Implicit Monads in Attribute Grammars

Dawn Michaelson and Eric Van Wyk

University of Minnesota

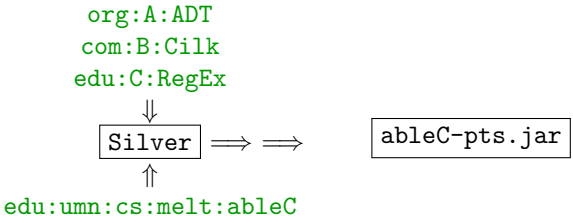
```
typedef datatype Tree Tree;
datatype Tree {
  Fork ( Tree*, Tree*, const char* );
  Leaf ( const char* );
};

cilk int count_matches (Tree *t) {
  match ( t ) {
    Fork(t1,t2,str): {
      int res_t, res_t1, res_t2;
      spawn res_t1 = count_matches( t1 );
      spawn res_t2 = count_matches( t2 );
      res_t = ( str =~ /foo[1-9]+/ ) ? 1 : 0;
      sync;
      cilk return res_t1 + res_t2 + res_t ;
    } ;
    Leaf(str): { return ( str =~ /foo[1-9]+/ ) ? 1 : 0; } ;
  }
}
```

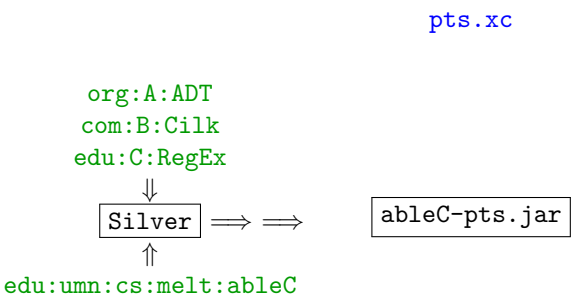
ableC- extensible specification of C11

```
org:A:ADT
com:B:Cilk
edu:C:RegEx
```

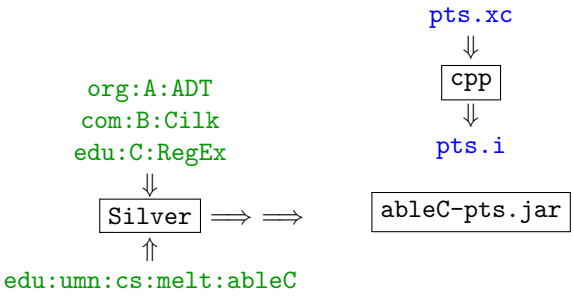
ableC- extensible specification of C11



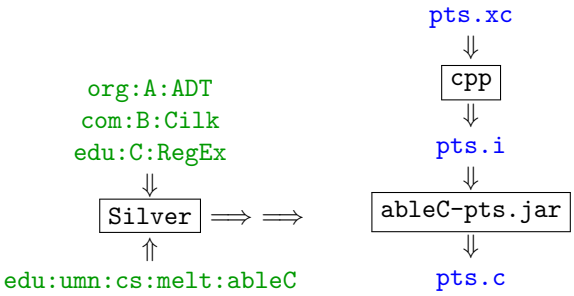
ableC- extensible specification of C11



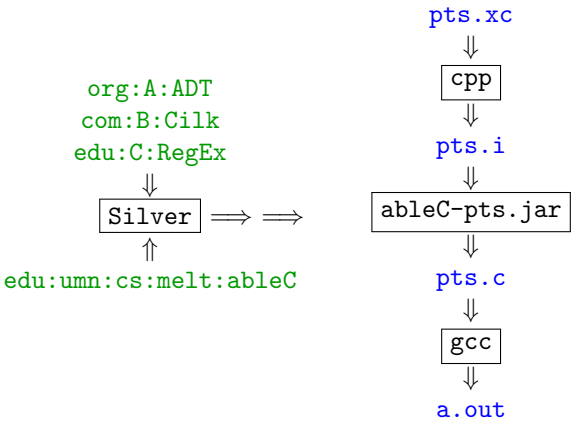
ableC- extensible specification of C11



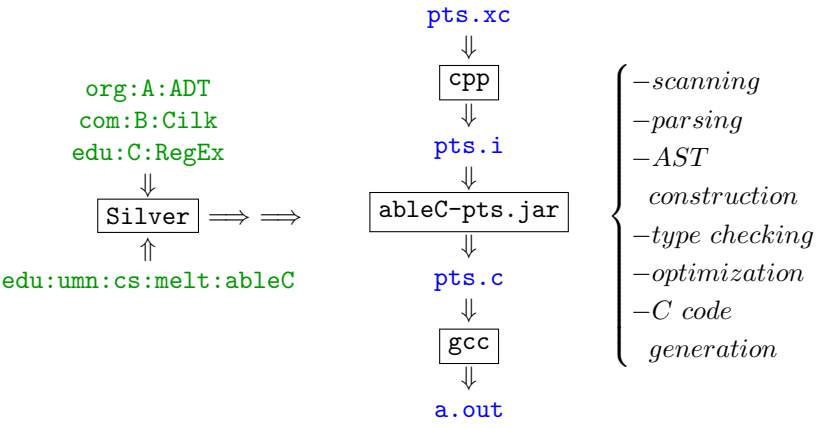
ableC- extensible specification of C11



ableC- extensible specification of C11



ableC- extensible specification of C11



Motivation

In reasoning about language specifications, we want

- the simplicity of structural operational semantics
- as in
  - concise inference rules, these yield, *e.g* typing derivations
  - rules need not be deterministic, *e.g.* evaluation

and

- the utility of attribute grammars in generating working language processing tools:
  - type checking
  - specifying error messages
  - pretty-printers
  - translation to other languages
  - etc.

Silver Attribute Grammars

TAPL, Pierce

Well-defined attribute grammars define attributes for **all** terms.

SOS derivations exist only for well-typed terms.

Silver Attribute Grammars

TAPL, Pierce

```
nonterminal Exp
```

$e ::= \lambda x : ty.e$

```
lambda:  a:Exp ::= x:Name ty:Typ b:Exp
```

Well-defined attribute grammars define attributes for **all** terms.

SOS derivations exist only for well-typed terms.

Silver Attribute Grammars

```
nonterminal Exp
attr typ:Maybe<Typ> occurs on Exp

lambda:  a:Exp ::= x:Name ty:Typ b:Exp
```

TAPL, Pierce

$\Gamma \vdash e : ty$

$e ::= \lambda x : ty.e$

Well-defined attribute grammars define attributes for **all** terms.

SOS derivations exist only for well-typed terms.

Silver Attribute Grammars

```
nonterminal Typ
int:   t:Typ ::= ϵ
arrow: t:Typ ::= in:Typ out:Typ

nonterminal Exp
attr typ:Maybe<Typ> occurs on Exp


lambda:  a:Exp ::= x:Name ty:Typ b:Exp
```

TAPL, Pierce

$$ty ::= int \mid ty \to ty$$

$$\Gamma \vdash e : ty$$

$$e ::= \lambda x : ty.e$$

Well-defined attribute grammars define attributes for **all** terms.

SOS derivations exist only for well-typed terms.

Silver Attribute Grammars

```
nonterminal Typ
int:   t:Typ ::= ϵ
arrow: t:Typ ::= in:Typ out:Typ

nonterminal Exp
attr typ:Maybe<Typ> occurs on Exp


lambda:  a:Exp ::= x:Name ty:Typ b:Exp
```

TAPL, Pierce

$$ty ::= int \mid ty \to ty$$

$$\Gamma \vdash e : ty$$

$$e ::= \lambda x : ty.e$$

$$\frac{\Gamma, x : ty \vdash b : bt}{\Gamma \vdash \lambda x : ty.b : ty \to bt}$$

Well-defined attribute grammars define attributes for **all** terms.

SOS derivations exist only for well-typed terms.

Silver Attribute Grammars

```
nonterminal Typ
int:   t:Typ ::= ϵ
arrow: t:Typ ::= in:Typ out:Typ

nonterminal Exp
attr typ:Maybe<Typ> occurs on Exp
attr gamma:  Context occurs on Exp

lambda:  a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
```

TAPL, Pierce

$$ty ::= int \mid ty \to ty$$

$$\Gamma \vdash e : ty$$

$$e ::= \lambda x : ty.e$$

$$\frac{\Gamma, x : ty \vdash b : bt}{\Gamma \vdash \lambda x : ty.b : ty \to bt}$$

Well-defined attribute grammars define attributes for **all** terms.

SOS derivations exist only for well-typed terms.

Silver Attribute Grammars

```
nonterminal Typ
int:   t:Typ ::= ϵ
arrow: t:Typ ::= in:Typ out:Typ

nonterminal Exp
attr typ:Maybe<Typ> occurs on Exp
attr gamma:  Context occurs on Exp

lambda:  a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
  a.typ = case b.typ of


}
```

TAPL, Pierce

$$ty ::= int \mid ty \to ty$$

$$\Gamma \vdash e : ty$$

$$e ::= \lambda x : ty.e$$

$$\frac{\Gamma, x : ty \vdash b : bt}{\Gamma \vdash \lambda x : ty.b : ty \to bt}$$

Well-defined attribute grammars define attributes for **all** terms.

SOS derivations exist only for well-typed terms.

Silver Attribute Grammars

```
nonterminal Typ
int:   t:Typ ::= ϵ
arrow: t:Typ ::= in:Typ out:Typ

nonterminal Exp
attr typ:Maybe<Typ> occurs on Exp
attr gamma:  Context occurs on Exp

lambda:  a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
  a.typ = case b.typ of
    | Just bt -> Just (arrow (ty, bt));
    | Nothing -> Nothing
}
```

TAPL, Pierce

$$ty ::= int \mid ty \to ty$$

$$\Gamma \vdash e : ty$$

$$e ::= \lambda x : ty.e$$

$$\frac{\Gamma, x : ty \vdash b : bt}{\Gamma \vdash \lambda x : ty.b : ty \to bt}$$

Well-defined attribute grammars define attributes for **all** terms.

SOS derivations exist only for well-typed terms.

```
attr typ:Maybe<Typ> occurs on Exp;

const: e:Exp ::= i:Int
{ e.typ = Just(int());
}
```

$$\Gamma \vdash i : int$$

```
attr typ:Maybe<Typ> occurs on Exp;

app: e:Exp ::= f:Exp a:Exp
{ e.typ = case f.typ of
    | Just(arrow(in,out)) -> (case a.typ of
          | Just (at) when in = at -> Just(out)
          | Nothing() -> Nothing() )
    | Just(_) -> Nothing()
    | Nothing() -> Nothing()
```

$$\frac{\Gamma \vdash f : t_{in} \to t_{out} \quad \Gamma \vdash a : t_{in}}{\Gamma \vdash f\ a : t_{out}}$$

```
attr typ:Maybe<Typ> occurs on Exp;

app: e:Exp ::= f:Exp a:Exp
{ e.typ = case f.typ of
    | Just(arrow(in,out)) -> (case a.typ of
          | Just (at) when in = at -> Just(out)
          | Nothing() -> Nothing() )
    | Just(_) -> Nothing()
    | Nothing() -> Nothing()

  e.errs =  = f.errs ++ a.errs ++
    case f.typ of
    | Just(arrow(in,out)) -> (case a.typ of
          | Just (at) when in = at -> []
          | Nothing() -> ["Type mismatch on application"] )
    | Just(_) -> ["Must be a function type"]
    | Nothing() -> []
}
```

$$\frac{\Gamma \vdash f : t_{in} \to t_{out} \qquad \Gamma \vdash a : t_{in}}{\Gamma \vdash f\ a : t_{out}}$$

Monads

- We assume three operations for a monad:
  - Return: T → M<T>
  - Bind: M<T> → (T → M<S>) → M<S>
    - Takes a monad and a function to apply to its inner value
    - Written as m >>= f
  - Fail: _ → M<T>
    - Represents a computation which does not succeed

Common Monads

- Maybe
  - Return x = Just x
  - m >>= f = case m of
    | Just t ⇒ f t
    | Nothing ⇒ Nothing
  - Fail _ = Nothing
- List
  - Return x = [x]
  - m >>= f = case m of
    | h::t ⇒ (f h) ++ (t >>= f)
    | [ ] ⇒ [ ]
  - Fail _ = [ ]
- Either a b = Left a | Right b
  - Return x = Left x
  - m >>= f = case m of
    | Left a ⇒ f a
    | Right b ⇒ Right b ++ (t >>= f)
  - Fail x = Right x

```
attr typ : Maybe<Typ>

const: e:Exp ::= i::Int
{ e.typ = Just( int() );
}
```

becomes

```
const: e:Exp ::= i::Int
{ e.typ = Return( int() );
}
```

```
attr typ : Maybe<Typ>

lambda: a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma};
  a.typ = case b.typ of
          | Just bt -> Just (arrow (ty, bt));
          | Nothing -> Nothing ;
}
```

becomes

```
lambda: a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
  a.typ = b.typ >>= \ bt . Return(arrow (ty, bt));
}
```

```
attr typ : Maybe<Typ>

app: e:Exp ::= f:Exp a:Exp
{ e.typ = case f.typ of
    | Just(arrow(in,out)) -> (case a.typ of
          | Just (at) when in = at -> Just(out)
          | Nothing() -> Nothing() )
    | Just(_) -> Nothing()
    | Nothing() -> Nothing()     }
```

becomes

```
app: e:Exp ::= f:Exp a:Exp
{ e.typ = f.typ >>= \ ft . case ft of
            | arrow(in,out) ->
                (a.typ >>= \ at . if in = at then Return(out)
                                  else Fail() )
            | _ -> Fail ()    }
```

And we haven't fixed the problem of duplication with defining `errs`.

```
attr typ : Maybe<Typ>
```

with separate equations for `typ` and `errs`

or it becomes

```
attr typ : Either<Typ, List<String>>
```

```
attr typ : Maybe<Typ>
```

with separate equations for typ and errs

or it becomes

```
attr typ : Either<Typ, List<String>>

app: e:Exp ::= f:Exp a:Exp
{ e.typ = f.typ >>= \ ft . case ft of
           | arrow(in,out) ->
               (a.typ >>= \ at . if in = at then Return(out)
                                 else Fail( ["Type mismatch on application"] ) )
           | _ -> Fail ( ["Must be a function type"] )
}
```

Implicit Monads

- Can we use monads "behind the scenes" and generate the monadic code above from something closer to the SOS specifications?
- Many attribute values flow up and down the tree within their own attribute (e.g. the type attribute is often computed based on the children's types), so users should rarely need to look within the abstraction to deal directly with the monad.
- This approach should ideally allow the declared types of attributes to be changed between different monads and still use the same code.
- Can we transform "nice specifications" into well-defined attribute grammar expressions? "nice" $\rightsquigarrow$ "not nice"

```
attr typ : Maybe<Typ>

const: e:Exp ::= i::Int
{ e.typ = int();
}
```

$$\Gamma \vdash i : int$$

```
attr typ : Maybe<Typ>

lambda: a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma};
  a.typ = arrow (ty, b.typ)
}
```

$$\frac{\Gamma, x : ty \vdash b : bt}{\Gamma \vdash \lambda x : ty.b : ty \to bt}$$

```
attr typ : Either<Typ, List<String>>

app: e:Exp ::= f:Exp a:Exp
{ e.typ =
    case f.typ of
    | arrow(in,out) when a.ty = in -> out
    | arrow(_,_) -> Fail( ["Type mismatch in application"] )
    | _ -> Fail( ["Applied term must have a function type"] )
}
```

$$\frac{\Gamma \vdash f : t_{in} \to t_{out} \quad \Gamma \vdash a : t_{in}}{\Gamma \vdash f\ a : t_{out}}$$

Judgments

Whether (and to what) a term translates depends on the types of subterms, so we present our rules with typing derivations.

$\Gamma \vdash$ e : T	Expression e has type T
$\Gamma \Vdash$ cs : T, S	Clauses cs match type T and evaluate to type S
$\Gamma \sim$ cs complete	Clauses cs match all values of their match type
$\Gamma \vdash$ a :: T	Attribute a is declared with type T
$\Gamma \vdash$ a @ T	Attribute a occurs on type T
$\Gamma \vdash$ lhs = rhs OK	Equation lhs = rhs does not cause type errors

If a is rewritten to b, we write a $\rightsquigarrow$ b in the typing rule.

- $\Gamma \vdash$ a $\rightsquigarrow$ b : T

Equations for synthesized attributes

$$\frac{\Gamma \vdash \mathsf{lhs} : \mathsf{T} \qquad \Gamma \vdash \mathsf{e} : \mathsf{T}}{\Gamma \vdash \mathsf{lhs} = \mathsf{e} \;\; \mathsf{OK}}$$

Equations for synthesized attributes

$$\frac{\Gamma \vdash \text{lhs} : \text{T} \qquad \Gamma \vdash \text{e} : \text{T}}{\Gamma \vdash \text{lhs} = \text{e} \;\; \text{OK}}$$

$$\frac{\Gamma \vdash \text{lhs} : \text{M<T>} \qquad \Gamma \vdash \text{e} : \text{T}}{\Gamma \vdash \text{lhs} = \text{e} \rightsquigarrow \text{lhs} = \text{Return(e)} \;\; \text{OK}}$$

Equations for synthesized attributes

$$\frac{\Gamma \vdash \text{lhs} : \text{T} \qquad \Gamma \vdash \text{e} : \text{T}}{\Gamma \vdash \text{lhs} = \text{e} \;\; \text{OK}}$$

$$\frac{\Gamma \vdash \text{lhs} : \text{M<T>} \qquad \Gamma \vdash \text{e} : \text{T}}{\Gamma \vdash \text{lhs} = \text{e} \rightsquigarrow \text{lhs} = \text{Return(e)} \;\; \text{OK}}$$

```
attr typ : Maybe<Typ>

const: e:Exp ::= i::Int
{ e.typ = int();
}
```

becomes

```
attr typ : Maybe<Typ>

const: e:Exp ::= i::Int
{ e.typ = Return(int());
}
```

Binary Operators

$$\frac{\Gamma \vdash \mathsf{a} : \mathsf{Int} \qquad \Gamma \vdash \mathsf{b} : \mathsf{Int}}{\Gamma \vdash \mathsf{a} + \mathsf{b} : \mathsf{Int}}$$

Binary Operators

$$\frac{\Gamma \vdash \mathsf{a} : \mathsf{Int} \qquad \Gamma \vdash \mathsf{b} : \mathsf{Int}}{\Gamma \vdash \mathsf{a} + \mathsf{b} : \mathsf{Int}}$$

$$\frac{\Gamma \vdash \mathsf{a} : \mathsf{M{<}Int{>}} \qquad \Gamma \vdash \mathsf{b} : \mathsf{Int}}{\Gamma \vdash \mathsf{a} + \mathsf{b} \rightsquigarrow \mathsf{a} >>= (\lambda \mathsf{x{:}Int.\ Return(x} + \mathsf{b})) : \mathsf{M{<}Int{>}}}$$

Binary Operators

$$\frac{\Gamma \vdash \mathsf{a} : \mathsf{Int} \qquad \Gamma \vdash \mathsf{b} : \mathsf{Int}}{\Gamma \vdash \mathsf{a} + \mathsf{b} : \mathsf{Int}}$$

$$\frac{\Gamma \vdash \mathsf{a} : \mathsf{M{<}Int{>}} \qquad \Gamma \vdash \mathsf{b} : \mathsf{Int}}{\Gamma \vdash \mathsf{a} + \mathsf{b} \rightsquigarrow \mathsf{a} >>= (\lambda \mathsf{x{:}Int.\ Return(x} + \mathsf{b})) : \mathsf{M{<}Int{>}}}$$

$$\frac{\Gamma \vdash \mathsf{a} : \mathsf{M{<}Int{>}} \qquad \Gamma \vdash \mathsf{b} : \mathsf{M{<}Int{>}}}{\Gamma \vdash \mathsf{a} + \mathsf{b} \rightsquigarrow \mathsf{a} >>= (\lambda \mathsf{x{:}Int.\ b} >>= (\lambda \mathsf{y{:}Int.\ Return(x} + \mathsf{y}))) : \mathsf{M{<}Int{>}}}$$

Tree Construction and Function Calls: Basic Rule

$$\frac{\Gamma \vdash \mathsf{p} : (\mathsf{T}_1, \mathsf{T}_2, ..., \mathsf{T}_n) \to \mathsf{T} \quad \Gamma \vdash \mathsf{x}_i : \mathsf{T}_i,\ i = 1...n}{\Gamma \vdash \mathsf{p}\ (\mathsf{x}_1, \mathsf{x}_2, \ ...\ \mathsf{x}_n) : \mathsf{T}}$$

The rules in this section apply to both function calls and tree construction, as both are typed in the same way.

Monadic Arguments: Only Two Arguments, Only One Monadic

$$\frac{\Gamma \vdash \mathsf{p} : (\mathsf{T}_1, \mathsf{T}_2) \to \mathsf{T} \qquad \Gamma \vdash \mathsf{x}_1 : \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{x}_2 : \mathsf{M}\langle\mathsf{T}_2\rangle}{\Gamma \vdash \mathsf{p}\ (\mathsf{x}_1, \mathsf{x}_2) \rightsquigarrow \mathsf{x}_2 \gg\!= (\lambda \mathsf{y}{:}\mathsf{T}_2.\ \mathsf{Return}(\mathsf{f}(\mathsf{x}_1, \mathsf{y}_2))) : \mathsf{M}\langle\mathsf{T}\rangle}$$

Monadic Arguments: Only Two Arguments, Only One Monadic

$$\frac{\Gamma \vdash \mathsf{p} : (\mathsf{T}_1, \mathsf{T}_2) \to \mathsf{T} \qquad \Gamma \vdash \mathsf{x}_1 : \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{x}_2 : \mathsf{M}\langle\mathsf{T}_2\rangle}{\Gamma \vdash \mathsf{p}\ (\mathsf{x}_1, \mathsf{x}_2) \rightsquigarrow \mathsf{x}_2 >\!\!>\!\!= (\lambda \mathsf{y}{:}\mathsf{T}_2.\ \mathsf{Return}(\mathsf{f}(\mathsf{x}_1, \mathsf{y}_2))) : \mathsf{M}\langle\mathsf{T}\rangle}$$

```
attr typ : Maybe<Typ>
lambda: a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
  a.typ = arrow (ty, b.typ)
}
```

Monadic Arguments: Only Two Arguments, Only One Monadic

$$\frac{\Gamma \vdash \mathsf{p} : (\mathsf{T}_1, \mathsf{T}_2) \to \mathsf{T} \qquad \Gamma \vdash \mathsf{x}_1 : \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{x}_2 : \mathsf{M}\langle\mathsf{T}_2\rangle}{\Gamma \vdash \mathsf{p}\ (\mathsf{x}_1, \mathsf{x}_2) \rightsquigarrow \mathsf{x}_2 >>= (\lambda \mathsf{y}{:}\mathsf{T}_2.\ \mathsf{Return}(\mathsf{f}(\mathsf{x}_1, \mathsf{y}_2))) : \mathsf{M}\langle\mathsf{T}\rangle}$$

```
attr typ : Maybe<Typ>
lambda: a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
  a.typ = arrow (ty, b.typ)
}
```

becomes

```
attr typ : Maybe<Typ>
lambda: a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
  a.typ = b.typ >>= \ bt . Return(arrow (ty, bt))
}
```

Monadic Arguments: Only Two Arguments, Only One Monadic

$$\frac{\Gamma \vdash p : (T_1, T_2) \to T \qquad \Gamma \vdash x_1 : T_1 \qquad \Gamma \vdash x_2 : M\langle T_2\rangle}{\Gamma \vdash p\ (x_1, x_2) \rightsquigarrow x_2 >>= (\lambda y{:}T_2.\ \mathsf{Return}(f(x_1, y_2))) : M\langle T\rangle}$$

```
attr typ : Maybe<Typ>
lambda: a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
  a.typ = arrow (ty, b.typ)
}
```

becomes

```
attr typ : Maybe<Typ>
lambda: a:Exp ::= x:Name ty:Typ b:Exp
{ b.gamma = [(x, ty)] ++ a.gamma;
  a.typ = b.typ >>= \ bt . Return(arrow (ty, bt))
}
```

- In general, we have a bind for each monadic argument, then place their bound variables in the correct places in the function call.
- If T = M<U>, we do not insert the Return() and the type is simply M<U>.

Case Expressions: Non-monadic rules

Case Expression:

$\Gamma \vdash e : T$
$\Gamma \Vdash cs : T, S$
$\Gamma \sim cs$ complete

$\Gamma \vdash$ case e of $|$ cs : S

Clauses:

$\Gamma \vdash A : (T_1, \ldots T_n) \to T$
$\Gamma, x_1{:}T_1, \ldots x_n{:}T_n \vdash e : S$
$\Gamma \Vdash cs : T, S$

$\Gamma \Vdash A(x_1, \ldots x_n) \Rightarrow e \mid cs : T, S$

```
attr typ : Either<Typ, List<String>>

app: e:Exp ::= f:Exp a:Exp
{ e.typ =
    case f.typ of
    | arrow(in,out) when a.ty = in -> out
    | arrow(_,_) -> Fail( ["Type mismatch in application"] )
    | _ -> Fail( ["Applied term must have a function type"] )
}
```

```
attr typ : Either<Typ, List<String>>

app: e:Exp ::= f:Exp a:Exp
{ e.typ =
    case f.typ of
    | arrow(in,out) when a.ty = in -> out
    | arrow(_,_) -> Fail( ["Type mismatch in application"] )
    | _ -> Fail( ["Applied term must have a function type"] )
}
```

becomes

```
app: e:Exp ::= f:Exp a:Exp
{ e.typ = f.typ >>= \ ft . case ft of
          | arrow(in,out) ->
              (a.typ >>= \ at . if in = at then Return(out)
                                else Fail( ["Type mismatch on application"] ) )
          | _ -> Fail ( ["Must be a function type"] )
}
```

Matching a Monad to the Scrutinee Type with Complete Patterns

$\Gamma \vdash$ e : M<T>
$\Gamma \Vdash$ cs : T, S
$\Gamma \sim$ cs complete

$\Gamma \vdash$ case e of | cs $\rightsquigarrow$
e >>= (λx:T. Return(case x of cs)) : M<S>

If S = M<U> for some type U, we leave out the Return() and the return type is simply M<U> to avoid unnecessarily having monads inside of monads.

Matching a Monad to the Inner Type with Incomplete Patterns

$$\frac{\Gamma \vdash \text{e} : \text{M<T>} \quad \Gamma \Vdash \text{cs} : \text{T, S} \quad \neg\,(\Gamma \sim \text{cs complete})}{\Gamma \vdash \text{case e of} \mid \text{cs} \rightsquigarrow \text{e} >>= (\lambda \text{x:T. case x of} \mid \text{returnify(cs)} \mid \_ \Rightarrow \text{Fail()}) : \text{M<S>}}$$

The returnify() function wraps each clause's result in a Return():

- A(...) $\Rightarrow$ e becomes A(...) $\Rightarrow$ Return(e)

We only use returnify() if S $\neq$ M<U> for some type U.

Making Return Types Monad-Consistent

$$\frac{\Gamma \vdash A : (T_1, \ldots\ T_n) \to T \quad \Gamma, x_1{:}T_1, \ldots\ x_n{:}T_n \vdash e : M\langle S\rangle \quad \Gamma \Vdash cs : T, S}{\Gamma \Vdash A(x_1, \ldots\ x_n) \Rightarrow e \mid cs \rightsquigarrow A(x_1, \ldots\ x_n) \Rightarrow e \mid \text{returnify}(cs) : T, M\langle S\rangle}$$

$$\frac{\Gamma \vdash A : (T_1, \ldots\ T_n) \to T \quad \Gamma, x_1{:}T_1, \ldots\ x_n{:}T_n \vdash e : S \quad \Gamma \Vdash cs : T, M\langle S\rangle}{\Gamma \Vdash A(x_1, \ldots\ x_n) \Rightarrow e \mid cs \rightsquigarrow A(x_1, \ldots\ x_n) \Rightarrow \text{Return}(e)\ cs : T, M\langle S\rangle}$$

Completing Incomplete Patterns

$$\frac{\Gamma \vdash e : T \quad \Gamma \Vdash cs : T, S \quad \neg\,(\Gamma \sim cs \text{ complete})}{\Gamma \vdash \text{case } e \text{ of } | \; cs \rightsquigarrow \text{case } e \text{ of } | \; returnify(cs) \; | \; \_ \Rightarrow Fail() : M\langle S\rangle}$$

- This allows us to make a monad when we want one without needing to explicitly state it or use one to build it.
- Since we require complete patterns in our rules, this case expression would not be considered well-typed without this rule.
- We should have an expected type coming down to know which monad to use in this rule.

Attribute Accesses: Rules

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash a :: T \quad \Gamma \vdash a \ @ \ S}{\Gamma \vdash e.a : T}$$

$$\frac{\Gamma \vdash e : M\langle S\rangle \quad \Gamma \vdash a :: T \quad \Gamma \vdash a \ @ \ S}{\Gamma \vdash e.a \rightsquigarrow \quad e >\!\!>\!= (\lambda x{:}S.\ \mathrm{Return}(x.a)) : M\langle T\rangle}$$

If $T = M\langle U\rangle$, we do not insert the Return() and the type is simply $M\langle U\rangle$.

Inherited Attributes on Monadic Trees

- We can't assign inherited attributes directly on a tree that is a monad
- We use a bind to access the tree, let expressions to give it inherited attributes, then access a synthesized attribute on it

synthesized attribute s::M<S>;
⋮
local attribute m::M<T> = e;
$m.i_1 = 1;$
$m.i_2 = 2;$
top.s = m.s;

$m.s \rightsquigarrow$
m >>=
(λt:T.decorate t with
$\{i_1 = 1; i_2 = 2\}$.s)

- The attribute being assigned into (top.s) must have a monadic type as the result of accessing an attribute off a monadic tree must be a monad.

Inherited Attribute Equations for Monadic Trees

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{M}\langle\mathsf{T}\rangle \quad \Gamma \vdash \mathsf{a} :: \mathsf{S} \quad \Gamma \vdash \mathsf{a} \ @ \ \mathsf{T} \quad \Gamma \vdash \mathsf{e} : \mathsf{S}}{\Gamma \vdash \mathsf{t.a} = \mathsf{e} \ \ \mathsf{OK}}$$

$$\mathsf{t.s} \rightsquigarrow \mathsf{t} >\!\!>\!\!= (\lambda \mathsf{x}{:}\mathsf{T}.\ \mathsf{decorate\ x\ with}\ \{\mathsf{a} = \mathsf{e}\}.\mathsf{s})$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{M}\langle\mathsf{T}\rangle \quad \Gamma \vdash \mathsf{a} :: \mathsf{S} \quad \Gamma \vdash \mathsf{a} \ @ \ \mathsf{T} \quad \Gamma \vdash \mathsf{e} : \mathsf{M}\langle\mathsf{S}\rangle}{\Gamma \vdash \mathsf{t.a} = \mathsf{e} \ \ \mathsf{OK}}$$

$$\mathsf{t.s} \rightsquigarrow \mathsf{t} >\!\!>\!\!= (\lambda \mathsf{x}{:}\mathsf{T}.\ \mathsf{e} >\!\!>\!\!= (\lambda \mathsf{y}{:}\mathsf{S}.\ \mathsf{decorate\ x\ with}\ \{\mathsf{a} = \mathsf{y}\}.\mathsf{s}))$$

- These rules allow us to check the types for the equations before translating them into decorate expressions for accesses of synthesized attributes

Monoids

```
syn attr next::List<Exp>;
production or
e:Exp ::= l:Exp r:Exp;
{ e.next = case l, r of
    | const true, _  -> const true
    | _, const true -> const true
    | _, _ -> or ( l.next, r )
    |_, _ -> or (l, r.next)
}
```

To allow nondeterminism, use `mplus` (M<T> → M<T> → M<T>) and `mzero` (M<T>) to combine the different cases that match the input (in addition to wrapping expressions in Return() when the type requires it).

Thanks for your attention.

Questions?

Pointers?

evw@umn.edu