Evaluation strategies for monadic computations

Tomas Petricek
University of Cambridge

~~Evaluation strategies for monadic computations~~

Computational semi-bimonads!

Tomas Petricek
University of Cambridge

Running example in Haskell

```
chooseSize :: Int → Int → Int
chooseSize new legacy =
    if new > 0 then new else legacy

resultSize :: Int
resultSize =
    chooseSize   (pureInput "new_size")
                 (pureInput "legacy_size")

main :: IO ()
main = print resultSize
```

Monadic call-by-name translation

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.\, e]\!] &= \text{unit}\ (\lambda x.\, [\![e]\!]) \\
[\![e_1\ e_2]\!] &= \text{bind}\ [\![e_1]\!]\ (\lambda f.\, f\ [\![e_2]\!])
\end{aligned}
$$

arguments & result are effectful

```
chooseSize :: IO Int → IO Int → IO Int
chooseSize new legacy = do
    newVal ← new
    if newVal > 0 then new else legacy

resultSize :: IO Int
resultSize =
    chooseSize   (pureInput "new_size")
                 (pureInput "legacy_size")
```

access new_size

access legacy_size or new_size (again!)

Monadic call-by-value translation

$$
\begin{aligned}
[\![x]\!] &= \text{unit } x \\
[\![\lambda x.\, e]\!] &= \text{unit } (\lambda x.\, [\![e]\!]) \\
[\![e_1\ e_2]\!] &= \text{bind } [\![e_1]\!]\ (\lambda f.\, \text{bind } [\![e_2]\!]\ f)
\end{aligned}
$$

arguments are effect-free

```
chooseSize :: Int → Int → IO Int
chooseSize new legacy =
    if new > 0 then return new else return legacy

resultSize :: IO Int
resultSize = do
    new    ← fileInput "new_size"
    legacy ← fileInput "legacy_size"
    chooseSize new legacy
```

access new_size

access legacy-size

Motivation

Standard translations

- Programs have **different structure**
- Difficult to switch strategies

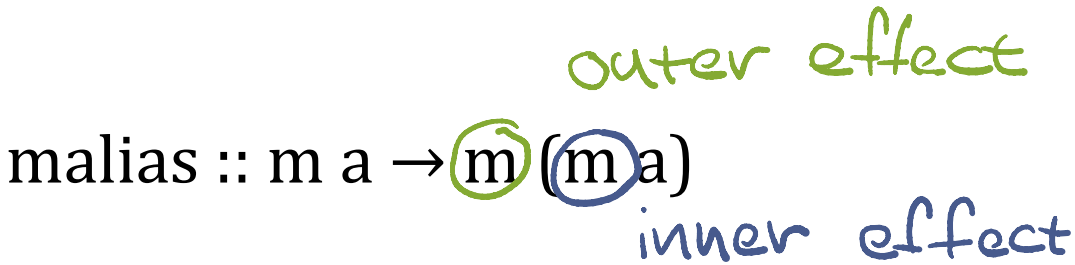
Two strategies only

- **Call-by-value** – unnecessary effects
- **Call-by-name** – repeats effects
- How to add other strategies?

Introducing **call-by-alias** translation for monadic computations

Introducing **call-by-alias** translation

Parameterized by an operation



Benefits of our translation

- Generalizes **call-by-value** and **call-by-name**
- **Call-by-need** for stateful monads
- Parameterize code by evaluation strategy

Monadic **call-by-name** translation

may have effects

$$[\![x]\!] = x$$
$$[\![\lambda x.\, e]\!] = \text{unit}\ (\lambda x.\, [\![e]\!])$$
$$[\![e_1\ e_2]\!] = \text{bind}\ [\![e_1]\!]\ (\lambda f.\, f\ [\![e_2]\!])$$

Monadic **call-by-value** translation

$$
\begin{aligned}
[\![x]\!] &= \mathrm{unit}\ x \\
[\![\lambda x.\, e]\!] &= \mathrm{unit}\ (\lambda x.\, [\![e]\!]) \\
[\![e_1\ e_2]\!] &= \mathrm{bind}\ [\![e_1]\!]\ (\lambda f.\, \mathrm{bind}\ [\![e_2]\!]\ f)
\end{aligned}
$$

may run effects

Monadic call-by-alias translation

may have effects

$$[\![x]\!] = x$$
$$[\![\lambda x.\, e]\!] = \text{unit}\ (\lambda x.\, [\![e]\!])$$
$$[\![e_1\ e_2]\!] = \text{bind}\ [\![e_1]\!]\ (\lambda f.\, \text{bind}\ (\text{malias}\ [\![e_2]\!])\ f)$$

may run effects

```
chooseSize :: IO Int → IO Int → IO Int
chooseSize new legacy = do
    newVal ← new
    if newVal > 0 then new else legacy

resultSize :: IO Int
resultSize = do
    new    ← malias (fileInput "new_size")
    legacy ← malias (fileInput "legacy_size")
    chooseSize new legacy
```

run inner effects

run outer effects

Defining strategies using
call-by-alias translation

Implementing call-by-name

```
malias :: m a → m (m a)
malias m = return m
```

```
chooseSize :: IO Int → IO Int → IO Int
chooseSize new legacy = do
    newVal ← new
    if newVal > 0 then new else legacy

resultSize :: IO Int
resultSize = do
    new    ← malias (fileInput "new_size")
    legacy ← malias (fileInput "legacy_size")
    chooseSize new legacy
```

Implementing call-by-value

```
malias :: m a → m (m a)
malias m = mapM return m
```

```
chooseSize :: IO Int → IO Int → IO Int
chooseSize new legacy = do
    newVal ← new
    if newVal > 0 then new else legacy

resultSize :: IO Int
resultSize = do
    new    ← malias (fileInput "new_size")
    legacy ← malias (fileInput "legacy_size")
    chooseSize new legacy
```

run all effects

Implementing **other** strategies

```
malias :: IO a → IO (IO a)
malias comp = do
    ref ← newIORef Nothing
    return $ do
        value ← readIORef ref
        case value of
            Nothing → comp >>= λv →
                writeIORef ref (Just v) ≫ return v
            Just v → return v
```

Call-by-need for monads with **state**

Call-by-future for monads with **parallelism**

Parameterize code by evaluation strategy

Towards the theory of **call-by-alias** translation

Theory of **call-by-alias**

Operation obeys soma laws

- Preserves **source equivalence**

$$\textbf{let } x = v \textbf{ in } e \equiv e[x \leftarrow v] \qquad \textbf{where } v = \lambda x.\, e$$
$$\textbf{let } x = e \textbf{ in } x \equiv e$$

- Preserves computational **effects**
- Obeys **natural transformation** laws

Monad with cojoin of a comonad

- Shares laws with **computational comonads**

Conclusions & Questions

Call-by-alias translation

- Uses additional operation
- Unifies **call-by-name** & **call-by-value**
- Support for **other** strategies
- Based on comonadic structure

Do we need **monadic notations**?

More information

The malias operation laws

Naturality and **associativity** laws

$$map\ (map\ f\ ) \circ\ malias \equiv malias \circ (map\ f)$$
$$map\ malias \circ malias \equiv malias \circ malias$$

Aliasing of a pure computation is pure

$$malias \circ unit = unit \circ unit$$

Monadic *join* is a **left inverse** of *malias*

$$join \circ malias = id$$

Computational semi-bimonads

Monad (T, η, μ)

$T: \mathcal{C} \to \mathcal{C}$

$\eta: I \to T$

$\mu: T^2 \to T$

Comonad (T, ε, δ)

$T: \mathcal{C} \to \mathcal{C}$

$\varepsilon: T \to I$

$\delta: T \to T^2$

Computational comonad $(T, \varepsilon, \delta, \gamma)$

(T, ε, δ) is a comonad

$\gamma: I \to T$

Computational semi-bimonads

Monad with δ of a computational comonad

Explaining **malias** laws (1)

Computationality $malias \circ unit = unit \circ unit$

Aliasing of pure computation is pure computation

Translation uses $unit$ on function values

$$\textbf{let}\ f = \lambda x.\, e_1\ \textbf{in}\ e_2 \ \equiv\ e_2[f \leftarrow \lambda x.\, e_1]$$

Identity law $join \circ malias = id$

Aliasing and immediately collapsing has no effect

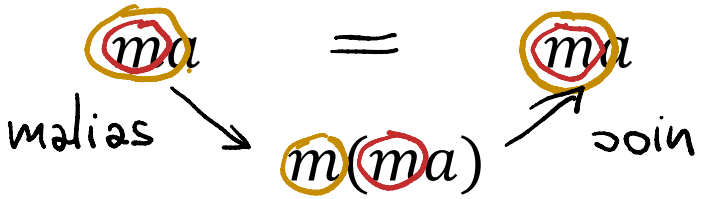
Allows removing redundant bindings

$$\textbf{let}\ v = e\ \textbf{in}\ v \ \equiv\ e$$

Explaining **malias** laws (2)

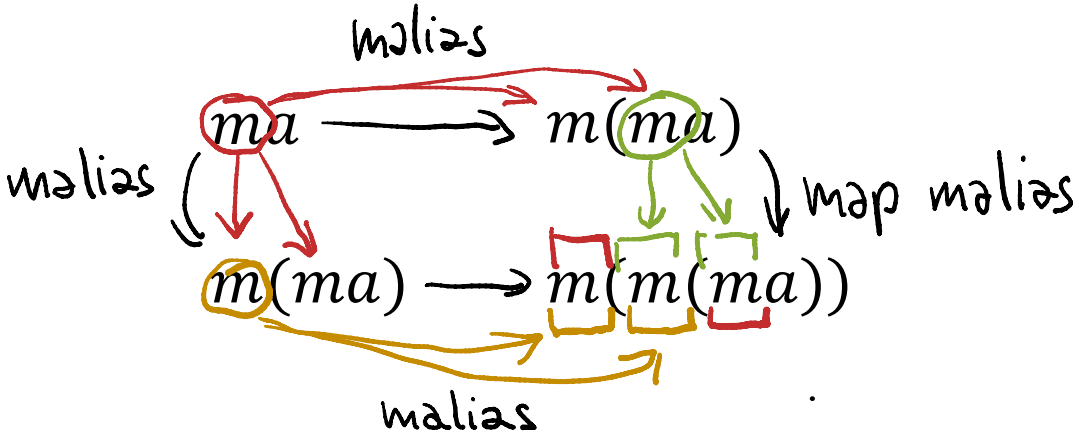
Identity law $join \circ malias = id$

Aliasing and collapsing should not change effects



Associativity $map\ malias \circ malias = malias \circ malias$

Splitting should not introduce asymmetries



Practical **applications**

Special strategies for some monads

- Lazy evaluation for **IO** or **State** monads
- Parallel evaluation for the **Par** monad

Extending monads with **call-by-need**

- Using monad transformer to add state
- Extended monad supports lazy evaluation

Code **parameterized** by evaluation strategy

- Using advanced Haskell constraints

Implementing call-by-future

```
malias :: Par a → Par (Par a)
malias comp = do
    ref ← spawn comp
    return $ get ref
```

fork (outer)

join (inner)

```
chooseSize :: IO Int → IO Int → IO Int
chooseSize new legacy = do
    newVal ← new
    if newVal > 0 then new else legacy
```

wait for completion

```
resultSize :: IO Int
resultSize = do
    new    ← malias (fileInput "new_size")
    legacy ← malias (fileInput "legacy_size")
    chooseSize new legacy
```

start in parallel

Extending monad with call-by-need

```
newtype CbL s m a = CbL { unCbL :: STT s m a }
```

add state to m

```
malias :: CbL s m a → CbL s m (CbL s m a)
malias (CbL marg) = CbL $ do
    r ← newSTRef  Nothing
    return (CbL $ do
        rv ← readSTRef r
        case rv of
            Nothing → marg >>= λv →
                writeSTRef r (Just v) ≫ return v
            Just v → return v)
```

allocate

run & cache