Generating Higher-order Maude from Haskell

Everett Hildenbrandt, Lucas Pena

Combining Algebra and Higher-Order Types

It's known that combining algebras and the simply typed lambda calculus can be done correctly (in a way that preserves the Church-Rosser property of both)[1]. Both of these can be expressed equationally, giving fully algebraic models of the combined theories.

In this small document, we demonstrate this possibility by defining terms which behave as higher-order functions and integrating that with other equational theories. All of this is expressed in many-sorted equational logic, with the sort-checking performing the type-checking for the higher-order functional part, and normal equational reduction performing the equivalent of beta-reduction.

Rather than offering lambda abstraction as an operation for defining lambda terms, we choose the combinator route. To define a "function", you provide a constant of the appropriate sort as well as equational definitions which look very functional in nature. We have demonstrated a few higher-order functions (`map` and `foldl`), as well as partial function application (with appropriate sort/type inference).

Original Haskell

```haskell
module HaskTest where

import Prelude hiding (Foldable, Maybe, map, Just, Nothing, foldl)

data Maybe a = Just a
             | Nothing
             deriving Show

infixr 5 :|
data Cons a = Nil
            | a :| Cons a
            deriving Show

class Mappable f where
    map :: (a -> b) -> f a -> f b

instance Mappable Maybe where
    map f Nothing = Nothing
    map f (Just a) = Just (f a)

instance Mappable Cons where
    map f Nil = Nil
    map f (a :| as) = f a :| map f as
```

```haskell
class Foldable f where
    foldl :: (b -> a -> b) -> b -> f a -> b

instance Foldable Cons where
    foldl f b Nil = b
    foldl f b (a :| as) = foldl f (f b a) as
```

Above is an example of two common Haskell algebraic datatypes; the `Maybe` datatype specifies the possible absence of data/result and the `Cons` datatype represents a singly-linked list. In addition, the typeclasses `Mappable` and `Foldable` are defined. We've provided `Mappable` instances for both `Maybe` and `Cons`, which amounts to defining the function `map` for each of them. We've also provided a `Foldable` instance for `Cons`. Note that both `map` and `foldl` above are higher order functions, and that when using `map`, you must infer whether you are mapping over a `Cons` or a `Maybe`.

We would like to be able to use Haskell-like higher order code within Maude, or even be able to use the above code directly in Maude. The following sections discuss how this and similar higher order Haskell modules can be converted into equivalent Maude modules.

Maude Code

Pre-Exists

```
fmod FUNCTION{X :: TRIV, Y :: TRIV} is
    sort =>{X,Y} .
    op __   : =>{X,Y} X$Elt -> Y$Elt [prec 40] .
    op _$_  : =>{X,Y} X$Elt -> Y$Elt [prec 60] .
    var f : =>{X,Y} .
    var x : X$Elt .
    eq f $ x = f x .
endfm

fmod FUNCTION-ID{X :: TRIV} is
    protecting FUNCTION{X,X} .
    op id : -> =>{X,X} .
    var x : X$Elt .
    eq id x = x .
endfm

fmod FUNCTION-COMP{X :: TRIV, Y :: TRIV, Z :: TRIV} is
    protecting FUNCTION{X,Y} .
    protecting FUNCTION{Y,Z} .
    protecting FUNCTION{X,Z} .

    op _._ : =>{Y,Z} =>{X,Y} -> =>{X,Z} [gather (E e) prec 44].

    var f : =>{X,Y} .
    var g : =>{Y,Z} .
    var x : X$Elt .
    eq (g . f) x = g (f x) .
endfm
```

The above code acts as a "prelude" for other higher-order modules. The most important of the three modules specified above is the `FUNCTION` module, parametrized on two `TRIV` theories `X` and `Y`. Given the

`Elt` sorts of these theories, it creates a new sort that represents functions `[X -> Y]`. `FUNCTION{X,Y}` also defines the `__` operator for function application. By defining the `__` operator in this way, we are able to rely on Maude's sort checker to rule out ill-formed simply-typed functional terms. We also define the `_$_` operator in this module, which is based off Haskell's function of the same name, and is used as a low-precedence function application operator.

The other modules are specified mostly for the user's convenience. The `FUNCTION-ID` module gives the identity function on `X`. The `FUNCTION-COMP` module is parametrized on `X`, `Y`, and `Z`, and allows one to compose a function from `Y` to `Z` with a function from `X` to `Y`, resulting in a function from `X` to `Z`. Here, using the `=>{.,.}` notation, higher order function composition can be expressed without much difficulty. Further, note that by our definition of `_._`, if two functions cannot be composed, Maude's sort checker will disallow usage of the `_._` operator.

Generated

We would like to generate the following code given the specification above.

Core Maude

```
fmod DATA-MAYBE{a :: TRIV} is
    sort Maybe{a} .
    op Nothing : -> Maybe{a} [ctor] .
    op Just_ : a$Elt -> Maybe{a} [ctor] .
endfm

fmod DATA-CONS{a :: TRIV} is
    sort Cons{a} .
    op Nil : -> Cons{a} [ctor] .
    op _:|_ : a$Elt Cons{a} -> Cons{a} [ctor] .
endfm
```

Because Haskell data-types are just Algebraic Data Types (ADTs), their representation in Maude is nearly identical to that in Haskell. Maude has many fewer restrictions on the allowed syntax for defining ADTs; for instance we had to choose the syntax `:|` for the Haskell above to be compilable but Maude has no such restriction.

Maude has open sorts, which means that we can also easily extend the `Maybe` or `Cons` data-types later with more data constructors. Adding data-constructors to a type in Haskell can be very painful - it requires adding definitions to all the places where that datatype is used. Additionally, Maude supports not just many-sorted equational logic, but order-sorted equational logic; this could conceivably be used to provide very natural data-subtyping, something that is not immediately present in Haskell.

Full Maude

```
load full-maude27.maude .

(
view Maybe{a :: TRIV} from TRIV to DATA-MAYBE{a} is
    sort Elt to Maybe{a} .
endv
)
```

```
(
view Cons{a :: TRIV} from TRIV to DATA-CONS{a} is
    sort Elt to Cons{a} .
endv
)

(
view =>{X :: TRIV, Y :: TRIV} from TRIV to FUNCTION{X,Y} is
    sort Elt to =>{X,Y} .
endv
)
```

To actually get usable datatypes and functions, we must instantiate the Maude modules above with the corresponding `TRIV` theories. Here, we provide some parameterized views (supported by Full Maude) which make this process easier. To get a function `[X -> Y]`, a user can use the view `=>{X,Y}`. As long as there are `TRIV` instances for both `X` and `Y` Full Maude will generate the appropriate view `=>{X,Y}` for the user.

In *Combining Algebra and Higher-Order Types*[1], a base type is a sort of some equational theory. These types can be combined using lambda-terms to form other more complex types. By having these parameterized views, we are declaring that anything of sort `TRIV` is a base type, as well as anything built from the data-constructors for `Maybe` and `Cons`. We've also added `=>{X,Y}` as a base-type here too, meaning we can build multi-argument functions and higher-order functions.

```
(
fmod INSTANCE-MAPPABLE-MAYBE{a :: TRIV, b :: TRIV} is
    extending FUNCTION{=>{a,b}, =>{Maybe{a},Maybe{b}}} .

    op map : -> =>{=>{a,b}, =>{Maybe{a},Maybe{b}}} .

    var f : =>{a,b} .
    var a : a$Elt .
    eq map f Nothing = Nothing .
    eq map f (Just a) = Just (f a) .
endfm
)

(
fmod INSTANCE-MAPPABLE-CONS{a :: TRIV, b :: TRIV} is
    extending FUNCTION{=>{a,b}, =>{Cons{a},Cons{b}}} .

    op map : -> =>{=>{a,b}, =>{Cons{a},Cons{b}}} .

    var f   : =>{a,b} .
    var a   : a$Elt .
    var as  : Cons{a} .
    eq map f Nil = Nil .
    eq map f (a :| as) = f a :| map f as .
endfm
)

(
fmod INSTANCE-FOLDABLE-CONS{a :: TRIV, b :: TRIV} is
    extending FUNCTION{=>{b, =>{a,b}}, =>{b, =>{Cons{a}, b}}} .

    op foldl : -> =>{=>{b,=>{a,b}}, =>{b, =>{Cons{a}, b}}} .
```

```
    var f   : =>{b, =>{a,b}} .
    var b   : b$Elt .
    var a   : a$Elt .
    var as  : Cons{a} .

    eq foldl f b Nil = b .
    eq foldl f b (a :| as) = foldl f (f b a) as .
endfm
)
```

From the instance and typeclass declarations above, we would like to generate this Full Maude code. The equational definitions of the `map` and `foldl` functions look nearly identical to the Haskell definitions above. To make sure that the appropriate sort-checking will be used, various `FUNCTION{X,Y}` instances must be included into this module.

To achieve the higher-order functionality here, all we have to do is use algebra. The sort-checking using `TRIV` views ensures that our terms are well-formed. Additionally, we get partial application of functions for free (as shown below).

Testing

Here is an example module which would use this higher-order functionality. We've provided it for demonstration purposes.

```
(
fmod TESTING is
    extending INSTANCE-MAPPABLE-CONS{Nat, Nat} .
    extending INSTANCE-MAPPABLE-CONS{Nat, Bool} .
    extending INSTANCE-FOLDABLE-CONS{Nat, Nat} .
    extending INSTANCE-FOLDABLE-CONS{Bool, Bool} .
    extending INSTANCE-MAPPABLE-MAYBE{Nat, Bool} .
    protecting FUNCTION-ID{Nat} .
    protecting FUNCTION-ID{Bool} .
    protecting FUNCTION-COMP{Nat,Nat,Nat} .
    protecting FUNCTION-COMP{Nat,Nat,Bool} .
    protecting FUNCTION-COMP{Nat,Bool,Bool} .
    protecting FUNCTION-COMP{Cons{Nat},Cons{Nat},Cons{Bool}} .

    vars N M : Nat .

    --- some constants (combinator-style functions) to play with
    op aanndd : -> =>{Bool, =>{Bool,Bool}} .
    eq aanndd true true     = true .
    eq aanndd true false    = false .
    eq aanndd false true    = false .
    eq aanndd false false   = false .

    op double : -> =>{Nat,Nat} .
    eq double N = 2 * N .

    op + : -> =>{Nat, =>{Nat,Nat}} .
    eq + N M = N + M .
```

```
    op even : -> =>{Nat,Bool} .
    eq even 0       = true .
    eq even 1       = false .
    eq even s(s(N)) = even N .

    op odd : -> =>{Nat,Bool} .
    eq odd N = not (even N) .

    --- some constants (data)
    op list1 : -> Cons{Nat} .
    eq list1 = 3 :| 5 :| 8 :| 2 :| 19 :| 20 :| Nil .

    op list2 : -> Cons{Nat} .
    eq list2 = 16 :| 100 :| 0 :| 3 :| 9 :| 19 :| 22 :| 101 :| Nil .
endfm
)
```

At the top of the module we include all of the instances we need just to make the sort-checking and function application of the `TESTING` module work. Ideally, a user would not have to import these manually, our tool would infer which modules need to be protected based on what the user writes. For example, if `map even` is used over a list, then we would like to infer that `INSTANCE-MAPPABLE-CONS{Nat, Bool}` should be included.

Further, one can define functions both algebraically and functionally. For example, `aanndd` is defined in a functional way (as it would appear in Haskell), explicitly specifying the value of `aanndd B1 B2` for all `B1` and `B2`. On the other hand, `double` and `+` are defined purely algebraically, using Maude's builtin `*` and `+` operators respectively.

The `even` and `odd` functions are a bit more interesting. `even` is defined purely functionally for the base-cases, but for the recursive case it has an algebraic term within a functional term. `odd` is the other way around - it is defined with a functional term (`even N`) inside an algebraic term (`not _`). This demonstrates how algebraic and higher-order functional definitions can be combined freely, leading to compact specifications.

```
--- map over Maybe type
--- -------------------
(reduce map even Nothing .)
    --- produces: Nothing

(reduce map odd (Just 3) .)
    --- produces: Just true

--- map over Cons type
--- ------------------
(reduce map odd list1 .)
    --- produces: true :| true :| false :| false :| true :| false :| Nil

(reduce map even list2 . )
    --- produces: true :| true :| true :| false :| false :| false :| true :| false :| Nil

--- function composition
--- --------------------
(reduce map (even . double) list1 . )
    --- produces: true :| true :| true :| true :| true :| true :| Nil

--- foldl numeric over Cons type
```

```
--- ----------------------------
(reduce foldl (+) 0 list1 .)
    --- produces: 57

--- foldl over Cons type and function composition, using `$` precedence operator
--- ------------------------------------------------------------------------------
(reduce foldl aanndd true $ map (id . even . id . double . id) list1 .)
    --- produces: true

--- map partially applied function over Cons type
--- ---------------------------------------
(reduce map (+ 3) list1 .)
    --- produces: 6 :| 8 :| 11 :| 5 :| 22 :| 23 :| Nil

--- composing two map examples with `$` precedence operator
--- -----------------------------------------------------
(reduce map even . map (+ 3) $ list1 .)
    --- produces: true :| true :| false :| false :| true :| false :| Nil
```

Above are a series of examples using code from the `TESTING` module. The first few examples are basic examples using `map` over `Maybe` and `Cons` datatypes. The next couple of examples show the use of function composition and basic uses of `foldl`. Note in the seventh example, `id` is used multiple times, and Maude's sort checker is able to infer when it is the identity function over `Nat` and when it is the identity function over `Bool`.

The next example is another basic use of `map` over a list, though note the function used is `(+ 3)`. Here, with no additional work, Maude gives us partial application. Unfortunately, a partially applied function like `map even` can be `=>{Maybe{Nat},Maybe{Bool}}` or `=>{Cons{Nat},Cons{Bool}}`, so currently we are unable to infer a generic sort for such partially applied functions without additional information. The last example shows function composition along with partial application, as well as another use of the `$` precedence operator. Here, since a list is used, Maude is able to infer the correct sort for the partially applied functions `map even`, `map (+ 3)`, and their composition.

Future Work

One nice thing we would like to do would be to actually generate much of the above Maude code using a Full Maude parser. This would allow users to write higher order functions in Haskell, and immediately see how that can be translated to Maude. One could then use some Maude-specific functionality, such as the ITP, to prove interesting properties about his or her code. Additionally, we can achieve slightly more compact representations of purely functional code using Haskell as input, then use it directly in other Maude modules.

Another interesting functionality would be adding support for translating Haskell's lambda abstraction. This is possible using one of the various `LAMBDA-2-CL` compilers discussed in class combined with some type inference to determine which instances of `FUNCTION{X,Y}` to include. A type inference algorithm would be useful in a more general sense though - it would allow for people to not have to write the includes at the top of a module where they want to use first/higher-order functional programming. Instead, using Full Maude, we could scan the module for uses of functional programming and infer the correct includes to use.

Finally, we could add support for more general sort inference when using partial application. With this, `map even` could initially be inferred with a more general sort such as `=>{f{a}, f{b}}`, then when instantiated with a list could be converted to the sort `=>{Cons{a}, Cons{b}}`. Currently, as previously mentioned, Maude's parser will not accept `map even` or similar partially applied functions if the sort is ambiguous (as

the two sorts it would infer for `Cons` and `Maybe` are in disconnected components). This would enable "true typeclass" support, as Haskell has.

References

[1] V. Breazu-Tannen, "Combining Algebra and Higher-Order Types." 1988.