04_impl

Baldur Blöndal

May 19, 2016

Contents

TODO: Minnast á að ég fann villu í nested if-then-else með aðstoð QuickCheck.

1 Hutton's Razor

Preliminary implementations focused on a simple arithmetic expression language rather than the language in its entirety:

```
data Exp = Val Int | Add Exp Exp
```

This simple language with integer constants and addition — often dubbed "Hutton's razor" — is a common vehicle (**rephrase**) for exploring compilation.[1] It is "unityped" since there is only a single type involved (`Int`).

An example of an expression of our simple language is an expression adding numbers from 1 to 4 $(1 + 2) + (3 + 4)$:

```
exp :: Exp
exp = Add (Add (Val 1) (Val 2))
          (Add (Val 3) (Val 4))
```

This is not a pleasant surface language for users. Making `Exp` an instance of `Num` will make it more convenient to work with:

```
instance Num Exp where
  (+) :: Exp -> Exp -> Exp
  (+) = Add

  fromInteger :: Integer -> Exp
  fromInteger n = Val (fromInteger n)
```

`exp` can then be rewritten as

```
exp :: Exp
exp = (1 + 2) + (3 + 4)
```

Note the other methods of the `Num` type class do not have a sensible implemention for our expression language and are therefore omitted. Calling those methods will result in a run-time error.

This simple expression language is deeply embedded and can be interpreted or observed in a variety of ways. One possible observer evaluates the expression by recursively replacing each `Val` with the identity function and each `Add` by addition:

```
eval :: Exp -> Int
eval (Val n)   = n
eval (Add a b) = eval a + eval b
```

which can be used to evaluate the `exp`

[1] Hutton, G.: Fold and unfold for program semantics. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland (1998) 280-288

```
> eval exp
10
```

This evaluation function will be extended with the EDSL and will play a role in testing the compilation process as a testing oracle.

Another observation is to print the expressions as strings:

```
toString :: Exp -> String
toString (Val n)   = printf "%d" n
toString (Add a b) = printf "(%s + %s)" (toString a) (toString b)
```

```
> toString exp
"((1 + 2) + (3 + 4))"
```

Since we're interested in producing an executable we are interested in a third interpretation: compiling the expression language into LLVM assembly. since our language includes no control flow the entire language can be compiled into an LLVM basic block without additional labels or jumps:

```
compile :: Exp -> String
```

```
> putStrLn (compile exp)
entry:
  %add1 = add i64 1, 2
  %add2 = add i64 3, 4
  %add3 = add i64 %add1, %add2
  ret i64 %add3
```

Plugging that basic block into a complete function definition gives

```
define i64 @foobar() {
entry:
  %add1 = add i64 1, 2
  %add2 = add i64 3, 4
  %add3 = add i64 %add1, %add2
  ret i64 %add3
}
```

which can then be interpreted or turned into an executable

```
compileRun :: Exp -> IO Word64
```

```
> compileRun exp
10
```

1.1 Testing

- QuickCheck QuickCheck was used to test properties of the compiler.

QuickCheck generates test cases and tests whether properties hold. In order to make test cases we need to indicate how to generate random terms of `Exp` by defining an `Arbitrary` instance for `Exp` and a generator `arbitrary :: Gen Exp` that generates expression.

A naive definition of `arbitrary` way means the generator is defined recursively has an issue:

```
instance Arbitrary Exp where
  arbitrary :: Gen Exp
  arbitrary = oneof
    [ Val <$> arbitrary,
      Add <$> arbitrary <*> arbitrary ]
```

The expressions it generates may become too large or even fail to terminate. Each QuickCheck generator carries with it a size parameter that slowly increases during testing, to avoid the previous situation we can use the combinator `sized` to modify the value of our size parameter:

```
sized :: (Int -> Gen a) -> Gen a
```

Using this we can cut the size parameter in two in the recursive cases:

```
instance Arbitrary Exp where
  arbitrary :: Gen Exp
  arbitrary = sized sizedExpr where

    sizedExpr :: Int → Gen Exp
    sizedExpr 0 = liftA I arbitrary
    sizedExpr n = let
      subtree = sizedExpr (n `div` 2)

      in oneof [ liftA  I    arbitrary,
                 liftA2 Add subtree subtree ]
```

A desirable property of the compiler is that its output agree with the evaluation function, using it as a *test oracle* which for our purposes is a source of expected results we can compare against. Then it is simple to

compile our expressions and compare the resulting value to the evaluation function to see if there is a mismatch.

Because the `compileRun` function returns an `IO Word64` action we use monadic QuickCheck (`Test.QuickCheck.Monadic`) to test the output:

```
prop_eval :: Exp -> Property
prop_eval exp = monadicIO $ do
  compiled <- run (compileRun exp)

  assert (eval exp == compiled)
```

and run it using:

```
> quickCheck prop_eval
+++ OK, passed 100 tests.
```

This generates 100 random expression trees, compiles them and compares them to our oracle and the outputs match making sure the compilation is at the very least consistent with the `eval` observer whose implementation is much simpler to verify.

This works as a simple sanity check.

2 if-then-else

TALK ABOUT KEEPING TRACK OF ENVIRONMENT OF THE BASIC BLOCKS

We augment our language with an *if*-expression

```
data Exp = ... | If Exp Exp Exp
```

where the arguments of `If` indicate the conditional, then and else branches respectively. For simplicity 0 indicates falsehood.

2.1 Testing

When testing an expression with an *if* expression we ideally want to test the branches with similar frequency.

This means the conditional tests should be biased towards expressions that evaluate to 0.

Simple solution:

```
zeroBias :: Gen Exp
zeroBias = suchThat arbitaray (\exp -> eval exp == 0)
```

more efficient solution that generates a smaller space:

```
zeroBias' :: Gen Exp
zeroBias' = do
  exp <- arbitrary
  return (exp + Val (- eval exp))
```

Again we need to be cognisant of the generator's size parameter, since `If` takes three arguments and the size of each generator should be split in three.

...

3 Ad-hoc polymorphism

We want to operate on different numeric values using the same operator,

```
(+) :: Exp Int8  -> Exp Int8  -> Exp Int8
(+) :: Exp Int32 -> Exp Int32 -> Exp Int32
```

without having a new constructor for each type:

```
Add8  :: Exp Int8  -> Exp Int8  -> Exp Int8
Add32 :: Exp Int32 -> Exp Int32 -> Exp Int32
```

Instead of going the route of Accelerate[2] which models its hierarchy via type classes, I create a data type to represent the hierarchy and use it to index our `Exp`. Consider a slightly richer language with scalar values such as Booleans as well as 8- and 32-bit numbers, our implementation has four components:

- Universe
  - `MkNumber I8` represents 8-bit integers, while `MkNotnum I1` represents Boolean values.

[2] `https://github.com/AccelerateHS/accelerate/blob/179fb230a6af1aa10789c96c1d9be45a2f627b13/Data/Array/Accelerate/Type.hs`

- Type-indexed type representation
  - `TypeRep (MkNumber I8)` is a representation of the type `MkNumber I8`. Unlike `Typeable` we decide to create one for each level of our hierarchy.

    This allows us to be more specific about our representation, `TypeRep` is a representation of every type while `NumberRep` is only a representation of numeric values.
- Implicit type representations (`Typeable`)
  - `typeRep :: Typeable ty => TypeRep ty`

    Like our type representation we have implicit representations for each layer

    ```
    numRep :: GetNumber num => NumberRep num
    notRep :: GetNotnum not => NotnumRep not
    ```
- A mapping from our universe `MkNumber I8` to the Haskell types `Int8`.

3.1 Universe

```
data Scalar = MkNumber Number | MkNotnum Number

data Number = I8 | I32

data Notnum = I1
```

3.2 Type-indexed type representation

Now we construct an indexed type representation:

```
data NumberRep :: Number -> Type where
  RepInt8  :: NumberRep I8
  RepInt32 :: NumberRep I32

data NotnumRep :: Notnum -> Type where
  RepBool :: NotnumRep I1

data TypeRep :: Scalar -> Type where
  RepNumber :: NumberRep num -> TypeRep (MkNumber num)
  RepNotnum :: NotnumRep num -> TypeRep (MkNotnum num)
```

This is similar to the recent development of type indexed `TypeRep` [3] with the exception that we can talk about subsets of our hierarchy.

3.3 Implicit type representation

What about our version of `Typeable`?

We have one class for each layer

```
class GetNumber num where  numRep :: NumberRep num
class GetNotnum not where  notRep :: NotnumRep not
class Typeable  ty  where typeRep ::   TypeRep ty
```

The 'leafs' have straightforward instances

```
instance GetNumber I8  where numRep = RepInt8  :: NumberRep I8
instance GetNumber I32 where numRep = RepInt32 :: NumberRep I32
instance GetNotnum I1  where notRep = RepBool  :: NotnumRep I1
```

While `Typeable` instances can build on the `GetNumber`, `GetNotnum` instances:

```
instance GetNumber num => Typeable (MkNumber num) where
  typeRep :: TypeRep (MkNumber num)
  typeRep = RepNumber getNum


instance GetNotnum not => Typeable (MkNotnum not) where
  typeRep :: TypeRep (MkNotnum not)
  typeRep = RepNotnum getNot
```

This is elegant, thus right.[4]

Unlike the approach taken by Accelerate, we don't need need multiple instances for the same type:

```
instance IsIntegral Int where
  integralType = TypeInt IntegralDict
instance IsIntegral Int8 where
  integralType = TypeInt8 IntegralDict

instance IsNum Int where
  numType = IntegralNumType integralType
instance IsNum Int8 where
  numType = IntegralNumType integralType
```

[3] `https://ghc.haskell.org/trac/ghc/wiki/Typeable`

[4] `https://en.wikipedia.org/wiki/Mathematical_beauty`

3.4 Mapping to Haskell types

This is trickier than you'd think, the naive approach might be:

```
type family
  ToType (a :: Scalar) = (res :: Type) where
  ToType (MkNumber I8)  = Int8
  ToType (MkNumber I32) = Int32
  ToType (MkNotnum I1)  = Bool
```

or splitting it up as we've done with the explicit and implicit type representation hierarchy

```
class GetNumber num where type NumToType num :: Type
class GetNotnum not where type NotToType not :: Type
class Typeable  ty  where type ToType    ty  :: Type

instance GetNumber I8  where type NumToType I8  = Int8
instance GetNumber I32 where type NumToType I32 = Int32
instance GetNotnum I1  where type NotToType I1  = Bool
```

And again our `Typeable` instance becomes rather elegant:

```
instance GetNumber num => Typeable (MkNumber num) where
  type ToType (MkNumber num) = NumToType num

instance GetNotnum not => Typeable (MkNotnum not) where
  type ToType (MkNotnum not) = NotToType not
```

So what's the problem?

With this type family we associate `Int8` to `MkNumber I8`, but we have no way of going the other way! That is to say, we would like a dependency that `Int8` imply `MkNumber I8` (we want `ToType` to be injective) — the problem here is that injectivity is not a compositional property![5]

We can certainly claim that `NumToType` and `NotToType` are injective, but that does not mean that

```
type ToType (MkNumber num) = NumToType num
type ToType (MkNotnum not) = NotToType not
```

[5] `http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/injective-type-families-acm.pdf`

is injective.

Why do we need injectivity? Well later we would like to define:

```
data Exp a where
  Val :: Typeable a => ToType a -> Exp a
```

and we would like

```
ghci> :type Val True
Val True :: Exp (MkNotnum I1)
```

but as it stands we get

```
ghci> :t Val True
Val True :: (ToType a ~ Bool, Typeable a) => Exp a
```

we know that for `ToType a ~ Bool` a must equal `MkNotnum I1` but GHC doesn't.

"Fine" you say, "just add a dependency"

```
type family
  ToType (a :: Scalar) = (res :: Type) | res -> a where
  ToType (MkNumber I8)  = Int8
  ToType (MkNumber I32) = Int32
  ToType (MkNotnum I1)  = Bool
```

This works but it becomes tricky when we add arrays and pairs to our language.

3.5 Our language

What does our language look like?

```
data Exp a where
  Val :: Typeable a => ToType a -> Exp a
  Add :: Exp (MkNumber n) -> Exp (MkNumber n) -> Exp (MkNumber n)
```

Now the type of addition is made precise, it adds two numbers and returns a number. But defining its `Num` instance isn't straightforward.

3.5.1 Every expression is a number... (as long as it's a number)

```
instance GetNumber num => Num (Exp (MkNumber num)) where
  (+) :: Exp (MkNumber num) -> Exp (MkNumber num) -> Exp (MkNumber num)
  (+) = Add

  fromInteger :: Integer -> Exp (MkNumber num)
  fromInteger = Val . fromInteger
```

However this is unsatisfactory for several reasons, it means that if we have a constructor

```
Min :: Exp a -> Exp a -> Exp a
```

applying it to two numeric literals doesn't pick our `Num` instance, GHC doesn't seem to know our expression language is a number:

```
ghci> :type Min 1 2
Min 1 2 :: Num (Exp a) => Exp a
```

This is a result of how GHC's instance resolution works, it tries to match against the head of the instance but there is no *Num* instance for `Exp _`, only for `Exp (MkNumber _)`.

To solve this I use a trick I haven't seen elsewhere:

```
instance (a ~ MkNumber num, GetNumber num) => Num (Exp a) where
```

Now GHC knows that any `Exp a` that is a number, must be of the form `Exp (MkNumber _)`. Applying `Min` to two numbers once again gives the desired result:

```
ghci> :type Min 1 2
Min 1 2 :: GetNumber num => Exp (MkNumber num)
```

3.5.2 What about constraints?

When we start implementing the `Num` methods we run into trouble, we don't know that `ToType a` is actually a number. When we start implementing:

```
instance (a ~ MkNumber num, GetNumber num) => Num (Exp a) where
  (+) :: Exp (MkNumber num) -> Exp (MkNumber num) -> Exp (MkNumber num)
  Val a + Val b = Val (a + b)
```

```
  a     + b     = Add a b

  fromInteger :: Integer -> Exp (MkNumber num)
  fromInteger = Val . fromInteger
```

GHC claims it can't deduce `Num (ToType ('MkNumber num))` from `Val (a + b)` and `Val . fromInteger`.

One solution is to add it to the context:

```
instance (Num (ToType a), a ~ MkNumber num, GetNumber num) => Num (Exp a) where
```

but we claim that every `MkNumber` is a number so we add a *Num* constraint on the `GetNumber` class — it is a property of our numbers.

We also want to be able to check that `Val 0 + b = b` so we add an `Eq` constraint as well.

```
class (Eq (ToType (MkNumber num)), Num (ToType (MkNumber num))) => GetNumber num where
```

and now we have our desired *Num* instance:

```
instance (a ~ MkNumber num, GetNumber num) => Num (Exp a) where
  (+) :: Exp (MkNumber num) -> Exp (MkNumber num) -> Exp (MkNumber num)
  Val 0 + b     = b
  a     + Val 0 = a
  Val a + Val b = Val (a + b)
  a     + b     = Add a b
```

with simple constant folding.

3.6 Overview

This way of modeling also means that we can define all these functions as specialisations of `typeRep`

```
getNum :: GetNumber num => TyRep (MkNumber num)
getNum = typeRep @(MkNumber _)

getNot :: GetNotnum not => TyRep (MkNotnum not)
getNot = typeRep @(MkNotnum _)

getTy :: Typeable a => TyRep a
getTy = typeRep
```

as can be seen from:

```
[getTy,getNum] :: GetNumber num => [TyRep (MkNumber num)]

[getTy,getNot] :: GetNotnum not => [TyRep (MkNotnum not)]
```

so we have a "proper subtyping" (ask Richard, this is probably **not** subtyping, rather subsumption?).

As I understand it, this means we do away with explicit **downcasting** since that is taken care off by subsumption.

Or was it upcasting? My anti-subtyping bias is showing.