

Syntax: Object Algebras

April 2, 2015

Notations

$N, T, \overline{N}, \overline{T}$: name, type, list of names, list of types.

$\&(\overline{T})$: intersecting a list of types.

$(,)(\overline{E})$: merging a list of expressions.

$[A/B]E$: substituting A for B in expression E .

1 Object Algebra Interface

1.1 Inheritance \times

1.1.1 Template

```
BEFORE:  sig  $N_{AI}[\overline{T_{AI}}]$  where  $\overline{N_{CS} : T_{CS}}$ ;
```

```
AFTER:   type  $N_{AI}[\overline{T_{AI}}]$  =  $\{\overline{N_{CS} : T_{CS}}\}$ ;
```

1.1.2 Example: ExpAlg[E]

```
BEFORE:  sig ExpAlg[E] where  
          lit : Int -> E,  
          add : E -> E -> E;
```

```
AFTER:   type ExpAlg[E] = {  
          lit : Int -> E,  
          add : E -> E -> E  
        };
```

1.2 Inheritance \surd

1.2.1 Template

```
BEFORE:  sig  $N_{AI}[\overline{T_{AI}}]$  extends  $\overline{N_{AI_2}[\overline{T_{AI_2}}]}$  where  $\overline{N_{CS} : T_{CS}}$ ;
```

```
AFTER:   type  $N_{AI}[\overline{T_{AI}}]$  =  $\&(\overline{N_{AI_2}[\overline{T_{AI_2}}]}) \ \& \ \{\overline{N_{CS} : T_{CS}}\}$ ;
```

NB. How to avoid multiple inheritance from the same interface? Throw an exception if two records with same labels are combined with an “&”?

1.2.2 Example: StatAlg[E, S]

BEFORE: **sig** StatAlg[E, S] **extends** ExpAlg[E] **where**
 seq : S -> S -> S,
 asn : String -> E -> S;

AFTER: **type** StatAlg[E, S] = (ExpAlg[E]) & {
 seq : S -> S -> S,
 asn : String -> E -> S
 };

2 Object Algebra

2.1 Inheritance \times

2.1.1 Template

BEFORE: **algebra** N_A **implements** $\overline{N_{AI}[T_A]}$ **where** $\overline{t@}(N_{CS} \ \overline{x}) = \overline{E}$;

AFTER: **let** $N_A = [\overline{T_A}/\overline{T_{AI}}]\{\overline{N_{CS}} = \lambda(\overline{x} : \overline{T_{CS}}). \{t = E\}\}$;

2.1.2 Example: EvalExpAlg

BEFORE: **type** IEval = { eval : Int };
algebra EvalExpAlg **implements** ExpAlg[IEval] **where**
 eval@(lit x) = x,
 eval@(add x y) = x.eval + y.eval;

AFTER: **type** IEval = { eval : Int };
let EvalExpAlg = {
 lit = \ (x : Int) -> { eval = x },
 add = \ (x : IEval) -> \ (y : IEval) -> { eval = x.eval + y.eval }
 };

2.2 Inheritance \surd

2.2.1 Template

BEFORE: **algebra** N_A **extends** $\overline{N_{A_2}}$ **implements** $\overline{N_{AI}[T_A]}$ **where** $\overline{t@}(N_{CS} \ \overline{x}) = \overline{E}$;

AFTER: **let** $N_A = ((,.) (\overline{N_{A_2}})) \ , \ , [\overline{T_A}/\overline{T_{AI}}]\{\overline{N_{CS}} = \lambda(\overline{x} : \overline{T_{CS}}). \{t = E\}\}$;

NB. The same for multiple inheritance.

2.2.2 Example: PrintStatAlg

BEFORE: **type** IPrint = { print : String };
algebra PrintExpAlg **implements** ExpAlg[IPrint] **where**
 print@(lit x) = "\{x}",
 print@(add x y) = "\{x.print} + \{y.print}";
algebra PrintStatAlg **extends** PrintExpAlg **implements** StatAlg[IPrint, IPrint] **where**
 print@(seq x y) = "\{x.print} || \{y.print}",
 print@(asn x y) = "\{x} = \{y.print}";

```

AFTER:  type IPrint = { print : String };
        let PrintExpAlg = {
          lit = \ (x : Int) -> { print = "\{x}" },
          add = \ (x : IPrint) -> \ (y : IPrint) -> { print = "\{x.print} + \{y.print}" };
        };
        let PrintStatAlg = PrintExpAlg ,, {
          seq = \ (x : IPrint) -> \ (y : IPrint) -> { print = "\{x.print} || \{y.print}" },
          asn = \ (x : String) -> \ (y : IPrint) -> { print = "\{x} = \{y.print}" };
        };

```

3 Datatype

3.1 Template

```

BEFORE:  data  $N_D[\overline{T_D}]$  from  $N_{AI}[\overline{T_{AI}}].S$ ;

```

```

AFTER:   type  $N_D[\overline{T_D}]$  = { accept : forall  $(\overline{T_{AI}} \setminus \overline{T_D}). N_{AI}[\overline{T_{AI}}] \rightarrow S$  };

```

NB. Usually $\overline{T_{AI}} \setminus \overline{T_D} = S$.

3.2 Example: List[A]

```

BEFORE:  sig ListAlg[A, L] where
          nil  : L,
          cons : A -> L -> L;
          data List[A] from ListAlg[A, L].L;

```

```

AFTER:   type ListAlg[A, L] = {
          nil  : L,
          cons : A -> L -> L
        };
        type List[A] = { accept : forall L. ListAlg[A, L] -> L };

```

4 Creating a Structure

4.1 Simple Structures

4.1.1 Template

```

BEFORE:  build  $N_S$  :  $N_D[\overline{T}] = E$ ;

```

```

AFTER:   let  $N_S$  = { accept =  $\Lambda(\overline{T_{AI}} \setminus \overline{T_D}). \lambda(\text{alg} : N_{AI}[\overline{(\overline{T}/\overline{T_D})T_{AI}}]). [\text{alg.N}_{CS}/N_{CS}]E$  };

```

NB. Potentially there could be name conflicts with `alg`. Also names of functions and constructors could probably overlap.

4.1.2 Example: Exp and List[Int]

```

BEFORE:  data Exp from ExpAlg[E].E;
          data List[A] from ListAlg[A, L].L;
          build exp : Exp = add (lit 3) (lit 5);
          build lst : List[Int] = cons 3 (cons 5 nil);

```

```

AFTER:  type Exp = { accept : forall E. ExpAlg[E] -> E };
type List[A] = { accept : forall L. ListAlg[A, L] -> L };
let exp = { accept = /\E -> \(alg : ExpAlg[E]) -> alg.add (alg.lit 3) (alg.lit 5) };
let lst = { accept = /\L -> \(alg : ListAlg[Int, L]) -> alg.cons 3 (alg.cons 5 alg.nil) };

```

4.2 Complicated Structures Created by Functions

4.2.1 Template

```

BEFORE:  build  $N_S$  ( $\bar{x} : \overline{T_1 \neq N_D[\bar{T}]}$ ) ( $\bar{y} : \overline{T_2 = N_D[\bar{T}]}$ ) :  $N_D[\bar{T}] = E$ ;

```

```

AFTER:  let  $N_S = \lambda(\bar{x} : \overline{T_1}). \lambda(\bar{y} : \overline{T_2}). \{$ 
       $\text{accept} = \Lambda(\overline{T_{AI}} \backslash \overline{T_D}). \lambda(\text{alg} : N_{AI}[\overline{[\bar{T}/\bar{T_D}]T_{AI}}]).$ 
       $\overline{[alg.N_{CS}/N_{CS}][(\overline{y.accept[\overline{T_{AI}} \backslash \overline{T_D}] alg})/\bar{y}]E}$ 
     $\};$ 

```

4.2.2 Example: myAdd and myCons

```

BEFORE:  build myAdd (e1 : Exp) (e2 : Exp) : Exp = add e1 e2;
build myCons (x : Int) (xs : List[Int]) : List[Int] = cons x xs;

```

```

AFTER:  let myAdd = \(e1 : Exp) -> \(e2 : Exp) -> {
      accept = /\E -> \(alg : ExpAlg[E]) -> alg.add (e1.accept[E] alg) (e2.accept[E] alg)
    };
let myCons = \(x : Int) -> \(xs : List[Int]) -> {
      accept = /\L -> \(alg : ListAlg[Int, L]) -> alg.cons x (xs.accept[L] alg)
    };

```

NB. Extension 1: What if there are BigLambdas in those functions?

NB. Extension 2: Does it make sense if there are two different instantiations from one datatype in a function? Namely $N_D[\bar{T}_1]$ and $N_D[\bar{T}_2]$?

NB. Extension 3: Recursive ones?

NB. IMPORTANT: Why don't we generate "add : Exp -> Exp -> Exp" and "cons : A -> List[A] -> List [A]" automatically for global use? My intuition is that for these two examples it's easy; however there could be non-trivial ones. For instance, a new constructor for ListAlg can be "f : L -> A", in which case some structures like "cons (f nil) nil" also make sense, but how to design a template for "f : List[Int] -> Int"? Instead it's easier to have "alg.cons (alg.f alg.nil) alg.nil".

5 Instantiation

5.1 Template

5.2 Example: