, but in future modules, we will add extra syntax for additional state; in either case, the syntax is our calculus. Morphisms are usually shown as arrows, often with text to specify exactly which morphism is being described; in fact, we’ve already seen a few morphisms, such as \(\beta\), but didn’t call them such at the time.

When describing morphisms, we are free to use other categories to do so. For instance, we could say that \((x + y \to z)\) if \((x + y = z)\), and we are now describing our language in terms of the language of arithmetic. It’s important to be clear, in such cases, what language is being described and what language is being used; for instance, in this case, it’s important to realize that the first + was part of the syntax of our language, and the second + was part of the syntax of arithmetic. Usually, this sort of mapping is too narrow to be called a functor—we haven’t actually described a complete rewriting of our language in terms of arithmetic, merely a step within our language—but in some cases, languages are actually described in terms of functors, by describing how to rewrite one language into another language. In fact, that’s a compiler, and proving things about compiler correctness involves proving the functor correct.

We won’t go any more deeply than this into category theory, because we’re not usually proving more broad things about categories. We’re only narrowly interested in proving things about our particular languages. But, you should now have some idea of the formal underpinnings we’re using to describe languages: programming language semantics are not an ad hoc invention, they are described in the language of categories.

\section{2 Semantics and Reality}

But is there anything to guarantee that the semantics we formally model are the same as the semantics implemented in real programming language implementations? The short answer is “usually not”.

There are systems that make formal semantics executable, but the resulting interpreters are usually unusably slow. The purpose of these systems is to have a ground truth for writing test cases. Even that is imperfect, however, since it’s always possible to write formal semantics which are consistent, but not what you intended.

There are also aspects of real implementations which are usually intentionally ignored in formal semantics. For instance, we won’t discuss what happens when the program state is too large to hold in memory. And, in later
modules, we won’t discuss garbage collection, even though it’s crucial to a correct implementation of many systems.

In a much later module on systems programming, we will discuss one counterexample, which successfully uses a formally-defined version of C both as a formal semantics and as a real compiler.

In reality, it’s impossible to prove, in the mathematical sense of the word, anything about how a program will behave on a real system. Aside from the pitfalls mentioned above, no formal system can model “a disgruntled employee took a pickaxe to my server”. Since we’re proving things about abstract calculi, rather than a real implementations, we can actually prove things, with all the rigor of mathematics. But since we’re not proving things about a real implementation, it is the job of the designer of a formal semantics to argue that the semantics correctly reflects the design of the language, and/or of some implementation of the language.

3 Review: Post System

If you are already keen on theoretical computer science (or just still remember the Post system introduced in CS 245), congratulations, you may skip this section. Otherwise, please read along.

The Post system, named after Emil Post, is an example of a deductive formal system, which can be used to reason about programming languages. There are three components to a Post system: a set of signs (which forms the alphabet of the system), a set of variables, and a set of productions. A term is a string of signs and variables, and a production is an expression of the form:

\[ t_1 t_2 t_3 \cdots t_n \]

Where \( t, t_1, \cdots, t_n (n \geq 0) \) are all terms. The \( t_i \) are called the premises of the production, and the \( t \) is the conclusion. Thus, a production with the form:

\[
\begin{array}{c}
\text{premises} \\
\text{conclusion}
\end{array}
\]

is read as “if premises are true, then the conclusion holds”. A production without premises is permitted, and is called an axiom.

Productions are the definitions within our system, so it is outside the scope of the Post system to prove that the productions themselves are correct. In our case, the conclusions are what define how our programming languages are evaluated; in essence, each conclusion is a step we can take, and the premises are the context in which we can take those steps.

Post systems are used to prove conclusions, where a proof is constructed from proofs of its premises. Proofs based on Post systems are constructed using the following rules:

1. An instance of an axiom is a proof of its conclusion;
2. If \( P_1, P_2, \cdots, P_n \) are proofs of \( t_1, t_2, \cdots, t_n \) respectively, and

\[
\begin{array}{c}
\text{t_1 t_2 t_3 \cdots t_n} \\
t
\end{array}
\]

is an instance of a production, then

\[
\begin{array}{c}
P_1 P_2 P_3 \cdots P_n \\
\text{t}
\end{array}
\]

is a proof of \( t \).
Thus, given a final conclusion, a proof of that conclusion can be formed by proving its premises, until no unproven premises remain. The result of such a proof is an upside down tree with the root (final conclusion) at the bottom and the leaves (axioms) at the top.

**Example 1.** As an example of a Post system, we can encode the logical operations of ‘and’, $\land$, and ‘or’, $\lor$, using the following three rules:

\[
\begin{align*}
A & \quad \frac{}{A \lor B} \\
B & \quad \frac{}{A \lor B} \\
A & \quad \frac{}{A \land B}
\end{align*}
\]

Using this small system, it is possible to show that the proof of $(A \lor B) \land (A \lor C)$ follows from a proof of $A$ alone:

\[
\begin{align*}
A & \quad \frac{}{A \lor B} \\
A & \quad \frac{}{A \lor C} \\
\hline
(A \lor B) \land (A \lor C)
\end{align*}
\]

Post systems are used extensively for describing formal semantics. You will see that formal semantics of programming languages, including type systems, are often described in Post systems.

## 4 Operational Semantics for (Vanilla) $\lambda$-calculus

We have already discussed the semantics of $\lambda$-terms in Module 2, when we discussed free and bound variables, substitution, $\alpha$-conversion, and $\beta$-reduction. Assuming that we have already established the notion of binding, substitution, and $\alpha$-conversion, $\beta$-reduction seems to be a suitable candidate for operational semantics, for it specifies a procedure for carrying out computation.

Let’s rewrite $\beta$-reduction as a formal set of rules. First of all, all expressions that have $\beta$-redex in the outermost level can be directly reduced, with no premises:

\[
(\lambda x. M) N \rightarrow_{\beta} M[N/x]
\]

This rule corresponds in the first part of the definition. However, the next part of the definition\(^1\), which describes the reduction of $\beta$-redices within an expression, cannot be simply interpreted using a single rule. We have to rely on the structure of the $\lambda$-expressions. Recall that $\lambda$-expressions are either abstractions, applications, or variables. A variable itself certainly doesn’t need any rules for $\beta$-reduction, but we can have reductions happening inside abstractions and applications. The above rule is the special case where the rator is an abstraction. We still need to take the case where there is reduction happening inside an abstraction and within the rator or the rand. The following rules capture those two cases:

\[
\begin{align*}
M & \rightarrow_{\beta} P \\
\lambda x. M & \rightarrow_{\beta} \lambda x. P
\end{align*}
\]

The most important fact from this rule is that in order to show that $\lambda x. M \rightarrow_{\beta} \lambda x. P$, we must either provide a proof of $M \rightarrow_{\beta} P$, or there must exist a rule that states $M \rightarrow_{\beta} P$ is an axiom.

---

\(^{1}\)Recall from Module 2: $C[(\lambda x. M) N] \rightarrow_{\beta} C[M[N/x]]$
For applications, remember that in the original description of \(\beta\)-reduction we didn’t specify a reduction strategy. That is, we can choose to start our reduction either in the rator and the rand. for those two cases, we need separate rules:

\[
\begin{align*}
M \rightarrow_\beta P \\
MN \rightarrow_\beta PN
\end{align*}
\]

\[
\begin{align*}
N \rightarrow_\beta P \\
MN \rightarrow_\beta MP
\end{align*}
\]

We have just described how computation proceeds in \(\lambda\)-calculus. However, because a \(\lambda\)-calculus expression may match more than one of these conditions, our description is non-deterministic; we haven’t describe a particular way of computing, but all valid ways of computing. In the previous module, we made this deterministic by focusing on the selection of redices, and we will now do the same formally.

5 Defining Evaluation Order

As we mentioned earlier, evaluation order is in fact very important, since many programming languages will not non-deterministically execute code. If we were to model actual programming languages using our calculus, it is crucial to choose a reduction strategy. In this section, we are going to discuss the operational semantics of \(\lambda\)-calculus under Normal Order Reduction and Applicative Order Evaluation. Let’s first consider NOR.

**Definition 1. (Small-Step Operational Semantics of the Untyped \(\lambda\)-Calculus, NOR)**

Let the metavariable \(M\) range over \(\lambda\)-expressions. Then a semantics of \(\lambda\)-terms in NOR is given by the following rules:

\[
\frac{M \rightarrow M'}{\lambda x. M \rightarrow \lambda x. M'}
\]

\[
\frac{M_1 \rightarrow M_1' \quad \forall x. \forall M_3. M_1 \neq \lambda x. M_3}{M_1 M_2 \rightarrow M_1'[M_2]}
\]

\[
\frac{M_2 \rightarrow M_2' \quad \forall M_1'. M_1 \not\leftrightarrow M_1' \quad \forall x. \forall M_3. M_1 \neq \lambda x. M_3}{M_1 M_2 \rightarrow M_1 M_2'}
\]

The first and the second rule stayed the same as in \(\beta\)-reduction. Similar to non-deterministic \(\beta\)-reduction, we can reduce an expression that is either a redex, or an abstraction which contains a redex. However, in order to enforce NOR, we have to add additional restrictions to the third and fourth reduction rules. First of all, if \(M_1\) can be reduced further, we should reduce \(M_1\) instead; this is the reason we introduced the premise \(\forall x. \forall M_3. M_1 \neq \lambda x. M_3\) (that is if \(M_1\) is an abstraction).

Video 3.1 (https://student.cs.uwaterloo.ca/~cs442/W21/videos/3.1/): Formal semantics of NOR
Now let’s look at AOE:

**Definition 2. (Small-Step Operational Semantics of the Untyped \(\lambda\)-Calculus, AOE)**

Let the metavariable \(M\) range over \(\lambda\)-expressions. Then a semantics of the \(\lambda\)-terms in AOE is given by the following rules:

\[
\forall M_1'.M_1 \not\rightarrow M_1' \quad (\lambda x. M_1)M_2 \rightarrow M_1[M_2/x]
\]

\[
M_1 \rightarrow M_1' \quad M_1M_2 \rightarrow M_1'M_2
\]

\[
M_2 \rightarrow M_2' \quad \forall M_1'.M_1 \not\rightarrow M_1' \quad M_1M_2 \rightarrow M_1'M_2'
\]

For AOE, the first rule has the added condition that the rand (i.e. the argument) can be applied only if it can’t be reduced further. Also, the abstraction rule is removed, since we can not reduce within an abstraction; similarly, in most programming languages, you can not evaluate inside a function you didn’t yet call. The last rule has the premise \(\forall x.M_1 \neq \lambda x. M_2\) removed since again we want the argument to be fully reduced before substituting itself into an abstraction first.

6 Terminal Values

In the previous section, we used the language of predicate logic (specifically, for-all) to conditionalize productions. While this is mathematically valid, it complicates the description of the language, and makes it more difficult to prove that a particular production is the right one to use: to show that we can use the first rule, we need to demonstrate that the rand cannot be used with any of the rules. Generally speaking, the rules and conditions become much clearer if we can instead syntactically define what expressions are terminal, or final; i.e., not capable of being reduced further.

There are a few choices that we could make for possible sets of terminal values in \(\lambda\)-calculus: we could choose \(\beta\)-normal form, weak normal form, or even head normal form (only the leftmost expression is required to be in normal form). If we use anything other than \(\beta\)-normal form, we are losing the guarantee given by the Church-Rosser Theorem. Even if we use \(\beta\)-normal form as the set of terminal value, we still need to be able to answer some important questions. For example, what is the semantics of \((\lambda x. xx)(\lambda x. xx)\)? The only response we can give is “no semantics”, since it does not have a normal form. And what about \((\lambda x. \lambda y. y)((\lambda x. xx)(\lambda x. xx))\)? It has a terminal value, but not all possible legal derivations will lead to it. Should this expression be given a final value of \(\lambda y. y\) since there is a possible reduction to it, or we should say that there is no meaning? In fact, the answer depends on what one needs to achieve by designing the semantics.

For now, we focus on the steps themselves rather than the possible terminal values. As a result, we can just let our final values be “the set of values our operational semantics would produce”. In the next module, we will discuss terminal values in greater detail.

7 Showcase: A Simple Proof

In this section, we will show that \((\lambda x. \lambda y. y)((\lambda x. xx)(\lambda x. xx))x\) indeed terminates and evaluates to \(x\) under NOR. This is not a formal proof by any means; however, we will use this example to give you an idea to how programming language theorists work with semantics.
The formal way to specify that the former expression terminates and evaluates to the latter is going to look like this:

\[(\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx))x \rightarrow (\lambda y.y)x \rightarrow x\]

This example is quite short. However, what should we do if we are dealing with larger examples? The answer is that we need a \(\rightarrow^*\) operator. It might be useful to formally define the \(\rightarrow^*\) operator so we can show every single step at once, instead of splitting them into separate proofs:

**Definition 3. (Sequencing)** Let the metavariables \(M\) range over \(\lambda\)-expressions and \(\rightarrow\) be the operator of “one step” in any small-step operational semantics. Then \(\rightarrow^*\) is defined as so:

\[
\frac{M \rightarrow^* M}{M_1 \rightarrow^* M_2 \quad M_2 \rightarrow M_3} \quad M_1 \rightarrow^* M_3
\]

**Aside:** \(\rightarrow^*\) is the reflexive and transitive closure of \(\rightarrow\).

To keep the proof text short, we will make the following definitions:

\[A = (\lambda x.\lambda y.y), B = (\lambda x.xx)(\lambda x.xx)\]

Now we can actually start our “proof”.

\[
ABx \rightarrow^* ABx \quad ABx \rightarrow (\lambda y.y)x \\
(\lambda y.y)x \rightarrow x
\]

\[ABx \rightarrow^* (\lambda y.y)x \quad (\lambda y.y)x \rightarrow x
\]

\[(\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx))x \rightarrow^* x\]

Although this proof is of course trivial, with proper abstraction, we can prove similar properties of entire classes of programs. We will look at some of those properties in the next module.

**Aside:** There are also many, many other kinds of semantics. In this aside, we showcase two of them since they are also used in the field of programming languages. One of the variations of operational semantics is big-step operational semantics, which describes the terminal values every expression will evaluate to directly, rather than as the closure of smaller steps. For example, this is the big-step operational semantics for the \(\lambda\)-calculus under AOE:

\[
V \downarrow V
\]

\[
M[V_1/x] \downarrow V_2 \\
(\lambda x. M)V_1 \downarrow V_2
\]

\[
M_1 \downarrow V_1 \quad V_1M_2 \downarrow V_2 \\
M_1M_2 \downarrow V_2
\]

\[
V_2M_1 \downarrow V_3
\]

In this example, \(V\) is the metavariable over values.

Another kind of semantics is denotational semantics. Denotational semantics are used to show the correspondence from language constructs to familiar mathematical objects. Our definition of functional language
8 Adding Primitives

Around the end of Module 2, we discussed the \(\lambda\)-calculus implementations of commonly seen data types. While those discussions are very useful in showcasing the power of \(\lambda\)-calculus in representing computation, the implementations presented are not particularly practical; furthermore, it is much more efficient to make use of the computer architecture we have and implement those computations in their terms. For instance, since all computer architectures support integers (of some limited range) natively, it would be absurd to implement integers as Church numerals in a real language. As a result, in the practice of modelling real programming languages, we tend to model those as primitives. To be specific, those data types will be treated as intrinsic (i.e. built-in) values of our language. In this section, we will describe the semantic rules required if we were to add those built-in entities, since most of the semantics we see in future modules will have such intrinsics.

8.1 Booleans and Conditionals

We will first introduce the syntactic elements, in Backus Normal Form (BNF). Note that we are adding new kinds of expressions in the definition of \(\langle Expr\rangle\); We will use “…” to denote the part of the definitions of expressions that was defined in Module 2.

\[
\langle \text{Boolexp} \rangle ::= \text{true} \mid \text{false} \\
\mid \text{not} \langle \text{Boolexp} \rangle \\
\mid \text{and} \langle \text{Boolexp} \rangle \langle \text{Boolexp} \rangle \\
\mid \text{or} \langle \text{Boolexp} \rangle \langle \text{Boolexp} \rangle \\
\langle \text{Expr} \rangle ::= \ldots \\
\mid \langle \text{Boolexp} \rangle \\
\mid \text{if} \langle \text{Boolexp} \rangle \text{then} \langle \text{Expr} \rangle \text{else} \langle \text{Expr} \rangle
\]

Errata: The above definitions of “not”, “and”, “or”, and “if” demand that the subexpressions be boolean expressions. The following definition of number binops and lists have a similar problem, restricting part of the expression to only expressions of a particular type. We got a bit ahead of ourselves, thinking about types in module 4; in all of these cases, any expression is allowed.

These syntactic elements are very similar to the ones you have seen from Module 2. However, they are now actually part of the syntax; that is, there is no \(\lambda\)-calculus representation for them. Programs in the \(\lambda\)-calculus with boolean primitives are simply \(\lambda\)-calculus expressions with additional syntax for boolean expressions, like so:

\[
\lambda x. \lambda y. \text{if } x \text{ then } y \text{ else false}
\]

We will now describe the operational semantics for this new language. Let the metavariables \(B\) and \(E\) range over all boolean expressions and all \(\lambda\)-expressions, respectively. We will start with “not”:

\[
\begin{align*}
\text{not true} & \rightarrow \text{false} \\
\text{not false} & \rightarrow \text{true} \\
\text{not } B & \rightarrow \text{not } B'
\end{align*}
\]
For “and” and “or”, we want the computation of the first parameter to happen first. In addition, we would like short-circuiting behavior for them; i.e., the evaluation of the second operand should not proceed if the synthesis is known from the first.

\[
\begin{align*}
\text{and false } B & \rightarrow \text{false} \\
\text{and true } B & \rightarrow B \\
\text{and false } B_1 & \rightarrow \text{false} \\
\text{and true } B_1 & \rightarrow \text{true} \\
\end{align*}
\]

The last two rules are how we describe “the first argument must be fully evaluated before the second one”. The first rule describes the short-circuiting behavior: whether the second argument is evaluated or not, as long as the first argument evaluates to “false”, the whole “and” evaluates to false.

Exercise 1. Write the semantic rules for “or”.

Now we add the rules for if statements:

\[
\begin{align*}
\text{if true then } E_1 \text{ else } E_2 & \rightarrow E_1 \\
\text{if false then } E_1 \text{ else } E_2 & \rightarrow E_2 \\
\end{align*}
\]

8.2 Numbers

Note that we will restrict our definition to natural numbers. Also, we are working in an “imaginary machine”, so we don’t care about overflows (i.e., we assume that we can represent numbers of an infinite range). We will make the following definitions in our syntax, again in BNF:

\[
\begin{align*}
\langle \text{Num} \rangle & ::= 0 \mid 1 \mid \cdots \\
& \mid \langle \text{Num} \rangle \langle \text{Num} \rangle \\
\langle \text{NumBinOps} \rangle & ::= + \mid - \mid \ast \mid / \\
\langle \text{Expr} \rangle & ::= \cdots \\
& \mid \langle \text{Num} \rangle
\end{align*}
\]

We will now consider the semantics of binary operations. Let \( M, N \) range over numeric expressions, \( a, b \) range over natural numbers. Starting from addition:

\[
\begin{align*}
\text{a + b = c} & \rightarrow (a + b) \rightarrow c \\
\text{M} & \rightarrow \text{M}' \\
\text{M} \rightarrow \text{M}' & \rightarrow (+M + N) \rightarrow (+M' + N) \\
\text{M} & \rightarrow \text{M}' \\
\text{M} \rightarrow \text{M}' & \rightarrow (+a M) \rightarrow (+a M')
\end{align*}
\]

Note that this set of rules forces the first argument (i.e. left hand side) to be evaluated before the second argument is evaluated. Also note that we’re describing our language, the \( \lambda \)-calculus with numbers, in terms of the language of arithmetic, with the predicate \( a + b = c \).

Let’s now look at subtraction.

\[
\begin{align*}
\text{a - b = c} & \rightarrow (a - b) \rightarrow c \\
\text{M} & \rightarrow \text{M}' \\
\text{M} \rightarrow \text{M}' & \rightarrow (-M + N) \rightarrow (-M' + N) \\
\text{M} & \rightarrow \text{M}' \\
\text{M} \rightarrow \text{M}' & \rightarrow (-a M) \rightarrow (-a M')
\end{align*}
\]

The semantics for subtraction is almost the same with addition, but there is one difference: to actually compute \( a - b \), we need to make sure that \( a - b \) is a natural number. With this expression, there is no rule to match expressions
like $(-2, 3)$, and so such expressions cannot be reduced. We describe this phenomenon as "getting stuck", and in the next module, we will dive into this issue and discuss the significance of an expression getting stuck. Another way of handling this is to actually allow such subtraction, but define the result as something arbitrary and perhaps counter-intuitive, such as 0:

$$a - b = c \quad c \notin \mathbb{N}$$

$$(-a b) \rightarrow 0$$

Although this definition is unintuitive, it is not incorrect: we are defining our language’s $-$, and if it doesn’t match perfectly with the $-$ of arithmetic, that is part of the definition. Indeed, in real programming languages with integers of a limited size, no mathematical operations match their arithmetic definitions perfectly, because of overflow, but these languages are still valid and well-defined.

**Exercise 2.** Write the semantic rules for $\ast$ and $/$ (use integer division; think about how do you handle zero division.)

**Exercise 3.** Propose changes to the syntax rules and add new semantic rules, so we have pred and succ, which are unary functions for getting predecessor and successor of a number, in our language. Note: pred $0 = 0$.

### 8.3 Lists

In this section we will discuss lists. We will use the representation you should be well familiar with: a list containing $1, 2, 3$ will be

$$(\text{cons } 1 \text{ (cons } 2 \text{ (cons } 3 \text{ empty)))) = [1, 2, 3]$$

We will use a short-hand in mathematics to make our semantic rules compact: $L_1 + L_2$ will be operator to append $L_1$ to the start of $L_2$. For example: $[1] + [2] = [1, 2]$. We will also assume that $+$ works for “empty”, the empty list.

Again, we will list the syntactic elements of lists here:

$$(\text{ListExpr}) ::= \text{empty} \mid (\text{cons } \langle \text{Expr} \rangle \langle \text{ListExpr} \rangle) \mid [\langle \text{Expr} \rangle \langle \text{ListRest} \rangle]$$

$$\langle \text{ListRest} \rangle ::= \varepsilon \mid , \langle \text{Expr} \rangle \langle \text{ListRest} \rangle$$

$$\langle \text{Expr} \rangle ::= \ldots$$

$$\mid \langle \text{ListExpr} \rangle$$

$$\mid \text{first } \langle \text{ListExpr} \rangle \mid \text{rest } \langle \text{ListExpr} \rangle$$

The recursive definition of lists is essentially identical to the recursive data definition of the Racket list you saw in first-year courses.

Here are the semantic rules. Let the metavariables $L, E$ range over list expressions and $\lambda$-expression respectively:

$$L_2 = [E] + L_1 \quad \forall E_1, E \not\rightarrow E_1 \quad (\text{cons } E L_1) \rightarrow L_2$$

$$L_1 = [E] + L_2$$

$$L_1 = [E] + L_2$$

$$(\text{first } L_1) \rightarrow E$$

$$(\text{rest } L_1) \rightarrow L_2$$

Note that the premises in the form $L_1 = [E] + L_2$ implies that $L_1$ is not empty.
At last, don’t forget that we want expressions to reduce inside those built-in functions:

\[
\begin{align*}
E_1 & \to E'_1 \\
(\text{cons } E_1 E_2) & \to (\text{cons } E'_1 E_2) \\
\forall E_3. E_1 \not\to E_3 \\
(\text{cons } E_1 E_2) & \to (\text{cons } E_1 E'_2) \\
E_1 & \to E'_1 \\
(\text{first } E_1) & \to (\text{first } E'_1) \\
E_1 & \to E'_1 \\
(\text{rest } E_1) & \to (\text{rest } E'_1)
\end{align*}
\]

Note that our definition of lists has been slightly less formal than our previous definitions, as we relied on an informally described mathematical language of lists for our predicates. It is not uncommon for formal semantics to have some quasi-formal “holes” like this, though obviously it is preferable to define everything as precisely as possible.

8.4 Sets

A set is a mathematical collection of distinct objects. In real programming languages, it is usually implemented by a hash-map. However, when formulating a semantics for sets, we usually do not need to worry about their actual implementation; we can just treat it as a mathematical object. So long as the implementation provides the same observable behavior, it is correct.

The syntax of sets will be as follows:

\[
\langle SetExpr \rangle ::= \text{empty} \mid \{\langle Expr \rangle \langle SetRest \rangle\} \\
\mid \text{insert } \langle Expr \rangle \langle SetExpr \rangle \\
\mid \text{remove } \langle Expr \rangle \langle SetExpr \rangle \\
\langle SetRest \rangle ::= \varepsilon \mid \langle Expr \rangle \langle SetRest \rangle \\
\langle Expr \rangle ::= \cdots \\
\mid \langle SetExpr \rangle \\
\langle Boolean \rangle ::= \cdots \\
\mid \text{contains? } \langle Expr \rangle \langle SetExpr \rangle
\]

Let metavariables \( S, E \) range over be set expressions and all \( \lambda \)-expressions respectively:

\[
\begin{align*}
\forall E_1. E \not\to E_1 \\
\text{(insert } E \text{ empty}) & \to \{E\} \\
\forall E_1. E \not\to E_1 \\
\text{(remove } E \text{ empty}) & \to \text{empty} \\
\forall E_1. E \not\to E_1 \\
E & \in S \\
\text{(contains? } E S \text{)} & \to \text{true} \\
\forall E_1. E \not\to E_1 \\
S' = S \cup \{E\} \\
\text{(insert } E S \text{)} & \to S' \\
\forall E_1. E \not\to E_1 \\
S' = S \setminus \{E\} \\
\text{(remove } E S \text{)} & \to S' \\
\forall E_1. E \not\to E_1 \\
E \not\in S \\
\text{(contains? } E S \text{)} & \to \text{false}
\end{align*}
\]

Exercise 4. Write the semantic rules for set where at least one argument is not fully reduced.

Note that again, we have described our own sets in terms of the language of set theory.
9 Fin

In the next module, we will introduce types, which allow us to prove certain properties of languages, including that the semantics do not “get stuck”, by categorizing the kinds of values that may undergo certain operations.

Rights

Copyright © 2020, 2021 Yangtian Zi, Gregor Richards, Brad Lushman, and Anthony Cox.
This module is intended for CS442 at University of Waterloo.
Any other use requires permission from the above named copyright holder(s).