Haskell, Do You Read Me?

Constructing and Composing Efficient Top-down Parsers at Runtime

Marcos Viera Doaitse Swierstra Eelco Lempsink

Instituto de Computación, Universidad de la República, Uruguay
Dept. of Information and Computing Sciences, Utrecht University

September 25, 2008

Symptoms

```
data T1 = T1 :<: T1
        | T1 :>: T1
        | C
     deriving (Read, Show)
infixl 5 :<:
infixr 6 :>:
x     :: T1
x  = C :<: C :<: C
```

```
*Main> x
(C :<: C) :<: C
```

Symptoms

```
data T1 = T1 :<: T1
        | T1 :>: T1
        | C
        deriving (Read, Show)
infixl 5 :<:
infixr 6 :>:
x, x'  :: T1
x  = C :<: C :<: C
x' = (read ∘ show) $ C :<: C :<: C
```

```
*Main> x'
(C :<: C) :<: C
```

Symptoms

```
data T1 = T1 :<: T1
        | T1 :>: T1
        | C
        deriving (Read, Show)
infixl 5 :<:
infixr 6 :>:
x, x′, x″ :: T1
x  = C :<: C :<: C
x′ = (read ∘ show) $ C :<: C :<: C
x″ = read "C :<: C :<: C"
```

```
*Main> x''
*** Exception: Prelude.read: no parse
```

Symptoms

```
data T1 = T1 :<: T1
        | T1 :>: T1
        | C
        deriving (Read, Show)
infixl 5 :<:
infixr 6 :>:
x, x', x'' :: T1
x   = C :<: C :<: C
x'  = (read ∘ show) $ C :<: C :<: C
x'' = read "C :<: C :<: C"
```

Ideally, you should be able to *read* every valid constant expression.

Parentheses

```
*Main> time (read "C" :: T1)
C
CPU Time: 0 ms
```

Parentheses

```
*Main> time (read "C" :: T1)
C
CPU Time: 0 ms

*Main> time (read "(((((C)))))" :: T1)
C
CPU Time: 74 ms
```

Parentheses

```
*Main> time (read "C" :: T1)
C
CPU Time: 0 ms

*Main> time (read "(((((C)))))" :: T1)
C
CPU Time: 74 ms

*Main> time (read "((((((C))))))" :: T1)
C
CPU Time: 389 ms
```

Parentheses

```
*Main> time (read "C" :: T1)
C
CPU Time: 0 ms

*Main> time (read "(((((C)))))" :: T1)
C
CPU Time: 74 ms

*Main> time (read "((((((C))))))" :: T1)
C
CPU Time: 389 ms

*Main> time (read "(((((((C)))))))" :: T1)
C
CPU Time: 1753 ms
```

Breadth-first Parsing

The language which is actually recognised by the generated *read* function is described by the non left-recursive grammar:

$$\begin{array}{rcl|l}
\mathit{T1}\ (n) & \rightarrow & \mathit{T1}\ (6)\ \texttt{":<:"}\ \mathit{T1}\ (6) & (n \leqslant 5) \\
 & \mid & \mathit{T1}\ (7)\ \texttt{":>:"}\ \mathit{T1}\ (7) & (n \leqslant 6) \\
 & \mid & \texttt{"C"} & (n \leqslant 10) \\
 & \mid & \texttt{"("}\ \mathit{T1}\ (0)\ \texttt{")"} & (n \leqslant 10)
\end{array}$$

Breadth-first Parsing

The language which is actually recognised by the generated $read$ function is described by the non left-recursive grammar:

$$\begin{array}{rcl|l}
\mathit{T1}\ (n) & \rightarrow & \mathit{T1}\ (6)\ \texttt{":<:"}\ \mathit{T1}\ (6) & (n \leqslant 5) \\
 & \mid & \mathit{T1}\ (7)\ \texttt{":>:"}\ \mathit{T1}\ (7) & (n \leqslant 6) \\
 & \mid & \texttt{"C"} & (n \leqslant 10) \\
 & \mid & \texttt{"("}\ \mathit{T1}\ (0)\ \texttt{")"} & (n \leqslant 10)
\end{array}$$

Three parallel parsers are started up for the first `'('`, and so on recursively.

Common Left-factors

Unfortunately the problem also shows up for more reasonable expressions such as $C :>: (C :>: (C :>: ...))$.

We remove the conditions, and encode them in the non-terminals.

$$\begin{array}{lcl}
\mathit{T1}\ (0 \mathinner{..} 5) & \to & \mathit{T1}\ (6)\ \texttt{":<:"}\ \mathit{T1}\ (6) \mid \mathit{T1}\ (6) \\
\mathit{T1}\ (6) & \to & \mathit{T1}\ (7)\ \texttt{":>:"}\ \mathit{T1}\ (7) \mid \mathit{T1}\ (7) \\
\mathit{T1}\ (7 \mathinner{..} 10) & \to & \texttt{"C"} \\
 & \mid & \texttt{"("}\ \mathit{T1}\ (0)\ \texttt{")"}
\end{array}$$

Common Left-factors

Unfortunately the problem also shows up for more reasonable expressions such as $C :>: (C :>: (C :>: ...))$.

We remove the conditions, and encode them in the non-terminals.

$$\begin{array}{lcl}
\mathit{T1}\ (0\,..\,5) & \rightarrow & \mathit{T1}\ (6)\ \texttt{":<:"}\ \mathit{T1}\ (6) \mid \mathit{T1}\ (6) \\
\mathit{T1}\ (6) & \rightarrow & \mathit{T1}\ (7)\ \texttt{":>:"}\ \mathit{T1}\ (7) \mid \mathit{T1}\ (7) \\
\mathit{T1}\ (7\,..\,10) & \rightarrow & \texttt{"C"} \\
 & \mid & \texttt{"("}\ \mathit{T1}\ (0)\ \texttt{")"}
\end{array}$$

We see that some alternatives start with the same non-terminal symbol.

The Problem

- Derived $read$ functions treat all operators as being *non-associative*, despite their declared associativities and precedences.

The Problem

- Derived $read$ functions treat all operators as being *non-associative*, despite their declared associativities and precedences.
- Derived $show$ functions generate the needed extra parentheses, in order to make $read \circ show$ work.

The Problem

- Derived $read$ functions treat all operators as being *non-associative*, despite their declared associativities and precedences.
- Derived $show$ functions generate the needed extra parentheses, in order to make $read \circ show$ work.
- These extra parentheses make parsing take exponential time.

The Problem

- Derived $read$ functions treat all operators as being *non-associative*, despite their declared associativities and precedences.
- Derived $show$ functions generate the needed extra parentheses, in order to make $read \circ show$ work.
- These extra parentheses make parsing take exponential time.
- Common left-factors have a similar effect.

How Does the Problem Arise?

```
infix 5 :+:
infix 6 :*:
data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
         deriving Read
t2 :: T2 (T2 Int)
t2 = read "(3 :*: C2) :*: C2"
```

The function *read* is a member of the class *Read*:

How Does the Problem Arise?

```
infix 5 :+:
infix 6 :*:
data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
          deriving Read
t2 :: T2 (T2 Int)
t2 = read "(3 :*: C2) :*: C2"
```

The function *read* is a member of the class *Read*:

- *read* functions are elements in dictionaries

How Does the Problem Arise?

```
infix 5 :+:
infix 6 :*:
data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
         deriving Read
t2 :: T2 (T2 Int)
t2 = read "(3 :*: C2) :*: C2"
```

The function *read* is a member of the class *Read*:

- *read* functions are elements in dictionaries
- **instance**-declarations compose new dictionaries out of existing dictionaries at run-time

How Does the Problem Arise?

```
infix 5 :+:
infix 6 :*:
data T2 a = T2 a :+: T2 a
            | a    :*: T2 a
            | C2
          deriving Read
t2 :: T2 (T2 Int)
t2 = read "(3 :*: C2) :*: C2"
```

The function *read* is a member of the class *Read*:

- *read* functions are elements in dictionaries
- **instance**-declarations compose new dictionaries out of existing dictionaries at run-time
- hence *read* functions are to be composed at run-time

The Bad News

- Bottom-up parsers do not compose at all, and all perform an analysis of the complete grammar (YACC, Happy)
- Top-down parsers do not compose efficiently for arbitrary grammars, and may lead to left-recursive parsers if no care is taken:

```
data T1 a = a        :*: Int deriving Read
data T2   = T1 T2 :+: Int deriving Read
```

The Bad and the Good News

- Bottom-up parsers do not compose at all, and all perform an analysis of the complete grammar (YACC, Happy)
- Top-down parsers do not compose efficiently for arbitrary grammars, and may lead to left-recursive parsers if no care is taken:

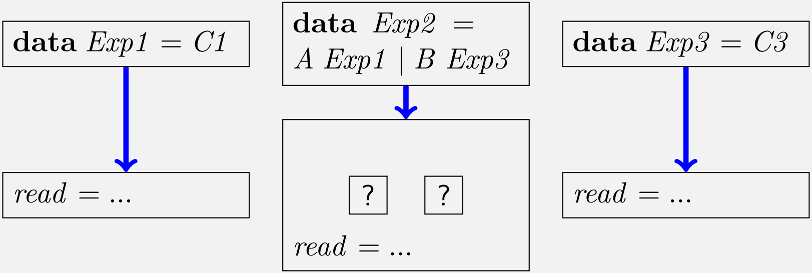
```
data T1 a = a       :*: Int deriving Read
data T2   = T1 T2 :+: Int deriving Read
```

- Grammars can be composed!

Using Grammars instead of Parsers

$$\boxed{\textbf{data}\ \mathit{Exp1} = \mathit{C1}} \quad \boxed{\begin{array}{l}\textbf{data}\ \mathit{Exp2}\ = \\ A\ \mathit{Exp1} \mid B\ \mathit{Exp3}\end{array}} \quad \boxed{\textbf{data}\ \mathit{Exp3} = \mathit{C3}}$$

Using Grammars instead of Parsers



$\textbf{data}\ \mathit{Exp1} = \mathit{C1}$

$\textbf{data}\ \mathit{Exp2} = A\ \mathit{Exp1} \mid B\ \mathit{Exp3}$

$\textbf{data}\ \mathit{Exp3} = \mathit{C3}$

$\mathit{read} = \ldots$

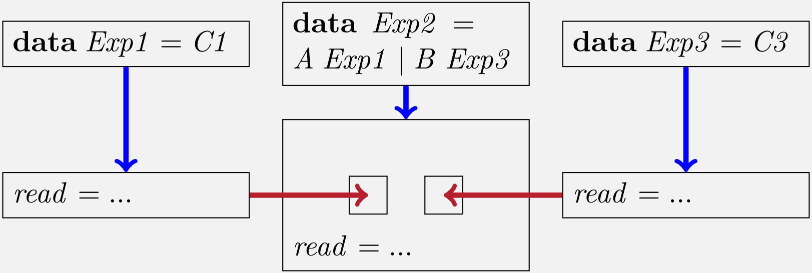
? ?

$\mathit{read} = \ldots$

$\mathit{read} = \ldots$

derive

Using Grammars instead of Parsers



data *Exp1* = *C1*

data *Exp2* =
A Exp1 | *B Exp3*

data *Exp3* = *C3*

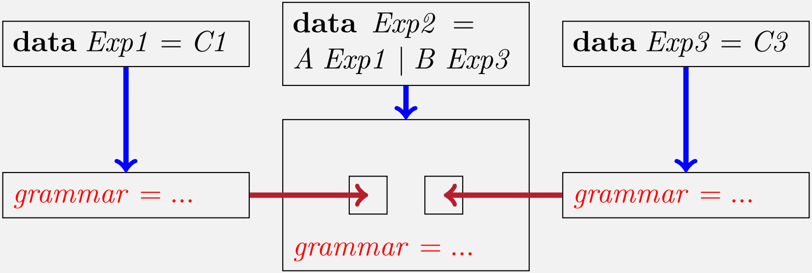
read = …

read = …

read = …

derive parameterise

Using Grammars instead of Parsers



data $Exp1 = C1$

data $Exp2 =$ $A\ Exp1 \mid B\ Exp3$

data $Exp3 = C3$

$grammar = \ldots$

$grammar = \ldots$

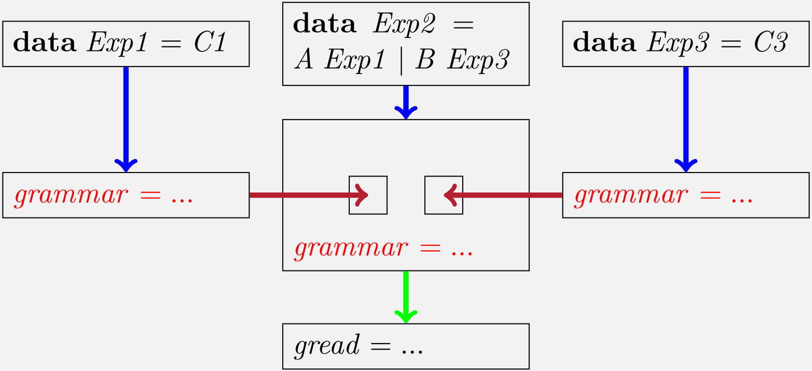
$grammar = \ldots$

derive parameterise

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Using Grammars instead of Parsers



data $Exp1 = C1$

data $Exp2 = A\ Exp1 \mid B\ Exp3$

data $Exp3 = C3$

$grammar = \ldots$

$grammar = \ldots$

$grammar = \ldots$

$gread = \ldots$

derive parameterise generate

The Class *Gram*

Instead of the class *Read* we introduce:

```
class Gram a where
  grammar :: DGrammar a
```

where *DGrammar* is a data type describing grammatical structures, including information about precedences.

The Class *Gram*

Instead of the class *Read* we introduce:

$$\begin{array}{l}\textbf{class}\ \mathit{Gram}\ a\ \textbf{where}\\ \quad \mathit{grammar} :: \mathit{DGrammar}\ a\end{array}$$

where *DGrammar* is a data type describing grammatical structures, including information about precedences.

Note that it is labelled with a type a, which is the data type described by a value of type $\mathit{DGrammar}\ a$.

The Class *Gram*

Instead of the class *Read* we introduce:

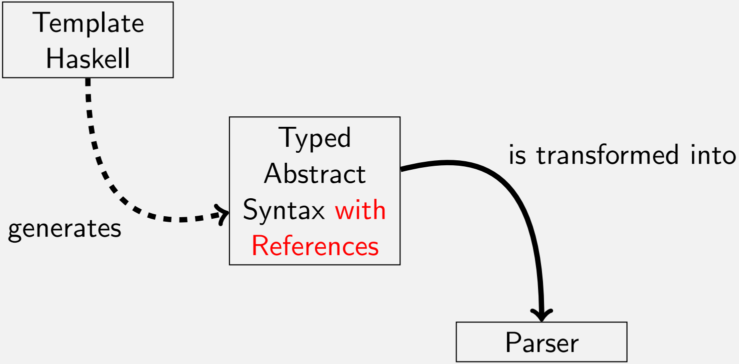
```
class Gram a where
   grammar :: DGrammar a
```

where *DGrammar* is a data type describing grammatical structures, including information about precedences.

Note that it is labelled with a type *a*, which is the data type described by a value of type *DGrammar a*.
Now we can, just as for *read* define:

```
read  :: Read a  ⇒ String → a
gread :: Gram a ⇒ String → a
```

Generating parsers from Data Types



Template Haskell

generates

Typed Abstract Syntax with References

is transformed into

Parser

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

The Steps to be Taken

group Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.

The Steps to be Taken

group	Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.
leftcorner	Remove possible left-recursion by applying the *Left-Corner Transform*

The Steps to be Taken

group	Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.
leftcorner	Remove possible left-recursion by applying the *Left-Corner Transform*
leftfactoring	Combine common prefixes of alternatives

The Steps to be Taken

group	Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.
leftcorner	Remove possible left-recursion by applying the *Left-Corner Transform*
leftfactoring	Combine common prefixes of alternatives
compile	Map the resulting $Grammar$ onto a parser

The Steps to be Taken

group	Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.
leftcorner	Remove possible left-recursion by applying the *Left-Corner Transform*
leftfactoring	Combine common prefixes of alternatives
compile	Map the resulting $Grammar$ onto a parser
parse	Use the parser to read a value

The Steps to be Taken

group	Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.
leftcorner	Remove possible left-recursion by applying the *Left-Corner Transform*
leftfactoring	Combine common prefixes of alternatives
compile	Map the resulting $Grammar$ onto a parser
parse	Use the parser to read a value

The Steps to be Taken

group	Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.
leftcorner	Remove possible left-recursion by applying the *Left-Corner Transform*
leftfactoring	Combine common prefixes of alternatives
compile	Map the resulting $Grammar$ onto a parser
parse	Use the parser to read a value

$$\begin{aligned} &gread :: Gram\ a \Rightarrow String \rightarrow a \\ &gread = (parse \circ compile \quad \circ leftfactoring \\ &\qquad\qquad\qquad \circ leftcorner \circ group)\ grammar \end{aligned}$$

Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals

Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals
- We want to be able to inspect and transform the grammar

Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals
- We want to be able to inspect and transform the grammar
- We want to inspect the underlying graph structure

Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals
- We want to be able to inspect and transform the grammar
- We want to inspect the underlying graph structure
- Of which the nodes are labelled with different types

Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals
- We want to be able to inspect and transform the grammar
- We want to inspect the underlying graph structure
- Of which the nodes are labelled with different types
- So we use heterogeneous collections, i.e. we use nested cartesian products, henceforth called Env-ironments

References and Environments I

We introduce natural numbers, labelled with a type a describing what is referred to, and a list of types env describing the structure in which this a labelled object lives:

```
data Ref a env where
  Zero ::                 Ref a (a, env)
  Suc  :: Ref a env′ → Ref a (x, env′)
data Equal a b where
  Eq :: Equal a a
match :: Ref a env → Ref b env → Maybe (Equal a b)
match Zero    Zero    = Just Eq
match (Suc x) (Suc y) = match x y
match _       _       = Nothing
```

References and Environments II

```
data Env t use def where
  Empty :: Env t use ()
  Cons  :: t a use → Env t use        def'
                   → Env t use (a, def')
```

$t\ a\ use$ :: a term of type t, describing a value of type a contains references pointing into an environment labelled by use. The parameter def describes the values actually existing in the Env. If use equals def the environment is closed.

DGrammar

```
data DGrammar a
   = ∀env.DGrammar (Ref a env)
                   (Env DGram env env)
data DGram a env = DGD (DLNontDefs a env)
                 | DGG (DGrammar a)
```

```
newtype DRef a env = DRef (Ref a env, Int)
```

```
newtype DLNontDefs a env
   = DLNontDefs [(DRef a env, DProductions a env)]
```

Continued ..

```
newtype DProductions a env
    = DPS{unDPS :: [DProd a env]}
```

```
data DProd a env where
    DSeq  :: DSymbol b env → DProd (b → a) env
                           → DProd a env
    DEnd :: a              → DProd a env
```

```
data DSymbol a env where
    DNont  :: DRef  a env → DSymbol a env
    DTerm :: Token        → DSymbol Token env
```

Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[
              _Exp                 ":+:"
              _Exp


,
             "C"




]
```

Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[
                         _Exp                ":+:"
                         _Exp


,
                         "C"
       ,                 "("              _Exp
                         ")"


]
```

Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[
               dNont (_Exp   ) .#. dTerm ":+:" .#.
               dNont (_Exp   )


,
               dTerm "C"
             , dTerm "(" .#. dNont (_Exp   ) .#.
               dTerm ")"

]
```

Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[
      DPS [dNont (_Exp, 6) .#. dTerm ":+:" .#.
           dNont (_Exp, 7)                    ]

,
      DPS [dTerm "C"
          , dTerm "(" .#. dNont (_Exp, 0) .#.
            dTerm ")"                       ]

]
```

Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[   (DRef (_Exp, 6)
    , DPS [dNont (_Exp, 6) .#. dTerm ":+:" .#.
           dNont (_Exp, 7)                    ]
    )
,   (DRef (_Exp, 10)
    , DPS [dTerm "C"
          , dTerm "(" .#. dNont (_Exp, 0) .#.
           dTerm ")"                        ]
    )
]
```

Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
  [   (DRef (_Exp, 6)
      , DPS [dNont (_Exp, 6) .#. dTerm ":+:" .#.
             dNont (_Exp, 7) .#. dEnd plus]
      )
  ,   (DRef (_Exp, 10)
      , DPS [dTerm "C" .#. dEnd (const C)
            ,dTerm "(" .#. dNont (_Exp, 0) .#.
             dTerm ")" .#. dEnd parenT]
      )
  ]
plus e1 _ e2 = e2 :+: e1
```

Typed Abstract Syntax

```
instance Gram Exp where
  grammar = DGrammar _0 envExp
envExp :: Env DGram (Exp, ()) (Exp, ())
envExp = consD (nonts _0) Empty
  where
     nonts _Exp = DLNontDefs
       [    (DRef (_Exp, 6)
            , DPS [dNont (_Exp, 6) .#. dTerm ":+:" .#.
                   dNont (_Exp, 7) .#. dEnd plus]
            )
       ,    (DRef (_Exp, 10)
            , DPS [dTerm "C" .#. dEnd (const C)
                  , dTerm "(" .#. dNont (_Exp, 0) .#.
                   dTerm ")" .#. dEnd parenT]
            )
       ]
     plus e1 _ e2 = e2 :+: e1
```

An Intermediate result

$$
\begin{array}{lcl}
A & \rightarrow & \texttt{"C1"}\ A\_C1 \mid \texttt{"("}\ A\_(\\
A\_A & \rightarrow & \texttt{":<:"}\ B\ A\_A \mid \texttt{":<:"}\ B \\
A\_B & \rightarrow & A\_A \mid \epsilon \\
A\_C & \rightarrow & \texttt{":>:"}\ B\ A\_B \mid A\_B \\
A\_C1 & \rightarrow & A\_C \\
A\_(& \rightarrow & A\ \texttt{")"}\ A\_C \\
B & \rightarrow & \texttt{"C1"}\ B\_C1 \mid \texttt{"("}\ B\_(\\
B\_C & \rightarrow & \texttt{":>:"}\ B \mid \epsilon \\
B\_C1 & \rightarrow & B\_C \\
B\_(& \rightarrow & A\ \texttt{")"}\ B\_C \\
C & \rightarrow & \texttt{"C1"}\ C\_C1 \mid \texttt{"("}\ C\_(\\
C\_C1 & \rightarrow & \epsilon \\
C\_(& \rightarrow & A\ \texttt{")"}
\end{array}
$$

1. We have introduced new non-terminals
2. Old non-terminals have new productions

The Transformations

All the transformations can be expressed in terms of an arrow-like type:

```
data Trafo m t a b =
   Trafo (∀env1.m env1 →
            ∃env2.
             (m env2
             ,∀s.   a s → T env2 s → Env t s env1 →
                   (b s,   T env1 s  ,  Env t s env2)
             )
```

Results I

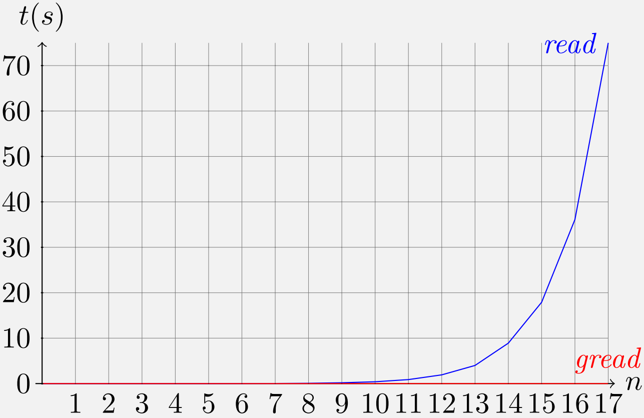
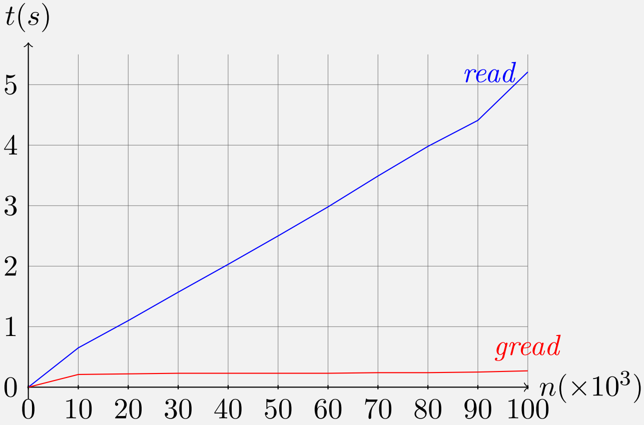


Figure: Execution times of reading $C :> (C :>: ...)$

Reading a Large Data Type



Overhead is very small, and that thanks to the use of the UU-parsers also parse times do hardly increase.

Why is this so complicated ...

1. The problem is complicated

Why is this so complicated ...

1. The problem is complicated
2. We do in 350 lines more than Bison (10.000 lines) is doing

Why is this so complicated ...

1. The problem is complicated
2. We do in 350 lines more than Bison (10.000 lines) is doing
3. Extra constructors are needed because we need existentials

Why is this so complicated ...

1. The problem is complicated
2. We do in 350 lines more than Bison (10.000 lines) is doing
3. Extra constructors are needed because we need existentials
4. If we have lazy evaluation, we also want it at the type level!

Why is this so complicated …

1. The problem is complicated
2. We do in 350 lines more than Bison (10.000 lines) is doing
3. Extra constructors are needed because we need existentials
4. If we have lazy evaluation, we also want it at the type level!

$$\begin{array}{l} f :: \forall a.(a \rightarrow \exists b\ (b, a, b \rightarrow b \rightarrow Int)) \\ \mathbf{let}\ (b, a, g) = f\ \ b \\ \mathbf{in}\ \ g\ \ b\ \ a \end{array}$$

To Take Home

- The transformation library has been used unmodified for all the transformations
- The library can be used for any collection of abstract syntax trees, which contain references to each other, and of which the structure has to be inspected