Optimal Effects via Token-Passing

Marvin Borner (Wilhelm-Schickard-Institut für Informatik)

University of Tübingen

Optimality with Side Effects?

Optimality = Full Laziness

Goal: Do the minimal amount of reduction work.

```
let x = factorial(100)          let x = factorial(100)
in  x + x                       in  maybeTrue() ? x : 0
```

$\Longrightarrow$ delay & share reduction via **Call-by-Need**?

```
let f = x => x + factorial(100) // <-- evaluated twice!
in  f(0) + f(1)
```

$\Longrightarrow$ *optimal reducers* also **share redexes**!

Functional Languages with Side Effects, Traditionally

λ-calculus + side effects: `let x = readInt() / readInt() // ?`

- strict evaluation (CbV) $\Longrightarrow$ no inherent sharing & parallelism
- lazy + monads/state-passing $\Longrightarrow$ complex sharing mechanisms

\+ parallel reduction is nontrivial without the *one-step diamond property*.

Optimal Solution

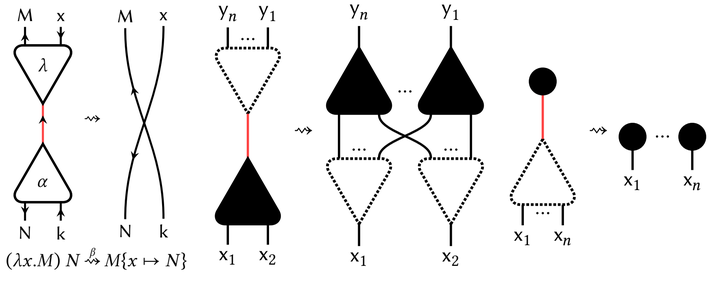
Ignore evaluation order completely for now. **Combine**:

- *Unordered reduction* (optimality/parallelism by choice)
- *Ordered execution* with side effects (+ controlled parallelism)
- One-step diamond property (strong confluence)
- Optimality property (maximal sharing, minimal redundancy)

Solution Strategy

- Translate to *effectful* interaction nets: $\{\triangle_\lambda, \triangle_\alpha, \blacktriangle, \bullet\} \cup \{\downarrow, \uparrow, \triangle\}$
- Duplicate iteratively using $\blacktriangle \Longrightarrow$ maximal sharing by default
- Require tokens ($\downarrow$) for execution of actors ($\triangle$: effectful)
- Pass tokens through the net, steered by reduction (returning $\downarrow$: $\uparrow$)

Graph rewrites are local & satisfy the one-step diamond property:

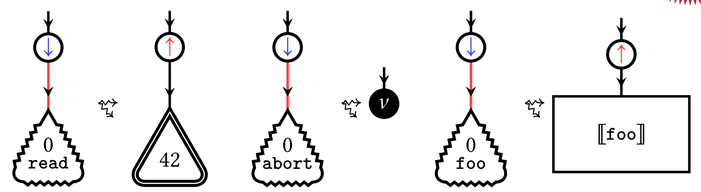


Beta Reduction — Iterative Memory Management

Executing Actors

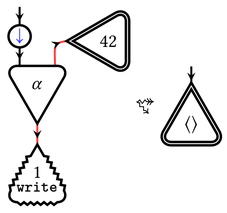
No oracle required!

Arbitrary Execution Behavior



Read Action — Abort Thread — Recursion/FFI/RPC/…

Asynchronous Effects ("`unsafePerformIO`")



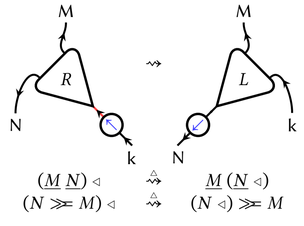
Code: `write(42)!`

- If an action is idempotent and commutes with every other action: Insert explicit token!
- Can then be executed asynchronously and will be shared maximally by iterative duplication: **only gets executed once**!

Reduction Paves the Path for Execution

Tokens are either part of asynchronous execution ("thread"), or a *token-passing semantics*. If tokens are "stuck" (e.g. at the continuation `k` of applications), they can only continue after interaction (e.g. β-reduction)

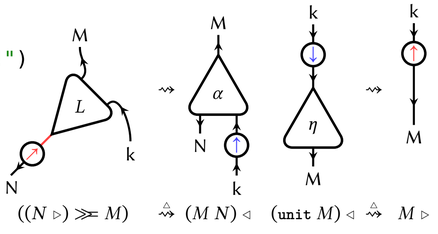
Execution Semantics from Interaction Rules



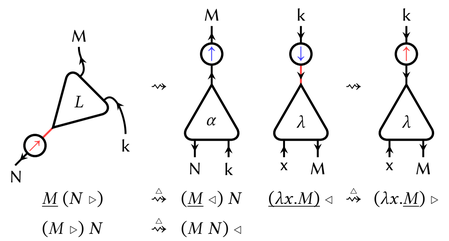
- *Monadic Style*: Translate `binds` to $\triangle_R$ and `units` to $\triangle_\eta$
- *Direct Style*: Replace $\triangle_\alpha$ with $\triangle_R$ in order to permit interaction with the application's continuation
- Redirect token by rotating agents

Monadic Style

```
let repl = do (
  _ <- print("Enter numbers!")
  x <- readInt()
  y <- readInt()
  _ <- writeInt(add(x, y)!)
  r <- repl
  r
) in repl
```



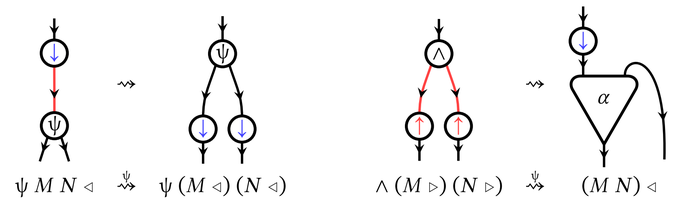
Direct-Style Call-by-Value: Monadic `bind` Everywhere!



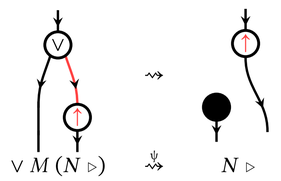

```
let repl = () => (
  writeInt(add(
    readInt(), readInt()))
  repl()
) in repl()
```

+ inference: lazy reduction & strict execution!

Controlled Parallel Execution

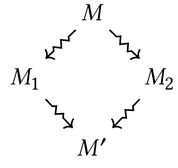


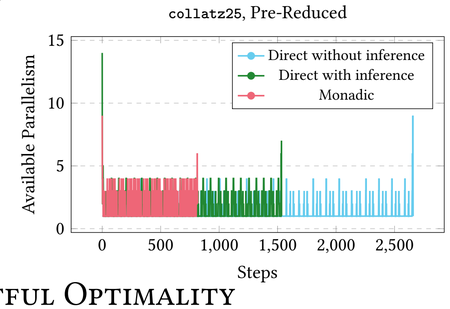

```
let data = race(
  fetch("www1.example.com/data"),
  fetch("www2.example.com/data"),
  fetch("www3.example.com/data"),
)! // promises to be "pure"
```



Massive Parallelism by Strong Confluence*

Step count is independent of reduction/execution order!





Effectful Optimality

- Only reduce "needed" terms (cf. Call-by-Need)
  - Now includes any net containing tokens & actions
  - *Future work*: reasoning with effect systems
- Share all reductions from same origin
  - By iterative duplication (including redexes)
  - Also: asynchronous actions via explicit tokens

*without `race`, full asynchronous execution, or optimal GC