Type System Extensions

Two Reasons to Extend a Type System

- We can't typecheck some programs!
  - We may know that a program is ok, would run fine in an interpreter, but type system can't prove it
  - Spoiler: our type system can't typecheck `mk-rec`
  - So we'll have to add `rec` to the language
  - New language constructs → new typing rule!
  - New typing rule → new way to prove programs ok!

- We want to extend our language
  - We'll add boxes and product types
  - New language constructs → new typing rules

Recursion

```
{with {mk-rec {fun {body}
                   {{fun {fX} {fX fX}}
                    {fun {fX}
                         {{fun {f} {body f}}
                          {fun {x} {{fX fX} x}}}}}}}
      {with {fib {mk-rec
                  {fun {fib}
                       {fun {n}
                            {if0 n
                                 1
                                 {if0 {- n 1}
                                      1
                                      {+ {fib {- n 1}}
                                         {fib {- n 2}}}}}}}}}
            {fib 4}}}
```

Typed Recursion

```
{with {mk-rec : (((number -> number) -> (number -> number))
                 -> (number -> number))
                {fun {body : ((number -> number)
                              -> (number -> number))}
                     {{fun {fX : … -> (number -> number)} {fX fX}}
                      {fun {fX : … -> (number -> number)}
                           {{fun {f : (number -> number)} {body f}}
                            {fun {x : number} {{fX fX} x}}}}}}}
      {with {fib : (number -> number)
                   {mk-rec
                    {fun {fib : (number -> number)}
                         {fun {n : number}
                              {if0 n
                                   1
                                   {if0 {- n 1}
                                        1
                                        {+ {fib {- n 1}}
                                           {fib {- n 2}}}}}}}}}
            {fib 4}}}
```

Nothing works in place of the …

Extending the Type System

When the type system rejects your prefectly good program, it may be time to extend the type system

In this case, we can add **`rec`** as a core form, again

```
{rec {fib : (number -> number)
           {fun {n : number}
                {if0 n
                     1
                     {if0 {- n 1}
                          1
                          {+ {fib {- n 1}}
                             {fib {- n 2}}}}}}}}
      {fib 4}}
```

TRCFAE Grammar

```
<TRCFAE>  ::=  <num>
            |  {+ <TRCFAE> <TRCFAE>}
            |  {- <TRCFAE> <TRCFAE>}
            |  <id>
            |  {fun {<id> : <TE>} <TRCFAE>}
            |  {<TRCFAE> <TRCFAE>}
            |  {if0 <TRCFAE> <TRCFAE> <TRCFAE>}        NEW
            |  {rec {<id> : <TE> <TRCFAE>} <TRCFAE>}   NEW
<TE>      ::=  number
            |  (<TE> -> <TE>)
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \textit{number} \qquad \Gamma \vdash \mathbf{e}_2 : \tau_0 \qquad \Gamma \vdash \mathbf{e}_3 : \tau_0}{\Gamma \vdash \{\texttt{if0}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3\} : \tau_0}$$

TRCFAE Grammar

```
<TRCFAE>  ::=  <num>
            |  {+ <TRCFAE> <TRCFAE>}
            |  {- <TRCFAE> <TRCFAE>}
            |  <id>
            |  {fun {<id> : <TE>} <TRCFAE>}
            |  {<TRCFAE> <TRCFAE>}
            |  {if0 <TRCFAE> <TRCFAE> <TRCFAE>}        NEW
            |  {rec {<id> : <TE> <TRCFAE>} <TRCFAE>}   NEW
<TE>      ::=  number
            |  (<TE> -> <TE>)
```

$$\frac{\Gamma[\texttt{<id>} \leftarrow \tau_0] \vdash \mathbf{e}_0 : \tau_0 \qquad \Gamma[\texttt{<id>} \leftarrow \tau_0] \vdash \mathbf{e}_1 : \tau_1}{\Gamma \vdash \{\texttt{rec } \{\texttt{<id>} : \tau_0\ \mathbf{e}_0\}\ \mathbf{e}_1\} : \tau_1}$$

TRCFAE Datatypes

```
(define-type TRCFAE
  ...
  [if0 (test-expr : TRCFAE)
       (then-expr : TRCFAE)
       (else-expr : TRCFAE)]
  [rec (name : symbol)
       (type : Type)
       (named-expr : TRCFAE)
       (body : TRCFAE)])
```

TRCFAE Type Checker

```
(define typecheck : (TRCFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TRCFAE fae
      ...
      [if0 (test-expr then-expr else-expr)
           (type-case Type (typecheck test-expr env)
             [numberT () (define then-type
                           (typecheck then-expr env))
                         (if (equal? then-type
                                     (typecheck else-expr env))
                             then-type
                             (error 'typecheck "type mismatch"))]
             [else (error 'typecheck "expected number")])])))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \mathit{number} \qquad \Gamma \vdash \mathbf{e}_2 : \tau_0 \qquad \Gamma \vdash \mathbf{e}_3 : \tau_0}{\Gamma \vdash \{\mathbf{if0}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3\} : \tau_0}$$

TRCFAE Type Checker

```
(define typecheck : (TRCFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TRCFAE fae
      ...
      [rec (name type named-expr body)
           (define new-env (aBind name type env))
           (if (equal? type (typecheck named-expr new-env))
               (typecheck body new-env)
               (error 'typecheck "type mismatch"))])))
```

$$\frac{\Gamma[\texttt{<id>} \leftarrow \tau_0] \vdash \texttt{e}_0 : \tau_0 \qquad \Gamma[\texttt{<id>} \leftarrow \tau_0] \vdash \texttt{e}_1 : \tau_1}{\Gamma \vdash \{\texttt{rec } \{\texttt{<id>} : \tau_0 \ \texttt{e}_0\} \ \texttt{e}_1\} : \tau_1}$$

TBFAE = FAE + Boxes + Types

```
<TBFAE>  ::=  <num>
           |  {+ <TBFAE> <TBFAE>}
           |  {- <TBFAE> <TBFAE>}
           |  <id>
           |  {fun {<id> : <TE>} <TBFAE>}
           |  {<TBFAE> <TBFAE>}
           |  {newbox <TBFAE>}                    NEW
           |  {setbox <TBFAE> <TBFAE>}            NEW
           |  {openbox <TBFAE>}                   NEW
           |  {seqn <TBFAE> <TBFAE>}              NEW

<TE>     ::=  number
           |  (<TE> -> <TE>)
           |  (boxof <TE>)                        NEW
```

TBFAE Typing Rules

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \tau_1}{\Gamma \vdash \{\mathbf{newbox}\ \mathbf{e}_1\} \ : \ (\mathbf{boxof}\ \tau_1)}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ (\mathbf{boxof}\ \tau_1) \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \tau_1}{\Gamma \vdash \{\mathbf{setbox}\ \mathbf{e}_1\ \mathbf{e}_2\} \ : \ \tau_1}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ (\mathbf{boxof}\ \tau_1)}{\Gamma \vdash \{\mathbf{openbox}\ \mathbf{e}_1\} \ : \ \tau_1}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \tau_1 \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \tau_2}{\Gamma \vdash \{\mathbf{seqn}\ \mathbf{e}_1\ \mathbf{e}_2\} \ : \ \tau_2}$$

TBFAE Datatypes

```
(define-type TBFAE
  ...
  [newbox   (init-expr : TBFAE)]
  [setbox   (box-expr : TBFAE)
            (new-value-expr : TBFAE)]
  [openbox  (box-expr : TBFAE)]
  [seqn     (expr1 : TBFAE)
            (expr2 : TBFAE)])

(define-type Type
  ...
  [boxT (contents-type : Type)])
```

TBFAE Type Checker

```
(define typecheck : (TBFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TBFAE fae
      ...
      [newbox (init-expr)
              (boxT (typecheck init-expr gamma))]
      ...)))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \tau_1}{\Gamma \vdash \{\mathbf{newbox}\ \mathbf{e}_1\} \ : \ (\mathbf{boxof}\ \tau_1)}$$

TBFAE Type Checker

```
(define typecheck : (TBFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TBFAE fae
      ...
      [setbox (box-expr new-value-expr)
              (define box-type (typecheck box-expr gamma))
              (define new-value-type
                (typecheck new-value-expr gamma))
              (unless (boxT? box-type)
                (error 'typecheck "expected box"))
              (unless (equal? new-value-type
                              (boxT-contents-type box-type))
                (error 'typecheck "type mismatch"))
              new-value-type]
      ...)))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : (\mathbf{boxof}\ \tau_1) \qquad \Gamma \vdash \mathbf{e}_2 : \tau_1}{\Gamma \vdash \{\mathbf{setbox}\ \mathbf{e}_1\ \mathbf{e}_2\} : \tau_1}$$

TBFAE Type Checker

```
(define typecheck : (TBFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TBFAE fae
      ...
      [openbox (box-expr)
               (define box-type (typecheck box-expr gamma))
               (unless (boxT? box-type)
                 (error 'typecheck "expected box"))
               (boxT-contents-type box-type)]
      ...)))
```

$$\frac{\Gamma \vdash \mathbf{e}_l \ : \ (\mathbf{boxof}\ \tau_l)}{\Gamma \vdash \{\mathbf{openbox}\ \mathbf{e}_l\} \ : \ \tau_l}$$

TBFAE Type Checker

```
(define typecheck : (TBFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TBFAE fae
      ...
      [seqn (expr1 expr2)
            (typecheck expr1 gamma) ; still need to look for errors!
            (typecheck expr2 gamma)]
      ...)))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \tau_1 \qquad \Gamma \vdash \mathbf{e}_2 : \tau_2}{\Gamma \vdash \{\mathbf{seqn}\ \mathbf{e}_1\ \mathbf{e}_2\} : \tau_2}$$

Product Types

Product as in *cartesian product*

Remember **`Value*Store`**?

- A pair of a **`Value`** and a **`Store`**
- Thinking types as sets, the set of all **`Value*Store`**s is the cartesian product of the set of all **`Value`**s and the set of all **`Store`**s

Can generalize to arbitrary types τ_1 and τ_2

- $\tau_1 \times \tau_2$ is the type for pairs of a τ_1 and a τ_2

TPFAE = FAE + Boxes + Types

```
<TPFAE>  ::=  <num>
           |  {+ <TPFAE> <TPFAE>}
           |  {- <TPFAE> <TPFAE>}
           |  <id>
           |  {fun {<id> : <TE>} <TPFAE>}
           |  {<TPFAE> <TPFAE>}
           |  {pair <TPFAE> <TPFAE>}        NEW
           |  {left <TPFAE> <TPFAE>}        NEW
           |  {right <TPFAE>}               NEW

<TE>     ::=  number
           |  (<TE> -> <TE>)
           |  (<TE> x <TE>)                 NEW
```

```
{with {p {pair 3 {fun {x : number} x}}}
      {+ 5 {left p}}}                  ⇒  8
{with {p {pair 3 {fun {x : number} x}}}
      {{right p} 12}}                  ⇒  12
```

TPFAE Typing Rules

$$\frac{\Gamma \vdash \mathbf{e}_1 : \tau_1 \qquad \Gamma \vdash \mathbf{e}_2 : \tau_2}{\Gamma \vdash \{\mathbf{pair}\ \mathbf{e}_1\ \mathbf{e}_2\} : (\tau_1\ \mathbf{x}\ \tau_2)}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 : (\tau_1\ \mathbf{x}\ \tau_2)}{\Gamma \vdash \{\mathbf{left}\ \mathbf{e}_1\} : \tau_1}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 : (\tau_1\ \mathbf{x}\ \tau_2)}{\Gamma \vdash \{\mathbf{right}\ \mathbf{e}_1\} : \tau_2}$$

TPFAE Datatypes

```
(define-type TPFAE
  ...
  [pair  (left-expr  : TPFAE)
         (right-expr : TPFAE)]
  [left  (pair-expr : TPFAE)]
  [right (pair-expr : TPFAE)])

(define-type Type
  ...
  [productT (left-type  : Type)
            (right-type : Type)])
```

TPFAE Type Checker

```
(define typecheck : (TPFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TPFAE fae
      ...
      [pair (left-expr right-expr)
            (productT (typecheck left-expr gamma)
                      (typecheck right-expr gamma))]
      ...)))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \tau_1 \qquad \Gamma \vdash \mathbf{e}_2 : \tau_2}{\Gamma \vdash \{\mathbf{pair}\ \mathbf{e}_1\ \mathbf{e}_2\} : (\tau_1\ \mathbf{x}\ \tau_2)}$$

TPFAE Type Checker

```
(define typecheck : (TPFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TPFAE fae
      ...
      [left (pair-expr)
            (define pair-type (typecheck pair-expr gamma))
            (unless (productT? pair-type)
              (error 'typecheck "expected product"))
            (productT-left-type pair-type)]
      ...)))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ (\tau_1 \ \mathbf{x} \ \tau_2)}{\Gamma \vdash \{\mathbf{left}\ \mathbf{e}_1\} \ : \ \tau_1}$$

TPFAE Type Checker

```
(define typecheck : (TPFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TPFAE fae
      ...
      [right (pair-expr)
             (define pair-type (typecheck pair-expr gamma))
             (unless (productT? pair-type)
               (error 'typecheck "expected product"))
             (productT-right-type pair-type)]
      ...)))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ (\tau_1 \ \mathbf{x} \ \tau_2)}{\Gamma \vdash \{\mathbf{right}\ \mathbf{e}_1\} \ : \ \tau_2}$$