Term Rewriting with Traversal Functions

Mark van den Brand
Paul Klint
Jurgen Vinju

April 18th, 2001
Overview

- Term Rewriting Problems
- Solutions
- Traversal functions
- Limitations
- Conclusion & Questions
Term Rewriting Problems
Term Rewriting Problems

- Term rewriting is expressive, but...
Term Rewriting Problems

- Term rewriting is expressive, but...
- Term traversal is implicit (e.g. innermost)
  - only sort preserving transformations
  - no extra arguments (i.e. context)
  - fixed path
Term Rewriting Problems

- Term rewriting is expressive, but...
- Term traversal is implicit (e.g. innermost)
  - only sort preserving transformations
  - no extra arguments (i.e. context)
  - fixed path
- Confluence
  - non-trivial trs’s are not confluent
Solutions
Solutions

- Encode strategies using functional style
  - ASF+SDF
- Strategy primitives ELAN
- Strategy & Traversal primitives Stratego
- Very nice! But:
  - Encoding is boring for large systems
  - ELAN Strategies do not solve term traversal
  - Stratego traversal primitives are not typed
- Strategic Programming is another paradigm
  - Traversal functions!
  - Less powerful
  - Fully typed
  - Simple
Solutions

- Encode strategies using functional style: ASF+SDF
- Strategy primitives: ELAN

Very nice! But:
- Encoding is boring for large systems
- ELAN Strategies do not solve term traversal
- Stratego traversal primitives are not typed

Strategic Programming is another paradigm
- Traversal functions: less powerful, fully typed, simple
Solutions

- Encode strategies using functional style \( \text{ASF+SDF} \)
- Strategy primitives \( \text{ELAN} \)
- Strategy & Traversal primitives \( \text{Stratego} \)
Solutions

- Encode strategies using functional style: **ASF+SDF**
- Strategy primitives: **ELAN**
- Strategy & Traversal primitives: **Stratego**

- Very nice! But:
Solutions

- Encode strategies using functional style  \text{ASF+SDF}
- Strategy primitives  \text{ELAN}
- Strategy & Traversal primitives  \text{Stratego}

\text{Very nice! But:}

\text{Encoding is boring for large systems}
Solutions

- Encode strategies using functional style \( \text{ASF+SDF} \)
- Strategy primitives \( \text{ELAN} \)
- Strategy & Traversal primitives \( \text{Stratego} \)

Very nice! But:

- Encoding is boring for large systems
- ELAN Strategies do not solve term traversal
Solutions

- Encode strategies using functional style: ASF+SDF
- Strategy primitives: ELAN
- Strategy & Traversal primitives: Stratego

- Very nice! But:
  - Encoding is boring for large systems
  - ELAN Strategies do not solve term traversal
  - Stratego traversal primitives are not typed
Solutions

- Encode strategies using functional style: ASF+SDF
- Strategy primitives: ELAN
- Strategy & Traversal primitives: Stratego

Very nice! But:

- Encoding is boring for large systems
- ELAN Strategies do not solve term traversal
- Stratego traversal primitives are not typed

_Strategic Programming_ is another paradigm
Solutions

- Encode strategies using functional style
  ASF+SDF
- Strategy primitives
  ELAN
- Strategy & Traversal primitives
  Stratego

- Very nice! But:
  Encoding is boring for large systems
  ELAN Strategies do not solve term traversal
  Stratego traversal primitives are not typed
  *Strategic Programming* is another paradigm

- Traversal functions!
  less powerful
  fully typed
  simple
Traversal functions

Characteristics:
- typed prefix functions
  - three generic types: `trafo`, `accu`, or both
  - two generic strategies: bottom-up (all), top-down (stop)
- simple semantics, easy implementation

The questions are:
- What can we do with them?
- What can we not do with them?
Traversal functions

- Characteristics:
Traversal functions

- Characteristics:
  typed prefix functions
Traversal functions

- Characteristics:
  - typed prefix functions
  - three generic types: trafo, accu or both

The questions are:

- What can we do with them?
- What can we not do with them?
Traversal functions

- Characteristics:
  - typed prefix functions
  - three generic types: trafo, accu or both
  - two generic strategies: bottom-up (all), top-down (stop)
Traversals functions

- Characteristics:
  
  typed prefix functions
  
  three generic types: \texttt{trafo}, \texttt{accu} or both
  
  two generic strategies: \texttt{bottom-up} (all), \texttt{top-down} (stop)
  
  simple semantics, easy implementation
Traversal functions

- Characteristics:
  - typed prefix functions
  - three generic types: trafo, accu or both
  - two generic strategies: bottom-up (all), top-down (stop)
  - simple semantics, easy implementation

- The questions are:
### Traversal functions

- **Characteristics:**
  - Typed prefix functions
  - Three generic types: `trafo`, `accu` or both
  - Two generic strategies: **bottom-up** (all), **top-down** (stop)
  - Simple semantics, easy implementation

- **The questions are:**
  - What can we do with them?
Traversal functions

- Characteristics:
  - typed prefix functions
  - three generic types: trafo, accu or both
  - two generic strategies: bottom-up (all), top-down (stop)
  - simple semantics, easy implementation

- The questions are:
  - What can we do with them?
  - What can we not do with them?
module Pico-syntax
imports Pico-Identifiers Pico-Integers Pico-Strings Types
exports
sorts PROGRAM
context-free syntax
  "begin" DECLS {STATEMENT ";"}* "end" -> PROGRAM
  "declare" {ID-TYPE ","}* ";" -> DECLS
  PICO-ID ":" TYPE -> ID-TYPE
  PICO-ID ":=" EXP -> STATEMENT
  "if" EXP "then" {STATEMENT ";"}* "else" {STATEMENT ";"}* "fi" -> STATEMENT
  "while" EXP "do" {STATEMENT ";"}* "od" -> STATEMENT
  PICO-ID -> EXP
  PICO-NAT-CON -> EXP
  PICO-STR-CON -> EXP
  EXP "+" EXP -> EXP {left}
  EXP "-" EXP -> EXP {left}
  EXP "+" EXP -> EXP {left}
Transformer

module Pico-Replace
imports Pico-syntax
exports
context-free syntax
  replace(PROGRAM,ID,ID) -> PROGRAM {traversal(trafo)}
equations
[] replace(Id1,Id1,Id2) = Id2

Figure 2:
Transformer (cont’d)

```
replace(
begin
    declare
        input  : natural,
        output : natural;
    input  := 12;
    output := 1;
    while input - 1 do
        output := output * 2;
        input := input - 1
    od
end, input, INPUT)
```

Figure 3:
begin
    declare
        INPUT  : natural,
        output : natural;
    INPUT  := 12;
    output := 1;
    while INPUT - 1 do
        output := output * 2;
        INPUT := INPUT - 1
    od
end
Accumulator

module Pico-Count
imports Pico-syntax
exports
count(PROGRAM,NAT) -> NAT {traversal(accu)}
equations
[] count(Id,Nat) = suc(Nat)

Figure 5:
Accumulating Transformer

module Pico-ReplaceCount
imports Pico-syntax
exports
context-free syntax
   replacecount(PROGRAM,NAT,ID,ID) -> PROGRAM # NAT
   {traversal(accu,trafo)}
equations
[] replacecount(Id1,Nat,Id1,Id2) = (Id2,suc(Nat))

Figure 6:
Experience

- COBOL transformations
- CASL transformations
- SDF tools: typechecker, statistics
- New DSL: Framework for SDF Transformation (FST)
Experience

- COBOL transformations
- CASL transformations
- SDF tools: typechecker, statistics
- New DSL: Framework for SDF Transformation (FST)

- Significant reduction in number of rules
- Interpreter efficient enough
Limitations

1. Types: no arbitrary non-sort-preserving traversals
2. Traversals: no arbitrary traversal paths through trees
3. Strategies: still no control over application of rewrite rules

1. Use traversal functions for the unaffected nodes
2. Use conditions and default rules to guide the path
3. Still encode strategies using functions
Limitations

1. Types: no arbitrary non-sort-preserving traversals
Limitations

1. Types: no arbitrary non-sort-preserving traversals
2. Traversals: no arbitrary traversal paths through trees
Limitations

1. Types: no arbitrary non-sort-preserving traversals
2. Traversals: no arbitrary traversal paths through trees
3. Strategies: still no control over application of rewrite rules
**Limitations**

1. Types: no arbitrary non-sort-preserving traversals
2. Traversals: no arbitrary traversal paths through trees
3. Strategies: still no control over application of rewrite rules

---

1. Use traversal functions for the unaffected nodes
Limitations

1. Types: no arbitrary non-sort-preserving traversals
2. Traversals: no arbitrary traversal paths through trees
3. Strategies: still no control over application of rewrite rules

1. Use traversal functions for the unaffected nodes
2. Use conditions and default rules to guide the path
Limitations

1. Types: no arbitrary non-sort-preserving traversals
2. Traversals: no arbitrary traversal paths through trees
3. Strategies: still no control over application of rewrite rules
   
   1. Use traversal functions for the unaffected nodes
   2. Use conditions and default rules to guide the path
   3. Still encode strategies using functions
Conclusions
Conclusions

• Automated tree traversal
Conclusions

- Automated tree traversal
- Statically typed
Conclusions

- Automated tree traversal
- Statically typed
- Simple semantics, first-order term rewriting
Conclusions

- Automated tree traversal
- Statically typed
- Simple semantics, first-order term rewriting
- It works (various applications)
Conclusions

• Automated tree traversal
• Statically typed
• Simple semantics, first-order term rewriting
• It works (various applications)

Future work:
  Efficient compilation
Questions