Coalgebraic Semantics of Recursion on Circular Data Structures

Baltasar Trancón y Widemann

(Technische Universität Berlin)
Universität Bayreuth

CALCO-Jnr
2011-08-29

The Story of a PhD Thesis

Timeline

2000–2002 Search

2002–2004 Experiments

2004–2006 Writing

2007 Success!

For my Thesis I Wanted to do

- something with functional programming,
- something with coalgebra,
- something funny, or at least surprising.

The Story of a PhD Thesis

Timeline

2000–2002	Search
2002–2004	Experiments
2004–2006	Writing
2007	Success!

For my Thesis I Wanted to do

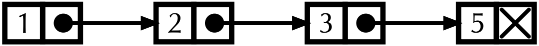
- something with functional programming,
- something with coalgebra,
- something funny, or at least surprising.

My Inspirations

- Karczmarczuk, Jerzy (1998). "The Most Unreliable Technique in the World to Compute PI". In: *Workshop at the 3rd International Summer School on Advanced Functional Programming*.
- Ruiz de Santayana, Jorge Augustín Nicolás (1906). *The Life of Reason*.

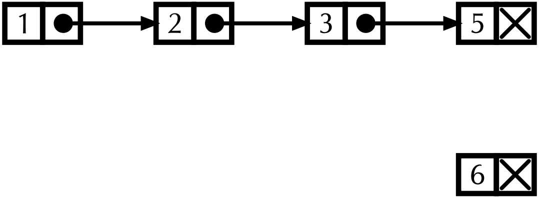
Basic Scenario

- Data structures as cells in memory
- Substructure relation as pointers between cells
- Computation by recursion along pointer chains



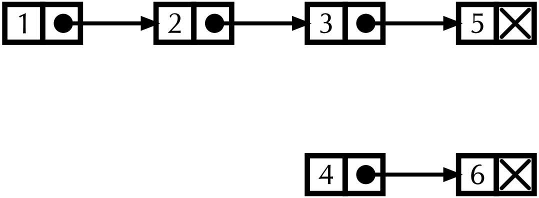
Basic Scenario

- Data structures as cells in memory
- Substructure relation as pointers between cells
- Computation by recursion along pointer chains



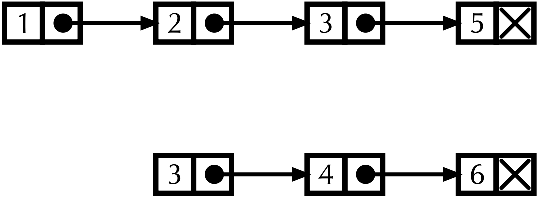
Basic Scenario

- Data structures as cells in memory
- Substructure relation as pointers between cells
- Computation by recursion along pointer chains



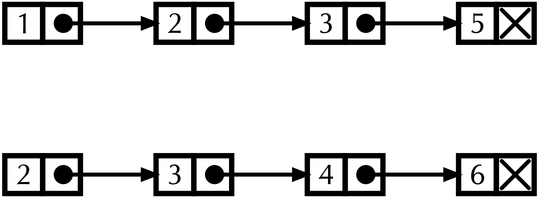
Basic Scenario

- Data structures as cells in memory
- Substructure relation as pointers between cells
- Computation by recursion along pointer chains



Basic Scenario

- Data structures as cells in memory
- Substructure relation as pointers between cells
- Computation by recursion along pointer chains

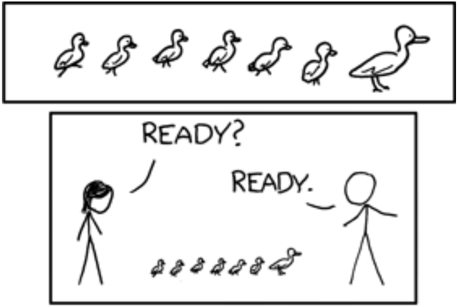


Extension #1: Laziness

Lazy Thunks (Ingerman 1961)

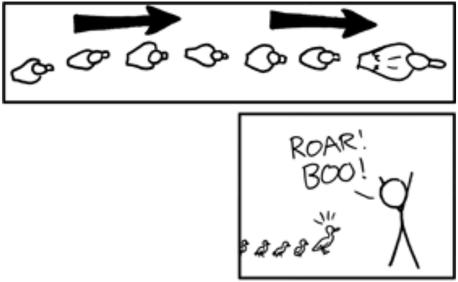
- Contain code to perform suspended computations
- Replaced on demand by data computed on the fly
- Allow for potentially infinite data
- Break temporal connection between call and result

Extension #2



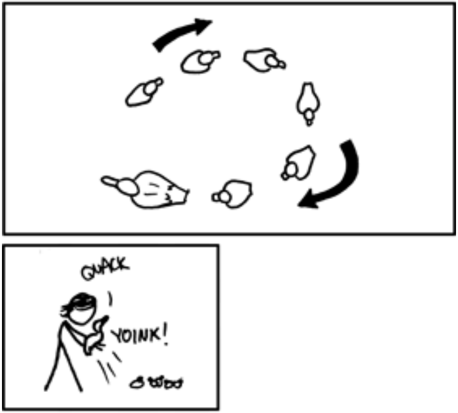
(xkcd 2009)

Extension #2



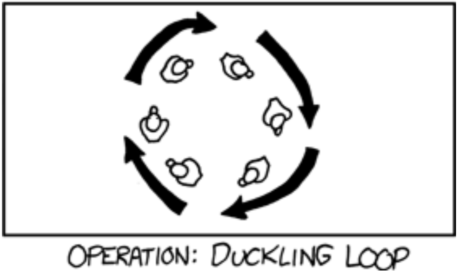
(xkcd 2009)

Extension #2



(xkcd 2009)

Extension #2



(xkcd 2009)

Extension #2: Cycles

- Pointer cycles arise naturally
  - ring lists, doubly linked lists, threaded trees, . . .
- Traditional segregation:

acyclic data; referential transparency; structural recursion
cyclic data; explicit mutable pointers; imperative updates

- Goal: Recursion in the presence of YOINK!

Extension #2: Cycles

- Pointer cycles arise naturally
  - ring lists, doubly linked lists, threaded trees, . . .
- Traditional segregation:

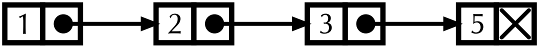
  acyclic data; referential transparency; structural recursion
  cyclic data; explicit mutable pointers; imperative updates
- Goal: Recursion in the presence of **YOINK!**

Heureka!

Motto (Ruiz de Santayana 1906)

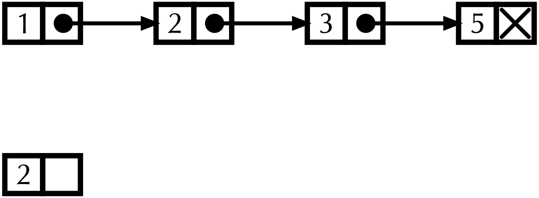
Those who cannot remember the past
are condemned to repeat it.

Order of Operations



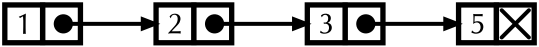
- Destination-passing style (Minamide 1998)
- Tail recursion modulo cons(tructor) (Warren 1980)
  - can be used to eliminate tail calls, or
  - alternatively allows to handle duckling loops!

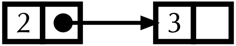
Order of Operations



- Destination-passing style (Minamide 1998)
- Tail recursion modulo cons(tructor) (Warren 1980)
  - can be used to eliminate tail calls, or
  - alternatively allows to handle duckling loops!

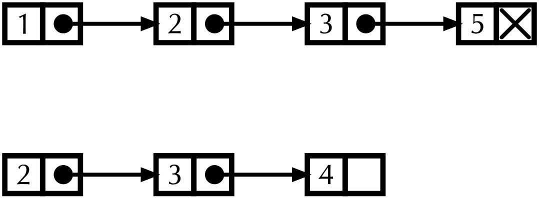
Order of Operations





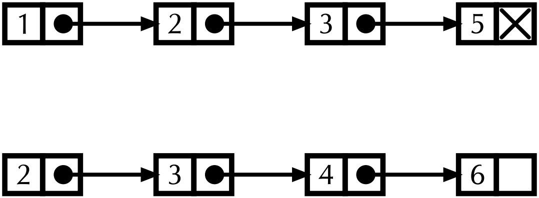
- Destination-passing style (Minamide 1998)
- Tail recursion modulo cons(tructor) (Warren 1980)
  - can be used to eliminate tail calls, or
  - alternatively allows to handle duckling loops!

Order of Operations



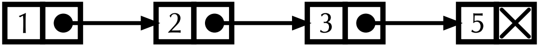
- Destination-passing style (Minamide 1998)
- Tail recursion modulo cons(tructor) (Warren 1980)
  - can be used to eliminate tail calls, or
  - alternatively allows to handle duckling loops!

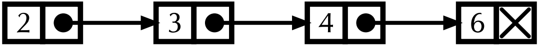
Order of Operations



- Destination-passing style (Minamide 1998)
- Tail recursion modulo cons(tructor) (Warren 1980)
  - can be used to eliminate tail calls, or
  - alternatively allows to handle duckling loops!

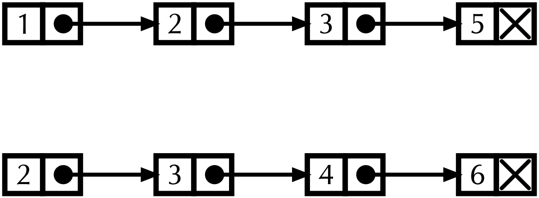
Order of Operations





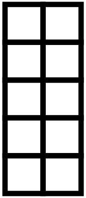
- Destination-passing style (Minamide 1998)
- Tail recursion modulo cons(tructor) (Warren 1980)
  - can be used to eliminate tail calls, or
  - alternatively allows to handle duckling loops!

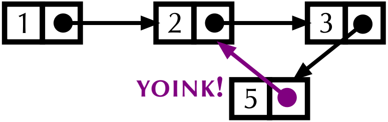
Order of Operations



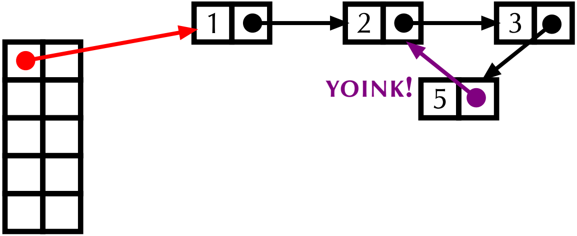
- Destination-passing style (Minamide 1998)
- Tail recursion modulo cons(tructor) (Warren 1980)
  - can be used to eliminate tail calls, or
  - alternatively allows to handle duckling loops!

Cycle Detection & Handling

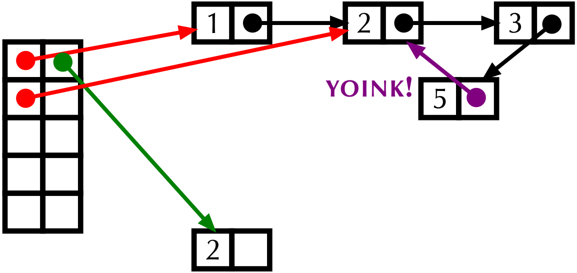




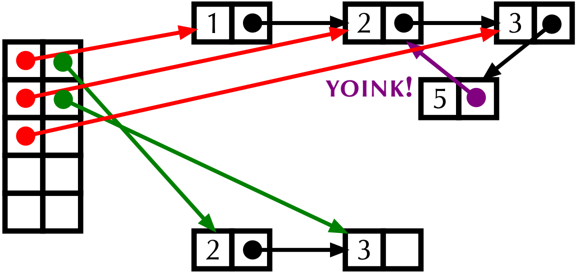
Cycle Detection & Handling



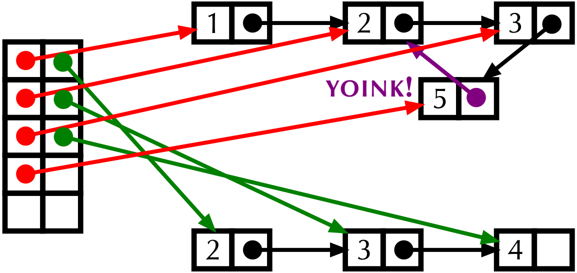
Cycle Detection & Handling



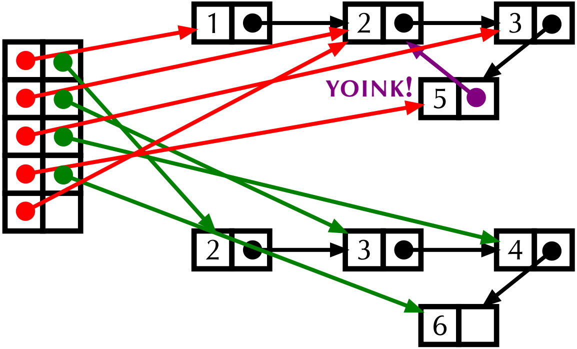
Cycle Detection & Handling



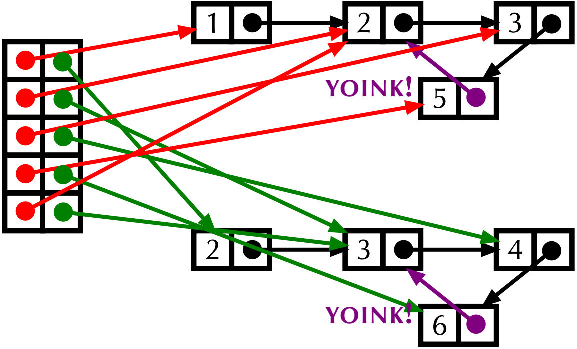
Cycle Detection & Handling



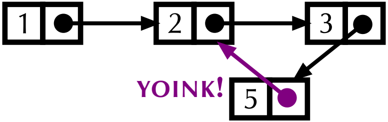
Cycle Detection & Handling



Cycle Detection & Handling



Search Problems

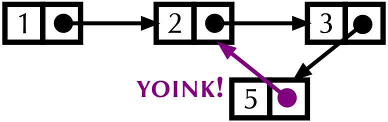


Search Problem Examples

1	Is there an even number?	YES	example	easy
2	Is there a perfect number?	NO	no example	hard
3	Are all numbers prime?	NO	counterex.	easy
4	Are all numbers Fibonacci?	YES	no counterex.	hard

- The hard cases require cycle detection
  - no need to look twice!
- Lazy languages condemned to repeat

Search Problems

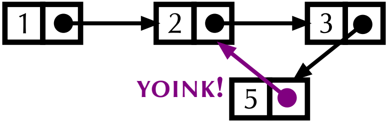


Search Problem Examples

1	Is there an even number?	YES	example	easy
2	Is there a perfect number?	NO	no example	hard
3	Are all numbers prime?	NO	counterex.	easy
4	Are all numbers Fibonacci?	YES	no counterex.	hard

- The hard cases require cycle detection
  - no need to look twice!
- Lazy languages condemned to repeat

Search Problems



Search Problem Examples

1	Is there an even number?	YES	example	easy
2	Is there a perfect number?	NO	no example	hard
3	Are all numbers prime?	NO	counterex.	easy
4	Are all numbers Fibonacci?	YES	no counterex.	hard

- The hard cases require cycle detection
  - no need to look twice!
- Lazy languages condemned to repeat

Implementation

Virtual Machine

- Similar to Java VM
  - memory management, safe references
  - destination-passing style calling conventions
  - cycle detection by stack inspection
- Alternative function body upon cycle detection
  - limited access to call stack (YOINK!)

Efficiency

- Blanket cycle detection on every call too slow
- Mark at least one edge per cycle
  - detection only for marked cases
  - eliminate tail calls for unmarked cases
  - trivial to maintain (YOINK!)
- No cycle ⇒ no mark ⇒ (almost) no cost

Implementation

Virtual Machine

- Similar to Java VM
  - memory management, safe references
  - destination-passing style calling conventions
  - cycle detection by stack inspection
- Alternative function body upon cycle detection
  - limited access to call stack (**YOINK!**)

Efficiency

- Blanket cycle detection on every call too slow
- Mark at least one edge per cycle
  - detection only for marked cases
  - eliminate tail calls for unmarked cases
  - trivial to maintain (**YOINK!**)
- No cycle $\Rightarrow$ no mark $\Rightarrow$ (almost) no cost

Pointer Coalgebra

- Functor $F(X) = \{0,1\}^* \times X^*$ (bits and pointers)
- Memory state as F-coalgebra

  Carrier live addresses

  Operation dereferencing
- Final coalgebra semantics
  - restricted to finite representatives
  - decidable semantic equivalence (bisimilarity)
- Referential transparency
  - monotonocity w.r.t. final semantics
  - modulo garbage collection
- A natural improvement over pointer algebra (Möller 1993)?

Pointer Coalgebra

- Functor $F(X) = \{0, 1\}^* \times X^*$ (bits and pointers)
- Memory state as F-coalgebra

  Carrier live addresses

  Operation dereferencing
- Final coalgebra semantics
  - restricted to finite representatives
  - decidable semantic equivalence (bisimilarity)
- Referential transparency
  - monotonocity w.r.t. final semantics
  - modulo garbage collection
- A natural improvement over pointer algebra (Möller 1993)?

Pointer Coalgebra

- Functor $F(X) = \{0,1\}^* \times X^*$ (bits and pointers)
- Memory state as F-coalgebra

  Carrier live addresses

  Operation dereferencing
- Final coalgebra semantics
  - restricted to finite representatives
  - decidable semantic equivalence (bisimilarity)
- Referential transparency
  - monotonocity w.r.t. final semantics
  - modulo garbage collection
- A natural improvement over pointer algebra (Möller 1993)?

Pointer Coalgebra

- Functor $F(X) = \{0, 1\}^* \times X^*$ (bits and pointers)
- Memory state as F-coalgebra

  Carrier live addresses

  Operation dereferencing
- Final coalgebra semantics
  - restricted to finite representatives
  - decidable semantic equivalence (bisimilarity)
- Referential transparency
  - monotonocity w.r.t. final semantics
  - modulo garbage collection
- A natural improvement over pointer algebra (Möller 1993)?

Pointer Coalgebra

- Functor $F(X) = \{0,1\}^* \times X^*$ (bits and pointers)
- Memory state as F-coalgebra
  - **Carrier** live addresses
  - **Operation** dereferencing
- Final coalgebra semantics
  - restricted to finite representatives
  - decidable semantic equivalence (bisimilarity)
- Referential transparency
  - monotonocity w.r.t. final semantics
  - modulo garbage collection
- A natural improvement over pointer algebra (Möller 1993)?

Structural Corecursion

- Recursion preserving **YOINK!** implements structural corecursion
  - primitive corecursion/coiteration
- Generic algorithm
  - given a coalgebra compatible with final semantics
  - performs referentially transparent memory operations
  - such that final semantics of result
  - equal image of final semantics of input
  - under unique homomorphism (anamorphism)
- Proof of correctness by coinduction

Structural Corecursion

- Recursion preserving **YOINK!** implements structural corecursion
  - primitive corecursion/coiteration
- Generic algorithm
  - given a coalgebra compatible with final semantics
  - performs referentially transparent memory operations
  - such that final semantics of result
  - equal image of final semantics of input
  - under unique homomorphism (anamorphism)
- Proof of correctness by coinduction

Structural Corecursion

- Recursion preserving **YOINK!** implements structural corecursion
  - primitive corecursion/coiteration
- Generic algorithm
  - given a coalgebra compatible with final semantics
  - performs referentially transparent memory operations
  - such that final semantics of result
  - equal image of final semantics of input
  - under unique homomorphism (anamorphism)
- Proof of correctness by coinduction

Cyclical Logic

- Given a search problem as a monotonic deduction system

  acyclic single fixpoint

  cyclic lattice of fixpoints (Tarski 1955)

- Generic algorithm
  - deduce recursively (depth-first search)
  - break cycles with *expectation* YES or NO
  - always NO → least fixpoint (∃)
  - always YES → greatest fixpoint (∀)
  - otherwise (some consistency conditions) → intermediate fixpoints
  - monotonic, modular choice
- Proof of correctness by lattice-theoretic methods
- Special case: bisimilarity as greatest fixpoint

Cyclical Logic

- Given a search problem as a monotonic deduction system

  acyclic single fixpoint

  cyclic lattice of fixpoints (Tarski 1955)
- Generic algorithm
  - deduce recursively (depth-first search)
  - break cycles with *expectation* **YES** or **NO**
  - always **NO** $\rightarrow$ least fixpoint ($\exists$)
  - always **YES** $\rightarrow$ greatest fixpoint ($\forall$)
  - otherwise (some consistency conditions) $\rightarrow$ intermediate fixpoints
  - monotonic, modular choice
- Proof of correctness by lattice-theoretic methods
- Special case: bisimilarity as greatest fixpoint

Cyclical Logic

- Given a search problem as a monotonic deduction system

  acyclic single fixpoint

  cyclic lattice of fixpoints (Tarski 1955)

- Generic algorithm
  - deduce recursively (depth-first search)
  - break cycles with *expectation* **YES** or **NO**
  - always **NO** $\rightarrow$ least fixpoint ($\exists$)
  - always **YES** $\rightarrow$ greatest fixpoint ($\forall$)
  - otherwise (some consistency conditions) $\rightarrow$ intermediate fixpoints
  - monotonic, modular choice
- Proof of correctness by lattice-theoretic methods
- Special case: bisimilarity as greatest fixpoint

Cyclical Logic

- Given a search problem as a monotonic deduction system

  acyclic single fixpoint

  cyclic lattice of fixpoints (Tarski 1955)

- Generic algorithm
  - deduce recursively (depth-first search)
  - break cycles with *expectation* **YES** or **NO**
  - always **NO** $\to$ least fixpoint ($\exists$)
  - always **YES** $\to$ greatest fixpoint ($\forall$)
  - otherwise (some consistency conditions) $\to$ intermediate fixpoints
  - monotonic, modular choice
- Proof of correctness by lattice-theoretic methods
- Special case: bisimilarity as greatest fixpoint

Rational Decimal Arithmetics

- Renaissance algorithms for decimal arithmetics (Ries 1522)
  - Extended to cyclic sequences of digits
  - With (Karczmarczuk 1998) in mind
- Addition/subtraction proceed right to left
  - but there is no right end to start with
- Half addition/subtraction compute local result and carrier independently (coiteration)
  - shift & repeat
  - each digit overflows at most once (iteration)
- Division computes digits by repeated subtraction (iteration)
  - eventually a remainder recurs (coiteration)
- Multiplication (directly) remains hard

Rational Decimal Arithmetics

- Renaissance algorithms for decimal arithmetics (Ries 1522)
  - Extended to cyclic sequences of digits
  - With (Karczmarczuk 1998) in mind
- Addition/subtraction proceed right to left
  - but there is no right end to start with
- Half addition/subtraction compute local result and carrier independently (coiteration)
  - shift & repeat
  - each digit overflows at most once (iteration)
- Division computes digits by repeated subtraction (iteration)
  - eventually a remainder recurs (coiteration)
- Multiplication (directly) remains hard

Rational Decimal Arithmetics

- Renaissance algorithms for decimal arithmetics (Ries 1522)
  - Extended to cyclic sequences of digits
  - With (Karczmarczuk 1998) in mind
- Addition/subtraction proceed right to left
  - but there is no right end to start with
- Half addition/subtraction compute local result and carrier independently (coiteration)
  - shift & repeat
  - each digit overflows at most once (iteration)
- Division computes digits by repeated subtraction (iteration)
  - eventually a remainder recurs (coiteration)
- Multiplication (directly) remains hard

Rational Decimal Arithmetics

- Renaissance algorithms for decimal arithmetics (Ries 1522)
  - Extended to cyclic sequences of digits
  - With (Karczmarczuk 1998) in mind
- Addition/subtraction proceed right to left
  - but there is no right end to start with
- Half addition/subtraction compute local result and carrier independently (coiteration)
  - shift & repeat
  - each digit overflows at most once (iteration)
- Division computes digits by repeated subtraction (iteration)
  - eventually a remainder recurs (coiteration)
- Multiplication (directly) remains hard

Rational Decimal Arithmetics

- Renaissance algorithms for decimal arithmetics (Ries 1522)
  - Extended to cyclic sequences of digits
  - With (Karczmarczuk 1998) in mind
- Addition/subtraction proceed right to left
  - but there is no right end to start with
- Half addition/subtraction compute local result and carrier independently (coiteration)
  - shift & repeat
  - each digit overflows at most once (iteration)
- Division computes digits by repeated subtraction (iteration)
  - eventually a remainder recurs (coiteration)
- Multiplication (directly) remains hard

Cyclic Lists

- Many list algorithms generalize to the cyclic case

  structural *map*, *insert*, *delete*, *concat*

  search *any*, *all*, *sorted*
- Man-or-boy test: *filter*
  - laziness fails if infinitely many consecutive elements are discarded (bust)
  - can be split in three phases:

    mark instance of *map*

    busted instance of *all*

    sweep easy for non-busted case
- With *filter*, *concat* and *sorted* we have *quicksort*!

Cyclic Lists

- Many list algorithms generalize to the cyclic case

  structural *map*, *insert*, *delete*, *concat*

  search *any*, *all*, *sorted*

- Man-or-boy test: *filter*
  - laziness fails if infinitely many consecutive elements are discarded (bust)
  - can be split in three phases:

    mark instance of *map*

    busted instance of *all*

    sweep easy for non-busted case

- With *filter*, *concat* and *sorted* we have *quicksort*!

Cyclic Lists

- Many list algorithms generalize to the cyclic case

  structural *map*, *insert*, *delete*, *concat*

  search *any*, *all*, *sorted*
- Man-or-boy test: *filter*
  - laziness fails if infinitely many consecutive elements are discarded (bust)
  - can be split in three phases:

    mark instance of *map*

    busted instance of *all*

    sweep easy for non-busted case
- With *filter*, *concat* and *sorted* we have *quicksort*!

Structural Subtyping

- Recursive type declarations with ad-hoc products & coproducts
- Structural subtyping by interface emulation
  - opposed to layout compatibility (OOP)
  - transitive, deep, safe
- Subtyping *witness* objects
  - cyclically dependent layout maps (cf. *vtables*)
  - for static checking
  - for dynamic casting
  - composition by cyclic computation
  - dynamic deep “conversion” in $\mathcal{O}(1)$ time

Structural Subtyping

- Recursive type declarations with ad-hoc products & coproducts
- Structural subtyping by interface emulation
  - opposed to layout compatibility (OOP)
  - transitive, deep, safe
- Subtyping *witness* objects
  - cyclically dependent layout maps (cf. *vtables*)
  - for static checking
  - for dynamic casting
  - composition by cyclic computation
  - dynamic deep “conversion” in $\mathcal{O}(1)$ time

Structural Subtyping

- Recursive type declarations with ad-hoc products & coproducts
- Structural subtyping by interface emulation
  - opposed to layout compatibility (OOP)
  - transitive, deep, safe
- Subtyping *witness* objects
  - cyclically dependent layout maps (cf. *vtables*)
  - for static checking
  - for dynamic casting
  - composition by cyclic computation
  - dynamic deep "conversion" in $\mathcal{O}(1)$ time

Looking Back

Some Lessons Learned

1. Aiming for a nice problem pays off.
2. A position with time to merely think is invaluable.
3. Coalgebra is very hard to sell to real programmers (and some theoreticians, too).
4. Weird theory sometimes makes natural examples.

Status of Implementation

The MALICE System

- Java Application
- Executable VM Model
  - assembly-style code format
  - interpreter
- Compiler to lower-level code
  - optimization, specialization
  - static + just-in-time
  - compiles to threaded code
- IDE
  - editors, browsers, interactive, demos

Open Problems I

Front-end Language

- Leverage capabilities of the VM
  - cyclic detection & handling
  - destination-passing & tail recursion
- Nice high-level notation
  - pattern-based recursion
  - referential transparency
  - safe operation order
  - declaration of cycle handling strategy
  - declaration of intended fixpoint

Open Problems II

Generalized Search Problems

- Proof of soundness
  - completeness (all fixpoints selectable)?
- Relies on Boolean lattice of truth values
  - other lattices?
  - application to abstract interpretation?

Open Problems III

Compiler to Machine Code

- All ingredients ready
  - memory management in the presence of **YOINK!**
  - portable cycle detection & handling

Trancón y Widemann, Baltasar (2008a). "A reference-counting garbage collection algorithm for cyclical functional programming". In: *ISMM*. Ed. by Richard Jones and Stephen M. Blackburn. ACM, pp. 71–80. ISBN: 978-1-60558-134-7. DOI: 10.1145/1375634.1375645.

— (2008b). "Stackless Stack Inspection. A Portable Escape Route from Vicious Circles". In: *Programmiersprachen und Rechenkonzepte*. Ed. by Michael Hanus and Sebastian Fischer. 0811.

Postscriptum

Vicious Circle (Reed 1976)

You're caught in a vicious circle
Surrounded by your so-called friends
You're caught in a vicious circle
And it looks like it will never end

You're caught in a vicious circle
You're caught in a vicious circle
You're caught in a vicious circle
You're caught in a vicious circle
Surrounded by all of your friends

Reed, Lou (1976). "Vicious Circle". In: *Rock and Roll Heart*.

5 **References**

6 Applications

- Arithmetics
- Lists
- Subtyping

Bibliography I

Ingerman, P. Z. (1961). "Thunks. A Way of Compling Procedure Statements with Some Comments on Procedure Declarations". In: *CACM* 4.1, pp. 55–58.

Karczmarczuk, Jerzy (1998). "The Most Unreliable Technique in the World to Compute PI". In: *Workshop at the 3rd International Summer School on Advanced Functional Programming.*

Minamide, Yasuhiko (1998). "A functional representation of data structures with a hole". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* POPL '98. San Diego, California, United States: ACM, pp. 75–84. ISBN: 0-89791-979-3. DOI: 10.1145/268946.268953.

Möller, Bernhard (1993). "Towards pointer algebra". In: *Science of Computer Programming* 21.1, pp. 57–90. ISSN: 0167-6423. DOI: 10.1016/0167-6423(93)90008-D.

Reed, Lou (1976). "Vicious Circle". In: *Rock and Roll Heart.*

Bibliography II

Ries, Adam (1522). *Rechenung auff der linihen und federn*. Erfurt.

Ruiz de Santayana, Jorge Augustín Nicolás (1906). *The Life of Reason*.

Tarski, Alfred (1955). "A lattice-theoretical fixpoint theorem and its applications". In: *Pacific Journal of Mathematics* 5.2, pp. 285–309.

Trancón y Widemann, Baltasar (2008a). "A reference-counting garbage collection algorithm for cyclical functional programming". In: *ISMM*. Ed. by Richard Jones and Stephen M. Blackburn. ACM, pp. 71–80. ISBN: 978-1-60558-134-7. DOI: 10.1145/1375634.1375645.

— (2008b). "Stackless Stack Inspection. A Portable Escape Route from Vicious Circles". In: *Programmiersprachen und Rechenkonzepte*. Ed. by Michael Hanus and Sebastian Fischer. 0811.

Warren, David H. D. (1980). DAI Research Report 141. University of Edinburgh.

xkcd (Sept. 31, 2009). *Ducklings*. URL: http://xkcd.com/537/.

5 References

6 **Applications**

- Arithmetics
- Lists
- Subtyping

Rational Decimal Arithmetics

$$
\begin{array}{rlllll}
 & 0\,.\,2 & 8 & (6 & 3) \\
+ & 0\,.\,1 & 3 & (8) & \\
\hline
 & \;\;. & & & \\
 & \;\;. & & & \\
\hline
 & \;\;. & & & \\
 & \;\;. & & &
\end{array}
$$

Rational Decimal Arithmetics

$$
\begin{array}{rcccc}
 & 0.2 & 8 & (6 & 3) \\
+ & 0.1 & 3 & (8) & 8 \\
\hline
 & . & & & \\
 & . & & & \\
\hline
 & . & & & \\
 & . & & &
\end{array}
$$

Rational Decimal Arithmetics

$$
\begin{array}{rl}
 & 0.2\;8\;(6\;\;3) \\
+ & 0.1\;3\;(8)\;8 \\
\hline
 & 0.3\;1\;(4\;\;1) \\
C & 0.0\;1\;(1\;\;1) \\
\hline
 & \vdots
\end{array}
$$

Rational Decimal Arithmetics

$$
\begin{array}{rcccccc}
 & 0 & . & 2 & 8 & (6 & 3) \\
+ & 0 & . & 1 & 3 & (8) & 8 \\
\hline
 & 0 & . & 3 & 1 & (4 & 1) \\
+ & 0 & . & 1 & (1 & 1) & \\
\hline
 & & \vdots & & & &
\end{array}
$$

Rational Decimal Arithmetics

$$
\begin{array}{rllll}
 & 0.2 & 8 & (6 & 3) \\
+ & 0.1 & 3 & (8) & 8 \\
\hline
 & 0.3 & 1 & (4 & 1) \\
+ & 0.1 & (1 & 1) & 1 \\
\hline
 & \vdots & & &
\end{array}
$$

Rational Decimal Arithmetics

	0.2	8	(6	3)	
+	0.1	3	(8)	8	
	0.3	1	(4	1)	
+	0.1	(1	1)	1	
	0.4	2	(5	2)	
C	0.0	0	(0	0)	

5 References

6 Applications

- Arithmetics
- Lists
- Subtyping

Cyclic Lists

I G O A L W A Y S (O N)

Cyclic Lists

I G O A L W A Y S (O N)

- I G O A L W A Y S (O N) ⇝ G A A

- I G O A L W A Y S (O N) ⇝ I
- I G O A L W A Y S (O N) ⇝ O L W Y S (O N)

Cyclic Lists

I G O A L W A Y S (O N)

- I G O A L W A Y S (O N) ⇝ G A A
  - G A A ⇝ **A A**
  - G A A ⇝ **G**
  - G A A ⇝
- I G O A L W A Y S (O N) ⇝ **I**
- I G O A L W A Y S (O N) ⇝ O L W Y S (O N)

Cyclic Lists

I G O A L W A Y S (O N)

- I G O A L W A Y S (O N) ⇝ G A A
  - G A A ⇝ **A A**
  - G A A ⇝ **G**
  - G A A ⇝
- I G O A L W A Y S (O N) ⇝ **I**
- I G O A L W A Y S (O N) ⇝ O L W Y S (O N)
  - O L W Y S (O N) ⇝ **L (N)**
  - O L W Y S (O N) ⇝ **O (O)**
  - O L W Y S (O N) ⇝ W Y S

Cyclic Lists

I G O A L W A Y S (O N)

- I G O A L W A Y S (O N) ⇝ G A A
  - G A A ⇝ **A A**
  - G A A ⇝ **G**
  - G A A ⇝
- I G O A L W A Y S (O N) ⇝ **I**
- I G O A L W A Y S (O N) ⇝ O L W Y S (O N)
  - O L W Y S (O N) ⇝ **L (N)**
  - O L W Y S (O N) ⇝ **O (O)**
  - O L W Y S (O N) ⇝ W Y S
    - W Y S ⇝ **S**
    - W Y S ⇝ **W**
    - W Y S ⇝ **Y**

Cyclic Lists

I G O A L W A Y S (O N)

- I G O A L W A Y S (O N) $\rightsquigarrow$ G A A
  - G A A $\rightsquigarrow$ **A A**
  - G A A $\rightsquigarrow$ **G**
  - G A A $\rightsquigarrow$
- I G O A L W A Y S (O N) $\rightsquigarrow$ **I**
- I G O A L W A Y S (O N) $\rightsquigarrow$ O L W Y S (O N)
  - O L W Y S (O N) $\rightsquigarrow$ **L (N)**
  - O L W Y S (O N) $\rightsquigarrow$ **O (O)**
  - O L W Y S (O N) $\rightsquigarrow$ W Y S
    - W Y S $\rightsquigarrow$ **S**
    - W Y S $\rightsquigarrow$ **W**
    - W Y S $\rightsquigarrow$ **Y**

A A G I L (N)

Recursive Structural Subtyping

$$\begin{aligned} \mathrm{BinTree}[\alpha] ::= {} & \mathrm{Branch}_2(\mathrm{BinTree}[\alpha], \mathrm{BinTree}[\alpha]) & \text{①} \\ \mid {} & \mathrm{Leaf}(\alpha) & \text{②} \end{aligned}$$

$$\begin{aligned} \mathrm{23Tree}[\beta] ::= {} & \mathrm{Branch}_3(\mathrm{23Tree}[\beta], \mathrm{23Tree}[\beta], \mathrm{23Tree}[\beta]) & \text{①} \\ \mid {} & \mathrm{Branch}_2(\mathrm{23Tree}[\beta], \mathrm{23Tree}[\beta]) & \text{②} \\ \mid {} & \mathrm{Leaf}(\beta) & \text{②} \end{aligned}$$

$$c : (\alpha <: \beta) \to (\mathrm{BinTree}[\alpha] <: \mathrm{23Tree}[\beta])$$

$$c(d) = \begin{bmatrix} \text{①} \to \text{②}(c(d), c(d)) \\ \text{②} \to \text{③}(d) \end{bmatrix}$$

Recursive Structural Subtyping

$$\mathsf{BinTree}[\alpha] ::= \mathsf{Branch}_2(\mathsf{BinTree}[\alpha], \mathsf{BinTree}[\alpha]) \quad ❶$$
$$\mid \ \mathsf{Leaf}(\alpha) \quad ❷$$

$$\mathsf{23Tree}[\beta] ::= \mathsf{Branch}_3(\mathsf{23Tree}[\beta], \mathsf{23Tree}[\beta], \mathsf{23Tree}[\beta]) \quad ❶$$
$$\mid \ \mathsf{Branch}_2(\mathsf{23Tree}[\beta], \mathsf{23Tree}[\beta]) \quad ❷$$
$$\mid \ \mathsf{Leaf}(\beta) \quad ❷$$

$$c : (\alpha <: \beta) \to (\mathsf{BinTree}[\alpha] <: \mathsf{23Tree}[\beta])$$

$$c(d) = \begin{bmatrix} ❶ \to ❷\big(c(d), c(d)\big) \\ ❷ \to ❸(d) \end{bmatrix}$$