Syntax: Object Algebras

April 2, 2015

1 Object Algebra Interface

1.1 Inheritance $\times$

1.1.1 Template

```
BEFORE:  sig N_AI[T_AI] where N_CS : T_CS;
```

```
AFTER:   type N_AI[T_AI] = {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[T_AI] extends N_AI2[T_AI2] where N_CS : T_CS;
```

```
AFTER:   type N_AI[T_AI] = &(N_AI2[T_AI2]) & {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 ×

2.1.1 Template

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

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 √

2.2.1 Template

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

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

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[T_D] from N_AI[T_AI].S;
```

```
AFTER:   type N_D[T_D] = { accept : forall (T_AI \ T_D). N_AI[T_AI] -> S };
```

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

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[T] = E;
```

```
AFTER:   let N_S = { accept = Λ(T_AI \ T_D). λ(alg : N_AI[[T/T_D]T_AI]). [alg.N_CS/N_CS]E };
```

NB. Potentially there could be name conflicts with `alg`.

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 from Functions

4.2.1 Template

4.2.2 Example: `Exp` and `List[Int]`

5 Instantiation

5.1 Template

5.2 Example: