EECS 662 Homework 1: Sums and Datatypes

J. Garrett Morris

September 18, 2017

Introduction

This homework will help you develop and demonstrate an understanding of *sum types*, one of the building blocks of data representation in programming languages, and their relationship with *datatypes*, a key programmer-facing abstraction of data representation primitives.

The homework distribution contains four Haskell files:

`Core.hs`	contains the interpreter for the *core* language
`Main.hs`	contains scaffolding code to run both parsers on either standard input or files
`Simple.hs`	contains a parser for the core language, extended with sums
`Sugar.hs`	contains parsing and desugaring functions for the *extended* language

The `Main.hs` file includes a driver program to simplify interaction with the interpreter. You can build Main either using the included `Makefile`, or using the following GHC command:

```
ghc --make Main.hs
```

There are two modes of interacting with the interpreter. If you just call the interpreter (without passing any file names on the command line) then it functions as an interactive interpreter. You can write expressions, and those expression will be parsed, checked, and evaluated. For example:

```
$ ./Main
> 2 + 2
VInt 4
> 2 + True
ERROR: type error
> :q
$
```

Or, you can pass the interpreter a file (or series of files) to interpret. For example, if the file `Add.stlc` contains the text `2+2`, then:

```
$ ./Main Add.stlc
VInt 4
$
```

If you pass the `--core` argument to the interpreter, it expects the core language (i.e., does not invoke the functions you should write in Problem 2); otherwise, it expects the sugared language (i.e., does expect that you have attempted Problem 2).

The homework distribution also includes several sample source files: `SmallSum.stlc`, `LargeSum.stlc`, `SmallData.stlc`, `LargeData.stlc`. The first two should be run with the `--core` flag. Each has its expected result as a comment on the first line of the file.

1 Sums

The first problem asks you to extend the core language (`Core.hs`) with support for *sum types*. Just as product types provide a generic way to combine a number of values into a single value, sum types provide a generic way to combine a number of options into a single value. Sums generalize several ideas that we have already seen in class, including Booleans and `Maybe` types. The sum of types t and u is written $t + u$; a value of type $t + u$ is *either* a value of type t (conventionally called the left case) *or* a value of type u (conventionally called the right case). We introduce three new syntactic forms for using values of sum type. The first two construct new sum values, given values of either summand type. This is conventionally called "injecting" values into the sum, and therefore the type constructs are called `Inl` (for injection on the left) and `Inr` (for injection on the right). Finally, we have a `case` construct for branching on values of sum type. Unlike the `if` construct we have already developed, `case` both branches and binds values.

Part 1. In the first part of this problem, you will extend the type checker (i.e., the `check` function in `Core.hs`) to support sum types and their terms. The typing rules for sums are as follows:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{Inl}[t+u]\ e : t+u} \quad \frac{\Gamma \vdash e : u}{\Gamma \vdash \texttt{Inr}[t+u]\ e : t+u} \quad \frac{\Gamma \vdash e : t_1 + t_2 \quad \Gamma, x_1 : t_1 \vdash e_1 : u \quad \Gamma, x_2 : t_2 \vdash e_2 : u}{\Gamma \vdash \texttt{case}\ e\ \texttt{of Inl}\ x_1\ \texttt{->}\ e_1 \mid \texttt{Inr}\ x_2\ \texttt{->}\ e_2 : u}$$

The `Inl` and `Inr` terms come with type annotations $t + u$. The unannotated versions of these terms

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{Inl}\ e : t+u} \quad \frac{\Gamma \vdash e : u}{\Gamma \vdash \texttt{Inr}\ e : t+u}$$

make type checking difficult. In the typing rule for `Inl`, the type u appears nowhere besides the result, so in trying to write a `check` equations for this case, we would have to invent the right u. By adding type annotations, we avoid this problem.

These following terms are well typed, and should have types computed as shown:

Expression	*Type*
$\texttt{Inl}[Int + Bool]\ 1$	`Int + Bool`
$\texttt{Inr}[Int + (Int + Bool)]\ (\texttt{Inl}[Int + Bool]\ 1)$	`Int + (Int + Bool)`
$\texttt{\textbackslash x : Int + Bool -> case x of Inl y -> y + 1 | Inr y -> 0}$	`(Int + Bool) -> Int`

On the other hand, the following terms are not well typed, and so should be rejected:

$\texttt{Inl}[Bool + Int]\ 1$
$\texttt{Inr}[Int + (Int + Bool)]\ (\texttt{Inl}[Int + Bool]\ True)$

Part 2. In the second part, you will extend the evaluator (i.e., the `eval` function) to support sum types and their terms. The evaluation rules for sums are as follows:

$$\frac{e \Downarrow w}{\texttt{Inl}[t+u]\ e \Downarrow \texttt{Inl}\ w} \quad \frac{e \Downarrow w}{\texttt{Inr}[t+u]\ e \Downarrow \texttt{Inr}\ w}$$

$$\frac{e \Downarrow \texttt{Inl}\ w \quad [x_1 \mapsto w]e_1 \Downarrow v}{\texttt{case}\ e\ \texttt{of Inl}\ x_1\ \texttt{->}\ e_1 \mid \texttt{Inr}\ x_2\ \texttt{->}\ e_2 \Downarrow \texttt{Inl}\ v} \quad \frac{e \Downarrow \texttt{Inr}\ w \quad [x_2 \mapsto w]e_2 \Downarrow v}{\texttt{case}\ e\ \texttt{of Inl}\ x_1\ \texttt{->}\ e_1 \mid \texttt{Inr}\ x_2\ \texttt{->}\ e_2 \Downarrow \texttt{Inr}\ v}$$

The first pair of evaluation rules capture that sums introduce two new kinds of values: values on the left of sum types $\texttt{Inl}[w+]$ and values on the right of sum types $\texttt{Inr}[w+]$. As we have gotten past typing, we no longer need to track the type annotations on values in sums. The second two rules describe how `case` behaves; they should be unsurprising. The following terms should reduce as shown:

Expression		*Result*
$\texttt{Inr}[Int + (Int + Bool)]\ (\texttt{Inr}[Int + Bool]\ True)$	$\Downarrow$	`Inr (Inr (VBool True))`
$\texttt{case Inl}[Int + Bool]\ 4\ \texttt{of Inl}\ x\ \texttt{->}\ x * x \mid \texttt{Inr}\ y\ \texttt{->}\ 0$	$\Downarrow$	`VInt 16`

2 Datatypes

The second problem asks you to extend the surface language with support for *datatypes*. This means we will extend the source language with: *datatype declarations* to new data types, *constructors* to build values of those types, and *case expressions* to eliminate them. These constructs will all be expressed in terms of the core calculus developed in part 1. Rather than further extending the core calculus, you will write a desugaring function to transform datatypes, constructors, and case expressions into the sum and product types and terms of the core language.

For a simple example of the extended language:

```
data T = K1 Int Int | K2 Bool in
case K1 2 3 of
  K2 b -> 0
| K1 x y -> x + y
```

The first line introduces a new data type. Unlike data types in Haskell, our data types will have scopes, like `let` constructs. The data type `T`, along with its constructors `K1` and `K2`, will be available in the expression following `in`. The expression `K1 2 3` constructs a value of type `T`. Finally, the `case` expression examines this `T` value and branches based on which constructor was used to build it. This expression will evaluate to 5.

Your implementation will all be in the file `Sugar.hs`, as extensions to the provided `desugar` function. You will have to add four new desugaring cases:

- Deugaring data types (like `T`) into their underlying implementations in terms of sums and products (like `(Int * Int) + Bool`);
- Desugaring constructor expressions (like `K1 2 3`) into constructors of products and sums; and,
- Desugaring `case` expressions for data types into `case` expressions for sums and `let` expressions to deconstruct products.

Each of these will depend on some kind of *environment* tracking the definitions of data types. The provided desugaring function has an argument for the environment (`dtypes`), and the homework provides one possible type for this environment. However, you should feel free to change this environment (change the type synonym `DtypeEnv`) if you want. So, your final desugaring task is:

- Eliminating data type declarations from the program, updating the data type environment accordingly.

Note. The resulting surface language will *not* have either sums of pairs directly. These ideas must be expressed using data types.

Types. Your first task is to desugar data types in core types. The basic idea is that you will interpret each individual constructor as a *product* of its arguments, and then build a *sum* of these products to represent the original type. The following table demonstrates several *possible* interpretations of data types as core types (products and sums):

Data type declaration	*Interpretation of data type*
`data T1 = KA Int | KB Bool in ...`	`Int + Bool`
`data T1' = KA' Int | KB' Bool in ...`	`Bool + Int`
`data T2 = KC Int Int | KD Bool | KE (Int -> Int) in ...`	`(Int * Int) + (Bool + (Int -> Int))`

As illustrated by the first and second examples, you have freedom in how you order and associate sums and products in your interpretation of data types. Of course, you must be consistent in your interpretation of data types throughout your implementation.

In desugaring data type declarations and data types, you have two tasks. The first is to remove data type declarations, while updating the data types environment correspondingly. Your code should remove `EDataType name ctors e` nodes from the input expression; you should provide an extended datatype environment in a recursive call to `desugar` to desugar `e` . The second is to rewrite type annotations to replace data type names with the corresponding sum-of-product types. For example:

Expression	*Translation*
`\x : T1 -> x`	`\x : Int + Int -> x`
`\x : T1 * Int -> let (y,z) = x in y`	`\x : (Int + Int) * Int -> let (y,z) = x in y`

The provided desugaring code includes a function `desugarTy` to handle desugaring of types. Your code should replace `TDatatype` nodes with corresponding `TSum` or `TProduct` expressions.

Terms. Your second task is to desugar data type constructor expressions into `Inl`, `Inr` and `(,)` expressions, and data type `case` expressions into sum `case` expressions and product `let` expressions. This desugaring is driven by your choice of interpretation, as shown by the following table:

Sugared expressions	*Core expression*
`KC 1 2`	`Inl[Int * Int + (Bool + (Int -> Int))] (1,2)`

`case z of`
`  KA x -> x + x`
`| KB y -> 0`
	`case z of`
	`  Inl x -> x + x`
	`| Inr y -> 0`

`case z of`
`  KC x y -> x + y`
`| KD b -> if b then 0 else 1`
`| KE f -> f 0`
	`case z of`
	`  Inl w -> let (x, y) = w in x + y`
	`| Inr w ->`
	`  case w of`
	`  Inl b -> if b then 0 else 1`
	`  | Inr f -> f 0`

In the first line, because we have interpreted type `T2` as `(Int * Int) + (Bool + (Int -> Int))`, our interpretation of `KC` must be the corresponding `Inl`. Similarly, the ordering of `case`s is also forced by the interpretation of the type: as shown in the third line, the `KC` case must come first, because we have chosen that to be the left-most summand in the interpretation of `T`.

Assumptions and hints. You are allowed to make some assumptions:

- Data types will contain at least one constructor
- Each constructor will contain at least one value. (You don't have to worry about something like `data Boolean = Truth | Falsity in Truth`.)

However, you are *not* allowed to make several other assumptions

- Do *not* assume that data types contain more than one constructor (i.e., are actually sums). You do have to support cases like `data Ints = K Int Int in K 1 2`.
- Do *not* assume that constructors contain more than one value (i.e., are actually products). You do have to support cases like `data T = K1 Int | K2 Int in K1 1`.
- Do *not* assume that branches in case statements will appear in the same order as they appeared in the data declaration. You do have to support cases like the following:
  `data T = K1 Int | K2 Int in`
  `case K1 1 of K2 x -> x | K1 y -> y`
  However, order of branches in case statements is not significant. You should compile the code above exactly the same as the following:
  `data T = K1 Int | K2 Int in`
  `case K1 1 of K1 y -> y | K2 x -> x`
- Do *not* assume that some user variable is free to use in your case declarations (such as the variable `w` in the examples above). However, you can create variables programmatically that cannot be used in user code. For example, `$v` cannot be used as a variable in user code (it will not parse), but you are perfectly welcome to create a variable like that in your desugarer (for example, by saying `EVar "$v"`).

Submission Instructions

Your submission should include the four Haskell files from the original homework distribution, modified as necessary for your solution, the test files from the original distribution, and (at least) four new test cases, stored in separate files. Each of your test files should being with a comment line (i.e., a line beginning with two dashes) containing their expected result. Each of the provided tests follows this format; check there for details. Please submit these as a single file (`.zip` or `.tar`), via Blackboard.

Point allocation:

Problem 1		30
Type checking	15	
Evaluation	15	
Problem 2		70
Types	10	
Constructors	25	
Case blocks	35	
Test cases (4)		16
Total		116