Compiling with Higher-order Effects

Jaro Reinders

Problem

- Existing languages and compilers are messy
- Hard to maintain
- Hard to understand
- Easy to get things wrong

Problem, Continued

Numbers.sdf3

```
context-free syntax

  Exp.Int      = IntConst

  Exp.Uminus   = [- [Exp]]
  Exp.Times    = [[Exp] * [Exp]]    {left}
  Exp.Divide   = [[Exp] / [Exp]]    {left}
  Exp.Plus     = [[Exp] + [Exp]]    {left}
  Exp.Minus    = [[Exp] - [Exp]]    {left}

  Exp.Eq       = [[Exp] = [Exp]]    {non-assoc}
  Exp.Neq      = [[Exp] <> [Exp]]   {non-assoc}
  Exp.Gt       = [[Exp] > [Exp]]    {non-assoc}
  Exp.Lt       = [[Exp] < [Exp]]    {non-assoc}
  Exp.Geq      = [[Exp] >= [Exp]]   {non-assoc}
  Exp.Leq      = [[Exp] <= [Exp]]   {non-assoc}

  Exp.And      = [[Exp] & [Exp]]    {left}
  Exp.Or       = [[Exp] | [Exp]]    {left}
```

static-semantics.stx

```
  typeOfExp(s, Int(i)) = INT() :-
    @i.lit := i.

rules // operators

  typeOfExp(s, Uminus(e)) = INT() :-
    typeOfExp(s, e) == INT().

  typeOfExp(s, Divide(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Times(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Minus(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().
```

to-ir.str

```

  to-ir-all = innermost(
    to-ir +
    to-ir-flatmap
  )

  // lhs |> rhs -> lhs ; flatMap(lhs)
  to-ir-flatmap: FlatMap(lhs, rhs) -> Seq(lhs, Apply
  // flatMap(lhs, flatMap(rhs)) -> flatMap(lhs) ; fl
  to-ir-flatmap: Apply(Var("flatMap"), [Seq(lhs, App
    Seq(Apply(Var("flatMap"), [lhs]), Apply(Var("fla
  // flatMap(lhs; rhs@(flatMap(_) ; _)) -> flatMap(l
  to-ir-flatmap: Apply(Var("flatMap"), [Seq(lhs, rhs
    Seq(Apply(Var("flatMap"), [lhs]), rhs)

  // Makes a strategy with an implicit input argumen
  to-ir: StrategyDef(name, params, body) -> Strategy
  with inputVar := "__input"     // TODO: Generate un

```

Syntax

Declaratively specify your syntax and pretty-printer using the Syntax Definition Formalism 3 (SDF3) language.

Static Semantics

Use Statix to declare the type system and name binding using *scope graphs*.

Term Transformations

Write an interpreter or compiler using term transformations in Stratego.

Problem, Continued

Numbers.sdf3

```
context-free syntax

  Exp.Int      = IntConst

  Exp.Uminus   = [- [Exp]]
  Exp.Times    = [[Exp] * [Exp]]   {left}
  Exp.Divide   = [[Exp] / [Exp]]   {left}
  Exp.Plus     = [[Exp] + [Exp]]   {left}
  Exp.Minus    = [[Exp] - [Exp]]   {left}

  Exp.Eq       = [[Exp] = [Exp]]   {non-assoc}
  Exp.Neq      = [[Exp] <> [Exp]]  {non-assoc}
  Exp.Gt       = [[Exp] > [Exp]]   {non-assoc}
  Exp.Lt       = [[Exp] < [Exp]]   {non-assoc}
  Exp.Geq      = [[Exp] >= [Exp]]  {non-assoc}
  Exp.Leq      = [[Exp] <= [Exp]]  {non-assoc}

  Exp.And      = [[Exp] & [Exp]]   {left}
  Exp.Or       = [[Exp] | [Exp]]   {left}
```

static-semantics.stx

```
  typeOfExp(s, Int(i)) = INT() :-
    @i.lit := i.

rules // operators

  typeOfExp(s, Uminus(e)) = INT() :-
    typeOfExp(s, e) == INT().

  typeOfExp(s, Divide(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Times(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Minus(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().
```

to-ir.str

```

to-ir-all = innermost(
  to-ir +
  to-ir-flatmap
)

// lhs |> rhs -> lhs ; flatMap(lhs)
to-ir-flatmap: FlatMap(lhs, rhs) -> Seq(lhs, Apply
// flatMap(lhs, flatMap(rhs)) -> flatMap(lhs) ; fl
to-ir-flatmap: Apply(Var("flatMap"), [Seq(lhs, App
  Seq(Apply(Var("flatMap"), [lhs]), Apply(Var("fla
// flatMap(lhs; rhs@(flatMap(_) ; _)) -> flatMap(l
to-ir-flatmap: Apply(Var("flatMap"), [Seq(lhs, rhs
  Seq(Apply(Var("flatMap"), [lhs]), rhs)

// Makes a strategy with an implicit input argumen
to-ir: StrategyDef(name, params, body) -> Strategy
with inputVar := "__input"     // TODO: Generate un

```

Syntax

Declaratively specify your syntax and pretty-printer using the Syntax Definition Formalism 3 (SDF3) language.

Static Semantics

Use Statix to declare the type system and name binding using *scope graphs*.

Term Transformations

Write an interpreter or compiler using term transformations in Stratego.

Monadic Semantics

- Denotational Semantics
- Denote a source program as a monadic computation
- Traditional Denotational Semantics:
  $[\![e_1 + e_2]\!] = [\![e_1]\!] + [\![e_2]\!]$
  Monadic Semantics:
  $[\![e_1 + e_2]\!] = [\![e_1]\!]\ \mathbf{bind}\ (\lambda x.\ [\![e_2]\!]\ \mathbf{bind}\ (\lambda y.\ \mathbf{add}\ x\ y))$
- Advantage: **`bind`** and **`add`** can accommodate different meanings

Monadic Semantics, Continued

- Identity:

```
M A = A
x bind k = k x
add x y  = x + y
```

Monadic Semantics, Continued

- Identity:

```
M A = A
x bind k = k x
add x y  = x + y
```

- Store:

```
M A = S → A × S
x bind k = λs. let (x′, s′) = x s in k x′ s′
add x y  = λs. (x + y, s)
```

Monadic Semantics, Continued

- Identity:
```
M A = A
x bind k = k x
add x y  = x + y
```
- Store:
```
M A = S → A × S
x bind k = λs. let (x′, s′) = x s in k x′ s′
add x y  = λs. (x + y, s)
```
- Partiality:
```
M A = Maybe A
Nothing  bind k = Nothing
(Just x) bind k = k x
add x y         = Just (x + y)
```

Monadic Semantics, Continued

- Identity:

```
M A = A
x bind k = k x
add x y  = x + y
```

- Store:

```
M A
x bi                                  k x′ s′
add
```

Do we have to rewrite the interpretation of all our operations every time?

- Partiality:

```
M A = Maybe A
Nothing  bind k = Nothing
(Just x) bind k = k x
add x y         = Just (x + y)
```

Algebraic Effects and Handlers

- Just keep the operations abstract
- But do optimize the monadic **`return`** and **`bind`** operations
- `⟦(1 + 2) + (3 + 4)⟧`

Algebraic Effects and Handlers

- Just keep the operations abstract
- But do optimize the monadic **`return`** and **`bind`** operations
-
```
⟦(1 + 2) + (3 + 4)⟧ = do
  x ← ⟦1 + 2⟧
  y ← ⟦3 + 4⟧
  add x y
```

Algebraic Effects and Handlers

- Just keep the operations abstract
- But do optimize the monadic **`return`** and **`bind`** operations
- ```
  ⟦(1 + 2) + (3 + 4)⟧ = do
    x ← do
      xx ← ⟦1⟧
      xy ← ⟦2⟧
      add xx xy
    y ← do
      yx ← ⟦3⟧
      yy ← ⟦4⟧
      add yx yy
    add x y
  ```
  ```

Algebraic Effects and Handlers

- Just keep the operations abstract
- But do optimize the monadic **`return`** and **`bind`** operations
-

```
⟦(1 + 2) + (3 + 4)⟧ = do
  x ← do
    xx ← return 1
    xy ← return 2
    add xx xy
  y ← do
    yx ← return 3
    yy ← return 4
    add yx yy
  add x y
```

Algebraic Effects and Handlers

- Just keep the operations abstract
- But do optimize the monadic `return` and `bind` operations
-

```
⟦(1 + 2) + (3 + 4)⟧ = do
  x ← add 1 2
  y ← add 3 4
  add x y
```

Algebraic Effects and Handlers

- Just keep the operations abstract
- But do optimize the monadic **`return`** and **`bind`** operations
- ⟦(1 + 2) + (3 + 4)⟧ = **do**

```
x ← add 1 2
y ← add 3 4
add x y
```

```
(add 1 2) (λx. do y ← add 3 4; add x y)

(add 3 4) (λy. add x y)

(add x y) return
```

Algebraic Effects and Handlers

- Just keep the operations abstract
- But do optimize the monadic **`return`** and **`bind`** operations
- $[\![(1 + 2) + (3 + 4)]\!]$ = **do**

```
x ← add 1 2
y ← add 3 4
add x y
```

```
(add 1 2) (λx. do y ← add 3 4; add x y)

(add 3 4) (λy. add x y)

(add x y) return
```

```
(add x y) k → k (x + y)
```

Denote a Simple Language

$$
\begin{aligned}
[\![n]\!] &= \mathbf{int}\ n \\
[\![\texttt{-e}]\!] &= \mathbf{do}\ x \leftarrow [\![e]\!]\ ;\ \mathbf{neg}\ x \\
[\![e_1 + e_2]\!] &= \mathbf{do}\ x \leftarrow [\![e_1]\!]\ ;\ y \leftarrow [\![e_2]\!]\ ;\ \mathbf{add}\ x\ y \\
[\![e_1 - e_2]\!] &= \mathbf{do}\ x \leftarrow [\![e_1]\!]\ ;\ y \leftarrow [\![e_2]\!]\ ;\ \mathbf{sub}\ x\ y \\
\\
[\![\mathrm{read}]\!] &= \mathbf{read} \\
\\
[\![v]\!] &= \mathbf{var}\ v \\
\\
[\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e_2]\!] &= \mathbf{let}\ x\ [\![e_1]\!]\ \ [\![e_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{int} &: \mathrm{Integer} \rightarrow m\ v \\
\mathbf{neg} &: v \rightarrow m\ v \\
\mathbf{add} &: v \times v \rightarrow m\ v \\
\mathbf{sub} &: v \times v \rightarrow m\ v \\
\\
\mathbf{read} &: m\ v \\
\\
\mathbf{var} &: \mathrm{String} \rightarrow m\ v \\
\\
\mathbf{let} &: \mathrm{String} \times m\ v \times m\ v \\
&\rightarrow m\ v
\end{aligned}
$$

Flatten `let`

(CBV)

```
assign : String → v → m ()
```

```
(let x e₁ e₂) k → do
  y ← e₁
  assign x y
  z ← e₂
  k z
```

Uniquify

```
(var v) k → do
  x ← lookupEnv v
  z ← var x
  k z

(let x e₁ e₂) k → do
  x′ ← gensym x
  z ← let x′ e₁ (do insertEnv x x′; e₂)
  k z
```

X86 With Variables

```
(int n) k → do
  z ← gensym "int"
  x ← imm n
  movq x z
  k z

(add x y) k → do
  z ← gensym "add"
  movq y z
  addq x z
  k z
```

```
read k → do
  z ← gensym "read"
  callq "_read_int"
  x ← reg Rax
  movq x z
  k z

(assign x y) k → do
  z ← var x
  movq y z
  k ()
```

```
imm   : Integer  → m v
reg   : Register → m v
movq  : v × v    → m ()
addq  : v × v    → m ()
callq : String   → m ()
```

X86 With Variables

```
(int n) k → do
  z ← gensym "in
  x ← imm n
  movq x z
  k z

(add x y) k → do
  z ← gensym "ad
  movq y z
  addq x z
  k z
```

`gensym` is a meta-operation and should be discharged separately.

It is still not clear if this is a workable approach.

Perhaps we need a built-in way to pass state while handling effects.

```
k ()
```

```
Integer  → m v
Register → m v
v × v    → m ()
v × v    → m ()
String   → m ()
```

Future Work

- Handling meta-effects like **`gensym`**
- Full compilation pipeline to real X86 by allocating variables on the stack
- Dealing with variable binding in a more satisfying way
- More complicated language features: conditionals, loops, closures, exceptions, parallelism
- Optimizations and analyses: register allocation, peephole optimizations
- Correctness by reasoning about algebraic laws