# FAE Datatypes in Caml

```
type fae =
    Num of int
  | Add of fae * fae
  | Sub of fae * fae
  | Id of string
  | Fun of string * fae
  | App of fae * fae

and faeValue =
    NumV of int
  | ClosureV of string * fae * subCache

and subCache =
    MTSub
  | ASub of string * faeValue * subCache
```

# FAE Lookup in Caml

```
exception Failed of string

let rec lookup = function
    (findName, MTSub) -> raise (Failed "free variable")
  | (findName, ASub(name, v, restSc)) ->
      if (name = findName)
      then v
      else lookup(findName, restSc)
```

# FAE Number Operations in Caml

```
let mkNumOp = fun op
  -> function
       (NumV(a), NumV(b)) -> NumV(op a b)
     | _ -> raise (Failed "not numbers")

let numPlus = mkNumOp (fun a b -> a + b)
let numMinus = mkNumOp (fun a b -> a - b)
```

# FAE Interpreter in Caml

```
let rec interp : (fae * subCache -> faeValue ) = function
    (Num(n), sc) -> NumV(n)
  | (Add(l, r), sc) -> numPlus(interp(l, sc),
                               interp(r, sc))
  | (Sub(l, r), sc) -> numMinus(interp(l, sc),
                                interp(r, sc))
  | (Id(name), sc) -> lookup(name, sc)
  | (Fun(param, body), sc) -> ClosureV(param, body, sc)
  | (App(fn, arg), sc) ->
      let funV = interp(fn, sc)
      and argV = interp(arg, sc)
      in match funV with
         ClosureV(param, body, sc) ->
           interp(body, ASub(param, argV, sc))
      | _ -> raise (Failed "not a function")
```

# TFAE Expressions and Types

```
type fae =
    …
  | Fun of string * te * fae
    …

and te =
    NumTE
  | BoolTE
  | ArrowTE of te * te

…

and ty =
    NumT
  | BoolT
  | ArrowT of ty * ty

and typeEnv =
    MTEnv
  | ABind of string * ty * typeEnv
```

# TFAE Interpreter

```
let rec interp : (fae * subCache -> faeValue ) = function
    …
  | (Fun(param, texpr, body), sc) -> ClosureV(param, body, sc)
    …
```

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
```

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (Num(n), env) -> NumT
    …
```

$$\Gamma \vdash \texttt{<num>} : \textit{num}$$

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
  | (Add(l, r), env) ->
      (match (typecheck(l,env), typecheck(r, env)) with
        (NumT, NumT) -> NumT
      | _ -> raise (NoType (Add(l, r), "+: not both nums")))
    …

exception NoType of fae * string
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \mathit{num} \qquad \Gamma \vdash \mathbf{e}_2 : \mathit{num}}{\Gamma \vdash \{\mathbf{+}\ \mathbf{e}_1\ \mathbf{e}_2\} : \mathit{num}}$$

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
  | (Sub(l, r), env) ->
      (match (typecheck(l,env), typecheck(r, env)) with
        (NumT, NumT) -> NumT
      | _ -> raise (NoType (Add(l, r), "-: not both nums")))
    …
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \mathit{num} \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \mathit{num}}{\Gamma \vdash \{\mathbf{-} \ \mathbf{e}_1 \ \mathbf{e}_2\} \ : \ \mathit{num}}$$

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
  | (Id(name), env) -> gettype(name, env)
    …
```

$$[\ldots\ \texttt{<id>} \leftarrow \tau\ \ldots] \vdash \texttt{<id>} : \tau$$

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
  | (Id(name), env) -> gettype(name, env)
    …

let rec gettype = function
    (findName, MTEnv) -> raise (Failed "free variable")
  | (findName, ABind(name, t, restEnv)) ->
      if (name = findName)
      then t
      else gettype(findName, restEnv)
```

[ … <id>←τ … ] ⊢ <id> : τ

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
  | (Fun(param, texpr, body), env) ->
      let argType = parseType(texpr)
      in ArrowT(argType,
                typecheck(body, ABind(param, argType, env)))
    …
```

$$\frac{\Gamma[\,\texttt{<id>}\leftarrow\tau_1\,] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{\{fun \{<id> : } \tau_1\texttt{\} e\}} : (\tau_1 \rightarrow \tau_2)}$$

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
  | (Fun(param, texpr, body), env) ->
      let argType = parseType(texpr)
      in ArrowT(argType,
                typecheck(body, ABind(param, argType, env)))
    …


let rec parseType = function
    NumTE -> NumT
  | BoolTE -> BoolT
  | ArrowTE(a, b) -> ArrowT(parseType a, parseType b)
```

# TFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    ...
  | (App(fn, arg), env) ->
      (match typecheck(fn, env) with
        ArrowT(paramType, resultType) ->
          let argType = typecheck(arg, env)
          in if (argType = paramType)
          then resultType
          else raise (NoType (App(fn,arg),"type mismatch"))
      | _ -> raise (NoType (App(fn, arg), "non-function")))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ (\tau_2 \rightarrow \tau_3) \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \tau_2}{\Gamma \vdash \{\mathbf{e}_1 \ \mathbf{e}_2\} \ : \ \tau_3}$$