Technische Universität Dresden

Steffen Märcker
Chair for Compiler Construction

Transducers in Smalltalk

Iteration - Here we go again

July 9, 2026

About Me

2010	Graduation from Leipzig University
2011	Software developer at itCampus / FleetBoard
2012–2020	PhD and research at Technische Universität Dresden
	▶ Formal verification (probabilistic model checking)
2021–2026	Postdoc and research at Technische Universität Dresden
	▶ Circuits based on reconfigurable field effect transistors (RFETs)
	▶ Asynchronous Logic (NCL)
	▶ Teaching in computer architecture

Smalltalk Projects

- Application for clinical trails in language parsing
- Simple XML to Object mapper (SimpleXO)

Outline

Iteration

This is how we do it

Iteration Examples

Sum of Even Squares: Plain Loop

```
sum := 0.                          "initial value"
numbers
  do: [:n |
      n even ifTrue:               "filter: select even"
        [| sqr |
        sqr := n squared.          "map:    square"
        sum := sum + sqr]]         "reduce: sum"
```

Iteration Examples

Sum of Even Squares: Classic Collection Methods

```
((numbers
  select: #even)                    "filter: select even"
    collect: #squared)              "map:    square"
      inject: 0                     "initial value"
      into: #+                      "reduce: sum"
```

Iteration Examples

Sum of Even Squares: Xtreams

```
((numbers reading
  selecting: #even)                "filter: select even"
    collecting: #squared)          "map:    square"
      inject: 0                    "initial value"
      into: #+                     "reduce: sum"
```

Observations

General Pattern

```
Source ~> Transformation ~> Drain
```

1. **Source**
   Provides elements for iteration (either push or pull), e. g.,
   collection, stream, . . .
2. **Transformation**
   Transforms elements before providing it to the drain, e. g.,
   select:, collect:, plain code, . . .
3. **Drain**
   Aggregates elements into a result, e. g.,
   summation, accumulation in collection, . . .

Observations

Transformations

- Implemented for each source type (or in plain code)
  - $\Rightarrow$ *Duplication*
  - $\Rightarrow$ *Subtle semantic differences*
  - $\Rightarrow$ *Not composable*
- Kernel is always the same

Observations

Transformations

- Implemented for each source type (or in plain code)
  - ⇒ *Duplication*
  - ⇒ *Subtle semantic differences*
  - ⇒ *Not composable*
- Kernel is always the same

Select (*aka Filter*)

```
(testBlock value: each) ifTrue: [anAction value: each]
```

Observations
Transformations

- Implemented for each source type (or in plain code)
  - ⇒ *Duplication*
  - ⇒ *Subtle semantic differences*
  - ⇒ *Not composable*
- Kernel is always the same

Select (*aka Filter*)

```
(testBlock value: each) ifTrue: [anAction value: each]
```

Collect (*aka Map*)

```
anAction value: (mapBlock value: each)
```

Transducers

Transformation Kernels

Origin

Credit Where Credit Is Due

- Standard library in Clojure: “Composable algorithmic transformations” developed by R. Hickey[1][2]
- Implementations available for other languages:
  - Haskel[3]
  - Python[4]
  - and more

[1] https://clojure.org/news/2012/05/08/reducers
[2] https://clojure.org/news/2014/08/06/transducers-are-coming
[3] https://github.com/hypirion/haskell-transducers
[4] https://pypi.org/project/transducer/

Aggreation

Loop Bodies (Reducing Functions)

- Perform *step-wise aggregation*
- Stateful: $States \times Values \rightarrow States$
- Method: `#inject:into:` (aka *reduce*)

Sources and Drains

Sources (Reducibles)

- Implement `#inject:into:`

Drains (Reducers)

- Hold a *reducing function*
- Implement `#init:reduce:` by sending `#inject:into:` to a *reducible*
- Support *early termination* by handling exception `Reduced`:
  `Reduced signalResult: 'I am done!'`

Transducers

Transformations (Transducers)

- Capture the *kernel* of a transformation
- Implement `#transform:` to wrap a reducing function into a block that applies a transformation on the fly
- Composable with *

```
(Filter select: #even)          "filter: select even"
  * (Map values: #squared)      "map:    square"
     transform: #+              "wrap:   sum"
```

Transducers

Implementation of `#transform:`

FilterSelect

```
[:result :each |
  (testBlock value: each)
    ifTrue: [aReducingFunction value: result value: each]
    ifFalse: [result]]
```

Transducers

Implementation of `#transform:`

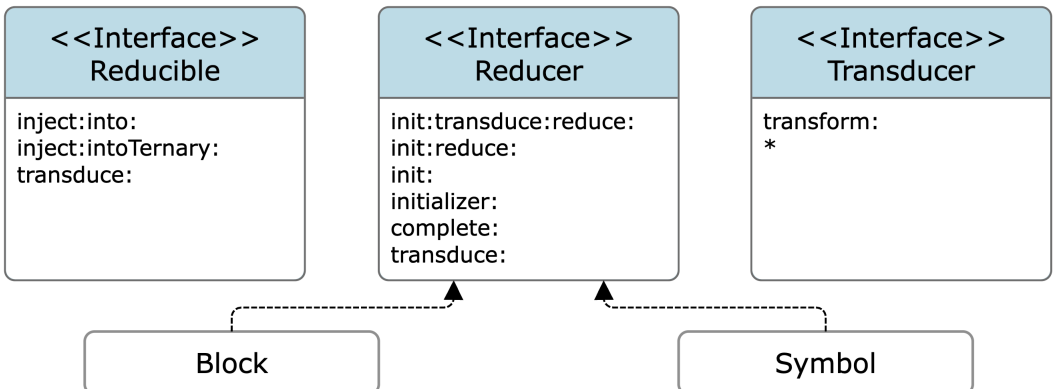
FilterSelect

```
[:result :each |
  (testBlock value: each)
    ifTrue: [aReducingFunction value: result value: each]
    ifFalse: [result]]
```

MapValues

```
[:result :each |
  aReducingFunction
    value: result value: (mapBlock value: each)]
```

Interfaces


<<Interface>>
Reducible
inject:into:
inject:intoTernary:
transduce:
<<Interface>>
Reducer
init:transduce:reduce:
init:reduce:
init:
initializer:
complete:
transduce:
<<Interface>>
Transducer
transform:
*
Block
Symbol

Usage: Plain Loop

Plain Loop

- Transform immediately
- Reduce immediately

```
((Filter select: #even)          "filter: select even"
  * (Map values: #squared)       "map:    square"
     transform: #+)              "wrap:   sum"
       init: 0                   "initial value"
       reduce: numbers           "reduce"
```

Composition
Initialization and Completion

Reducers

- Bind *initial value*
- Perform *completing action* on last result

```
compSum  := #+ init: 0.                          "compute sum"
compMean := compSum complete: [:s | s / n].      "compute mean"

mean := compMean reduce: numbers.                "mean of numbers"
```

Composition
Transformed Source

Reducible + Transducer

- *Virtual sequence* computed on demand
- Bind transducer with `#transduce:`
- Do transformation later

```
"deviation from mean"
deviations := numbers transduce: (Map values: [:x | x - mean]).
```

Composition

Transformed Drain

Reducer + Transducer

- *Reusable computation*
- Bind transducer with `#transduce:`
- Do transformation later

```
"mean absolut deviation"
compAbsDev   := compMean transduce: (Map values: #abs).
"variance"
compVariance := compMean transduce: (Map values: #squared).

absDev   := compAbsDev reduce: deviations.
variance := compVariance reduce: deviations.
```

Composition
Adaptors

Reducible Blocks

- *Sequence* of block evaluations using `#iterate` or `#iterateValue:`
- Adapt to different sources (ad-hoc)
- Compute (infinite) sequences

```
source := [[xtream get] on: Incomplete do: [#stop]]
            iterate
              transduce: (Take whileFalse: [:e | e = #stop]).

naturals := [:n | n + 1] iterateValue: 0.
```

Parallelization

Overview

Parallelization

Concept

$$1 + 2 + 3 + 4 \quad = \quad (1 + 2) + (3 + 4)$$

Requirements

- Reducing function is associative
- Transducers are stateless

Iterating in Parallel

1. Split source
2. Reduce slices in parallel
3. Combine the results (with the same function)

Parallelization

Example (Experimental)

```
((Filter select: #even) * (Map values: #squared)
  transform: (#+ init :0))
      combine: #+
      split: 100000
      fold: numbers
```

- Uses *MatriX* from VisualWorks
- Defaults to 3 child processes
- API subject to change

Experiments

Preliminary Results

Experiments
Enumeration Protocol

- Replaced all implementations of enumeration methods, e. g., `#collect:`, `#groupedBy:`, `#reject:`, ...
- No problems or crashes observed, performance impact imperceptible
- Demand for (comprehensive) tests and benchmark suites

```
Collection>>select: aBlock
  ^((Filter select: aBlock)
    transform: [:newCol :each | newCol add: each; yourself])
      init: self species new
      reduce: self
```

Experiments

Benchmarks

Plain loop	Classic collections	Xtreams	Transducers	Transducers*
100 %	+300 %	+1610 %	+134 %	+52 %

- Demand for (comprehensive) tests and benchmark suites
- Spot checks, e. g., sum of even squares
- Performance relative to a plain loop
- * assumes that all blocks are inlined

Performance Remarks

Considerations

- Avoid full blocks (slow, prevent parallelization)
- Optimize implementations of `#inject:into:`
- Potential (JIT) compiler optimizations
  - Plain loops can be inlined
  - Transformation can be “compiled” similar to Regex
  - Depend on what is know at compile time, e. g.,
    Which transformations can evaluated by the compiler?

Wrap-Up

Wrap-Up

Summary

- Composability enables construction and reuse of complex pipelines
- Encapsulation of the transformation kernels avoid code duplication
- No intermediate collections
- Parallelization processing
- Code for VisualWorks available in public store

Future Work

- Stabilization of streaming and parallel API
- Port to Pharo
- Benchmarks !

Fin
Merci encore.

Backup Slides

Available Transducers

Transducer	Methods
Cat	—
Dedupe	—
Drop	everyNth:, everyNth:Offset:, first:, last:, whileFalse:, whileTrue:
Filter	reject:, select:
Flatten	—
Group	by:, size:, size:offset:
Map	keys:, keys:andValues:, values:
NonNil	—
RandomSample	probability:
Reductions	of:
Replace	lookup:
Take	everyNth:, everyNth:Offset:, first:, last:, whileFalse:, whileTrue
Tee	to:

Transducers

Example: Coin Flipping

```
| scale label sample protocol coin result |
"Transducers"
scale := Map values: [:x | (x * 2 + 1) floor].
label := Replace lookup: #(heads tails).
sample := Take first: 1000.
"Drain"
protocol := [:bag :c | bag add: c; yourself].
"Source"
coin := Random new transduce: scale * label.
result := (sample transform: protocol)
            init: Bag new
            reduce: coin
```

Performance of Base Methods

VisualWorks

Method Method	Array	Ordered- Collection	Dictionary	Linked- List	Linked- List*
`do:`	100 %	±0 %	+500 %	−33 %	−33 %
`inject:into:`	100 %	+40 %	+380 %	±0 %	±0 %
`keysAndValuesDo:`	100 %	+33 %	+400 %	t.o.	±0 %
`inject:intoTernary:`	100 %	±0 %	+49 %	t.o.	−49 %

Iteration Examples

Transducers Inlined

```
numbers
  inject: 0
  into:
    [:result :value |
      value even
        ifTrue: [result + value squared]
        ifFalse: [result]]
```

Iteration Examples

Plain Loops in Java

```
int sum = 0;           // initial value
for (int i = numbers.length - 1; i >= 0; i--) {
  int n = numbers[i];
  if (n % 2 == 0) {    // filter: select even
    int sqr = n * n;   // map:    square
    sum += sqr;        // reduce: sum
  }
}
```

Iteration Examples

Streams in Java

```
Arrays.stream(numbers)
  .filter((int n) -> n % 2 == 0)
     .map((int n) -> n * n)
       .reduce(0, (int s, int n) -> s + n)
```

Iteration Examples

FunctionalIterables in Java

```
FunctionalIterable.ofInt(numbers)
  .filter((int n) -> n % 2 == 0)
    .mapToInt((int n) -> n * n)
      .reduce(0, (int s, int n) -> s + n)
```