CMSC631 Program Analysis and Understanding: Class notes

October 4, 2014

Contents

Preface

These are the course notes to accompany *CMSC631 Program Analysis and Understanding* for the Fall semester of 2014 at the University of Maryland. This course has been taught many times, over several years, by a variety of distinguished and seasoned professors, who have put together a wide array of fine supplementary materials, all of which have been foolishly discarded by your current professor, who is neither distinguished, nor seasoned. "Why?" is the perfectly reasonable thought that should be running through your mind at this point.

Bret Victor, one of the more interesting philosophers of programming around today, wrote a little essay[1] on teaching in which he said:

> When I write or talk, it comes out of trying to understand a way of thinking that's deeply personal and valuable to me, and then trying to share this understanding. It's more than mere passion — anyone can be passionate about anything. It's a kind of honesty that comes from distilling and passing on *my own genuine insights and experience*.

And so that's really the reason your unseasoned, undistinguished professor has thrown it all out. These notes are an approximation of his own insights and experiences, and are surely full of mistakes and shortcomings, reflecting gaps in his own understanding.

Since your professor has thrown out all of the prepared materials *and* this is only the second time he's taught this course, these notes will unfortunately be prepared in a *just-in-time* fashion. As a kind of apology, the source code for these notes have been made available in a public repository. If you spot errors, have better ways of presenting ideas, or would like to raise questions not answered in the notes, you are encouraged to create issues and pull requests:

`https://github.com/cmsc631/notes`

Contributions to the notes will be acknowledged explicitly in the text, and implicitly in the participation component of your grade.

[1] `http://worrydream.com/SomeThoughtsOnTeaching/`

1 Introduction

Software is arguably the most complicated and varied kind of artifact humans produce. People who make software at the highest professional level think deeply about the meaning and correctness of the programs they write. A command of software design at this level requires developing an understanding how we give meaning to phrases of a programming language and how to build analytic tools for proving properties of programs.

This course covers basic theoretical ideas and practical techniques for modeling and analyzing programming languages; and leveraging those techniques to mechanically reason about programs.

2 Syntax, Semantics, & Machines for Arithmetic

2.1 Modelling Syntax with Inductive Sets

The *syntax* of a programming language is a set of rules for the arrangement of words and phrases to create well-formed sentences in the language. In other words, it is the grammar of programs. Syntax comes in two forms: *concrete syntax* describes the way programs actually look at the level of braces, semicolons, whitespace, etc., while *abstract syntax* describes the structure of programs without worrying over the superficial details of concrete syntax. Concrete syntax, while the stuff of frenzied fervor, is actually not all that significant for formally reasoning about programming languages so we focus exclusively on abstract syntax.

Programs generally have a tree-like structure of nesting phrases and expressions, so defining the abstract syntax of a language is no more complicated than defining an inductive set. As a case study, let's look at a very simple programming language, the language of arithmetic expressions. To keep things as simple as possible, the language will include integers, a couple binary operators like multiplication and addition, a unary operator for successor and predecessor.

Here is an inductive mathematical definition of the set $\mathcal{A}$. It is the smallest set satisfying the following constraints:

$$i \in \mathbb{Z} \Rightarrow i \in \mathcal{A} \tag{1}$$

$$e \in \mathcal{A} \Rightarrow \mathit{Pred}(e) \in \mathcal{A} \tag{2}$$

$$e \in \mathcal{A} \Rightarrow \mathit{Succ}(e) \in \mathcal{A} \tag{3}$$

$$e_1 \in \mathcal{A} \wedge e_2 \in \mathcal{A} \Rightarrow \mathit{Plus}(e_1, e_2) \in \mathcal{A} \tag{4}$$

$$e_1 \in \mathcal{A} \wedge e_2 \in \mathcal{A} \Rightarrow \mathit{Mult}(e_1, e_2) \in \mathcal{A} \tag{5}$$

Like any inductive definition, this can be viewed simultaneously as a recipe for *constructing* members of the set $\mathcal{A}$ and as a procedure for *checking* if a value is a member of the set $\mathcal{A}$.

Interpreted as a recipe, you can see that every integer is an $\mathcal{A}$ program; so 5 is an $\mathcal{A}$ program by (1). Since 5 is an $\mathcal{A}$ program, $\mathit{Pred}(5)$ is an $\mathcal{A}$ program by (2). And therefore

Mult(*Pred*$(5), 5)$ is an $\mathcal{A}$ program by (5). The recipe lets us build up bigger and bigger expressions from other expressions.

Interpreted as a checking procedure, we can answer the question "is *Plus*$(4,$ *Succ*$(2))$ an $\mathcal{A}$ program?" It is, according to (4), if both 4 and *Succ*(2) are $\mathcal{A}$ programs. Since 4 is an integer, it is an $\mathcal{A}$ program; by (3), *Succ*(2) is an $\mathcal{A}$ program if 2 is an $\mathcal{A}$ program, which of course it is, by (1).

An alternative notation for defining exactly the same set $\mathcal{A}$ is to use a BNF grammar:

$$\begin{array}{llll} \mathbb{Z} & i & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \\ \mathcal{A} & e & ::= & i \\ & & \mid & \textit{Pred}(e) \\ & & \mid & \textit{Succ}(e) \\ & & \mid & \textit{Plus}(e, e) \\ & & \mid & \textit{Mult}(e, e) \end{array}$$

In both of these definitions, the occurrences of "i" and "e" are occurrences of **meta-variables**—the are variables in the language describing the programming language (in this case $\mathcal{A}$). The describing language itself (in this case math, but often it is another programming language) is called the **meta-language**, while the described languaged ($\mathcal{A}$) is called the **object-language**. By convention, the name of a meta-variable signals the set of values it ranges over. So for example a meta-variable i may refer to 5 or 17, but never *Succ*(3).

Yet another notation for defining exactly the same set is to use inference rules, which is a two-dimensional notation for writing implications. The general form of an **inference rule** is:

$$\frac{H_1 \qquad H_2 \qquad \ldots \qquad H_n}{C}$$

Here H_1 through H_n are hypotheses and C is the conclusion. The inference rule states that if H_1 through H_n are true, then conclusion must be true as well. In other words, the inference rule is just a notation for $H_1 \wedge H_2 \wedge \cdots \wedge H_n \Rightarrow C$. With that in mind, it is straightforward to transliterate the first formulation of $\mathcal{A}$ into a set of inference rules:

$$\frac{i \in \mathbb{Z}}{i \in \mathcal{A}}\ (1) \qquad \frac{e \in \mathcal{A}}{\textit{Pred}(e) \in \mathcal{A}}\ (2) \qquad \frac{e \in \mathcal{A}}{\textit{Succ}(e) \in \mathcal{A}}\ (3) \qquad \frac{e_1 \in \mathcal{A} \qquad e_2 \in \mathcal{A}}{\textit{Plus}(e_1, e_2) \in \mathcal{A}}\ (4) \qquad \frac{e_1 \in \mathcal{A} \qquad e_2 \in \mathcal{A}}{\textit{Mult}(e_1, e_2) \in \mathcal{A}}\ (5)$$

This set of inference rules establishes a very simple *proof system* for constructing proofs that an expression is in the set $\mathcal{A}$. A proof is simply a tree constructed following the inference rules above. As an example, here is a proof that *Plus*$(4,$ *Succ*$(2)) \in \mathcal{A}$.

$$\frac{\dfrac{4 \in \mathbb{Z}}{4 \in \mathcal{A}} \qquad \dfrac{\dfrac{2 \in \mathbb{Z}}{2 \in \mathcal{A}}}{\textit{Succ}(2) \in \mathcal{A}}}{\textit{Plus}(4, \textit{Succ}(2)) \in \mathcal{A}}$$

It's easy to express any BNF grammar as an inductively defined set, which in turn is easy to express as a set of inference rules. But not all inductive definitions or inference rules can be expressed as grammars. Consequently, syntax is often defined using BNF, while more sophisticated relations are defined using inference rules.

2.2 Modelling Syntax with Data

Modelling programs with mathematics is a powerful idea that predates computers, but it's arguably more useful to model programs with programs. In other words, we can write programs that operate over data representations of programs. From this perspective, the abstract syntax of a programming language is just an inductive data type definition.

Here is the data type definition for $\mathcal{A}$ that corresponds to the earlier mathematical definition[2], written in the OCaml language:

```
type arith = Int of int
           | Pred of arith
           | Succ of arith
           | Plus of arith * arith
           | Mult of arith * arith
```

The only difference, which is inconsequential, is that OCaml, unlike math, requires unions to be formed by disjoint types, so integers must be "tagged" with the `Int` constructor to make them distinct from integers.

We can rely on the type system of OCaml to verify when a value is a member of the $\mathcal{A}$ set:

```
# Plus (Int 4, Succ (Int 2));;
- : arith = Plus (Int 4, Succ (Int 2))
```

2.3 Natural Semantics

Perhaps the simplest semantics we can give arithmetic expressions is to define a relation between an expression and the integer value it simplifies to according to the usual rules of arithmetic.

To do this, we define a binary relation $\Downarrow \subseteq \mathcal{A} \times \mathbb{Z}$. When an expression e is related to an integer i, it means e *evaluates* to i. Following the usual convention, we write $e \Downarrow i$ to mean $(e, i) \in \Downarrow$ (it's much easier to read $3 < 4$ instead of $(3, 4) \in <$, right?).

Just as we defined the set of $\mathcal{A}$ programs inductively, we can define the evaluation relation $\Downarrow$ inductively as well.

$$\frac{}{i \Downarrow i} \qquad \frac{e \Downarrow i}{\mathit{Pred}(e) \Downarrow i - 1} \qquad \frac{e \Downarrow i}{\mathit{Succ}(e) \Downarrow i + 1} \qquad \frac{e_1 \Downarrow i \qquad e_2 \Downarrow j}{\mathit{Plus}(e_1, e_2) \Downarrow i + j} \qquad \frac{e_1 \Downarrow i \qquad e_2 \Downarrow j}{\mathit{Mult}(e_1, e_2) \Downarrow i \cdot j}$$

[2]Already a lie has crept in: OCaml's `int` is not the same as $\mathbb{Z}$. We're going to ignore this discrepancy for the time being. The problem could be resolved by using a big integer library, at the cost of some notational overhead in our examples.

Note on notation: To be truly pedantic, we should add hypotheses to each of the inference rules that state $e \in \mathcal{A}$ and $i \in \mathbb{Z}$ and so on. The convention that you'll frequently see, and which is used in these notes, is that certain meta-variables range only over a restricted set of values. So if you see the meta-variable "e", you can reasonably read it as e such that $e \in \mathcal{A}$. If you see e being used as something that is not an arithmetic expression, then it's a mistake. Likewise, i and j range only over integers.

The simplest $\mathcal{A}$ expression is an integer, which cannot be simplified further, so an integer evaluates to itself as shown in the leftmost rule. If an expression e evaluates to an integer i, then *Succ*(e) evaluates to $i + 1$ as shown in the second rule. The remaining rules are similar.

The proof that an expression evaluates to some integer can be given as a proof tree using these inference rules. For example, here is the proof that *Plus*$(4,$ *Succ*$(2)) \Downarrow 7$.

$$\dfrac{\dfrac{}{4 \Downarrow 4} \qquad \dfrac{\dfrac{}{2 \Downarrow 2}}{\textit{Succ}(2) \Downarrow 3}}{\textit{Plus}(4, \textit{Succ}(2)) \Downarrow 7}$$

When reading these inference rules, it's important to notice that the right hand side of the evaluation relation is a mathematical expression that denotes an integer; not a peice of syntax. As the previous example shows, *Plus*$(4,$ *Succ*$(2))$ evaluates to 7; not a representation of the expression "4+(2+1)". If it helps to see this, a completely equivalent formulation of, for example, the *Plus* rule that stresses evaluation produces a single integer is:

$$\dfrac{e_1 \Downarrow i \qquad e_2 \Downarrow j \qquad k = i + j}{\textit{Plus}(e_1, e_2) \Downarrow k}$$

Evaluation is defined as a relation, but it is actually a special case of a relation: it is a function. Can you convince yourself of this? That is, can you prove that if $e \Downarrow i$ and $e \Downarrow j$, then $i = j$? Assuming you can, that means we can write the $\Downarrow$ function as a function in a programming language such as OCaml. It naturally is a recursive function over the data type of syntax:

```
let rec eval (e : arith) : int =
    match e with
      Int i -> i
    | Pred e -> (eval e) - 1
    | Succ e -> (eval e) + 1
    | Plus (e1, e2) -> (eval e1) + (eval e2)
    | Mult (e1, e2) -> (eval e1) * (eval e2)
```

The evaluator can be used as a calculator for arithmetic expressions:

```
# eval (Plus (Int 4, Succ (Int 2)));;
- : int = 7
```

The above semantics are called a "**big-step**" or "**natural**" semantics. It has a simple correspondence with a structurally recursive functional program. The meaning of a program is given by the irreducible value it produces, so the semantics tells you *what* a program evaluates to. If the "what" is all you are interested in, then this semantics is adequate.

This brings up several questions. What is a semantics for? Why would one semantics be preferred over another? What other kind of semantics could there possible be?

2.4 Reduction Semantics

A semantics is useful for establishing properties of programs. There are all kinds of properties of programs, so there are all kinds of semantics of programs; just like math or logic, there is no "one true semantics." The program properties of interest depend on what you are trying to accomplish. Are you trying to write a correct interpreter? The natural semantics is probably what you need since it tells you what value the interpreter should produce. Are you trying to prove upper bounds on the cost of runnning some programs? The natural semantics won't work because it doesn't say *anything* about cost. Ditto if you want to prove a program can run forever without using all available memory. So a semantics should be tailored to properties of interest.

With that in mind, let's investigate a semantics that informs us of each of the steps a computation goes through toward reaching a value. This is useful for determing, for example, if two programs reach the same intermediate state, or in which order are operations performed during a computation. In other words, such a semantics would tell us more about *how* a program evaluates.

Fundamentally, we do this like before: by specifying relations on syntax. But instead of defining a "evaluates to" relation, we define a "steps to" relation **a** that captures the basic laws of arithmetic reduction. The new relation does not relate expressions to integers, but instead relates expressions to expressions, i.e. $\mathbf{a} \subseteq \mathcal{A} \times \mathcal{A}$. The meaning of related expressions e_1 and e_2 is that e_1 reduces in one step to e_2.

$$\frac{}{\mathit{Pred}(i)\ \mathbf{a}\ i-1} \qquad \frac{}{\mathit{Succ}(i)\ \mathbf{a}\ i+1} \qquad \frac{}{\mathit{Plus}(i,j)\ \mathbf{a}\ i+j} \qquad \frac{}{\mathit{Mult}(i,j)\ \mathbf{a}\ i\cdot j}$$

These axioms[3] reflect the basic facts of reduction in arithmetic, but they're still not enough to capture computation, since for example $\mathit{Plus}(4, \mathit{Succ}(2))$ doesn't step to anything. The reason is that our axioms only apply when the arguments to operators are integers, which is not the case here. It is true that $\mathit{Succ}(2)$ steps to 3, but none of the rules allows us to evaluate inside of a nested expression.

Let's define another relation , $\rightarrow_{\mathbf{a}} \subseteq \mathcal{A} \times \mathcal{A}$, that allows steps within nested expressions. For the language of $\mathcal{A}$ this amounts to taking the **compatible closure** of **a** over the grammar of expressions. The compatible closure of a relation **r** over some grammar g derives a new relation $\mathbf{r}'$ that allows **r** to be distributed through non-terminals in g. Writing this out explicitly for the case of **a** and e is the following:

[3] An axiom is just a inference rule with no hypotheses.

$$\frac{e\ \mathbf{a}\ e'}{e \to_{\mathbf{a}} e'} \qquad \frac{e \to_{\mathbf{a}} e'}{\mathit{Pred}(e) \to_{\mathbf{a}} \mathit{Pred}(e')} \qquad \frac{e \to_{\mathbf{a}} e'}{\mathit{Succ}(e) \to_{\mathbf{a}} \mathit{Succ}(e')} \qquad \frac{e_1 \to_{\mathbf{a}} e_1'}{\mathit{Plus}(e_1, e_2) \to_{\mathbf{a}} \mathit{Plus}(e_1', e_2)}$$

$$\frac{e_2 \to_{\mathbf{a}} e_2'}{\mathit{Plus}(e_1, e_2) \to_{\mathbf{a}} \mathit{Plus}(e_1, e_2')} \qquad \frac{e_1 \to_{\mathbf{a}} e_1'}{\mathit{Mult}(e_1, e_2) \to_{\mathbf{a}} \mathit{Mult}(e_1', e_2)} \qquad \frac{e_2 \to_{\mathbf{a}} e_2'}{\mathit{Mult}(e_1, e_2) \to_{\mathbf{a}} \mathit{Mult}(e_1, e_2')}$$

Using $\to_{\mathbf{a}}$, we can see that $\mathit{Plus}(4, \mathit{Succ}(2)) \to_{\mathbf{a}} \mathit{Plus}(4, 3)$ and $\mathit{Plus}(4, 3) \to_{\mathbf{a}} 7$. Evaluation of a program can be viewed a series of related expressions arriving at a final answer.

If we'd like to capture the notion of "e steps to e' in any number of steps," we can define yet another relation, $\to_{\mathbf{a}}^\star \subseteq \mathcal{A} \times \mathcal{A}$, as the reflexive, transitive closure of $\to_{\mathbf{a}}$. The **reflexive closure** of a relation $\mathbf{r} \subseteq X \times X$ is a relation $\mathbf{r}' \subseteq X \times X$ such that $x \in X \Rightarrow x\ \mathbf{r}'\ x$ and $x_1 \in X \wedge x_2 \in X \wedge x_1\ \mathbf{r}\ x_2 \Rightarrow x_1\ \mathbf{r}'\ x_2$. In other words, the reflexive closure of $\mathbf{r}$ relates every thing in $\mathbf{r}$ plus it relates everything to itself. The reflexive closure of $\to_{\mathbf{a}}$ captures the notion of "steps in zero or one step." The **transitive closure** of a relation $\mathbf{r} \subseteq X \times X$ is a relation $\mathbf{r}' \subseteq X \times X$ such that $x_1\ \mathbf{r}\ x_2 \Rightarrow x_1\ \mathbf{r}'\ x_2$ and $x_1\ \mathbf{r}'\ x_2 \wedge x_2\ \mathbf{r}'\ x_3 \Rightarrow x_1\ \mathbf{r}'\ x_3$. In other words, the transitive closure of $\mathbf{r}$ relates x_1 to x_2 if $\mathbf{r}$ does, but also includes everything x_2 relates to, and anything related to that, and so on. The transitive closure of $\to_{\mathbf{a}}$ captures the notion of "steps in one or more steps". Composing these closure operations gives $\to_{\mathbf{a}}^\star$, which is "steps in zero or more steps." Writing it out explicitly results in the following set of inference rules:

$$\frac{e \to_{\mathbf{a}} e'}{e \to_{\mathbf{a}}^\star e'} \qquad \frac{}{e \to_{\mathbf{a}}^\star e} \qquad \frac{e \to_{\mathbf{a}}^\star e' \qquad e' \to_{\mathbf{a}}^\star e''}{e \to_{\mathbf{a}}^\star e''}$$

The above semantics are a "**small-step**" or "**reduction**" semantics. Unlike the natural semantics, the reduction semantics accounts for each step of a computation. Although it's immaterial for the $\mathcal{A}$ language, the small-step approach has some advantages over big-step (as we should expect considering it is considerably more involved); one of the most important advantages comes into play when the object language is sufficiently powerful to include non-terminating computations. Since the natural semantics is concerned only with final answers, it doesn't say much about non-terminating programs, while reduction semantics can still be used to reason about the (infinite) steps of such a computation.

One important observation to make is that although we've constructed an alternative semantics, we can formally relate these two semantics. In particular, we can recover a big-step evaluation relation from the reduction semantics. Consider the following relation $\downarrow \subseteq \mathcal{A} \times \mathbb{Z}$:

$$\frac{e \to_{\mathbf{a}}^\star i}{e \downarrow i}$$

which is a subset of $\to_{\mathbf{a}}^\star$, restricted to the case of the right hand side being an integer. This relation effectively forgets any intermediate terms in a computation and just relates expressions to their irreducible values. This relation, although defined differently, is the same relation as $\Downarrow$. Seen this way, it is accurate to say natural semantics are an **abstraction** of reduction semantics, and reduction semantics are a **refinement** of natural semantics.

To model the reduction semantics in OCaml, we can first define the axiomatic reduction relation, **a**. First, observe that **a** is a *function*; if e **a** e' and e **a** e'', then $e' = e''$. Therefore we can model the relation as a function on expressions. However, **a**, when viewed as a function, is *partial*; not all expressions reduce according to **a**. For example, $i \not\to_{\mathbf{a}} e$, for any e.

We can model a partial function from expressions to integers as a total function from expressions to an integer options. Optional types are a useful way to encode either getting something or nothing:

```
type 'a option =
  | None
  | Some of 'a
```

This type is so useful, it's built in. Now defining **a** in OCaml consists in translating the defining judgements into an OCaml function:

```
let a (e : arith) : arith option =
  match e with
    | Pred (Int i) -> Some (Int (i-1))
    | Succ (Int i) -> Some (Int (i+1))
    | Plus (Int i, Int j) -> Some (Int (i+j))
    | Mult (Int i, Int j) -> Some (Int (i*j))
    | _ -> None
```

Some examples:

```
# a (Int 4);;
- : arith option = None

# a (Succ (Int 5));;
- : arith option = Some (Int 6)

# a (Plus (Succ (Int 5), Succ (Int 4)));;
- : arith option = None
```

To characterize $\to_{\mathbf{a}}$, which is not a function, but a more general relation, which means we need to represent $\to_{\mathbf{a}}$ as a function from expressions to *sets of* expressions. We can encode this set as a list. Therefore the signature of `step_a` will be `arith -> arith list`:

```
let rec step_a (e : arith) : arith list =
  match a e with
    | None ->
        (match e with
           | Int i -> []
           | Pred e -> List.map (fun e' -> Pred e') (step_a e)
           | Succ e -> List.map (fun e' -> Succ e') (step_a e)
           | Plus (e1, e2) ->
               let f e1s e2s = List.map
```

```
                  (fun (e1',e2') -> Plus (e1', e2'))
                  (cartesian e1s e2s)
              in
                (f (step_a e1) [e2]) @ (f [e1] (step_a e2))
          | Mult (e1, e2) ->
              let f e1s e2s = List.map
                (fun (e1',e2') -> Mult (e1', e2'))
                (cartesian e1s e2s)
              in
                (f (step_a e1) [e2]) @ (f [e1] (step_a e2)))
    | Some e' -> [e']
```

where

```
let cartesian (l : 'a list) (l' : 'b list) : ('a * 'b) list =
  List.concat (List.map (fun e -> List.map (fun e' -> (e,e')) l') l)
```

Some examples:

```
# step_a (Int 4);;
- : arith list = []

# step_a (Succ (Int 5));;
- : arith list = [Int 6]

# step_a (Plus (Succ (Int 5), Succ (Int 4)));;
- : arith list = [Plus (Int 6, Succ (Int 4)); Plus (Succ (Int 5), Int 5)]
```

Exercise 1. *Prove* $e \Downarrow i \iff e \downarrow i$. □

Exercise 2. *Is* $\mathbf{a}$ *a function? Is* $\rightarrow_{\mathbf{a}}$*? In each case, either prove the relation is a function or give an example of an expression that relates to two distinct expressions.* □

Exercise 3. *The* $\rightarrow_{\mathbf{a}}$ *relation codifies the notion of "takes exactly one step of arithmetic reduction." In other words, any proof tree of* $e \rightarrow_{\mathbf{a}} e'$ *involves exactly one use of the* $\mathbf{a}$ *rule. (You could prove this if you'd like.) In terms of grade-school arithmetic, this is the "show each step of your work" semantics, which doesn't let you skip any steps (*$\Downarrow$ *lets you skip all of the steps and just give the answer) or do any work in parallel. Design an alternative semantics,* $\rightrightarrows_{\mathbf{a}}$*, that allows multiple subexpressions to reduce (one step) in parallel. So for example,* $\mathit{Plus}(\mathit{Succ}(3), \mathit{Pred}(5)) \rightrightarrows_{\mathbf{a}} \mathit{Plus}(4, 4)$*. Note that this semantics should not let you conclude that* $\mathit{Plus}(\mathit{Succ}(3), \mathit{Pred}(5)) \rightrightarrows_{\mathbf{a}} 8$ *in just one step.*

You have a design choices to make for $\rightrightarrows_{\mathbf{a}}$*: it can allow any amount of parallelism, or it can impose a maximal amount of parallelism. Let's call the more lax any amount of parallelism relation* $\rightrightarrows_{\mathbf{a}}$ *and the maximal one* $\rightrightarrows'_{\mathbf{a}}$*. The key difference is that* $\rightrightarrows_{\mathbf{a}}$ *should relate* $\mathit{Plus}(\mathit{Succ}(3), \mathit{Pred}(5))$ *to* $\mathit{Plus}(4, \mathit{Pred}(5))$*, while* $\rightrightarrows'_{\mathbf{a}}$ *should not since there was more work that could have been done in parallel. Design both. Is* $\rightrightarrows_{\mathbf{a}}$ *a function? Is* $\rightrightarrows'_{\mathbf{a}}$*? Is* $\rightrightarrows_{\mathbf{a}}$ *equal to* $\rightarrow^{\star}_{\mathbf{a}}$*? Is* $\rightrightarrows'_{\mathbf{a}}$*?* □

Exercise 4. *To give away an answer to an earlier question,* $\rightarrow_{\mathbf{a}}$ *is* not *a function. To see why, consider* $\mathit{Plus}(\mathit{Succ}(3), \mathit{Pred}(5))$*. In one step, this program can either become* $\mathit{Plus}(4, \mathit{Pred}(5))$ *or*

Plus$(\mathit{Succ}(3), 4)$. *So modelling* $\rightarrow_{\mathbf{a}}$ *as a function in, say, OCaml can't be accomplished quite so easily as with* $\Downarrow$. *One idea for modelling a finite relation in a functional language is to rely on the isomorphism*

$$(X \times X) \cong (X \longrightarrow \mathcal{P}(X)).$$

This is to say, we can represent a relation as a function from an element to the set *of elements it relates to. With this idea in mind, write an OCaml function* `step` *that represents* $\rightarrow_{\mathbf{a}}$. □

Exercise 5. *To give away yet another answer,* **a** *is a function, but it is a* ***partial function****—it is not defined all elements of* $\mathcal{A}$. *One idea for representing a partial function in a functional language is to rely on the isomorphism*

$$(X \rightharpoonup X) \cong (X \longrightarrow (\emptyset + \{X\})).$$

This is to say, we can represent a partial function as a total function to either the empty set (interpreted as undefined) or a single set consisting of the element the partial function maps to. This has the nice property that it is consistent with the functional view of relations from the previous exercise, since

$$(X \longrightarrow (\emptyset + \{X\})) \subseteq (X \longrightarrow \mathcal{P}(X)).$$

Write an OCaml function `reduce` *that represents* **a** *using the above idea.* □

Exercise 6. *The compatible closure of a relation over the syntax of* $\mathcal{A}$ *expressions can be seen as a function that consumes an* $\mathcal{A}$ *relation and produces a new* $\mathcal{A}$ *relation. From the functional perspective of relations, this means the compatible closure operation (for* $\mathcal{A}$*) is a function with the following signature:*

$$\texttt{compat} : (\mathcal{A} \longrightarrow \mathcal{P}(\mathcal{A})) \longrightarrow (\mathcal{A} \longrightarrow \mathcal{P}(\mathcal{A}))$$

Design an OCaml function `compat` *that computes the compatible closure of its argument. Test the following conjecture:* `step = compat reduce`. □

Exercise 7. *The transitive and reflexive closure operations are functions that consume a relation and produce a new relation. Using the functional view of relations, this means they are functions with the following signatures:*

$$\begin{aligned}\texttt{refl} &: (X \longrightarrow \mathcal{P}(X)) \longrightarrow (X \longrightarrow \mathcal{P}(X)) \\ \texttt{trans} &: (X \longrightarrow \mathcal{P}(X)) \longrightarrow (X \longrightarrow \mathcal{P}(X))\end{aligned}$$

Design OCaml functions for `refl` *and* `trans`. *Test the following conjecture:* `refl (trans step) = trans (refl step)`. *The* `refl` *function is easy. The* `trans` *function is less so because it's not obvious when to stop iterating the given relation, but this exercise demonstrates a powerful idea we'll see again and again, which is the iterative computation of* ***fixed points****.*

The idea here is that `trans`*, given a relation* $\mathbf{r}$*, computes a relation* $\mathbf{r}'$*, written here as a function:*

$$\begin{aligned}\mathbf{r}'(x) &= \{x' \mid x \,\mathbf{r}\, x'\} && (6)\\ &\cup \{x' \mid x \,\mathbf{r}\, x_0 \,\mathbf{r}\, x'\} && (7)\\ &\cup \{x' \mid x \,\mathbf{r}\, x_0 \,\mathbf{r}\, x_1 \,\mathbf{r}\, x'\} && (8)\\ &\vdots && (9)\end{aligned}$$

The elipsis are elliding an infinite number of equations here, but if the codomain of $\mathbf{r}'$ *is finite, we know that only a finite number of equations will ever be used. Moreover, notice that the* x_0 *of (7) is just the* x' *of (6). Likewise, the* x_1 *of (8) is just the* x' *of (7). So if you want to compute* $\mathbf{r}'(x)$ *you can start with the set* $\mathbf{r}(x)$*. For each* x' *in this set, compute* $\mathbf{r}(x')$ *and add it in to the set (you've reached a fixed point!). Keep doing this until* $\mathbf{r}$ *doesn't add anything new to the set. This is* $\mathbf{r}'(x)$*.* □

2.5 Reduction in Context

When an expression reduces, there is a subexpression being reduced according to a reduction axiom. This occurs within a surrounding expression, or *context*. We can formalize the notion of context and by doing so, enable new kinds of reductions.

A **context** can be thought of as an expression with a hole in it, which is just a term in the following language:

$$\begin{array}{rrcl} \textit{Context} & \mathcal{C} & = & \Box \\ & & \mid & \textit{Pred}(\mathcal{C}) \mid \textit{Succ}(\mathcal{C}) \\ & & \mid & \textit{Plus}(\mathcal{C}, e) \mid \textit{Plus}(e, \mathcal{C}) \\ & & \mid & \textit{Mult}(\mathcal{C}, e) \mid \textit{Mult}(e, \mathcal{C}) \end{array}$$

A context is either a hole, written "□," or it's a term constructor with a context in place of a subexpression. An expression can be plugged into a context to obtain another expression, which is written "$\mathcal{C}[e]$," which means "replace the occurrence of □ in $\mathcal{C}$ with e." To be more formal, the plug function is defined as:

$$\begin{aligned} \Box[e] &= e \\ \textit{Pred}(\mathcal{C})[e] &= \textit{Pred}(\mathcal{C}[e]) \\ \textit{Succ}(\mathcal{C})[e] &= \textit{Succ}(\mathcal{C}[e]) \\ \textit{Plus}(\mathcal{C}, e')[e] &= \textit{Plus}(\mathcal{C}[e], e') \\ \textit{Plus}(e', \mathcal{C})[e] &= \textit{Plus}(e', \mathcal{C}[e]) \\ \textit{Mult}(\mathcal{C}, e')[e] &= \textit{Mult}(\mathcal{C}[e], e') \\ \textit{Mult}(e', \mathcal{C})[e] &= \textit{Mult}(e', \mathcal{C}[e]) \end{aligned}$$

The notation "$\mathcal{C}[e]$" is also overloaded to mean "an expression e' such that $e' = \mathcal{C}[e]$." We say e' can be "decomposed" into $\mathcal{C}$ and e.

This notation allows us to give an alternative, but equivalent, formulation of the compatible closure of **a** as:

$$\frac{e \ \mathbf{a} \ e'}{\mathcal{C}[e] \rightarrow_{\mathbf{a}} \mathcal{C}[e']}$$

This rule states: "if an expression can be decomposed into a context $\mathcal{C}$ with e in the hole, and e reduces to e' by the reduction axiom, then the program steps to $\mathcal{C}$ with e' plugged in the hole."

This context-based formulation may seem like just a somewhat more compact notation for specifying the tedious compatibility inference rules that allow reduction to happen

inside of subexpressions, but it also does something more signficant: it gives a name to the context in which a reduction occurs; naming something gives us control it.[4] We will examine some of the possibilities later in the notes.

2.6 Standard Reduction Machine

The reduction semantics for $\mathcal{A}$ allows a single program to reduce in multiple different ways, just as we may simplify $(2+4)\cdot((6-1)+(1+1))$ in a number of different ways by selecting different orders in which to simplify subexpressions. We know from the consistency of the language that no matter which order we choose, if we get an answer, it is *the* answer; redoing the calculation in another order will not change the final result.

Having a "calculus of programs," as embodied by the equational theory for our language, is important to reason about programs and equivalences between them, but when trying to find what a program reduces it, it's better if we had a strategy for the mechanical calculuation of answers. For the $\mathcal{A}$ language, it's easy to see that any strategy will do: as long as you keep reducing something, you'll always find the correct answer eventually; it doesn't matter what order you go in. For richer languages, it's not always clear that there's a fixed strategy for finding answers (if an answer even exists!).

Establishing a strategy for the order in which a machine applies reduction axioms to obtain an answer is called a **standard reduction semantics**; it represent a canonical way in which programs are reduced.

Let's develop a standard reduction relation, called $\longmapsto_{\mathbf{a}}$, for $\mathcal{A}$. (Take note of the long barred arrow used here; don't confuse the $\longmapsto$ and $\rightarrow$ arrows!) The idea here is to make the one-step reduction relation a function by standardizing the reduction strategy:

$$\frac{e\ \mathbf{a}\ e'}{e \longmapsto_{\mathbf{a}} e'} \qquad \frac{e_1 \longmapsto_{\mathbf{a}} e_1'}{\mathit{Plus}(e_1, e_2) \longmapsto_{\mathbf{a}} \mathit{Plus}(e_1', e_2)} \qquad \frac{e \longmapsto_{\mathbf{a}} e'}{\mathit{Plus}(v, e) \longmapsto_{\mathbf{a}} \mathit{Plus}(v, e')}$$

$$\frac{e_1 \longmapsto_{\mathbf{a}} e_1'}{\mathit{Mult}(e_1, e_2) \longmapsto_{\mathbf{a}} \mathit{Mult}(e_1', e_2)} \qquad \frac{e \longmapsto_{\mathbf{a}} e'}{\mathit{Mult}(v, e) \longmapsto_{\mathbf{a}} \mathit{Mult}(v, e')} \qquad \frac{e \longmapsto_{\mathbf{a}} e'}{\mathit{Succ}(e) \longmapsto_{\mathbf{a}} \mathit{Succ}(e')}$$

$$\frac{e \longmapsto_{\mathbf{a}} e'}{\mathit{Pred}(e) \longmapsto_{\mathbf{a}} \mathit{Pred}(e')}$$

Notice that none of the rules overlap. To see this, consider a more elaborate, but equivalent formulation of the rules for *Mult*:

$$\frac{e_1 \longmapsto_{\mathbf{a}} e_1' \qquad e_1 \notin \mathit{Val}}{\mathit{Mult}(e_1, e_2) \longmapsto_{\mathbf{a}} \mathit{Mult}(e_1', e_2)} \qquad \frac{e \longmapsto_{\mathbf{a}} e' \qquad e \notin \mathit{Val}}{\mathit{Mult}(v, e) \longmapsto_{\mathbf{a}} \mathit{Mult}(v, e')}$$

The added hypotheses are in fact redundant because $e \longmapsto_{\mathbf{a}} e'$ implies e is not a value, but it's now easy to see by case analysis that at most one rule applies to an expression of the form $\mathit{Mult}(e_1, e_2)$ because either e_1 is not a value, in which case the leftmost rule applies,

[4] Computer science is in many ways the science of names.

or e_1 is a value, but e_2 is not, in which case the rightmost rule applies, or both e_1 and e_2 are values, in which case the **a** rule applies.

The rules for reducing inside expressions are not quite compatibility rules because they do not allow a reduction axiom to be apply to *any* subexpression. Instead, they encode a **reduction strategy** that specifies the one subexpression where an axiom may be applied.

We can represent the $\longmapsto_{\mathbf{a}}$ relation in OCaml like we did for **a**:

```
let rec standard_step_a (e : arith) : arith list =
  match a e with
    | None ->
        (match e with
           | Int i -> []
           | Pred e -> List.map (fun e' -> Pred e') (step_a e)
           | Succ e -> List.map (fun e' -> Succ e') (step_a e)
           | Plus (e1, e2) ->
               let f e1s e2s = List.map
                 (fun (e1',e2') -> Plus (e1', e2'))
                 (cartesian e1s e2s)
               in
                 (match e1 with
                   | Int i -> (f [e1] (standard_step_a e2))
                   | _ -> (f (standard_step_a e1) [e2]))
           | Mult (e1, e2) ->
               let f e1s e2s = List.map
                 (fun (e1',e2') -> Mult (e1', e2'))
                 (cartesian e1s e2s)
               in
                 (match e1 with
                    | Int i -> (f [e1] (standard_step_a e2))
                    | _ -> (f (standard_step_a e1) [e2])))
    | Some e' -> [e']
```

Some examples:

```
# standard_step_a (Int 4);;
- : arith list = []

# standard_step_a (Plus (Succ (Int 5), Succ (Int 4)));;
- : arith list = [Plus (Int 6, Succ (Int 4))]
```

You should be able to convince yourself that `standard_step_a` always produces either an empty list or a singleton list. In other words, $\longmapsto_{\mathbf{a}}$ is a partial function, being encoded here as a relation. We could rewrite the code to produce an `arith option`, or we could use a function to provide that interface:

```
let rel_to_partial_func (r : ('a -> 'a list)) : ('a -> 'a option) =
  fun a ->
```

```
  match r a with
    | [] -> None
    | a::_ -> Some a
```

Now we can compute an alternative interface for standard reduction:

```
let standard_step_a' = rel_to_partial_func standard_step_a
```

Some examples:

```
# standard_step_a' (Int 4);;
- : arith option = None

# standard_step_a' (Plus (Succ (Int 5), Succ (Int 4)));;
- : arith option = Some (Plus (Int 6, Succ (Int 4)))
```

Instead of specifying the strategy by way of progress rules, the same strategy can also be defined through the use of contexts, but instead of using arbitrary contexts, we will specify the subset of contexts in which reduction may occur. These contexts are called *evaluation contexts*:

$$\begin{array}{rrcl} \textit{EvalContext} & \mathcal{E} & = & \Box \\ & & | & \textit{Pred}(\mathcal{E}) \mid \textit{Succ}(\mathcal{E}) \\ & & | & \textit{Plus}(\mathcal{E}, e) \mid \textit{Plus}(v, \mathcal{E}) \\ & & | & \textit{Mult}(\mathcal{E}, e) \mid \textit{Mult}(v, \mathcal{E}) \end{array}$$

And standard reduction is just:

$$\frac{e \ \mathbf{a} \ e'}{\mathcal{E}[e] \longmapsto_{\mathbf{a}} \mathcal{E}[e']}$$

It's possible to read off the reduction strategy from the grammar. For example, the production $\textit{Plus}(\mathcal{E}, e)$ says "until the left side of *Plus* is a value, reduce the left side of *Plus*" and $\textit{Plus}(v, \mathcal{E})$ says "after the left side is value, reduce the right side." So when you see $\textit{Plus}(e_1, e_2)$, you know all of the reductions will first happen for e_1, and only after there are no more will all of the reductions for e_2 happen.

The standard semantics has the following desirable properties:

1. if the standard semantics produces a value, it is consistent with the reduction semantics,
$$e \longmapsto^{\star}_{\mathbf{a}} v \Rightarrow e \rightarrow^{\star}_{\mathbf{a}} v,$$
2. if the reduction semantics produces a value, the standard semantics does too,
$$e \rightarrow^{\star}_{\mathbf{a}} v \Rightarrow e \longmapsto^{\star}_{\mathbf{a}} v.$$

Our goal with the standard semantics was to settle on a canonical strategy for reducing programs. The properties above tell us that this canonical strategy (1) never produces something inconsistent with the semantics, and (2) always produces a value, if there is one.

These two properties tell us the standard semantics we have is a good basis for the specification of an interpreter, which can be defined by the iteration of the standard reduction function.

Note on the literature: Since the canonical reduction strategy is often what we are concerned with when reasoning about programs, most PL papers formulate only the standard reduction and are not concerned with the syntactic theory of their language, which establishes algebraic laws for proving the equality of programs.

Exercise 8. *Prove* $e \longmapsto_{\mathbf{a}}^{\star} v \iff e \to_{\mathbf{a}}^{\star} v$. □

We can model evaluation contexts as a data type in OCaml:

```
type ecxt  = Hole
           | EPred of ecxt
           | ESucc of ecxt
           | EPlusL of ecxt * arith
           | EPlusR of int * ecxt
           | EMultL of ecxt * arith
           | EMultR of int * ecxt
```

The function to plug an expression into a context is straightforward:

```
let rec plug (c : ecxt) (e : arith) : arith =
  match c with
    Hole -> e
  | EPred c -> Pred (plug c e)
  | ESucc c -> Succ (plug c e)
  | EPlusL (c, e') -> Plus (plug c e, e')
  | EPlusR (i, c) -> Plus (Int i, plug c e)
  | EMultL (c, e') -> Mult (plug c e, e')
  | EMultR (i, c) -> Mult (Int i, plug c e)
```

Decomposing a program into an evaluation context and a potential redex is easy too, although decomposition is undefined on values, so we need to model decompose as a partial function:

```
let rec decompose (e : arith) : (ecxt * arith) option =
  match e with
    Int i -> None
  | Pred (Int i) -> Some (Hole, e)
  | Succ (Int i) -> Some (Hole, e)
  | Plus (Int i, Int j) -> Some (Hole, e)
  | Mult (Int i, Int j) -> Some (Hole, e)
  | Pred e ->
    let Some (c', e') = decompose e in Some (EPred c', e')
  | Succ e ->
    let Some (c', e') = decompose e in Some (ESucc c', e')
  | Plus (Int i, e) ->
    let Some (c', e') = decompose e in Some (EPlusR (i, c'), e')
  | Plus (e, e2) ->
    let Some (c', e') = decompose e in Some (EPlusL (c', e2), e')
```

```
  | Mult (Int i, e) ->
    let Some (c', e') = decompose e in Some (EMultR (i, c'), e')
  | Mult (e, e2) ->
    let Some (c', e') = decompose e in Some (EMultL (c', e2), e')
```

This code relies on the following (easy to prove) fact that if an expression is not a value, then it decomposes into an evaluation context and potential redex. This means the non-exhaustive pattern matching warnings the compiler emits are spurious.

The evaluation function just iterates the decompose, reduce, plug process until a value is reached:

```
let rec eval (e : arith) : int =
  match e with
    Int i -> i
  | _ ->
    let Some (c, e') = decompose e in
    let Some r = a e' in
    eval (plug c r)
```

In `decompose e` we know that `e` is not a value, so `decompose` is defined on `e` (i.e. we cannot get `None`), and further more, `e'` is a redex, hence `a e'` is defined, so none of these pattern matches can fail.

A quick "proof by example" confirms that `eval` produces the same results as the OCaml implementation of the natural semantics from section 2.3:

```
# eval (Plus (Int 4, Succ (Int 2)));;
- : int = 7
```

2.7 Compiling

What is a compiler?

One view is the following: a compiler is a function that consumes a program and produces a program, which when run produces the same result as the given progam.

In other words,

```
let compile (e : arith) : (unit -> int) = ...
```

such that

```
∀e. compile e () = eval e
```

Make a correct compiler is trivial:

```
(* Correct compiler, ground truth *)
let rec compile_0 (e : arith) : (unit -> int) =
  (fun () -> eval e)

(* Simple test *)
let p = (Plus (Int 1, Int 2)) in
  eval p = compile_0 p ();;
```

While this compiler is trivial, it does create an explicit distinction between *compiling* and *running*. Code outside the `fun` is the compile-time code; code inside the `fun` is run-time code. The trivial compiler says at compile time, do nothing; at run-time, run the interpreter. Perfectly sensible.

We could however have the compiler actually do some work and hope to reap the benefits at run-time. Taken to its logical conclusion, the compiler could just do *all* of the work, then generate code that returns the result:

```
(* Optimal inlining compiler *)
let rec compile_1 (e : arith) : (unit -> int) =
  let i = eval e in
    (fun () -> i)
```

This compiler runs the program at compile-time and produces a constant function that returns the result. This approach won't work for richer languages, but it's not a bad way to optimize languages with decidable evaluation relations (such as $\mathcal{A}$).

A more scalable thing to do is to compile away the run-time overhead of syntactic case analysis for the expression being run:

```
(* Tagless run-time *)
let rec compile_2 (e : arith) : (unit -> int) =
  match e with
    | Int i -> (fun () -> i)
    | Pred e ->
       let c = compile_2 e in
         (fun () -> c () - 1)
    | Succ e ->
       let c = compile_2 e in
         (fun () -> c () + 1)
    | Plus (e1, e2) ->
       let c1 = compile_2 e1 in
       let c2 = compile_2 e2 in
         (fun () -> c1 () + c2 ())
   | Mult (e1, e2) ->
       let c1 = compile_2 e1 in
       let c2 = compile_2 e2 in
         (fun () -> c1 () * c2 ())
```

Notice that at run-time, the code does not dispatch on the kind of expression is being run; there's nothing left at run-time but OCaml's arithmetic operations and function calls to run code.

2.8 Again, with Ambiguity

Let's now replay the development of the various formulations of $\mathcal{A}$ with the slighest addition to the language. Let's add the following form of expression: *Amb*(e_1, e_2), which we'll take to mean "evaluate, ambiguously, to either the value of e_1 or e_2".

2.8.1 Natural Semantics

It's straightforward to extend the evaluation relation to cover the *Amb* form:

$$\frac{e_1 \Downarrow i}{\mathit{Amb}(e_1, e_2) \Downarrow i} \qquad \frac{e_2 \Downarrow i}{\mathit{Amb}(e_1, e_2) \Downarrow i}$$

However, this changes $\Downarrow$ from being a function from expressions to integers to a *relation* from expressions to integers. Because of this, we will run into trouble with our `eval` function in OCaml. Consider the following naive translation of the mathematical definition of $\Downarrow$ into OCaml in the presence of *Amb*:

```
type arith =
  | Int of int
  | Pred of arith
  | Succ of arith
  | Plus of arith * arith
  | Mult of arith * arith

let rec eval (e : arith) : int =
    match e with
    | Int i -> i
    | Pred e -> (eval e) - 1
    | Succ e -> (eval e) + 1
    | Plus (e1, e2) -> (eval e1) + (eval e2)
    | Mult (e1, e2) -> (eval e1) * (eval e2)
    | Amb (e1, e2) -> eval e1
    | Amb (e1, e2) -> eval e2
```

The problem here is that the left-hand sides of the *Amb* cases overlap. Since OCaml performs pattern matching in order, the last case is not reachable, and the OCaml compiler will tell us as much:

```
    | Amb (e1, e2) -> eval e2;;
      ^^^^^^^^^^^^
Warning 11: this match case is unused.
val eval : arith -> int = <fun>
```

The problem here is the mismatch between the eval function and the $\Downarrow$ relation. An idea to address the problem is to represent the relation as a function from expressions to *sets of* integers. While OCaml has a nice, general purpose set library, an easy hack is to represent a set as a list. Therefore the signature of `eval` will be `arith -> int list`.

Let's take the function in peices:

```
let rec eval (e : arith) : int list =
  match e with
```

The base case is easy: when given an integer, produce the list of its results, which is just a list consisting of that integer:

```
  | Int i -> [i]
```

When we recursively evaluate subexpressions, we no longer get an integer back, but instead a list of integers. In the case of *Pred*, we must then subtract one from each element to obtain the final result:

```
  | Pred e' -> List.map (fun v -> v-1) (eval e')
```

Successor is similar:

```
  | Succ e' -> List.map (fun v -> v+1) (eval e')
```

In the case of multiple subexpressions, we will have multiple lists of integers to combine into the final result. Suppose we have *Plus*(e_1, e_2) where

```
eval e1 = [1; 2; 3]
eval e2 = [5; 10]
```

Then we want to compute the sums of

```
[(1, 5); (2, 5); (3, 5); (1, 10); (2, 10); (3, 10)]
```

which is

```
[6; 7; 8; 11; 12; 13]
```

To compute the first list, we just use our helper function for computing the Cartesian product of two given lists. Now the cases for addition and multiplication are easy:

```
  | Plus (e1, e2) ->
      List.map (fun (v1,v2) -> v1+v2) (cartesian (eval e1) (eval e2))
  | Mult (e1, e2) ->
      List.map (fun (v1,v2) -> v1*v2) (cartesian (eval e1) (eval e2))
```

Finally we have the case of *Amb*. Suppose we have *Amb*(e_1, e_2) where, as before,

```
eval e1 = [1; 2; 3]
eval e2 = [5; 10]
```

Here we want to produce a list of all of the elements in either list, so the thing to do is append the two lists together:

```
  | Amb (e1, e2) -> eval e1 @ eval e2
```

Putting it all together, we have:

```
let rec eval (e : arith) : int list =
  match e with
  | Int i -> [i]
  | Pred e ->
      List.map (fun v -> v-1) (eval e)
  | Succ e ->
      List.map (fun v -> v+1) (eval e)
  | Plus (e1, e2) ->
      List.map (fun (v1,v2) -> v1+v2) (cartesian (eval e1) (eval e2))
  | Mult (e1, e2) ->
      List.map (fun (v1,v2) -> v1*v2) (cartesian (eval e1) (eval e2))
  | Amb (e1, e2) -> eval e1 @ eval e2
```

2.8.2 Reduction Semantics

Adding reduction axioms for *Amb* are straightforward:

$$\frac{}{\mathit{Amb}(i,j)\ \mathbf{a}\ i} \qquad\qquad \frac{}{\mathit{Amb}(i,j)\ \mathbf{a}\ j}$$

But just as $\Downarrow$ went from being a function to a relation once *Amb* was added, **a**, goes from a (partial) function to a relation. So we can no longer represent **a** as a partial function in OCaml and switch to a relation encoding:

```
let a (e : arith) : arith list =
  match e with
    | Pred (Int i) -> [Int (i-1)]
    | Succ (Int i) -> [Int (i+1)]
    | Plus (Int i, Int j) -> [Int (i+j)]
    | Mult (Int i, Int j) -> [Int (i*j)]
    | Amb (Int i, Int j) -> [Int i; Int j]
    | _ -> []
```

This necessitates a change to `step_a` and `standard_step_a`.

3 Variables, Conditionals, & Errors

The $\mathcal{A}$ language is about as simple a programming language as you're likely to find, although it served us well in demonstrating many of the core ideas behind syntax and semantics. As programmers, we like programming in rich, expressive languages, which $\mathcal{A}$ ain't. But as we enrich languages, we must also enrich our models. Let's look at a minimal increment to the $\mathcal{A}$ language and what it entails for our models. Let's add a new binary operator, *Div*, which performs integer division. Let's add a new binary operator, *Eq*, which determines if two integers are equal. Since an answer to the question "are these two numbers equal?" is either a yes or a no, let's add a new kind of value, Booleans, to the language to represent such answers. Booleans are useful for conditionally selecting computations, let's add a conditional form, $\mathit{If}(e_1, e_2, e_3)$, which will select e_2 when e_1 evaluates to *True* and e_3 when it evaluates to *False*. Finally, for good measure, let's throw in variables so that programs run on given inputs. We'll call this language $\mathcal{B}$ to distinguish it from $\mathcal{A}$.

The syntax of $\mathcal{B}$ is straightfoward to define. We add a couple nullary constructors for Booleans, a binary constructor for equality and for division, and a ternary constructor for conditionals. For variables, we assume there is an infinite, enumerable set of variable names, *Var*, and use lower-case, italic names to denote elements of that set. Two variables are considered "the same" if they are spelled the same:

$$\begin{array}{llcl}
\mathit{Var} & x & = & x \mid y \mid z \mid \ldots \\
\mathit{Bool} & b & = & \mathit{True} \mid \mathit{False} \\
\mathcal{B} & e & = & x \mid b \mid \mathit{Eq}(e,e) \mid \mathit{Div}(e,e) \mid \mathit{If}(e,e,e) \\
 & & \mid & i \mid \mathit{Pred}(e) \mid \mathit{Succ}(e) \mid \mathit{Plus}(e,e) \mid \mathit{Mult}(e,e)
\end{array}$$

Before delving into the details of the semantics, it's worth taking time to ponder over a few $\mathcal{B}$ programs and think informally about what they should mean:

1. *x*
2. *Eq*(*False*, 7)
3. *Div*(7, 0)
4. *If*(4, 1, 2)
5. *If*(*False*, *Div*(7, 0), 3)

Let's take the first one. What should *x* mean? Consider this question in the context of a mathematical formula: $x^2 + 4$. What does x mean here? The answer is: it depends. You can answer the question without first giving a meaning to the variable. So $x^2 + 4$ means 13 when $x = 3$ and it means 29 when $x = 5$. The meaning of the formula varies with the meaning of the variable. The same should be true for our programming language.

What about *Eq*(*False*, 7)? Here you could make different design choices. You might follow the JavaScript approach and produce false since *False* is different than 7. Or maybe true because the pointer representation of 7 and *False* turn out to be the same. Or maybe you return 42, because why not? Or you might do something sensible and raise an error because a numerical operator was applied to the wrong kind of argument?

In this section, we are going to take the simplest path forward. We will only consider well-behaved programs. In the case of the natural semantics, this means programs such as *If*$(4, 1, 2)$ are literally *meaningless*: the semantic relation will be undefined on such programs. In the case of the reduction semantics, such programs "get stuck": they reach a bad state that cannot proceed further; errors manifest as irreducible programs that are not values.

3.1 Natural Semantics

Let's identify a set of *values* and a set of *answers*. Values include integers and Booleans. Answers, which represent the final results of evaluation are for the moment just values:

$$\begin{array}{lrcl} \textit{Val} & v & = & b \mid i \\ \textit{Ans} & a & = & v \end{array}$$

The evaluation of programs containing variables depends on an environment which gives meaning to these variables. In the natural semantics is modelled as a ternary relation $(\cdot \vdash \cdot \Downarrow \cdot) \subseteq (\textit{Var} \rightharpoonup \textit{Val}) \times \mathcal{B} \times \textit{Ans}$. The relation includes an **environment** which maps variables to values. Environments, ranged over by meta-variable ρ, are partial functions from variables to values. Values are self evaluating. Variables evaluate to the value given by the environment. Division performs the integer division of its arguments when they evaluate to numeric values and the denominator is non-zero. Equality is defined as you might expect, producing true when its arguments evaluate the same number, false when they evaluate to different numbers. Conditionals select which value to produce based on

the evaluation of the test expression.

$$\frac{}{\rho \vdash v \Downarrow v} \qquad \frac{\rho(x) = v}{\rho \vdash x \Downarrow v} \qquad \frac{\rho \vdash e_1 \Downarrow i \qquad \rho \vdash e_2 \Downarrow j \qquad j \neq 0 \qquad k = \lfloor i/j \rfloor}{\rho \vdash \mathit{Div}(e_1, e_2) \Downarrow k}$$

$$\frac{\rho \vdash e_1 \Downarrow i \qquad \rho \vdash e_2 \Downarrow j \qquad i = j}{\rho \vdash \mathit{Eq}(e_1, e_2) \Downarrow \mathit{True}} \qquad \frac{\rho \vdash e_1 \Downarrow i \qquad \rho \vdash e_2 \Downarrow j \qquad i \neq j}{\rho \vdash \mathit{Eq}(e_1, e_2) \Downarrow \mathit{False}}$$

$$\frac{\rho \vdash e_1 \Downarrow \mathit{True} \qquad \rho \vdash e_2 \Downarrow a}{\rho \vdash \mathit{If}(e_1, e_2, e_3) \Downarrow a} \qquad \frac{\rho \vdash e_1 \Downarrow \mathit{False} \qquad \rho \vdash e_3 \Downarrow a}{\rho \vdash \mathit{If}(e_1, e_2, e_3) \Downarrow a}$$

We can also assume the omitted rules for handling expressions from $\mathcal{A}$, which are just like before but carrying an environment.

The important to note here is what is missing. There's no definition for what happens, for example, when x is not defined in ρ, nor when $j = 0$ in the rule for *Div*. Likewise, a program that compares non-numeric values for equality will not have an answer according to $\Downarrow$.

This semantics still has the nice property of giving the same answer whenever it gives an answer, but unlike the natural semantics for $\mathcal{A}$, it does not give an answer for all programs. It is therefore a partial function.

3.2 Compiling, without Errors

Here's a compiler for the $\mathcal{B}$ language, minus errors.

```
type barith =
  | Int of int
  | Pred of barith
  | Succ of barith
  | Plus of barith * barith
  | Mult of barith * barith
  (* new stuff *)
  | Var of string
  | Bool of bool
  | If of barith * barith * barith
  | Div of barith * barith

type value =
  | VInt of int
  | VBool of bool
  | VError of string

type env = (string * value) list

let rec lookup (r : env) (x : string) : value =
```

```
  match r with
    | (y,v)::r' -> if y=x then v else lookup r' x

let rec compile (e : barith) : (env -> value) =
  match e with
    | Int i -> (fun _ -> VInt i)
    | Bool b -> (fun _ -> VBool b)
    | Pred e ->
        let c = compile e in
        (fun env ->
           let VInt i = c env in
             VInt (i-1))
    | Succ e ->
        let c = compile e in
          (fun env ->
             let VInt i = c env in
               VInt (i+1))
    | Mult (e1, e2) ->
        let c1 = compile e1 in
        let c2 = compile e2 in
          (fun env ->
             let VInt i = (c1 env) in
             let VInt j = (c2 env) in
               VInt (i*j))
    | Plus (e1, e2) ->
        let c1 = compile e1 in
        let c2 = compile e2 in
          (fun env ->
             let VInt i = (c1 env) in
             let VInt j = (c2 env) in
               VInt (i+j))
    | Div (e1, e2) ->
        let c1 = compile e1 in
        let c2 = compile e2 in
          (fun env ->
             let VInt i = (c1 env) in
             let VInt j = (c2 env) in
               VInt (i/j))
    | If (e1, e2, e3) ->
        let c1 = compile e1 in
        let c2 = compile e2 in
        let c3 = compile e3 in
          (fun env ->
             let VBool b = c1 env in
               if b=true then c2 env else c3 env)
```

```
| Var x ->
    (fun env -> lookup env x)
```

3.3 Reduction Semantics

Modelling $\mathcal{B}$ with reduction semantics is pretty easy. The reduction axioms are just the following (and we assume **a**-like axioms, too):

$$\frac{\rho(x) = v}{\rho \vdash x \ \mathbf{b} \ v} \qquad \frac{i = j}{\rho \vdash \mathit{Eq}(i,j) \ \mathbf{b} \ \mathit{True}} \qquad \frac{}{\rho \vdash \mathit{If}(\mathit{True}, e_1, e_2) \ \mathbf{b} \ e_1} \qquad \frac{}{\rho \vdash \mathit{If}(\mathit{False}, e_1, e_2) \ \mathbf{b} \ e_2}$$

$$\frac{i \neq j}{\rho \vdash \mathit{Eq}(i,j) \ \mathbf{b} \ \mathit{False}} \qquad \frac{j \neq 0 \qquad k = \lfloor i/j \rfloor}{\rho \vdash \mathit{Div}(i,j) \ \mathbf{b} \ k}$$

Notice that like the natural semantics, the meaning of a program is given in terms of an environment.

The $\rightarrow_{\mathbf{b}}$ relation is the closure of **b** for compatibility with the syntax of expressions; The $\rightarrow_{\mathbf{b}}^{\star}$ relation is the relfexive, transitive closure of the $\rightarrow_{\mathbf{b}}$ relation. The $=_{\mathbf{b}}$ equivalence relation is the symmetric closure of the $\rightarrow_{\mathbf{b}}^{\star}$ relation. The syntactic theory is consistent and standard reduction relation $\longmapsto_{\mathbf{b}}$ can be defined as the contextual closure of **b** over the following grammar of evaluation contexts (plus the grammar of $\mathcal{E}$ for $\mathcal{A}$):

$$\begin{array}{lrcl} \mathit{EvalContext} & \mathcal{E} & = & \ldots \\ & & \mid & \mathit{Eq}(\mathcal{E}, e) \mid \mathit{Eq}(v, \mathcal{E}) \\ & & \mid & \mathit{Div}(\mathcal{E}, e) \mid \mathit{Div}(v, \mathcal{E}) \\ & & \mid & \mathit{If}(\mathcal{E}, e, e) \end{array}$$

The usual standardization property holds for $\mathcal{B}$.

Unlike the natural semantics, we can give a partial meaning to programs that reach bad states because they reduce until the reduction axiom cannot be applied, for example:

$$\mathit{Plus}(3, \mathit{Mult}(2, \mathit{Div}(5, \mathit{Pred}(1)))) \ \rightarrow_{\mathbf{b}} \ \mathit{Plus}(3, \mathit{Mult}(2, \mathit{Div}(5, 0)))$$

The expression on the right cannot take any further steps. Since it is not a value and cannot step further, we say it is **stuck**.

3.4 Meaningful Errors

The development of the previous section, which essentially ignores errors in the semantic models of $\mathcal{B}$, can be a useful and simple way of conceiving of a programming language. It can also lead to issues, which depending on the setting, can be problematic. Undefined behavior in languages such as C are the source of major security vulnerabilities, for example. Sometimes it is worth modelling errors explicitly. For example, we may want to know what kinds of errors are possible, and if errors are possible, which code component is at fault for the error. In the remainder of this section, we look at various ways of modelling errors for our simple language. Although this is a simpified setting, we will see that errors,

which are a primitive form of **effect**, complicate reasoning about programs. In particular, we must be careful in designing the syntactic theory in order to remaing consistent.

For a proper account of error behavior, we need to designate a set of errors and include them in the set of answers that a program may compute:

$$\begin{array}{llll} \mathit{Ans} & a & = & \ldots \mid r \\ \mathit{Err} & r & = & \mathit{Err}_\ell \text{ where } \ell \text{ is in some set} \end{array}$$

We'll assume there are an infinite number of different errors and write subscripts to distinguish them (ℓ ranges over these subscripts).

3.5 Natural Semantics for Imprecise Errors

Errors can arise when a variable is unbound (it is not defined in the environment), when division is performed with a denominator of 0, or when the wrong kind of value is used for an operation, such as adding a Boolean and an integer or using *If* with a non-Boolean test. A variable is unbound when it is not in the domain of the environment; formally: $\mathsf{dom}(\rho) = \{x \mid \rho(x) = v \text{ for some } v\}$. The following rules define when an error is created:

$$\frac{x \notin \mathsf{dom}(\rho)}{\rho \vdash x \Downarrow \mathit{Err}_x} \qquad \frac{\rho \vdash e_1 \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Eq}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Eq}1}} \qquad \frac{\rho \vdash e_2 \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Eq}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Eq}2}}$$

$$\frac{\rho \vdash e_1 \Downarrow i \qquad \rho \vdash e_2 \Downarrow j \qquad j = 0}{\rho \vdash \mathit{Div}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Div}0}} \qquad \frac{\rho \vdash e_1 \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Div}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Div}1}} \qquad \frac{\rho \vdash e_2 \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Div}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Div}2}}$$

$$\frac{\rho \vdash e_1 \Downarrow v \qquad v \notin \mathit{Bool}}{\rho \vdash \mathit{If}(e_1, e_2, e_3) \Downarrow \mathit{Err}_{\mathit{If}}}$$

For completeness, we should also give the rules for expressions from $\mathcal{A}$:

$$\frac{\rho \vdash e \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Pred}(e) \Downarrow \mathit{Err}_{\mathit{Pred}}} \qquad \frac{\rho \vdash e \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Succ}(e) \Downarrow \mathit{Err}_{\mathit{Succ}}} \qquad \frac{\rho \vdash e_1 \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Plus}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Plus}1}}$$

$$\frac{\rho \vdash e_2 \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Plus}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Plus}2}} \qquad \frac{\rho \vdash e_1 \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Mult}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Mult}1}} \qquad \frac{\rho \vdash e_2 \Downarrow v \qquad v \notin \mathit{Int}}{\rho \vdash \mathit{Mult}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Mult}2}}$$

While the above rules take care of creating errors when they arise, no rules have dealt with the propagation of errors. For example, we should expect $\mathit{Eq}(\mathit{Div}(3, 0), 4)$ to produce a divide by zero error, although as it stands the semantics is undefined. Adding the following rules propagates errors from subexpressions in *Eq*:

$$\frac{\rho \vdash e_1 \Downarrow r}{\rho \vdash \mathit{Eq}(e_1, e_2) \Downarrow r} \qquad \frac{\rho \vdash e_2 \Downarrow r}{\rho \vdash \mathit{Eq}(e_1, e_2) \Downarrow r}$$

We need similar rules for all of the remaining syntactic constructors.

Properties of the semantics: Now that the semantics is in place, let's step back and consider some of its properties. The addition of errors has significantly complicated the formal development—we had to add a bunch of rules for signal and propogating errors. But dealing with errors and establishing the meaning of errors is a crucial aspect of semantic engineering. Remember that a semantics is useful for defininining properties of programs, and one of the more useful properties is "this program causes no errors." So to even talk about this, we need to confront error behaviors in the semantics.

But beyond the additional rules, errors, at least as we have formulated them, have also caused an important change to our semantics. Unlike in the error-free case of $\mathcal{A}$, our evaluation relation is no longer a function. To see this, consider $\mathit{Eq}(\mathit{Div}(3,0),\mathit{False})$. This expression either produces a divide by zero error ($\mathit{Err}_{\mathit{Div}0}$), or a type error ($\mathit{Err}_{\mathit{Eq}2}$) for giving a Boolean to *Eq*. This is concerning for anyone who wants to write an interpreter for, or reason about, $\mathcal{B}$ programs. If we were writing an interpreter, we should wonder "what should it produce for $\mathit{Eq}(\mathit{Div}(3,0),\mathit{False})$?" One answer is an interpreter only has to find *some* answer that is consistent with the semantics; so if `eval` $\rho\ e = v$, then $\rho \vdash e \Downarrow v$, but the reverse direction would not hold (the interpreter then, is an abstraction of the natural semantics). Many languages take this approach. Haskell for example, uses an imprecise exception semantics[5], basically following the outline we have above. Other languages choose to *specify* which error should be signalled by determinizing the language and restoring the functional nature of evaluation.

3.6 Natural Semantics for Precise Errors

The thing leading to the observed non-determinism of the evaluator is the overlap in hypothesis in the different inference rules. Consider the error propagation rules for *Eq*. It's possible that $\rho \vdash e_1 \Downarrow b$ *and* $\rho \vdash e_2 \Downarrow b'$, thus either rule could apply leading to distinct answers. By grade school analogy, it's as if the answer to a basic algebra problem depended on the order in which you simplified terms. A solution then is to specify more rigorously the order in which subexpressions should be evaluated, which can be achieved by making the hypotheses of the inference rules non-overlapping. For example, if we want to evaluate *Eq* expressions left-to-right, we can use the following rules for error generation and propogation:

$$\frac{\rho \vdash e_1 \Downarrow b \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash \mathit{Eq}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Eq}1}} \qquad\qquad \frac{\rho \vdash e_1 \Downarrow i \qquad \rho \vdash e_2 \Downarrow b}{\rho \vdash \mathit{Eq}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Eq}2}}$$

Exercise 9. *Implement an interpreter for the imprecise error semantics as a function in OCaml. Your interpreter is free to produce any answer that is consistent with the semantics.* □

Exercise 10. *Following the idea above, develop an alternative semantics for $\mathcal{B}$ that is deterministic, i.e. the evaluation relation is a function. Prove that it is in fact a function. Implement this function in OCaml. Is this function a satisfying solution to the previous problem?* □

[5] `http://research.microsoft.com/en-us/um/people/simonpj/papers/imprecise-exn.htm`

3.7 Consistent Syntactic Theory

Now that we've seen the complications errors introduce in the setting of natural semantics, let's look at the same development using reduction semantics.

First, we will need to include *Err*s in the syntax of $\mathcal{B}$ programs to model reduction:

$$\mathit{Exp} \quad e \quad = \quad \ldots \mid r$$

Next, let's define two reduction axioms. The first, which we'll call **err**, creates errors. The second, **prop**, propagates them. Some of rules for **err** are obvious:

$$\frac{x \notin \mathsf{dom}(\rho)}{\rho \vdash x \ \mathbf{err}\ \mathit{Err}_x} \qquad \frac{}{\rho \vdash \mathit{Div}(i,0)\ \mathbf{err}\ \mathit{Err}_{\mathit{Div}0}} \qquad \frac{v \notin \mathit{Bool}}{\rho \vdash \mathit{If}(v,e_1,e_2)\ \mathbf{err}\ \mathit{Err}_{\mathit{If}}}$$

$$\frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Pred}(v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Plus}}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Succ}(v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Succ}}}$$

But we would be wrong to add the following axioms:

$$\frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Plus}(e,v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Plus}1}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Plus}(v,e)\ \mathbf{err}\ \mathit{Err}_{\mathit{Plus}2}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Mult}(e,v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Mult}1}}$$

$$\frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Mult}(v,e)\ \mathbf{err}\ \mathit{Err}_{\mathit{Mult}2}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Eq}(e,v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Eq}1}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Eq}(v,e)\ \mathbf{err}\ \mathit{Err}_{\mathit{Eq}2}}$$

$$\frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Div}(e,v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Div}1}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Div}(v,e)\ \mathbf{err}\ \mathit{Err}_{\mathit{Div}2}}$$

The problem with these rules is they would immediately make the syntactic theory inconsistent and identify all errors as equal. For example, we can have $\vdash \mathit{Eq}(\mathit{Err}_1,\mathit{Err}_2)\ \mathbf{err}\ \mathit{Err}_1$ and $\vdash \mathit{Eq}(\mathit{Err}_1,\mathit{Err}_2)\ \mathbf{err}\ \mathit{Err}_2$, which induces the equivalence $\vdash \mathit{Err}_1 =_{\mathbf{err}} \mathit{Err}_2$. We also have to be careful with how errors are propagated. At first glance, you might think to propagate errors past each of the syntactic constructors. So for example, you might have the following propagation rules for *Eq*, and similar rules for all the other constructors:

$$\frac{}{\rho \vdash \mathit{Eq}(r,e)\ \mathbf{prop}\ r} \qquad \frac{}{\rho \vdash \mathit{Eq}(e,r)\ \mathbf{prop}\ r}$$

But this also causes inconsistency. Moreover, any program which *syntactically* contains an error expression (or an expression that reduces to an error) would be equated with that error in the theory. That's not a useful equational theory.

Instead, we want to limit the **err** relation in the following way. In order for a type error

to be created, the subexpressions must be values:

$$\frac{v_1 \notin \mathit{Int}}{\rho \vdash \mathit{Plus}(v_1, v_2)\ \mathbf{err}\ \mathit{Err}_{\mathit{Plus}1}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Plus}(i, v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Plus}2}} \qquad \frac{v_1 \notin \mathit{Int}}{\rho \vdash \mathit{Mult}(v_1, v_2)\ \mathbf{err}\ \mathit{Err}_{\mathit{Mult}1}}$$

$$\frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Mult}(i, v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Mult}2}} \qquad \frac{v_1 \notin \mathit{Int}}{\rho \vdash \mathit{Eq}(v_1, v_2)\ \mathbf{err}\ \mathit{Err}_{\mathit{Eq}1}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Eq}(i, v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Eq}2}}$$

$$\frac{v_1 \notin \mathit{Int}}{\rho \vdash \mathit{Div}(v_1, v_2)\ \mathbf{err}\ \mathit{Err}_{\mathit{Div}1}} \qquad \frac{v \notin \mathit{Int}}{\rho \vdash \mathit{Div}(i, v)\ \mathbf{err}\ \mathit{Err}_{\mathit{Div}2}}$$

Likewise, when propagating errors, we only allow propagation when subexpressions to the left of the error are values:

$$\frac{}{\rho \vdash \mathit{Eq}(r, e)\ \mathbf{prop}\ r} \qquad \frac{}{\rho \vdash \mathit{Eq}(v, r)\ \mathbf{prop}\ r}$$

The remaining rules for *Pred*, *Succ*, *Div*, *Plus*, and *Mult* are similar. One notable exception is *If*. We only allow errors to propagate past an *If* if the error occurs in the test position:

$$\frac{}{\rho \vdash \mathit{If}(r, e_1, e_2)\ \mathbf{prop}\ r}$$

We can now define a small-step reduction relation. Let's call it $\rightarrow_{\mathbf{bep}}$ to distinguish it from $\rightarrow_{\mathbf{b}}$. We define $\rightarrow_{\mathbf{bep}}$ as the compatible closure of $(\mathbf{b} \cup \mathbf{err} \cup \mathbf{prop})$ over the grammar of $\mathcal{B}$ expressions; $\rightarrow^{\star}_{\mathbf{bep}}$ is the reflexive transitive closure of $\rightarrow_{\mathbf{bep}}$, and $=_{\mathbf{bep}}$ as the symmetric closure of $\rightarrow^{\star}_{\mathbf{bep}}$.

Note that this theory allows us to reduce inside of *If* expressions, but if an error occurs, we cannot propogate it out until the test expression produces a value and the *If* is reduced to the consequent or alternative based on it.

Exercise 11 (Consistency)**.** *Prove if* $\rho \vdash e =_{\mathbf{bep}} a$ *and* $\rho \vdash e =_{\mathbf{bep}} a'$*, then* $a = a'$*.* □

Exercise 12. *Prove* $\rho \vdash e \ \rightarrow^{\star}_{\mathbf{bep}}\ v \iff \rho \vdash e \ \rightarrow^{\star}_{\mathbf{b}}\ v$*.* □

The approach of using evaluation contexts to specify canonical reductions scales up to more complicated languages such as $\mathcal{B}$ with errors. In fact, the notion of evaluation contexts does not change from what has already been defined for $\mathcal{B}$, so the standard reduction relation can be defined as:

$$\frac{\rho \vdash e\ (\mathbf{b} \cup \mathbf{err} \cup \mathbf{prop})\ e'}{\rho \vdash \mathcal{E}[e] \ \longmapsto_{\mathbf{bep}}\ \mathcal{E}[e']}$$

Exercise 13 (Standardization)**.** *Prove* $\rho \vdash e \ \longmapsto^{\star}_{\mathbf{bep}}\ a \iff \rho \vdash e \ \rightarrow^{\star}_{\mathbf{bep}}\ a$*.* □

Exercise 14. *Implement the standard reduction relation as a function in OCaml. Implement an interpreter for* $\mathcal{B}$ *by iterating the standard reduction relation until an answer is produced.* □

3.8 Discarding the Context

When introducing contexts in section 2.5, we noted that formulating reductions explicitly in terms of contexts gives us expressive power to write new kinds of reductions that are difficult to do without a context.

To see an example, consider how programs produce errors in our reduction semantics. We have some expression, say $\mathit{Plus}(3, \mathit{Mult}(2, \mathit{Div}(5, 0)))$, which introduces an error on the next step and then step by step propagates the error outward until it is the final answer:

$$\mathit{Plus}(3, \mathit{Mult}(2, \mathit{Div}(5, 0))) \rightarrow_{\mathbf{bep}} \mathit{Plus}(3, \mathit{Mult}(2, \mathit{Err}_{\mathit{Div}0})) \rightarrow_{\mathbf{bep}} \mathit{Plus}(3, \mathit{Err}_{\mathit{Div}0}) \rightarrow_{\mathbf{bep}}$$

$$\mathit{Err}_{\mathit{Div}0}$$

Achieving this step-by-step bubbling up of errors tedious because it required a bunch of reduction axioms of the form $\mathit{Mult}(v, \mathit{Err}_\ell)$ **prop** Err_ℓ. We can avoid the propagating reductions and intermediate steps in the standardized semantics.

Consider the following alternative standard reduction relation:

$$\frac{\rho \vdash e \ (\mathbf{b} \cup \mathbf{err}) \ e'}{\rho \vdash \mathcal{E}[e] \longmapsto_{\mathbf{be}} \mathcal{E}[e']} \qquad \frac{\mathcal{E} \neq \square}{\rho \vdash \mathcal{E}[\mathit{Err}_\ell] \longmapsto_{\mathbf{be}} \mathit{Err}_\ell}$$

The example now immediately jumps to the final error state:

$$\rho \vdash \mathit{Plus}(3, \mathit{Mult}(2, \mathit{Err}_{\mathit{Div}0})) \longmapsto_{\mathbf{be}} \mathit{Err}_{\mathit{Div}0}.$$

Exercise 15. *Prove* $\rho \vdash e \longmapsto^\star_{\mathbf{be}} a \iff \rho \vdash e \longmapsto^\star_{\mathbf{bep}} a$. □

4 Redex: a Domain-Specific Language for Semantics

Redex is a scripting language and set of associated tools supporting the conception, design, construction, and testing of semantic systems such as programming languages, type systems, program logics, and program analyses. As a scripting language, it enables an engineer to create executable specifications of common semantic elements such as grammars, reduction relations, judgments, and metafunctions; the basic elements of formal systems. It includes a number of software engineering inspired features to make models robust and includes tools for typesetting mod- els and generating algebraic steppers for exploring program behaviors. In brief, Redex supports all phases of the semantic engineering life-cycle.

4.1 Racket: a General-Purpose Host Language

Redex is a domain-specific language (DSL) embeded within the general-purpose language Racket. Racket is a descendant of Lisp and adopts much of Lisp's fully parenthesized prefix notation. It comes with an interactive development environment, called DrRacket, and has a large corpus of libraries. It has been developed over the last twenty or so years by the PLT research group, which is distributed across several different universities in the USA.

The group has developed Racket in part to support their educational work and in part as a vehicle for researching programming languages.

The basic unit of code in Racket is a module, which must first declare which language it is written in with the `#lang` directive:

```
#lang racket
```

It turns out Racket is more than just a single programming language but a ecosystem of many different languages than can interoperate. The `racket` language is the core language of the system, but there are many others, such as a typed variant of `racket` called `typed/racket`, Datalog, Algol 60, a documentation language called `scribble`, and many more. We will only be concerned with a very small subset of `racket`, though.

After the language declaration, a module may declare what it requires and provides, followed by a series of definitions or expressions. For example, this module defines and then uses the factorial function:

```
#lang racket
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))

(fact 5)
```

When run inside DrRacket (by pressing the "Run" button), the window is split into two parts: the definitions area, which contains the source code of the module, and the interactions area, which can be used to interact with the program above. An example is shown in figure 1.

The interactions area has a prompt for accepting expressions to evaluate. Any definitions from the module are avaiable within the interactions window, so we can experiment and calculate other results of factorial:

```
> (fact (fact 5))
66895029134491270575881180540903725867527463331380298102956713523016335572449629893668741652719849813081576378932140905525344085894081218598984811143896500059649605212569600000000000000000000000000000000
```

Changes that are made to the module are not immediately reflected in the interactions window. The program has to be re-run, which creates a fresh interactions window, discarding anything you may have entered. (But scrolling through a history of expressions is possible with Cmd+Up.)

4.2 Defining a Language

To construct a Redex model of a language, the first thing to do is declare the host language (`racket`) and require the Redex library:

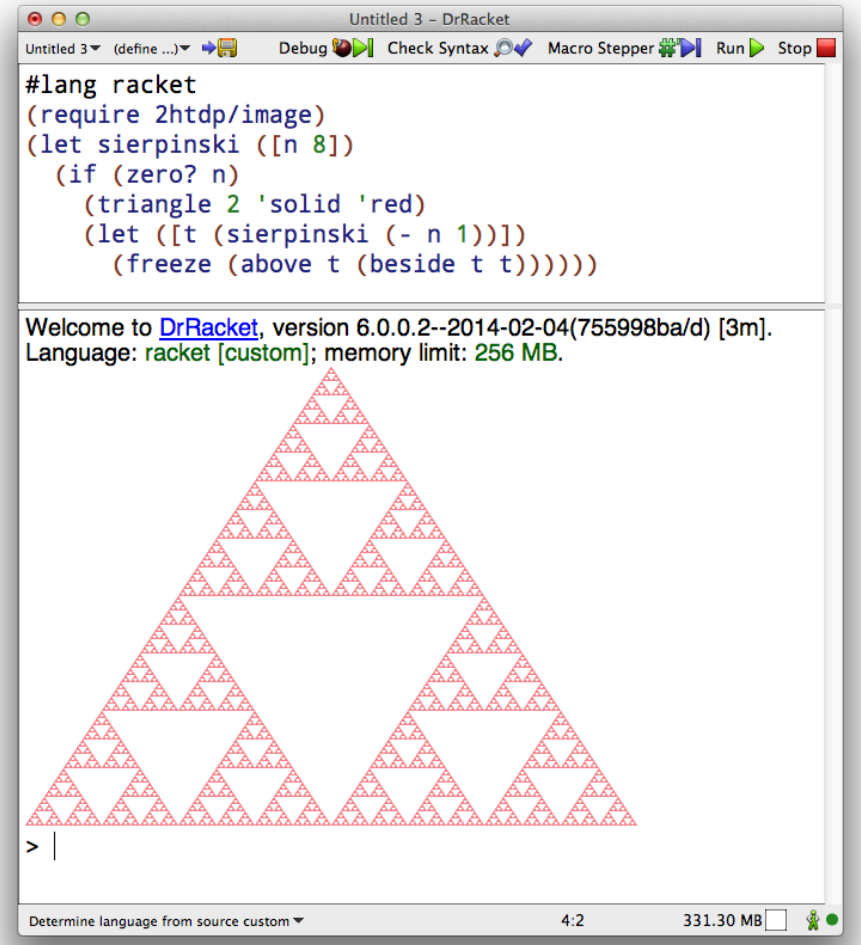


Figure 1: The DrRacket interactive development environment.

```
#lang racket
(require redex)
```

Defining the syntax of a language is accomplished with `define-language`, which gives the grammar of terms for the language:

```
(define-language A
  (e ::=
     v
     (Pred e)
     (Succ e)
     (Plus e e)
     (Mult e e))
  (v ::= i)
  (i j k ::= integer))
```

This establishes the set of terms in the language `A`, and also establishes a number of meta-variable names which can be used to pattern match terms.

For example, we can query the language to see if the term `4` is in the set of values ranged over by meta-variable `e`:

```
> (redex-match? A e (term 4))
#t
```

It's also true that `i` ranges over `4`:

```
> (redex-match? A i (term 4))
#t
```

But while `(Pred 4)` is in `e`, it is not in `i`:

```
> (redex-match? A e (term (Pred 4)))
#t
> (redex-match? A i (term (Pred 4)))
#f
```

4.3 Terms

Terms in Redex are constructed with the `term` form. To a first approximation, `term` is just a synonym for `quote`, which constructs literal data. In fact we could have constructed our examples in the previous section with `quote`, which can be written `(quote e)` or `'e` for short:

```
> (redex-match? A e '(Pred 4))
#t
> (redex-match? A i '(Pred 4))
#f
```

The difference between `term` and `quote` is that `term` interprets some of the literal data. The simplest example of this is that `term` interprets occurrences of bound meta-variables and substitutes their values for the occurrences. For example, we can define a term variable with `define-term` and then refer to it within `term` expressions:

```
> (define-term five 5)
> (term five)
5
> (quote five)
'five
```

As we'll see, Redex comes with a powerful pattern matcher that let's us match terms against term patterns and bind variables for use within terms.

Another feature of `term` is that it is possible to escape out of the term language and back in to Racket. This is accomplished with `unquote`. The notation `,e` is a Racket shorthand for `(unquote e)` and it signals to Redex not to interpret `e` as a term, but rather as a Racket expression which evaluates to a term. In other words. Using the `term` constructor within the expression re-enters the Redex world, and this nesting can be arbitrarily deep. Here are some examples:

```
> (term (Plus (add1 5)))
'(Plus (add1 5))
> (term (Plus ,(add1 5)))
'(Plus 6)
> (term (Plus ,(add1 (term five))))
'(Plus 6)
```

4.4 Defining a Reduction Relation

A reduction relation is a value in Redex, constructed with the `reduction-relation` form. Defining the **a** relation is achieved with:

```
(define a
  (reduction-relation
   A
   (--> (Pred i) ,(sub1 (term i)) pred)
   (--> (Succ i) ,(add1 (term i)) succ)
   (--> (Plus i j) ,(+ (term i) (term j)) plus)
   (--> (Mult i j) ,(* (term i) (term j)) mult)))
```

A `reduction-relation` specifies which language it operates on, in this case `A`. The language determines the interpretation of the meta-variables (and what whether a peice of syntax is a meta-variable). The relation is written as a set of axioms, each of which has the form `(-->` L R *name*`)` where L is a **pattern** and R is a **template** and *name* is an optional name for the rule. Applying a reduction relation to a term matches the term against each pattern and if the match is succesful produces the template with each meta-variable replaced by the corresponding substructure of the pattern match.

A reduction relation can be applied with `apply-reduction-relation`:

```
> (apply-reduction-relation a (term (Pred 5)))
'(4)
```

When the relation is not defined on a term, the empty list is produced:

```
> (apply-reduction-relation a (term (Mult (Pred 4) 5)))
'()
```

There are a few operations on reduction relations that produce new reduction relations. For example, `compatible-closure` computes the compatible closure of a relation:

```
> (apply-reduction-relation (compatible-closure a A e)
                            (term (Mult (Pred 4) 5)))
'((Mult 3 5))
```

Like any value, we can name this relation using `define`:

```
(define ->a (compatible-closure a A e))
```

The `apply-reduction-relation*` function computes the set of irreducible terms that are in the transitive closure of a given relation:

```
> (apply-reduction-relation* ->a (term (Mult (Pred 4) 5)))
'(15)
```

The `traces` function will visualize the reduction semantics as a graph of related terms. For example,

```
> (traces ->a (term (Mult (Plus (Succ 2) (Mult 2 2)) (Plus 5 5))))
```

will launch a window displaying the graph in figure 2.

4.5 Defining metafunctions

A metafunction is a function that is interpreted within the term language. Here is the natural semantics function written as a metafunction in Redex:

```
(define-metafunction A
  [(eval v) v]
  [(eval (Pred e)) ,(sub1 (term (eval e)))]
  [(eval (Succ e)) ,(add1 (term (eval e)))]
  [(eval (Plus e_1 e_2))
   ,(+ (term (eval e_1))
       (term (eval e_2)))]
  [(eval (Mult e_1 e_2))
   ,(* (term (eval e_1))
       (term (eval e_2)))])
```

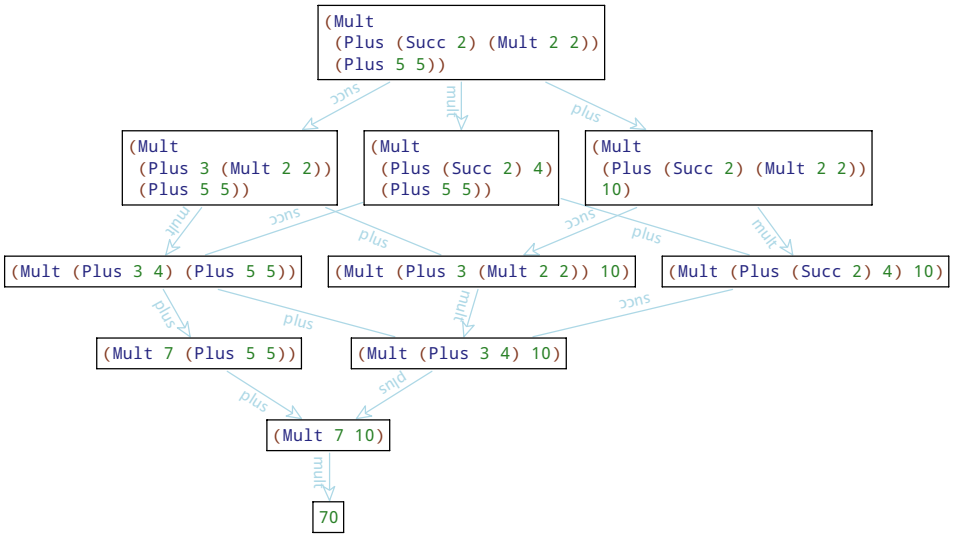


Figure 2: Example reduction graph.

The function is defined by a series of pattern and template clauses, much like a reduction relation but without the `-->` notation. Unlike a reduction relation, the patterns are tried in order and the result of the metafunction application is the instantiated template of the first matching clause. An error is signalled if no clause matches.

Once a metafunction is defined, it is applied whenever it occurs with a `term`:

```
> (term (eval (Mult (Pred 5) (Succ 8))))
36
> (term (Mult (Pred 5) (eval (Succ 8))))
'(Mult (Pred 5) 9)
```

It's important to notice that this defines a *function*, not a *relation*. Since we know that $\Downarrow$ for $\mathcal{A}$ is a function, this is a correct model of $\Downarrow$, but it would not be if $\Downarrow$ were realy a relation, such as with the natural semantics of imprecise errors for $\mathcal{B}$. To model relations (that are not reduction relations), we need another mecahnism: `define-judgment-form`.

4.6 Contexts

Evaluation contexts can also be specified as part of a language's grammar. We can add the following to the `define-language` for `A`:

```
(E ::=
   hole
   (Pred E)
   (Succ E)
   (Mult E e)
   (Mult v E)
```

```
    (Plus E e)
    (Plus v E))
```

Within a reduction relation and other patterns, `in-hole` can be used to match terms within a context. For example, the standard reduction relation can be defined as:

```
(define -->a
  (reduction-relation
   A
   (--> (in-hole E (Pred i)) (in-hole E ,(sub1 (term i))))
   (--> (in-hole E (Succ i)) (in-hole E ,(add1 (term i))))
   (--> (in-hole E (Mult i j)) (in-hole E ,(* (term i) (term j))))
   (--> (in-hole E (Plus i j)) (in-hole E ,(+ (term i) (term j))))))
```

Since each of these rules has a repeated structure of using `in-hole`, it's possible to define a shortcut:

```
(define -->a
  (reduction-relation
   A
   (~~> (Pred i) ,(sub1 (term i)))
   (~~> (Succ i) ,(add1 (term i)))
   (~~> (Mult i j) ,(* (term i) (term j)))
   (~~> (Plus i j) ,(+ (term i) (term j)))
   with
   [(--> (in-hole E e_1) (in-hole E e_2))
    (~~> e_1 e_2)]))
```

Or, we could simple compute the relation by using `context-closure` and the existing reduction relation `a`:

```
(define -->a
  (context-closure a A E))
```

4.7 Defining a Judgment (Relation)

Defining a relation is accomplished with the `define-judgment-form` notation:

```
(define-judgment-form A
  #:mode (evalr I O)
  [(evalr v v)]
  [(evalr e v)
   -----------
   (evalr (Pred e) ,(sub1 (term v)))]
  [(evalr e v)
   -----------
   (evalr (Succ e) ,(add1 (term v)))]
  [(evalr e_1 v_1)
```

```
   (evalr e_2 v_2)
   ---------------
   (evalr (Plus e_1 e_2) ,(+ (term v_1) (term v_2)))]
  [(evalr e_1 v_1)
   (evalr e_2 v_2)
   ---------------
   (evalr (Mult e_1 e_2) ,(* (term v_1) (term v_2)))])
```

A judgment is a relation that is being modelled computationally as a function. The `#:mode` keyword gives a mode specification which declares which parts of the relation can be consider inputs (`I`) and which parts can be considered outouts (`O`), which can be thought of as specifying which positions are inputs and outputs in this function.

A judgment is used with `judgment-holds` form. It can be used in two ways, the first is to check if the relation holds on particular values. So for example, we can check that `(Succ (Succ 4))` evaluates to `6` with:

```
> (judgment-holds (evalr (Succ (Succ 4)) 6))
#t
```

It's also possible to compute the set of all values related to `(Succ (Succ 4))` with:

```
> (judgment-holds (evalr (Succ (Succ 4)) v) v)
'(6)
```

The set of related terms is given as a list of results (in case, it's always a singleton list).

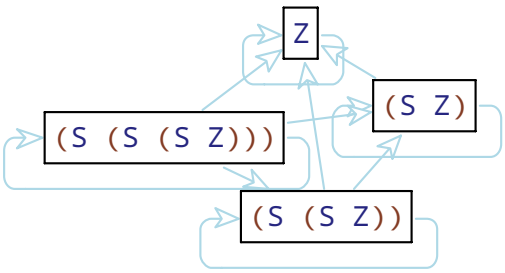
4.8 From Judgements to Reduction Relations

The Redex `reduction-relation` form is tailored toward writing relations as a set of axioms. Most of the inductive parts of a reduction relation are computed as contextual or compatible closures, which use the grammar of contexts or terms to guide the induction. This is often, but not always, what is needed. In cases where a reduction relation requires a more general set of inference rules, a useful modeling strategy is to define the relation as a judgment and then define a reduction relation in terms of the judgment. As an example, here is a model of the natural numbers and a formulation of "$\geq$" as a reduction relation using this strategy:

```
#lang racket
(require redex)

(define-language Natural
  (N ::= Z (S N)))

(define-judgment-form Natural
  #:mode (>= I O)
  [(>= N N)]
  [(>= N_0 N_1)
   ---------------
```

```
   (>= (S N_0) N_1)])

(define ->>=
  (reduction-relation
   Natural
   (--> N_0 N_1
        (judgment-holds (>= N_0 N_1)))))

(traces ->>= (term (S (S (S Z)))))
```

Running the reduction relation on the example of 3 produces the following graph:



4.9 Property-Based Testing

Semantic models are often buggy, just like programs. One of the easiest and quickest ways to test code is to state properties you believe to be true and then test that the property holds for a bunch of random inputs.

We believe the standard reductions semantics and natural semantics correspond in the following way, for all `e`:

```
  (equal? (apply-reduction-relation* -->a (term e))
          (judgment-holds (evalr e v) v))
```

That is, for any term, the set of irreducible values reached by iterating the standard reduction relation are exactly equal to the set of values related by the natural semantics relation. Redex will generate random expressions and search for a counterexample with `redex-check`:

```
> (redex-check A e
               (equal? (apply-reduction-relation* -->a (term e))
                       (judgment-holds (evalr e v) v)))
redex-check: no counterexamples in 1000 attempts
```

This generates 1000 random examples of `e`s from the language `A` and checks the expression, interpreted as a predicate universally quantified over the pattern variable `e`. We can use arbitrary patterns in place of `e` to generate other kinds of data.

4.10 Extending Languages

Semantics engineers often start with small, simple models and grow them over time into more realistic and rich languages. We've seen this with the development of $\mathcal{A}$ and $\mathcal{B}$ already. Redex supports exactly this kind of growth.

First, put the following at the top of the `A` model and save it in a file called `A.rkt`.

```
(provide (all-defined-out))
```

In another file, start with:

```
#lang racket
(require "A.rkt")
(require redex/reduction-semantics)
```

This will import all of the defintions form the model for `A`

We can now extend the syntax of `A` to obtain `B`:

```
(define-extended-language B A
  (e ::= ....
     x
     (Div e e)
     (Eq e e)
     (If e e e))
  (x ::= variable)
  (v ::= .... b)
  (b ::= True False)
  (ρ ::= ((x v) ...))
  (E ::= ....
     (Div E e)
     (Div v E)
     (Eq E e)
     (Eq v E)
     (If E e e)))
```

The "`e ::= ....`" (that's *four* dots) grammar means everything that's an `e` from `A` plus the new productions that follow. It is really as if you had copied the grammar of `e` from `A` in place of the ellipsis; all of the occurrences of the non-terminal `e`s in the grammar of `A` range over `e`s drawn from `B`.

The reduction relation `a` can be lifted to work on `B` expressions and unioned with the new reduction axioms:

```
(define (b ρ)
  (union-reduction-relations
   (extend-reduction-relation a B)
   (reduction-relation
    B
    (--> x v (where v (lookup ,ρ x)))
```

```
    (--> (Div i j) ,(quotient (term i) (term j))
         (side-condition (not (zero? (term j)))))
    (--> (Eq i i) True)
    (--> (Eq i j) False
         (side-condition (not (= (term i) (term j)))))
    (--> (If True e_1 e_2) e_1)
    (--> (If False e_1 e_2) e_2))))
```

Note that this relation is indexed by an environment ρ. The `lookup` metafunction looks up the value of a given variable if it exists, or produced `#f`:

```
(define-metafunction B
  lookup : ρ x -> v or #f
  [(lookup ((x v) (x_0 v_0) ...) x) v]
  [(lookup ((x_0 v_0) (x_1 v_1) ...) x)
   (lookup ((x_1 v_1) ...) x)]
  [(lookup () x) #f])
```

The first line of this metafunction specifies a signature which states it consumes an environment and variable and produces a value or `#f`. It will result in an error if any of these conditions are violated. Although Redex doesn't have a type system, it's still useful to catch these kinds of errors.

After the signature, there are three clauses, each consisting of a pattern and a template. The patterns are matched in order. The first clause matches when the first element in the environment contains the variable `x`, which is the same variable `x` being looked up; notice that the name `x` is used twice in this pattern. In this case, `lookup` produces the associated value from the environment. The second clause matches when the environment is non-empty and—knowing the first clause wasn't taken—the first element is not what's being looked up. In this case, `lookup` looks up the variable in the rest of the environment. The third and final clause matches when the environment is empty, meaning `x` is not in the environment.

A pattern of the form "`p ...`" (that's *three* dots!), where `p` is some pattern, means "match `p` zero or more times." (You can think of it as Kleene star or "$*$" from regular expressions.) So the pattern `(i ...)` matches `()`, `(1)`, `(1 2 3)`, etc. So in the first clause of `lookup`:

```
  [(lookup ((x v) (x_0 v_0) ...) x) v]
```

This pattern matches an environment which starts with `(x v)` and then has zero or more `(x_0 v_0)`s. An easy mistake to make is to write "`...`" as if it were a "match anything" pattern signifying you don't care about the remaining data in the environment. Thinking this way, you might instead write the clause as:

```
  [(lookup ((x v) ...) x) v]
```

But this does not mean what you think it means. The problem here is we're matching many `x`s since `x` is part of the pattern repeated by "`...`", and therefore we bind many `v`s,

but the template only has one occurrence. This kind of mismatch in number will result in a syntax error complaining about a binder at different depths.

Now let's look at the second clause:

```
[(lookup ((x_0 v_0) (x_1 v_1) ...) x)
 (lookup ((x_1 v_1) ...) x)]
```

This pattern matches whenever the environment contains one or more elements. This is achieved by writing a pattern `(x_0 v_0)` for the one element and then `(x_1 v_1) ...` for the remaining zero or more elements. This is a fairly common idiom for matching the first and rest of a list. The result calls `lookup` with a structurally smaller environment.

Programming with Redex's souped-up pattern matcher takes some getting used to. Try things out. Make conjectures with test cases.

Exercise 16. *Design a metafunction* `pukool` *which is like* `lookup` *but searches through the environment from right to left. The following test cases demonstrate the difference:*

```
(test-equal (term (lookup ((x 1) (x 5)) x))
            (term 1))
(test-equal (term (pukool ((x 1) (x 5)) x))
            (term 5))
```

□

We can further develop the semantics for errors as a further extension:

```
(define-extended-language BE B
  (e ::= .... r)
  (r ::= (Err variable))
  (a ::= v r))

(define err
  (reduction-relation
   BE
   (--> (Div i 0) (Err Div0))
   (--> (Div b v) (Err Div1))
   (--> (Div v b) (Err Div2))
   (--> (Eq b v) (Err Eq1))
   (--> (Eq v b) (Err Eq2))
   (--> (If i e_1 e_2) (Err If))))

(define prop
  (reduction-relation
   BE
   (--> (Pred r) r)
   (--> (Succ r) r)
   (--> (Plus r e) r)
   (--> (Plus v r) r)
```

```
   (--> (Mult r e) r)
   (--> (Mult v r) r)
   (--> (Div r e) r)
   (--> (Div v r) r)
   (--> (Eq r e) r)
   (--> (Eq v r) r)
   (--> (If r e_0 e_1) r)))

(define (bep ρ)
  (union-reduction-relations
   (extend-reduction-relation (b ρ) BE)
   err prop))
```

5 Reasoning About All Program Executions

It would be nice to eliminate all programs that have errors in them. In general, it's not possible to mechanically determine if a program will cause an error when run, so much of program analysis, which is the study of approximations to program properties, is concerned with detecting or proving the absense of errors in programs. One of the most common forms of this kind of program analysis are **type systems**:

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.
>
> — Benjamin C. Pierce

In its simplest form, a type system can prove the absence of errors that arise from applying operations to the wrong kinds of values. Like any program analysis, the classification of programs into those that have these kind of errors and those that do not must be approximate. So the classification will err on the side of caution and classify programs as *definitely* not having this kinds of errors or *potentially* having them. In other words, a type system will misclassify some good programs as bad, but no bad program will be classified as good.

5.1 Classifying phrases according to the kinds of answers they compute

If we were to consider classifying phrases according to the kinds of answers they compute, the most precise characterization would be to classify phrases by *the set of answers* they compute.

For some simple languages, calculating such a set is trivial. For example, for any $\mathcal{A}$ program e, it's easy to compute the set $\{i \mid e \Downarrow i\}$. Even for $\mathcal{B}$, if we consider characterizing phrases e with a *given environment* ρ, then it's easy to compute $\{a \mid \rho \vdash e \Downarrow a\}$. It's easy to compute this characterization because it consists of just a single run of the program and $\mathcal{B}$ is such a computationally feeble language, computing the evaluation of programs is decidable. But we might also want to reason about all possible executions of a $\mathcal{B}$ program.

If we consider the environment as a model of program inputs, provided by the external world, then it's reasonable to consider the answers a program computes as a function of the environment. In other words, characterize phrases e by the set $\{a \mid \rho \vdash e \Downarrow a, \text{ for some } \rho\}$. The trouble is this set is potentially infinite, so computing it could take a while.

For this reason, it may be useful to compute a more approximating classification of the kinds of answers phrases compute. Designing useful approximating classifications is the art and science of *abstract interpretation*.

Let's see a simple instance of this.

Suppose we want to classify phrases not by the precise value they compute but rather by whether they compute:

- an integer,
- a boolean,
- an error, or
- some combination of the above.

For example, $\mathit{Mult}(3, 4)$ should be characterized as computing an integer. $\mathit{If}(\mathit{True}, 3, \mathit{False})$ computes a boolean. $\mathit{If}(x, 3, \mathit{False})$ computes a boolean, an integer, or an error, dependending on the environment.

We therefore define the following *abstract answers*:

$$\mathit{AbstractAnswer} \quad a^\# \quad = \quad \mathit{Bool} \mid \mathit{Int} \mid \mathit{Err}$$

Abstract answers denote sets of answers (sometimes called "concrete" answers to contrast abstract answers). Their meaning is defined as follows:

$$\begin{array}{rcl} [\![\mathit{Bool}]\!] & = & \{\mathit{True}, \mathit{False}\} \\ [\![\mathit{Int}]\!] & = & \{\ldots, -1, 0, 1, \ldots\} \\ [\![\mathit{Err}]\!] & = & \{\mathit{Err}_{\mathit{Div0}}, \mathit{Err}_{\mathit{If}}, \mathit{Err}_{\mathit{Plus}}, \mathit{Err}_{\mathit{Succ}}, \mathit{Err}_{\mathit{Plus1}}, \ldots\} \end{array}$$

Now let's explore an abstract evaluation relation for $\mathcal{B}$ based around the above notion of abstract values. The idea here is that the formulate an alternative relation that operates on abstract values instead of values. There are many abstract evaluation relations. What makes them admissable is if they are *sound*, i.e. if it possible for an expression to compute a certain kind of answer, that answer is in the (denotation) of the abstract answer. Formally:

Claim 5.1. *If $\rho \vdash e \Downarrow a$ for some ρ, then there exists an $a^\#$ such that $\vdash e \Downarrow a^\#$ and $a \in [\![a^\#]\!]$.*

It's quite easy to make a sound, but useless abstract evaluator:

$$\frac{}{\vdash e \Downarrow \mathit{Int}} \qquad \frac{}{\vdash e \Downarrow \mathit{Bool}} \qquad \frac{}{\vdash e \Downarrow \mathit{Err}}$$

It's almost always impossible to go the other direction to design an abstract evaluator that is *complete*, i.e.:

Claim 5.2. *If* $\vdash e \Downarrow a^{\#}$*, then there exists an* a *and* ρ *such that* $\vdash e \Downarrow a$ *and* $a \in [\![a^{\#}]\!]$.

If we wanted to do better than the useless abstract evaluator, one approach is to use the concrete evaluator to guide its design. Let's consider some concrete and abstract cases in tandem:

$$\frac{\rho(x) = v}{\rho \vdash x \Downarrow v} \qquad \frac{}{\vdash x \Downarrow \mathit{Bool}} \qquad \frac{}{\vdash x \Downarrow \mathit{Int}}$$

On the left is the concrete evaluation relation for variables, which are given meaning in terms of a particular environent. The abstract evaluator must handle all possible environments; since an environment may map a variable to an integer or a boolean, the rule states that a variable occurrence may evaluate to either *Bool* or *Int*.

This may seem overly conservative; variables range over all possible values, which mean that programs involving variables that are treated as integers will be abstract interpreted as having errors since they variables could stand for booleans. This is necessary for soundness – indeed we could imagine running such programs with environments that don't respect our intentions for how these variables are used. But we will return to this question in a bit and show how we can reason based on assumptions of how variables will be used.

Here are the concrete and abstract rules for literals:

$$\frac{}{\rho \vdash i \Downarrow i} \qquad \frac{}{\rho \vdash b \Downarrow b} \qquad \frac{}{\vdash i \Downarrow \mathit{Int}} \qquad \frac{}{\vdash b \Downarrow \mathit{Bool}}$$

As you can see, the abstract evaluator is losing information; it "forgets" which particular number or boolean a literal evaluators to and instead just signals the appropriate set of values the literal belongs to.

Here are the concrete and abstract rules for *Pred* (*Succ* is similar):

$$\frac{\rho \vdash e \Downarrow i}{\rho \vdash \mathit{Pred}(e) \Downarrow i - 1} \qquad \frac{\vdash e \Downarrow \mathit{Int}}{\vdash \mathit{Pred}(e) \Downarrow \mathit{Int}}$$

The error propogation and creation rules can likewise be read-off from the concrete rules:

$$\frac{\rho \vdash e \Downarrow r}{\rho \vdash \mathit{Pred}(e) \Downarrow r} \qquad \frac{\rho \vdash e \Downarrow b}{\rho \vdash \mathit{Pred}(e) \Downarrow \mathit{Err}_{\mathit{Pred}}} \qquad \frac{\vdash e \Downarrow \mathit{Err}}{\vdash \mathit{Pred}(e) \Downarrow \mathit{Err}} \qquad \frac{\vdash e \Downarrow \mathit{Bool}}{\vdash \mathit{Pred}(e) \Downarrow \mathit{Err}}$$

The *Plus* and *Mult* cases follow a similar pattern.

Finally, let's consider *Div*. When a subexpression produces an error or evaluates a boolean follows the same pattern as above, so let's focus soley on the "real" cases for *Div*. First, the concrete cases:

$$\frac{\rho \vdash e_1 \Downarrow i \qquad \rho \vdash e_2 \Downarrow j}{\rho \vdash \mathit{Div}(e_1, e_2) \Downarrow \lfloor i/j \rfloor} \qquad \frac{\rho \vdash e_1 \Downarrow i \qquad \rho \vdash e_2 \Downarrow 0}{\rho \vdash \mathit{Div}(e_1, e_2) \Downarrow \mathit{Err}_{\mathit{Div}0}}$$

And now the abstract cases:

$$\frac{\rho \vdash e_1 \Downarrow \mathit{Int} \qquad \rho \vdash e_2 \Downarrow \mathit{Int}}{\rho \vdash \mathit{Div}(e_1, e_2) \Downarrow \mathit{Int}} \qquad \frac{\rho \vdash e_1 \Downarrow \mathit{Int} \qquad \rho \vdash e_2 \Downarrow \mathit{Int}}{\rho \vdash \mathit{Div}(e_1, e_2) \Downarrow \mathit{Err}}$$

The thing to notice here is that in the abstract rules the hypotheses are the same in both of these rules. Since our abstraction of integers to *Int* is too course-grained to distinguish between zero and non-zero integers, there's no way to tell if evaluating a division operation is safe. Consequently *any* program that includes a division operation will be reported as potentially causing an error. Even obviously error free programs like $\mathit{Div}(10, 2)$.

5.2 Type Judgments

In the language of $\mathcal{B}$ expressions, there are two kinds of data: Booleans and integers. We can formalize a judgment on programs that classifies them according to the data they produce, eliminating the possibility of errors arising from misusing data.

A type is either *Bool* or *Int*:

$$\mathit{Type} \quad t \quad = \quad \mathit{Bool} \mid \mathit{Int}$$

A **type judgment** is a ternary relation on environments, expressions, and types:

$$\rho \vdash e : t$$

If a program (ρ, e) is related to t it means the program evaluates to some value of that type:

$$\rho \vdash e : \mathit{Int} \Rightarrow \rho \vdash e \Downarrow i$$
$$\rho \vdash e : \mathit{Bool} \Rightarrow \rho \vdash e \Downarrow b$$

Thus the relation captures a class of well-behaved programs that do no cause type errors. The natural semantics employed here could either be the one that considers erroneous programs meaningless or the explicit error semantics.

There is a small wrinkle here having to do with divide-by-zero errors, and even without seeing the definition of the relation, you may be (rightfully) doubting the above claims. Let us punt on the wrinkle for now and consider *Div* banished from $\mathcal{B}$ (we will restore it later).

At first approximation, you might think of the ":" as a strange kind of evaluation relation, which gives an approximation of the "real" relation "$\Downarrow$." Instead of saying exactly what an expression evaluates to, the ":" says what set of values the result belongs to. That intuition can guide the definition of the relation. So for example, an integer expression evaluates to something in *Int*, and a Boolean evaluates to someting in *Bool*:

$$\frac{}{\rho \vdash i : \mathit{Int}} \qquad \frac{}{\rho \vdash b : \mathit{Bool}}$$

Likewise, variables bound to integers and Booleans behave similarly:

$$\frac{\rho(x) = i}{\rho \vdash x : \mathit{Int}} \qquad \frac{\rho(x) = b}{\rho \vdash x : \mathit{Bool}}$$

If an expression e evaluates to something in *Int*, so does $\mathit{Succ}(e)$ and $\mathit{Pred}(e)$:

$$\frac{\rho \vdash e : \mathit{Int}}{\rho \vdash \mathit{Pred}(e) : \mathit{Int}} \qquad \frac{\rho \vdash e : \mathit{Int}}{\rho \vdash \mathit{Succ}(e) : \mathit{Int}}$$

If expressions e_1 and e_2 evaluate to values in *Int*, then *Plus*(e_1, e_2) and *Mult*(e_1, e_2) evaluate to values in *Int* and *Eq*(e_1, e_2) evaluates to values in *Bool*:

$$\frac{\rho \vdash e_1 : \textit{Int} \qquad \rho \vdash e_2 : \textit{Int}}{\rho \vdash \textit{Plus}(e_1, e_2) : \textit{Int}} \qquad \frac{\rho \vdash e_1 : \textit{Int} \qquad \rho \vdash e_2 : \textit{Int}}{\rho \vdash \textit{Mult}(e_1, e_2) : \textit{Int}} \qquad \frac{\rho \vdash e_1 : \textit{Int} \qquad \rho \vdash e_2 : \textit{Int}}{\rho \vdash \textit{Eq}(e_1, e_2) : \textit{Bool}}$$

Finally, there is the matter of *If*. It's clear that the test expression *If*(e_1, e_2, e_3) should evaluate to some value in *Bool*, but what should the whole expression evaluate to? If all we know of e_1 is it evaluates to a Boolean, then the whole expression either evaluates to the value of e_1 or e_2. But which is it? One approach is to require e_1 and e_2 to evaluate to values within the same set. You've probably encountered this before: both branches of an *If* must have the same type. Adopting this approach, the rule is:

$$\frac{\rho \vdash e_1 : \textit{Bool} \qquad \rho \vdash e_2 : t \qquad \rho \vdash e_3 : t}{\rho \vdash \textit{If}(e_1, e_2, e_3) : t}$$

Now we have a typing relation, although it's not exactly clear what we have gained. After all, we could have just run a program to discover if it caused an error. That said, it still may be cheaper, even for this simple type system, to compute the type than to compute the evaluation. As our language grows, the gap between the resources required to run and analyze a program can grow without bound, so this point shouldn't be entirely disregarded. However, the real value of this system comes into play after making the following observation: the environment plays no role in the type relation other than to signify the types of variables by supplying a witness to the type. In other words, in the typing derivation of

$$[x \mapsto 4] \vdash \textit{If}(\textit{Eq}(x, 7), \textit{False}, \textit{True}) : \textit{Bool}$$

it doesn't matter at all that x is bound to 4 in particular. Any integer would have sufficed to prove this program produces a Boolean. Intuitively, we can make a much stronger claim about this program, which is that on any integer input, the program does not produce an error. By reasoning about a single "abstract" run of the program, we can conclude facts about all possible "concrete" runs: none of them produce errors.

We can make this claim precise with a relation on environments. First, let's define:

$$\textit{typeof}(i) = \textit{Int}$$
$$\textit{typeof}(b) = \textit{Bool}$$

Two environments "agree," written $\rho \sim \rho'$ when they map variables to values of the same type:

$$\frac{\forall x \in \mathsf{dom}(\rho).\textit{typeof}(\rho(x)) = \textit{typeof}(\rho'(x))}{\rho \sim \rho'}$$

Claim 5.3. *If $\rho \vdash e : t$ and $\rho \sim \rho'$, then $\rho' \vdash e : t$.*

Proof. By structural induction on the derivation of $\rho \vdash e : t$. The integer and Boolean cases are trivial. The variable cases follow from the definition of $\sim$. The remaining cases follow by induction. □

Since the environment really only informs the system of the type of each variable, we can replace the value environment with a **type environment** that directly maps a variable to its type. We use the metavariable Γ to range over type environments. The type judgment is obtained simply by replacing ρ with Γ in all the rules except the variable rules, which are replaced with:

$$\frac{\Gamma(x) = t}{\Gamma \vdash e : t}$$

We can extend the "$\sim$" relation to relate type environments and value environments:

$$\frac{\forall x \in \mathsf{dom}(\Gamma).\Gamma(x) = \mathit{typeof}(\rho(x))}{\Gamma \sim \rho}$$

The soundness of the typing judgment formalizes our understanding of the classification of programs into those that do not produce errors and those that may:

Claim 5.4. *If $\Gamma \vdash e : t$ and $\Gamma \sim \rho$, then $\rho \vdash e \Downarrow v$ and $\mathit{typeof}(v) = t$.*

Proof. By structural induction on the derivation of $\Gamma \vdash e : t$. □

Type systems are arguably the most effective and widely used formal verification tool in use today. In the development above, we've come up with a fairly simple type system that can prove the absence of certain errors in programs. In our case, the class of errors is simple; it rules out errors arising from branching on non-Booleans and applying numeric operators to non-numeric arguments. Much of the work in developing type systems has been in developing richer notions of errors and corresponding type judgements that safely classify error free programs. But every type system must necessarily throw good programs out with the bad. For example, this type system rejects the following error free programs:

$$\mathit{If}(x, 7, \mathit{False}), \text{ where } x : \mathit{Bool}$$
$$\mathit{If}(\mathit{False}, \mathit{Eq}(\mathit{True}, 1), 7)$$
$$\mathit{If}(x, \mathit{If}(x, 7, \mathit{Eq}(\mathit{True}, 1)), 8), \text{ where } x : \mathit{Bool}$$

Note on Runtime Errors: Type systems often only rule out a certain class of errors. Which class depends on the particular type system. The system we've just developed rules out all errors except $\mathit{Err}_{\mathit{Div0}}$ errors, which were ommitted from the presentation for simplicity. To be precise about divide-by-zero errors would require some tedious and uninteresting additions to the semantics and conditions on the claims.

5.3 Abstract Interpretation with Types

In this section, let's take an alternative perspective on the type judgement of the previous section. If "$:$" is a funny analog of "$\Downarrow$", a natural question is what is the reduction semantics equivalent of "$:$"?

First, let's embed types in the language of $\mathcal{B}$ expressions:

$$e \quad ::= \quad \ldots \mid t$$

The reduction axioms are easy to read-off from the typing judgement:

$$\frac{}{\Gamma \vdash i \ \mathbf{t} \ \mathit{Int}} \qquad \frac{}{\Gamma \vdash b \ \mathbf{t} \ \mathit{Bool}} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x \ \mathbf{t} \ t} \qquad \frac{}{\Gamma \vdash \mathit{Pred}(\mathit{Int}) \ \mathbf{t} \ \mathit{Int}} \qquad \frac{}{\Gamma \vdash \mathit{Succ}(\mathit{Int}) \ \mathbf{t} \ \mathit{Int}}$$

$$\frac{}{\Gamma \vdash \mathit{Plus}(\mathit{Int}, \mathit{Int}) \ \mathbf{t} \ \mathit{Int}} \qquad \frac{}{\Gamma \vdash \mathit{Mult}(\mathit{Int}, \mathit{Int}) \ \mathbf{t} \ \mathit{Int}} \qquad \frac{}{\Gamma \vdash \mathit{Eq}(\mathit{Int}, \mathit{Int}) \ \mathbf{t} \ \mathit{Bool}}$$

$$\frac{}{\Gamma \vdash \mathit{If}(\mathit{Bool}, t, t) \ \mathbf{t} \ t}$$

We can define $\rightarrow_{\mathbf{t}}$ as the compatible closure of $\mathbf{t}$ over the grammar of expressions and $\rightarrow_{\mathbf{t}}^{\star}$ as the reflexive, transitive closure of $\rightarrow_{\mathbf{t}}$. Analogs of the type soundness property hold in this system, too.

Claim 5.5. *If* $\Gamma \vdash e \ \rightarrow_{\mathbf{t}}^{\star} \ t$ *and* $\Gamma \sim \rho$*, then* $\rho \vdash e \ \rightarrow_{\mathbf{b}}^{\star} \ v$ *and* $\mathit{typeof}(v) = t$.

Exercise 17. *Prove claim 5.5.* □

More than just a change of notation, the reduction semantics view opens up some new design possibilities. For example, the *If* axiom requires both branches to be reduced to (abstract) values, i.e. types, before proceeding. What if we mimicked the original semantics more closely and replace the *If* rule with the following:

$$\frac{}{\Gamma \vdash \mathit{If}(\mathit{Bool}, e_1, e_2) \ \mathbf{t}' \ e_1} \qquad \frac{}{\Gamma \vdash \mathit{If}(\mathit{Bool}, e_1, e_2) \ \mathbf{t}' \ e_2}$$

While the above is a more precise abstraction of the $\mathbf{b}$ semantics, it leads to some potentially undesirable properties. The semantics are not consistent and claim 5.5 is not true. Consider the example from earlier:

$$\mathit{If}(x, 7, \mathit{False}), \text{ where } x : \mathit{Bool}$$

In the $\mathbf{t}'$ semantics, this program reduces to both *Int* and *Bool*.

Exercise 18. *Construct a counterexample to claim 5.5 for* $\mathbf{t}'$. □

The $\rightarrow_{\mathbf{t}'}^{\star}$ semantics, however, does offer some strong guarantees. In particular, the semantics is useful for informing us of what the program *does not evaluate to*. If ruling out behaviors such as type errors is all we care about, this is just as useful as the previous system which informed of us of what the program does evaluate to. After all, saying a program always evaluates to an integer is a way of saying what it does not evaluate to, also.

So we can formalize this with the following claim:

Claim 5.6. *If* $\neg(\Gamma \vdash e \ \rightarrow^\star_{\mathbf{t}'} \ t)$ *and* $\Gamma \sim \rho$*, then* $\neg(\rho \vdash e \ \rightarrow^\star_{\mathbf{b}} \ v)$ *where* $\mathit{typeof}(v) = t$*.*

While true, this negative formulation doesn't quite let us conclude that some programs are error free. The solution is to again mimick the original semantics more closely and explicitly characterize the error behavior of programs.

Suppose we add rules corresponding the axioms for creating and propogating errors (only showing a small excerpt):

$$\frac{t \neq \mathit{Bool}}{\Gamma \vdash \mathit{If}(t, e_1, e_2) \ \mathbf{t}' \ \mathit{Err}_{\mathit{If}0}} \qquad \frac{}{\Gamma \vdash \mathit{If}(r, e_1, e_2) \ \mathbf{t}' \ r} \qquad \frac{}{\Gamma \vdash \mathit{Eq}(r, e) \ \mathbf{t}' \ r} \qquad \frac{}{\Gamma \vdash \mathit{Eq}(t, r) \ \mathbf{t}' \ r}$$

Programs now reduce to a type or an error, which we'll call a type answer:

$$\mathit{TyAns} \quad ta \quad ::= \quad t \mid r$$

We need to relate type answers and answers, which we do by a relation $a \sqsubseteq ta$:

$$\frac{\mathit{typeof}(v) = t}{v \sqsubseteq t} \qquad\qquad \frac{}{r \sqsubseteq r}$$

We can now state a generalization of claim 5.6:

Claim 5.7. *If* $\neg(\Gamma \vdash e \ \rightarrow^\star_{\mathbf{t}'} \ ta)$ *and* $\Gamma \sim \rho$*, then* $\neg(\rho \vdash e \ \rightarrow^\star_{\mathbf{b}} \ a)$ *where* $a \sqsubseteq ta$*.*

We can also state this claim in an equivalent way which says that if a program can evaluate to some answer, then an appoximation of that answer is in the abstract evaluation of the program:

Claim 5.8 (Soundness)**.** *If* $\rho \vdash e \ \rightarrow^\star_{\mathbf{b}} \ a$ *and* $\Gamma \sim \rho$*, then* $\Gamma \vdash e \ \rightarrow^\star_{\mathbf{t}'} \ ta$ *where* $a \sqsubseteq ta$*.*

Exercise 19. *Prove claim 5.8.* □

The above formulation lets us talk about error-freedom in a more refined way since we can say exactly what kind of errors a program may or may not have. In fact, it's easy now to deal with divide-by-zero errors by including reduction axioms for *Div*:

$$\frac{}{\Gamma \vdash \mathit{Div}(\mathit{Int}, \mathit{Int}) \ \mathbf{t}' \ \mathit{Int}} \qquad\qquad \frac{}{\Gamma \vdash \mathit{Div}(\mathit{Int}, \mathit{Int}) \ \mathbf{t}' \ \mathit{Err}_{\mathit{Div}0}}$$

The case on the right may seem unfortunate because it says that *any* program that includes a *Div* operation which may be evaluated can cause a divide-by-zero error, including things like $\mathit{Div}(12, 4)$. But that's no different that the guarantee OCaml gives you when it says an expression is of type `int`. In the system above, at least we can say that if an expression does not (abstractly) reduce to $\mathit{Err}_{\mathit{Div}0}$, then it definitely cannot produce a divide-by-zero error.

The development of the reduction-based formulation of type system is an instance of **abstract interpretation**. And in fact, the original type judgement can be seen as a further abstract interpretation of the reduction semantics. Abstract interpretation (AI) is a general theory of semantic approximation. Therefore it serves as a good foundation for the theory of static analysis since static analysis consists of computable semantic approximations. (AI is also useful beyond static analysis as it can be used to related different forms of semantics such as natural semantics and reduction semantics.)

5.4 Abstract Interpretation with Intervals

In this section, let's develop another abstract interperation of $\mathcal{B}$, this time with a more refined abstract domain. On integers, we'll use intervals and interpret the integer operations using interval arithmetic and on Booleans we'll do no approximation. Intervals will be represented as a pair of extended integers, i.e. integers extended with "∞" and "$-\infty$". We'll use $i^\dagger$ and $j^\dagger$ to range over extended integers and $\vec{i}$ and $\vec{j}$ to range over intervals, i.e. pairs of extended integers $(i^\dagger, j^\dagger)$ where we assume $i^\dagger$ is either $-\infty$ or an integer and $j^\dagger$ is either ∞ or an integer. An interval is interpreted as a non-empty set of integers in the following way:

$$\begin{aligned}(-\infty, \infty) &= \mathbb{Z} \\ (-\infty, j) &= \{i' \mid i' \leq j\} \\ (i, \infty) &= \{i' \mid i \leq i'\} \\ (i, j) &= \{i' \mid i \leq i' \leq j\}\end{aligned}$$

We define the $\mathcal{BI}$ language as $\mathcal{B}$, but with intervals in place of integers in the set of values:

$$\begin{array}{llll} \textit{Exp} & e & ::= & \ldots \mid i \mid \vec{i} \\ \textit{Val} & v & ::= & \vec{i} \mid b \end{array}$$

Instead of a type environment as used in **t**, we'll let Γ range over interval environments that map variables to intervals (or Booleans). The reduction axioms are easy in many cases:

$$\frac{}{\Gamma \vdash i \;\mathbf{i}\; (i,i)} \qquad \frac{\Gamma(x) = v}{\Gamma \vdash x \;\mathbf{i}\; v} \qquad \frac{}{\Gamma \vdash \textit{Pred}(\vec{i}) \;\mathbf{i}\; \vec{i} - (1,1)} \qquad \frac{}{\Gamma \vdash \textit{Succ}(\vec{i}) \;\mathbf{i}\; \vec{i} + (1,1)}$$

$$\frac{}{\Gamma \vdash \textit{Plus}(\vec{i}, \vec{j}) \;\mathbf{i}\; \vec{i} + \vec{j}} \qquad \frac{}{\Gamma \vdash \textit{Mult}(\vec{i}, \vec{j}) \;\mathbf{i}\; \vec{i} \cdot \vec{j}} \qquad \frac{}{\Gamma \vdash \textit{If}(\textit{True}, e_1, e_2) \;\mathbf{i}\; e_1}$$

$$\frac{}{\Gamma \vdash \textit{If}(\textit{False}, e_1, e_2) \;\mathbf{i}\; e_2}$$

These definitions make use of extended interval arithmetic operations "$+$", "$-$", and "$\cdot$". Interval arithmetic is well understood mathematical domain, and these operations can be illustrated with just a few examples:

$$\begin{aligned}(1,5) + (3,9) &= (1,14) \\ (2,5) \cdot (3,9) &= (6,45) \\ (-\infty,5) \cdot (3,9) &= (-\infty,45)\end{aligned}$$

Things get trickier with equality. Because an interval represents a set of potential integer values, $\textit{Eq}(\vec{i}, \vec{j})$ must produce *True* whenever there are two equal integers in the sets denoted by $\vec{i}$ and $\vec{j}$ and *False* whenever there are two unequal integers in the sets.

$$\frac{\exists i \in \vec{i}. \exists j \in \vec{j}. i = j}{\Gamma \vdash \textit{Eq}(\vec{i}, \vec{j}) \;\mathbf{i}\; \textit{True}} \qquad \frac{\exists i \in \vec{i}. \exists j \in \vec{j}. i \neq j}{\Gamma \vdash \textit{Eq}(\vec{i}, \vec{j}) \;\mathbf{i}\; \textit{False}}$$

These definitions give the most precise relation our interval abstraction allows; i.e. this is the best approximation possible without changing the domain. The preconditions form a good specification, but clearly an implementation should not enumerate the (potentially infinite) integers in $\vec{i}$ and $\vec{j}$ to check the for (in-)equality. Instead, produce *True* whenever $\vec{i}$ and $\vec{j}$ overlap at all. Determining when to produce *False* requires a little more thought. You want to produce *False* almost all the time. When should *False* not be one of the results?

Things get even trickier with division. First, here are some examples of integer division on intervals:

$$(12, 24) \setminus (3, 4) = (3, 8)$$
$$(12, 24) \setminus (3, \infty) = (0, 8)$$
$$(12, \infty) \setminus (3, \infty) = (0, \infty)$$

Interval division is not defined when 0 is in the denominator interval.

It's tempting to define $\mathbf{i}$ for *Div* as the following:

$$\frac{0 \notin \vec{j}}{\Gamma \vdash \mathit{Div}(\vec{i}, \vec{j})\ \mathbf{i}\ \vec{i} \setminus \vec{j}} \qquad \frac{0 \in \vec{j}}{\Gamma \vdash \mathit{Div}(\vec{i}, \vec{j})\ \mathbf{i}\ \mathit{Err}_{\mathit{Div0}}}$$

But this definition is not sound. To see why, let's formalize the soundness property. Again, it will be in terms of a refinement relation between answers of $\mathcal{B}$ and $\mathcal{BI}$:

$$\frac{i \in \vec{i}}{i \sqsubseteq \vec{i}} \qquad \frac{}{r \sqsubseteq r} \qquad \frac{}{b \sqsubseteq b}$$

This relation is lifted to environments as usual:

$$\frac{\forall x \in \mathsf{dom}(\rho).\rho(x) \sqsubseteq \Gamma(x)}{\rho \sqsubseteq \Gamma}$$

Assuming $\rightarrow^\star_{\mathbf{i}}$ is the reflexive transitive closure of the compatible closure of $\mathbf{i}$, the soundness claim is then:

Claim 5.9 (Soundness)**.** *If* $\rho \vdash e \ \rightarrow^\star_{\mathbf{b}} \ a$ *and* $\rho \sqsubseteq \Gamma$*, then* $\Gamma \vdash e \ \rightarrow^\star_{\mathbf{i}} \ a'$ *where* $a \sqsubseteq a'$*.*

Exercise 20. *Construct a counterexample to claim 5.9 using the faulty definition of* $\mathbf{i}$ *for Div.* □

Exercise 21. *Design the best sound alternative of* $\mathbf{i}$ *for Div. By sound, it should validate claim 5.9; by best, it should be the smallest relation that does so.* □

5.5 Refinement Types

In section 5.2 we developed a simple type system that ruled out type errors. By viewing the typing judgement as a approximate evaluation relation, we explored an equivalent formulation of the type system as an approximate reduction relation in section 5.3. We then revealed a more precise semantics for proving the absence of type errors. In section 5.4, we developed an interval abstraction of the reduction semantics for $\mathcal{B}$. We can also go in

the opposite direction and derive a further approximation of the interval abstraction in the form of a natural semantics. This semantics takes the form of a type judgment, where the the language of types is refined by an interval in the case of integers:

$$\textit{Type} \quad t \quad ::= \quad \textit{Bool} \mid \textit{Int}(\vec{i})$$

The type $\textit{Int}(\vec{i})$ is known as a **refinement type**, which is a type qualified by a predicate which holds for every value of that type. For our purposes, the predicates only include intervals, but richer refinement type systems can be defined by embeding richer logics of predicates into the types. Refinement types, while not quite mainstream yet, are featured in several mature research project such as the refinement type system for F# called F7[6] and a variant of Haskell with refinements called Liquid Haskell[7].

To derive the type system from the reduction semantics, we start with a straightfoward reformulation of most of the axioms:

$$\frac{}{\Gamma \vdash i : \textit{Int}(i,i)} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash e : \textit{Int}(\vec{i})}{\Gamma \vdash \textit{Pred}(e) : \textit{Int}(\vec{i} - (1,1))} \qquad \frac{\Gamma \vdash e : \textit{Int}(\vec{i})}{\Gamma \vdash \textit{Succ}(e) : \textit{Int}(\vec{i} + (1,1))}$$

$$\frac{\Gamma \vdash e_1 : \textit{Int}(\vec{i}) \qquad \Gamma \vdash e_2 : \textit{Int}(\vec{j})}{\Gamma \vdash \textit{Plus}(e_1, e_2) : \textit{Int}(\vec{i} + \vec{j})} \qquad \frac{\Gamma \vdash e_1 : \textit{Int}(\vec{i}) \qquad \Gamma \vdash e_2 : \textit{Int}(\vec{j})}{\Gamma \vdash \textit{Mult}(e_1, e_2) : \textit{Int}(\vec{i} \cdot \vec{j})}$$

$$\frac{\Gamma \vdash e_1 : \textit{Int}(\vec{i}) \qquad \Gamma \vdash e_2 : \textit{Int}(\vec{j})}{\Gamma \vdash \textit{Eq}(e_1, e_2) : \textit{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \textit{Bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad t = t_2 \sqcup t_3}{\Gamma \vdash \textit{If}(e_1, e_2, e_3) : t}$$

The rule for *If* joins the type of its branches with the $\sqcup$ function, which is defined as:

$$\frac{}{\textit{Bool} \sqcup \textit{Bool} = \textit{Bool}} \qquad \frac{i^\dagger = \min(i_1^\dagger, i_2^\dagger) \qquad j^\dagger = \max(j_1^\dagger, j_2^\dagger)}{(i_1^\dagger, j_1^\dagger) \sqcup (i_2^\dagger, j_2^\dagger) = (i^\dagger, j^\dagger)}$$

The rule for *Div* can be quite conservative and require that 0 not be in the denominator. This rules out many programs, but is sound. If we require the typing judgment to be a function, it is the only option we have:

$$\frac{\Gamma \vdash e_1 : \textit{Int}(\vec{i}) \qquad \Gamma \vdash e_2 : \textit{Int}(\vec{j}) \qquad 0 \notin \vec{j}}{\Gamma \vdash \textit{Div}(e_1, e_2) : \textit{Int}(\vec{i} \setminus \vec{j})}$$

Claim 5.10. *If $\Gamma \vdash e : t$ and $\rho \sqsubseteq \Gamma$, then $\rho \vdash e \Downarrow v$ and $v \sqsubseteq t$.*

This claim relies on the following definition of $\sqsubseteq$:

$$\frac{}{b \sqsubseteq \textit{Bool}} \qquad \frac{i \in \vec{i}}{i \sqsubseteq \textit{Int}(\vec{i})} \qquad \frac{\forall x \in \mathsf{dom}(\rho).\rho(x) \sqsubseteq \Gamma(x)}{\rho \sqsubseteq \Gamma}$$

[6] `http://research.microsoft.com/en-us/projects/f7/`

[7] `http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/about/`

Exercise 22. *Interval refinements as defined above are a course-grained approximation because an interval is given by a single pair of extended integers; this abstraction cannot express, for example, that an integer i is approximated by a set such as $(-5 \leq i \leq -2) \vee (2 \leq i \leq 5)$. However interval arithmetic is perfectly capable of operating on finite unions of intervals. Design a refinement type system that refines integers by sets of intervals. It should prove the safety of the following program:*

$$[x \mapsto \mathit{Int}\{(-5,-2),(2,5)\}] \vdash \mathit{Div}(5,x) : \mathit{Int}\{(-2,-1),(1,2)\}$$

□

5.6 Symbolic Execution

Symbolic execution is another approach to reasoning about all possible executions of program that is achieved by considering inputs to a program symbolically. In other words, if a program takes as input an integer *x*, we will run the program using "*x*" as a placeholder for any possible integer. This is not unlike how we reason mathematically. If you see the function

$$f(x) = x \cdot 0,$$

this is equivalent to $f(x) = 0$; there's no need to wait until the function is applied to a particular input to come to this conclusion.

Similarly, if we have the following program:

$$\mathit{If}(\mathit{Eq}(\mathit{Plus}(x,x),\mathit{Mult}(2,x)),3,4),$$

it's easy to see it can never reduce to 4, no matter what value *x* takes on.

Symbolic reasoning often involves reasoning by implication. For example, consider a slight tweak to the above program:

$$\mathit{If}(\mathit{Eq}(x,y),\mathit{If}(\mathit{Eq}(\mathit{Plus}(x,x),\mathit{Mult}(2,y)),3,4),8).$$

This program also cannot reduce to 4, no matter what values *x* or *y* take on. The reasoning is obvious: if *x* is not equal to *y*, the *If* takes the alternative branch and the result is 8. If *x* is equal to *y*, the *If* takes the consequent branch, which is just the example from before, but with an *x* replaced with a *y*. But we've already assumed *x* equals *y*, so of course the same facts should hold as before.

We call the logical assertion $\mathit{Eq}(x,y)$ a **path condition** on the execution of the consequent branch. Likewise, there is a (unneeded) path condition on the alternative, which asserts $\neg\mathit{Eq}(x,y)$.

The idea behind symbolic execution is to augment our reduction semantics to deal with symbolic data, namely variables, and to build up facts about the program as a logical statement about what must be true at any given point during the execution. The form of the logical statement is a conjunction of conditions which must hold at that point in the execution. If an inconsistent set of facts are ever reached, we can conclude that state of execution is infeasible and cannot actually be reached.

Symbolic value	sv	::=	$x \mid v$
Path condition	ϕ	::=	$\{c, \dots\}$
Condition	c	::=	$x \mid \neg c \mid \mathit{Eq}(x, \mathit{Pred}(x)) \mid \mathit{Eq}(x, \mathit{Succ}(x))$
		$\mid$	$\mathit{Eq}(x, \mathit{Mult}(sv, sv)) \mid \mathit{Eq}(x, \mathit{Div}(sv, sv)) \mid \mathit{Eq}(sv, sv)$

The reduction of an expression is formulated as a reduction relation on expression and path condition pairs: $\vdash (e, \phi)\ \mathbf{s}\ (e', \phi')$. The meaning of such a reduction is that e, under the assumptions of ϕ, may reduce to to e', implying conditions ϕ'.

We use a relation on path conditions $\phi \models c$, which denotes that ϕ "allows" c to be true; in other words $\phi \cup \{c\}$ is not a contradiction. You can think of the $\phi \models c$ judgement as being an embedded model checker in our language. We can produce a tighter characterization of a program's behavior (i.e. produce a smaller reduction graph) if we use a powerful model checker, but so long as the model checker is sound, the semantics will be sound. The simplest sound model checker is just the one that allows everything:

$$\frac{}{\phi \models c}$$

A slightly more refined one is the following:

$$\frac{\neg c \notin \phi}{\phi \models c} \qquad \frac{c \notin \phi}{\phi \models \neg c} \qquad \frac{x : \mathit{Bool} \notin \phi}{\phi \models x : \mathit{Int}} \qquad \frac{x : \mathit{Int} \notin \phi}{\phi \models x : \mathit{Bool}}$$

In general what we want is:

$$\frac{\phi \cup \{c\} \text{ is satisfiable}}{\phi \models c}$$

In our case, the logical formulas are expressed in first-order logic with equality, using a theory of integer arithmetic. Such satisfiability problems are ubiquitous in computer science. The **Satisfiability Modulo Theories** (SMT) problem is a decision problem for logical formulas with background theories expressed in first-order logic with equality. You've probably heard of the Boolean satisfiability problem (SAT), which is just an instance of the SMT problem, using Boolean arithmetic as the underlying theory. Other theories might include the theories of real numbers, intervals, data structures, and so on. There are many mature SMT solvers available (e.g. Z3, CVC4), which can effecitvely decide $\phi \models c$. If we had a richer language, the path conditions would involve richer theories, which may not be decidable. At that point, a sound approximation would need to be used.

Let's now look at some of the reduction rules:

$$\frac{\phi \models x}{\vdash \mathit{If}(x, e_1, e_2), \phi\ \mathbf{s}\ e_1, \phi \cup \{x\}} \qquad \frac{\phi \models \neg x}{\vdash \mathit{If}(x, e_1, e_2), \phi\ \mathbf{s}\ e_2, \phi \cup \{\neg x\}}$$

Here, an *If* takes the consequent branch whenever the test variable may be true (determined by using $\phi \models x$) and adds x to the path condition. Symmetrically, *If* takes the alternative whenever the test may be false and adds $\neg x$ to the path condition.

You may wonder why x is added to ϕ if $\phi \models x$ already holds. The answer is that $\phi \models x$ only determines if it is possible that x is true. When x is added to the path condition, we are asserting that x *is* true. Consider this example:

$$\mathit{If}(x, \mathit{If}(x, 1, 2), 3)$$

This program can produce either 1 or 3 because x may be either true or false, but it cannot produce 2 because that would require x to be true in the first conditional and false in the second. This is reflected in the reduction semantics because $\emptyset \models x$ and $\emptyset \models \neg x$ in the first conditional. But when the consequent branch is taken, we add x to the path condition, which rules out producing 2 since it's not the case that $\{x\} \models \neg x$.

Here are some rules for arithmetic operations:

$$\frac{}{\vdash \mathit{Pred}(x), \phi \ \mathbf{s}\ x', \phi \cup \{\mathit{Eq}(x', \mathit{Pred}(x))\}} \qquad \frac{}{\vdash \mathit{Succ}(x), \phi \ \mathbf{s}\ x', \phi \cup \{\mathit{Eq}(x', \mathit{Succ}(x))\}}$$

$$\frac{}{\vdash \mathit{Plus}(sv_1, sv_2), \phi \ \mathbf{s}\ x', \phi \cup \{\mathit{Eq}(x', \mathit{Plus}(sv_1, sv_2))\}}$$

$$\frac{}{\vdash \mathit{Mult}(sv_1, sv_2), \phi \ \mathbf{s}\ x', \phi \cup \{\mathit{Eq}(x', \mathit{Mult}(sv_1, sv_2))\}}$$

In each of these rules, the variable x' is introduced by reduction. These variables are understood to be "fresh", i.e. they are variables that have not been used before.

The *Div* rule is more interesting since we have pre- and post-conditions on the denominator:

$$\frac{\phi \models \neg \mathit{Eq}(0, sv_2)}{\vdash \mathit{Div}(sv_1, sv_2), \phi \ \mathbf{s}\ x', \phi \cup \{\mathit{Eq}(x', \mathit{Div}(sv_1, sv_2)), \neg \mathit{Eq}(0, sv_2)\}}$$

$$\frac{\phi \models \mathit{Eq}(0, sv_2)}{\vdash \mathit{Div}(sv_1, sv_2), \phi \ \mathbf{s}\ \mathit{Err}_{\mathit{Div0}}, \phi \cup \{\mathit{Eq}(0, sv_2)\}}$$

Finally, we have the rule for *Eq*:

$$\frac{\phi \models \mathit{Eq}(sv_1, sv_2)}{\vdash \mathit{Eq}(sv_1, sv_2), \phi \ \mathbf{s}\ \mathit{True}, \phi \cup \{\mathit{Eq}(sv_1, sv_2)\}} \qquad \frac{\phi \models \neg \mathit{Eq}(sv_1, sv_2)}{\vdash \mathit{Eq}(sv_1, sv_2), \phi \ \mathbf{s}\ \mathit{False}, \phi \cup \{\neg \mathit{Eq}(sv_1, sv_2)\}}$$

5.7 Type Inference

Type inference is the process of searching for a proof of the type correctness of a program. In terms of the $\mathcal{B}$ language, we can think of it as the problem of determining for an expression e if there is a type environment Γ and type t that exist, such that $\Gamma \vdash e : t$.

Type inference is a common feature of some typed programming languages. For example, OCaml is able to infer the types of programs which make no mention of types in their source text:

```
# let rec fact n =
    if (n = 0)
    then 1
    else n * fact (n-1);;
val fact : int -> int = <fun>
```

Here, OCaml is telling us that it successfully found a proof that the program is well-typed, and moreover has the type `int -> int`, i.e. it consumes and produces integers. It is able to do this without any explicit mention of types in the program.

The process of type inference is not unlike the process a programmer might use to discover the type of this program. Imagine we start knowing nothing about `fact`. We can see from the shape of it's definition that it is a function, so we must determine it's input type and output type. Looking at the body of the function, we see `n` is compared to `0`. This constrains `n` to be a integer, since `n` is a argument to the integer equality operation. Later, `n` is an argument to both subtraction and multiplication operations, which also implies `n` must be an integer. To determine the output type of the function, we can see that `fact` produces `1` in one branch of the conditional. Thus it could produce an integer. In the other branch, it produces the result of multiplication, which also implies the output type is integer. Finally the fact that `fact` is the argument to a multiplication operation also suggests it needs to produce an integer.

In this informal analysis what we've done is collect a set of contraints on the usage of variables. If that set of constraints is consistent, i.e. a variable is not used as an integer in one place and as a Boolean in another, then we conclude the program is type correct. This informal process can be made rigorous to produce a type inference algorithm. The idea will be to generate a set of type constraints—equations between types and variables—and then try to solve the system of equations. If the equations have a solution, the program is type correct. If they don't, we can conclude the program is ill-typed.

In this setting, a type constraint is an equation between types $t = t'$, but which may contain variables (we'll call a type with no variables in it a ground type). A constraint set $\mathcal{T}$ is a set of type constraints.

$$\begin{array}{lrcl} \textit{Types} & t & ::= & \textit{Int} \mid \textit{Bool} \mid x \\ \textit{Type contraints} & \mathcal{T} & ::= & \{t = t', \dots\} \end{array}$$

The way that type constraints are generated is by a type constraint judgement that produces a type (or variable) and a set of constraints. The judgement $\vdash e : t, \mathcal{T}$ is defined for all expressions. This is unlike the type judgement from section 5.2, which was only defined for "good" programs. Instead the type constraint judgement will produce a set of constraints for any program, but ill-typed programs generate inconsistent constraint sets.

Here is the inference rule for *Pred*:

$$\frac{\vdash e : t, \mathcal{T}}{\vdash \textit{Pred}(e) : \textit{Int}, \mathcal{T} \cup \{t = \textit{Int}\}}$$

Compare this with the type judgement for *Pred* from section 5.2:

$$\frac{\Gamma \vdash e : \textit{Int}}{\Gamma \vdash \textit{Pred}(e) : \textit{Int}}$$

The key difference is that rather than place any restrictions on the precondition of the rule, in this case that e has type *Int*, the constraint version expresses this by adding $t = \textit{Int}$ to the set of constraints. In order for $\textit{Pred}(e)$ to be well-typed, these constraints must have a solution, which means the type of e will be *Int*.

Many other cases are straightforward following this blueprint. In each case, we place no conditions in the premise of a rule, but instead move them to the set of constraints that must be satisfied. Here is the rule for *Mult*, for example:

$$\frac{\vdash e_1 : t_1, \mathcal{T}_1 \qquad \vdash e_2 : t_2, \mathcal{T}_2}{\vdash \mathit{Mult}(e_1, e_2) : \mathit{Int}, \mathcal{T}_1 \cup \mathcal{T}_2 \cup \{t_1 = \mathit{Int}, t_2 = \mathit{Int}\}}$$

The remaining interesting rules are for variables and conditionals:

$$\frac{}{\vdash x : x, \emptyset} \qquad \frac{\vdash e_i : t_i, \mathcal{T}_i \qquad i \in \{1,2,3\}}{\vdash \mathit{If}(e_1, e_2, e_3) : t_2, \bigcup_{i \in \{1,2,3\}} \mathcal{T}_i \cup \{t_1 = \mathit{Bool}, t_2 = t_3\}}$$

A variable generates no constraints, because a variable occurrence alone doesn't imply anything about how the value of the variable is used. But the type of a variable is just the variable's name. So if the variable occurrs in some context that constrains its type, a type constraint will be generated for the variable. For example, $\mathit{Pred}(x)$ generates the constraint $x = \mathit{Int}$. If a variable occurs in multiple contexts that use the variable with different types will generate an inconsistent set of constraints. For example, $\mathit{If}(x, \mathit{Succ}(x), 8)$ will generate the unsolvable equations $\{x = \mathit{Bool}, x = \mathit{Int}\}$.

An *If* expression generates type constraints for all its subexpressions and then constrains the type of the test expression to be a Boolean and the types of the consequent and alternative to be equal. Since they are constrained to be equal, we could have used t_1 instead of t_2 as the result type and it wouldn't make a difference.

Once a constraint set is generated, there is the issue of finding out if the constraints have a solution.

The general problem of determing if a set of equations between terms has a solution is known as the **unification problem**. The way we will represent a solution to an instance of the unification problem is as a substitution that unifies all of the equations. A substitution is partial function from variables to types. So if given a set of constraints $\mathcal{T}$, we will try to construct a substitution ψ such that $\psi(\mathcal{T})$ produces a set of trivial equations: $\psi(\mathcal{T}) = \{t_1 = t_1, t_2 = t_2, \dots\}$.

The unification of a set of equations is given by the following function, $\mathit{unify}(\mathcal{T})$:

$$\mathit{unify}(\mathcal{T}) = U(\mathcal{T}, \emptyset) \text{ where}$$

$$U(\emptyset, \psi) = \psi \tag{10}$$

$$U(\{t = t\} \cup \mathcal{T}, \psi) = U(\mathcal{T}, \psi) \tag{11}$$

$$U(\{x = t\} \cup \mathcal{T}, \psi) = \begin{cases} U(\mathcal{T}, \psi[t/x] \circ [x \mapsto t]) & \text{if } \psi(x) = \bot \\ U(\mathcal{T} \cup \{t = t'\}, \psi) & \text{if } \psi(x) = t' \end{cases} \tag{12}$$

$$U(\{t = x\} \cup \mathcal{T}, \psi) = U(\{x = t\} \cup \mathcal{T}, \psi) \tag{13}$$

The algorithm works by iteratively removing constraints, building a substitution, until there are no constraints left. If a constraint is trivial, i.e. it is of the form $t = t$, then it is removed. If a constraint is between a variable and a type, and the variable is not already in the solution, then the type replaces the variable in the substitution (denoted $\psi[t/x]$)

and the mapping from the variable to the type is added to the solution. If the variable is already in the substitution, a new constraint is added to equate the type and what's given by the substitution.

The unification is sound and complete: it produces a substitution whenver one exists, and the substitution produced is always a solution. If a solution exists, the program is well-typed under some typing environment:

Claim 5.11. *If* $\vdash e : t, \mathcal{T}$ *and unify*$(\mathcal{T}) = \psi$*, then there exists* Γ *and* t' *such that* $\Gamma \vdash e : t'$.

The typing environment is *almost* the same as the substituion ψ, but we have to be a little careful about underconstrained variables. For example in this program: *If*(*True*, *x*, *y*), the variables *x* and *y* must have the same type, which also the type of the whole program, but the solution to the constraints generated by this program is either $[x \mapsto y]$ or vice versa, which is not a type environment.

6 Functions

At this point, we've investigated the syntax, semantics, and analysis of very simple programming languages. This has made our job easier, but made that of a programmer's basically impossible because these languages are so computationally limited.

One of the simplest and most powerful computational mechanisms is that of a function. Functions enable code-reuse as they can be applied many times to different arguments; they are an essential abstraction mechanism. Functions also enable computations to be packaged as values, delaying their evaluation until the function is applied. While functions are a powerful programming mechanism, they significantly complicate reasoning about programs (as should be expected). For starters, programs with functions may run indefinitely long and may not ever produce values. Analysis methods must be careful to stay within the delicate realm of computability.

6.1 Syntax of functions

Let's develop a language $\mathcal{F}$ that is like $\mathcal{B}$ but with the addition of functions. The syntax, modelled in OCaml, includes everything from $\mathcal{B}$:

```
type fexp =
    Int of int
  | Bool of bool
  | Var of string
  | Pred of fexp
  | Succ of fexp
  | Plus of fexp * fexp
  | Mult of fexp * fexp
  | Div of fexp * fexp
  | If of fexp * fexp * fexp
```

and additionally it contains unary functions and applications:

```
  | App of fexp * fexp
  | Fun of string * fexp
```

A function consists of a bound variable, represented with a string, and a body expression. Functions are applied with `App`. For example `App(Fun("x", Var "x"), Int 0)` represents a $\mathcal{F}$ expression corresponding to the program `((fun x -> x) 0)` written in OCaml.

6.2 Semantics of functions

The meaning of a $\mathcal{B}$ program is an *answer*, which is either a value or an error. A value is either a integer or a boolean:

```
type ans =
    Val of value
  | Err of string
```

```
and value =
    VInt of int
  | VBool of bool
```

Evaluation `eval : fexp -> env -> ans` is defined in terms of an environment, which is represented as a list of variable, value pairs:

```
type env = (string * value) list
```

$\mathcal{F}$ programs are similar, but now there's a new kind of value: functions. How should we represent functions?

How did we represent booleans and integers? With booleans and integers. So how should we represent functions? With functions? Let's explore that idea. We can add another case to the `value` type:

```
  | VFun of (value -> ans)
```

If this is going to representation of function values, how should we handle evaluating `Fun(x,e)`? It should a function that receives a value `v` as input and then evaluates `e`. Since `e` may contain references to `x`, we need to evaluate `e` in an environment that maps `x` to `v`:

```
    | Fun (x, e) ->
        Val (VFun (fun v -> eval e ((x,v)::r)))
```

where `r` is the environment that `Fun(x,e)` was evaluated in.

For applications, we first evaluate each subexpression as usual, propagating errors

```
    | App (e1, e2) ->
        (match eval e1 r with
             Err s -> Err s
           | Val v1 ->
               (match eval e2 r with
                    Err s -> Err s
                  | Val v2 -> ...))
```

At this point, we have two values, `v1` and `v2` in hand. To evaluate the application requires apply `v1` (if it's a function) to `v2`:

```
   match (v1, v2) with
       VFun f, v -> f v
     | _, _ -> Err "type error: not a function"
```

Here's a little example calculation to test things out:

```
let two = Fun ("s", Fun ("z", App (Var "s", App (Var "s", Var "z"))))
let succ = Fun ("x", (Succ (Var "x")))
let v1 = eval
  (App ((App ((App (two, (App (two, App (two, two))))), succ)),
        Int 0))
  []
```

A Acknowledgments

Thanks to Michael Hicks, Becca MacKenzie, Garrett Katz, and Javran Cheng for comments and catching errors.