Type-safe self-inspecting code

Arthur I. Baars and S. Doaitse Swierstra

Utrecht University
{arthurb,doaitse}@cs.uu.nl

November 24, 2004
Outline

Domain-Specific-Languages

Combinator Parsers

Automatic left-recursion detection and removal
  Typed abstract syntax

Constructing Grammars

Syntax Macros

Conclusions
Domain Specific Languages

A DSL is:
A programming language tailored for a particular application domain, which captures precisely the semantics of the application domain.

DSL examples:
- Lex / Yacc (lexing and parsing)
- \LaTeX (for document mark-up)
- Tcl/Tk (GUI scripting)
- MatLab (numerical computations)
Domain Specific Languages

Advantages:
  ▶ Using a DSL, programs:
    • are easier to understand
    • are quicker to write
    • are easier to maintain
    • can be written by non-programmers

Disadvantages:
  ▶ High start-up cost
    • design and implementation of a new language is hard
  ▶ Lack of "general purpose" features (e.g. abstraction)
  ▶ Little tool support
Domain Specific *Embedded Languages*

Embed a DSL as a library in a general purpose host language. Advantages:

- inherit general purpose features from host language
  - abstraction mechanism
  - type system
- inherit compilers and tools
- good integration with host language
- many DSL’s can easily be used together
DSEL’s in Haskell

Haskell is a very suitable host for DSEL’s:

- Polymorphism
- Lazy evaluation
- Higher-order functions
- infix syntax
- list and monad comprehension
- type classes

Examples:

- Parser combinators
- Pretty printing libraries
- HaskellDB
- QuickCheck
- GUI libraries
- WASH/CGI
- Haskore
Combinator Parsers

Parser combinators are an embedding of BNF in Haskell:

\[
\begin{align*}
\text{infix} & \ 6 <\$>; \ \text{infix} \ 5 <\*>; \ \text{infix} \ 4 \ <\mid> \\
succeed & :: a \rightarrow \text{Parser} \ a \\
symbol & :: \text{Char} \rightarrow \text{Parser} \ \text{Char} \\
(\mid>) & :: \text{Parser} \ a \rightarrow \text{Parser} \ a \rightarrow \text{Parser} \ a \\
(<\*>) & :: \text{Parser} \ (a \rightarrow b) \rightarrow \text{Parser} \ a \rightarrow \text{Parser} \ b \\
\text{failp} & :: \text{Parser} \ a \\
(<\$>) & :: (a \rightarrow b) \rightarrow \text{Parser} \ a \rightarrow \text{Parser} \ b
\end{align*}
\]

For example the BNF grammar:

\[
S \rightarrow \ 'a' \ S \\
| \ \epsilon
\]

is written using combinators as:

\[
\begin{align*}
\text{s} & :: \text{Parser} \ \text{[Char]} \\
\text{s} & = (:) <\$> \text{symbol} \ 'a' <\*> \text{s} \\
& <\mid> \text{succeed} \ \text{[]} 
\end{align*}
\]
Combinator Parsers

A direct translation of:

\[
Expr \to Expr \ '-' \ Integer \\
| \ Expr \ '+' \ Integer \\
| \ Integer
\]

would be:

\[
expr :: Parser Integer \\
expr = (\lambda x \ y \to x - y) <\$> expr <\$> symbol ' - ' <\$> integer \\
<|> (\lambda x \ y \to x + y) <\$> expr <\$> symbol ' + ' <\$> integer \\
<|> integer
\]

\[
integer :: Parser Integer \\
integer = ... \\
\]

Unfortunately, combinator parsers cannot deal with left-recursion.
Combinator Parsers: \textit{chain}

Transformed grammar, without left-recursion:

\[
\text{Integer } (('+') \mid '-') \text{ Integer}\star
\]

Repetition combinator:

\[
\text{many} :: \text{Parser } a \rightarrow \text{Parser } [a]
\]
\[
\text{many } p = (:) \ <$ > p \ <* > \text{ many } p < | > \text{ succeed } [ ]
\]

Chain combinator, parses as right-recursion, and converts the result to associate left

\[
\text{chain} :: \text{Parser } (a \rightarrow a \rightarrow a) \rightarrow \text{Parser } a \rightarrow \text{Parser } a
\]
\[
\text{chain } op \ p = f \ <$ > p \ <* > \text{ many } (\text{flip } <$ > op <* > p)
\]
\[
\text{ where } f \ e \ fs = \text{ foldl } (\text{flip } (\$)) \ e \ fs
\]

Revised expr:

\[
\text{expr} :: \text{Parser } \text{ Integer}
\]
\[
\text{expr} = \text{chain } op \ \text{integer}
\]
\[
\text{ where } op = \text{ const } (\text{\texttt{-}}) \ <$ > \text{ symbol } \text{'}-\text{'}
\]
\[
< | > \text{ const } (\text{\texttt{+}}) \ <$ > \text{ symbol } \text{'}+\text{'}
\]
Combinator Parsers: chain

The grammar can easily be extended with \( \ast \) and \( / \), by reusing the chain combinator:

\[
\begin{align*}
\text{expr} &:: \text{Parser Integer} \\
\text{expr} &= \text{chain \( \text{op} \) term} \\
\text{where} \quad \text{op} &= \text{const \( (\quad) \langle\rangle \text{symbol} \;\,'-\,' \rangle} \\
&\quad <\mid> \text{const \( (\quad) \langle\rangle \text{symbol} \;\,'+\,' \rangle}
\end{align*}
\]

\[
\begin{align*}
\text{term} &:: \text{Parser Integer} \\
\text{term} &= \text{chain \( \text{op} \) integer} \\
\text{where} \quad \text{op} &= \text{const \( \text{div} \langle\rangle \text{symbol} \;\,'/\,' \rangle} \\
&\quad <\mid> \text{const \( (\quad) \langle\rangle \text{symbol} \;\,'\ast\,' \rangle}
\end{align*}
\]
Combinator Parsers

Often left-factoring is needed to gain reasonably efficient parsers. For example:

```
S → 'a' S  
   | 'a' 'b' S
```

can be rewritten to:

```
S → 'a' (S | 'b' S)
```

For more complex grammars, left-factoring can be very tedious:

```
stat → pat '<-' exp  
     | exp  
     | 'let' decls
```
Combinator Parsers

Combinator parsers

- good integration with host language
- mimic BNF-syntax in Haskell using operators
- reuse through abstraction mechanism
- many cases of left-recursion can be rewritten using chain combinator
- complex forms of left-recursion (e.g. in mutually recursive definitions) are hard to remove
- left-factoring is often needed, and can be very tedious
Combinator Parsers

Parser Generator, such as Happy
- separate tool
- BNF like syntax
- lack abstraction mechanism, so lots of code duplication
- analyse grammar
  - perform factoring automatically
  - remove left-recursion automatically
Automatic left-recursion detection and removal

Example grammar:

\[ p \rightarrow q \ 'a' \]
\[ q \rightarrow p \ 'b' \]
\[ | \ 'c' \]

Is written using combinators as follows:

\[ p, q :: Parser [Char] \]
\[ p = flip (: ) <$> q <*> symbol 'a' \]
\[ q = flip (: ) <$> p <*> symbol 'b' \]
\[ <| ( : [ ]) <$> symbol 'c' \]
Automatic left-recursion detection and removal

Inlining $q$ in $p$ leads to:

$$
\begin{align*}
p &= (\lambda xs \ b \ a \rightarrow a : b : xs) \ <\> p \ <\> symbol \ 'b' \ <\> symbol \ 'a' \\
    & \ | \ (\lambda c \ a \rightarrow [c, a]) \ <\> symbol \ 'c' \ <\> symbol \ 'a'
\end{align*}
$$

The parser $p$ is left-recursive, so we need to remove the left-recursion:

$$
p = f \ <\> non\_left \ <\> many\_left
$$

where $non\_left = (\lambda c \ a \rightarrow [c, a]) \ <\> symbol \ 'c' \ <\> symbol \ 'a'$

$$
left = (\lambda b \ a \ xs \rightarrow a : b : xs) \ <\> symbol \ 'b' \ <\> symbol \ 'a'$

$$
f \ x \ fs = foldl \ (flip \ ($)) \ x \ fs
$$
Automatic left-recursion detection and removal

- Type preserving transformations
  - explicit representation of parsers: typed abstract syntax
- Make calls to other parser observable
  - Custom fix-point operator instead of normal recursion
  - Explicit representation of references
Typed abstract syntax

Datatype for representing parsers; the variable $t$ labels each parser with its type.

```haskell
data Parser t = Succeed t
  `|` Symbol Char
  `|` Choice (Parser t) (Parser t)
  `|` $\exists x.\text{Seq} (Parser (x \rightarrow t)) (Parser x)$
  `|` Fail
  `|` $\exists x.\text{Many} (Parser x)$
```

```haskell
wrong = Succeed (+1) 'Seq' Symbol 'a'
```

The type of `Symbol 'a'` is too general, it should be `Parser Char` instead of `Parser t`. 
Typed abstract syntax: smart constructors

Define smart constructors, that make sure only type-correct parser-terms can be constructed:

\[
\begin{align*}
\text{succeed} & :: t \rightarrow \text{Parser } t \\
\text{succeed} &= \text{Succeed} \\
\text{symbol} & :: \text{Char} \rightarrow \text{Parser } \text{Char} \\
\text{symbol} &= \text{Symbol} \\
(\langle|\rangle) & :: \text{Parser } t \rightarrow \text{Parser } t \rightarrow \text{Parser } t \\
(\langle|\rangle) &= \text{Choice} \\
(\langle*\rangle) & :: \text{Parser } (a \rightarrow t) \rightarrow \text{Parser } a \rightarrow \text{Parser } t \\
(\langle*\rangle) &= \text{Seq} \\
\text{failp} & :: \text{Parser } t \\
\text{failp} &= \text{Fail} \\
\text{many} & :: \text{Parser } t \rightarrow \text{Parser } [t] \\
\text{many} &= \text{Many}
\end{align*}
\]

Now \text{wrong} is rejected:

\[
\text{wrong} = \text{succeed } (+1) \langle*\rangle \text{ symbol 'a'}
\]
Suppose we have a real parser library is a module called $P$

\[
\text{eval} :: \text{Parser } a \rightarrow P.\text{Parser } a \\
\text{eval } (\text{Succeed } t) = P.\text{succeed } t \\
\text{eval } (\text{Symbol } c) = P.\text{symbol } c \quad --\text{ wrong, why?}
\]
Typed abstract syntax

The datatype \textit{Equal} is a proof that two types are equal. The only non-bottom value of this type is \texttt{self :: Equal a a}.

\begin{verbatim}
data Parser t =  Succeed t
  |  Symbol (Equal Char t) Char
  |  Choice (Parser t) (Parser t)
  |  \exists x. Seq (Parser (x \rightarrow t)) (Parser x)
  |  Fail
  |  \exists x. Many (Equal [x] t) (Parser x)

symbol = Symbol self
many    = Many self
\end{verbatim}
Compiling Parsers, second try

Suppose we have a real parser library is a module called \( P \)

\[
\text{castF} :: \text{Equal } a \ b \rightarrow (f \ a \rightarrow f \ b)
\]

\[
\text{eval} :: \text{Parser } a \rightarrow P.\text{Parser } a
\]
\[
\text{eval} (\text{Succeed } t) = P.\text{succeed } t
\]
\[
\text{eval} (\text{Symbol } eq \ c) = \text{castF } eq (P.\text{symbol } c)
\]
\[
\text{eval} (\text{Choice } p \ q) = \text{eval } p P. <\|> \text{eval } q
\]
\[
\text{eval} (\text{Seq } p \ q) = \text{eval } p P. <\ast> \text{eval } q
\]
\[
\text{eval Fail} = \text{failp}
\]
\[
\text{eval Many } eq \ p = \text{castF } eq (P.\text{many } p)
\]
Equality Type

newtype Equal a b = Eq (forall f. f a → f b)

self :: Equal a a
symm :: Equal a b → Equal b a
trans :: Equal a b → Equal b c → Equal a c
pairParts :: Equal (a, b) (c, d) → (Equal a c, Equal b d)

cast :: Equal a b → (a → b)
castF :: Equal a b → (f a → f b)
data Ref env a
   = ∃ env'.Zero (Equal env (a, env'))
   | ∃x env'.Suc (Equal env (x, env')) (Ref env' a)

As before we define smart constructors that pass the equality proof self to the corresponding constructor functions:

code
zero :: Ref (a, env) a
zero = Zero self
suc :: Ref env' a → Ref (b, env') a
suc = Suc self

The number of Suc-nodes in a reference determines to which value in the environment a reference points.
Two arbitrary references can be compared for equality as long as they point into environments that are labeled with the same sequence of types.

\[
\text{equalRef} :: \text{Ref env } a \rightarrow \text{Ref env } b \rightarrow \text{Maybe (Equal } a \ b\text{)}
\]

\[
\text{equalRef} \ (\text{Zero eq1}) \ (\text{Zero eq2}) \ = \ \text{let} \ (eq, _) = \text{pairParts} \ (\text{inv eq1 'trans' eq2}) \\
\text{in} \ \text{Just eq}
\]

\[
\text{equalRef} \ (\text{Suc eq1 ref1}) \ (\text{Suc eq2 ref2}) \ = \ \text{let} \ (_, eq) = \text{pairParts} \ (\text{inv eq1 'trans' eq2}) \\
\text{in} \ \text{equalRef} \ (\text{cast (subF2 eq self) ref1}) \ \text{ref2}
\]

\[
\text{equalRef} \ _ \ _ \ = \ \text{Nothing}
\]
A grammar is a collection of parsers. These parser may contain references to parsers in the grammar. Using nested pairs leads to infinite types.

```
grammar = (flip (:)) <$> (NT suc zero) <*> symbol 'a'
 ,
   flip (:) <$> NT zero <*> symbol 'b'
   <[|> ([:]]) <$> symbol 'c'
)
**Grammars**

```haskell
-- smart constructors
empty :: Env f ()
empty = EMPTY

infixr 1 'ext'

ext :: f a → Env f env → Env f (a, env)
ext = EXT self
```

```haskell
derefEnv :: Ref env a → (Env f env → f a)

type Grammar env = Env (Parser env) env
```
grammars

grammar :: Grammar ([Char], ([Char], ()))
grammar = flip (: <$> (NT suc zero) <*> symbol 'a'
  'ext'
  flip (: <$> NT zero <*> symbol 'b'
  <|> [: []] <$> symbol 'c'
  'ext'
  empty
Compiling Grammars

\[
\text{compileGrammar} :: \text{Grammar env} \to \text{Env P.Parser env}
\]

\[
\text{compileGrammar gram} =
\begin{align*}
&\text{let parsers} = \text{mapEnv (compile parsers) gram} \\
&\text{in parsers}
\end{align*}
\]

\[
\text{mapEnv :: (forall a.f a \to g a) \to Env f env \to Env g env}
\]

\[
\text{mapEnv f EMPTY} = \text{EMPTY}
\]

\[
\text{mapEnv f (EXT eq x rest)} = \text{EXT eq (f x) (mapEnv f rest)}
\]
Compiling Grammars

```
compile :: Env P.Parser env
   → Parser env a
   → P.Parser a
compile parsers prod =
   case prod of
     NT ref     → derefEnv ref parsers
     Choice p q → comp pP. <|> comp q
     Symbol eq c → castF eq (P.symbol c)
     Succeed x  → P.succeed x
     Seq p q    → comp pP. <*> comp q
     Many eq p  → castF eq (P.many (comp p))
     Fail       → P.failp
   where comp = compile parsers
```
Summary

- typed abstract syntax
- explicit references
- compile grammars to real parsers
- use of equality data type can be tedious
  - but our transformations are guaranteed to be type preserving
- loss of notational convenience when writing grammars using \textit{zero}, and \textit{suc}
infixr 1 'andalso'

example :: Grammar ([Char], ([Char], ()))
example =
  fixRefs
  (λ∼(p, (q, _)) →
   flip (:) <$> q <*> symbol 'a'
   'andalso'
   flip (:) <$> p <*> symbol 'b'
   <$> (:[ ]) <$> symbol 'c'
   'andalso'
   done
  )
Maybe **mdo** can help?

```
example = mdo p ← flip (:) <$> q <*> symbol 'a'
         q ← flip (:) <$> p <*> symbol 'b' <|> ([:]) <$> symbol 'c'
         ...
```

Unfortunately, our grammars are not a *Monad*. The type of the state grows whenever we bind another parser.
Invent our own syntax

Inventing better syntax can make a combinator library much easier to use. For example Arrow syntax for the Arrows library.

```
example :: Grammar ([Char], ([Char], ()))
example = grammar
    p ← flip (:) <$> q <*> symbol 'a';
    q ← flip (:) <$> p <*> symbol 'b' <|> [ ];
    [ ] <$> symbol 'c';
```

Hard-wiring special syntax into a compiler for parser combinators does not make sense.
Syntax Macros

- Generic mechanism to extend a programming language
- Defines a mapping of new concrete syntax into the core language
- Language extension modules can be loaded to facilitate a combinator library with a domain specific notation
Syntax Macros

nonterminals:
varid  :: String -- from Haskell Report
exp    :: Exp    -- from Haskell Report
exp10  :: Exp    -- from Haskell Report
prods  :: (Pat, Exp)

The new notation of grammar expressions is defined by the following macro rules:

rules:
exp10 ::= "grammar" (ids,ps)=prods
        => [| fixRefs (\ ~ $ids -> $ps ) |]

prods ::= => (WildP, [|done|])
prods ::= v=varid "<-" e=exp ";" (ids,ps)=prods
        => let var = VarP v
           in ( [|p ($var, $ids) |]
               , [| $e 'andalso' $ps|]
               )
Conclusions

- Typed abstract syntax
  - Represent programming structures that contain internal references,
- Custom fix-point operator instead of normal recursion
  - Programs inspect their own call-graph
- Not only for combinator parsers
- Type preserving transformations
- Syntax Macros
  - provide convenient notation
  - would make Haskell an even better tool for implementing DSEL’s
Related Work

A. Baars and D. Swierstra. Syntax macros. 
http://www.cs.uu.nl/groups/ST/Center/SyntaxMacros.

A. I. Baars and S. D. Swierstra. Typing dynamic typing.

L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping.

J. Cheney and R. Hinze. First-Class Phantom Types.

E. Pasalic and N. Linger. Meta-programming with typed object-language representations.