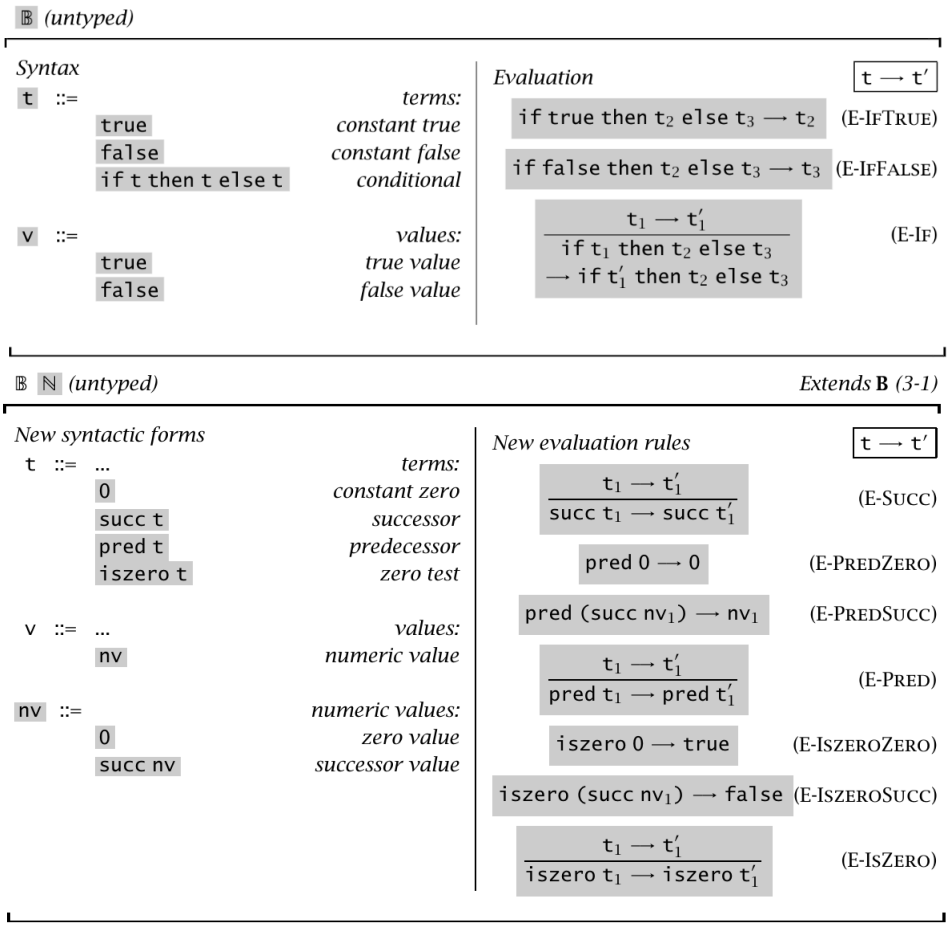
COMPSCI 3MI3 : Project 1 - Untyped Arithmetic Expressions

Fall 2021

Nicholas Moore

Project Due: Sunday October 10th at 11:59 PM

Recall our language of Untyped Arithmetic Expressions from topics 3 and 4.



Each part of this project will require you to submit a separate Haskell source file. The file from part 1 is used in part 2, and the file from part 2 is used in part 3. Nevertheless, please keep the part 1, 2 and 3 material separated for the purposes of marking.

1 Implementation of Small Step Semantics

In Assignment 2, you were asked to enocode the grammar of UAE in the type system of Haskell. The time has come to add the small-step semantics.

When you have finished these tasks, save your Haskell source file as `UAE1.hs`, and submit it to the P1 Avenue dropbox.

1.1 Modifying Our Datatype (10 points)

We will need to begin by creating two more data types, to represent both values and numeric values. Remove the value terms from your term data type, and create two new ones, conforming to the grammar given above.

This will become important later on in the project.

- Think about the way `v` and `nv` are used in the small step sematics above.

$$\mathcal{NV} \subset \mathcal{V} \subset \mathcal{T}$$

In order for UAE to work correctly, your two new value types will need to be **subtypes** of the term type. You can achieve this a couple different ways, but the easiest is as follows:

```
data SuperType = A | B | C | D SubType | E SubType SubType

data SubType = X | Y | Z
```

The one complication to this is that you'll need to specify both the supertype and subtype when pattern matching:

```
myfunc :: SuperType -> ...
myfunc (A) = ...
myfunc (B) = ...
myfunc (C) = ...
myfunc (D X) = ...
myfunc (D Y) = ...
myfunc (D Z) = ...
```

If you set things up in the manner specified above, you will cause a naming conflict between the successor function already defined in your term datatype and your new datatype for numeric values. If this is the case, rename the successor function in the numeric values datatype.

1.2 Single Step Semantics (25 points)

Next write a Haskell function which encodes the small-step operational semantics given above. This function should accept an argument of the term type, and produce an output also of the term type. Call this function `ssos`, which stands for *small-step opeartional semantics*.

Just as your data type for terms should not have any of Haskell's native data types in it (like boolean truth values or numbers), neither should your function. Points will be deducted for using Haskell's native types.

In addition to the rules above, please add rules which cause all values to evaluate to themselves (i.e., add reflexivity for values, both boolean and numeric). This will be necessary later on. Some rules will rely on the distinction between `succ t` and `succ nv`. Anywhere in the above small step semantics you see the successor taking a numeric value, you should interpret that as the successor function in your $\mathcal{NV}$ datatype.

If you used the subtyping method from the previous question, you will need to add one more rule to the above to transform a successor to a value-successor. This is relatively straightforward:

$$\texttt{succ nv} \rightarrow \texttt{succval nv}$$

Where `succval` is the version of the successor defined in your $\mathcal{NV}$ set.

- The reason for the above addition is this. Our semantics depend on the ability of certain terms to be able to distinguish between terms that are numeric values and terms that are not. Consider E-PredSucc and E-IsZero. These terms only operate over numeric values. This becomes a problem with expressions such as the following:

  `iszero succ succ succ 0` (1)

  The only successor function that "knows" that it holds a numeric value is the last one. The others don't have that information available. If however, we add the above evaluation rule, we can resolve this issue. A small amount of effort is also necessary to make sure that the other evaluation rules are looking for the correct version of `succ`, but once you've got it, the following:

  `pred succ succ succ pred pred succ succ succ 0` (2)

  Will evaluate to...

  `succval succval succval 0` (3)

  Otherwise, your final result will still have some instances of `pred` in it.

Here are some test cases, in case you want to check your work before submitting.

```
>> Pred $ Succ $ Succ $ Succ $ Pred $ Pred $ Succ $ Succ $ Succ $ NV Z
>> IfThenElse (IsZero (Succ (NV Z))) (V T) (Succ (NV Z))
>> IsZero $ V T
```

1.3 Multi-Step Semantics (5 points)

Now that we have our small-step semantics roughed out, write a function to evaluate our terms, called `eval`. This function should repeatedly apply the single-step semantic to a term until the term can't be evaluated further. How to figure out when a term can't be evaluated further is left as an exercise to the student, but as a general hint, in UAE not all irreducible terms are values, so you can't just evaluate until you get a value and call it a day!

2 Dealing with Wrongness (20 points)

We now have our grammar encoded, but we can still have behaviour within our language that we don't want. Specifically, not all evaluation chains in our system terminate in a value (as you may have noticed). This is because our language lacks the ability to produce runtime errors for syntactically correct expressions. With expressions such as:

`iszero true` (4)

There is no rule which applies. The expression can't be evaluated further, and the expression is not a value.

Let's add a new value, `wrong`, which will be the result of evaluating nonsense terms, like the one above. The following is an incomplete set of the rules needed to implement `wrong`. In the following, $\mathcal{BV}$ will be the set of Boolean values (`true` and `false`), and $\mathcal{NV}$ will be the set of numeric values, as defined above.

$$\forall n \in \mathcal{NV}, \forall t_2, t_3 \in \mathcal{T}, \texttt{if } n \texttt{ then } t_2 \texttt{ else } t_3 \rightarrow \texttt{wrong} \quad \text{(E-If-Wrong)}$$

$$\forall b \in \mathcal{BV}, \texttt{succ } b \rightarrow \texttt{wrong} \quad \text{(E-Succ-Wrong)}$$

$$\forall b \in \mathcal{BV}, \texttt{pred}\ b \rightarrow \texttt{wrong} \qquad \text{(E-Pred-Wrong)}$$

$$\forall b \in \mathcal{BV}, \texttt{iszero}\ b \rightarrow \texttt{wrong} \qquad \text{(E-IsZero-Wrong)}$$

In addition, you will need to add some more evaluation rules so that the following requirements are met. You are required to use the small-step semantic style to deal with these problems. (You can't just call `error`!):

- If, during normal execution, a `wrong` value is produced, the entire expression should evaluate to `wrong`.
- You may need to add one or two additional rules to make your `wrong` term work the same way as the other values.

Nothing special is required in our evaluation function, an expression evaluating to `wrong` is good enough. When finished, name your file `UAE2.hs`, and submit it to the P1 Avenue dropbox.

3 Big Step Semantics (20 points)

The essential difference between big step and small step semantics is that, where the small step semantics must be looped to find an expression's normal form, our big step semantics compute the entire result in one go.

Write a function, called `bsos` (Big Step Operational Semantics), which implements the following evaluation rules, which comprise the big-step semantics of UAE:

$$\mathsf{v} \Downarrow \mathsf{v} \qquad \text{(B-Value)}$$

$$\frac{\mathsf{t}_1 \Downarrow \texttt{true} \quad \mathsf{t}_2 \Downarrow \mathsf{v}_2}{\texttt{if}\ \mathsf{t}_1\ \texttt{then}\ \mathsf{t}_2\ \texttt{else}\ \mathsf{t}_3 \Downarrow \mathsf{v}_2} \qquad \text{(B-IfTrue)}$$

$$\frac{\mathsf{t}_1 \Downarrow \texttt{false} \quad \mathsf{t}_3 \Downarrow \mathsf{v}_3}{\texttt{if}\ \mathsf{t}_1\ \texttt{then}\ \mathsf{t}_2\ \texttt{else}\ \mathsf{t}_3 \Downarrow \mathsf{v}_3} \qquad \text{(B-IfFalse)}$$

$$\frac{\mathsf{t}_1 \Downarrow \mathsf{nv}_1}{\texttt{succ}\ \mathsf{t}_1 \Downarrow \texttt{succ}\ \mathsf{nv}_1} \qquad \text{(B-Succ)}$$

$$\frac{\mathsf{t}_1 \Downarrow 0}{\texttt{pred}\ \mathsf{t}_1 \Downarrow 0} \qquad \text{(B-PredZero)}$$

$$\frac{\mathsf{t}_1 \Downarrow \texttt{succ}\ \mathsf{nv}_1}{\texttt{pred}\ \mathsf{t}_1 \Downarrow \mathsf{nv}_1} \qquad \text{(B-PredSucc)}$$

$$\frac{\mathsf{t}_1 \Downarrow 0}{\texttt{iszero}\ \mathsf{t}_1 \Downarrow \texttt{true}} \qquad \text{(B-IszeroZero)}$$

$$\frac{\mathsf{t}_1 \Downarrow \texttt{succ}\ \mathsf{nv}_1}{\texttt{iszero}\ \mathsf{t}_1 \Downarrow \texttt{false}} \qquad \text{(B-IszeroSucc)}$$

You are free to introduce the `Wrong` term anywhere it may be appropriate. In general, your big-step semantics should evaluate expressions to the same normal form given by the multi-step small step semantics. That is,

$$t \rightarrow^* t' \wedge t \Downarrow t'' \implies t' = t'' \qquad (5)$$

When finished, name your file `UAE3.hs`, and submit it to the P1 Avenue dropbox.