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:

```
infix 6 <$>; infix 5 <*>; infix 4 <|>
succeed :: a            → Parser a
symbol  :: Char         → Parser Char
(<|>)   :: Parser a            → Parser a → Parser a
(<*>)   :: Parser (a → b)      → Parser a → Parser b
failp   :: Parser a
(<$>)   :: (a → b)             → Parser a → Parser b
```

For example the BNF grammar:

$$S \rightarrow \texttt{'a'}\ S \mid \epsilon$$

is written using combinators as:

```
s :: Parser [Char]
s = (:) <$> symbol 'a' <*> s
  <|> succeed []
```

Combinator Parsers

A direct translation of:

$$
\begin{aligned}
Expr &\rightarrow Expr\ \texttt{'-'}\ Integer \\
&\mid\ Expr\ \texttt{'+'}\ Integer \\
&\mid\ Integer
\end{aligned}
$$

would be:

$$
\begin{aligned}
expr &:: Parser\ Integer \\
expr &= (\lambda x\ \_\ y \rightarrow x - y)\ \texttt{<\$>}\ expr\ \texttt{<*>}\ symbol\ \texttt{'-'}\ \texttt{<*>}\ integer \\
&\quad \texttt{<|>}\ (\lambda x\ \_\ y \rightarrow x + y)\ \texttt{<\$>}\ expr\ \texttt{<*>}\ symbol\ \texttt{'+'}\ \texttt{<*>}\ integer \\
&\quad \texttt{<|>}\ integer \\
integer &:: Parser\ Integer \\
integer &= \ldots
\end{aligned}
$$

Unfortunately, combinator parsers cannot deal with left-recursion

Combinator Parsers: *chain*

Transformed grammar, without left-recursion:

```
Integer (('+' | '-') Integer) *
```

Repetition combinator:

```
many   :: Parser a → Parser [a]
many p = (:) <$> p <*> many p <|> succeed []
```

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

```
chain :: Parser (a → a → a) → Parser a → Parser a
chain op p = f <$> p <*> many (flip <$> op <*> p)
  where f e fs = foldl (flip ($)) e fs
```

Revised *expr*:

```
expr :: Parser Integer
expr = chain op integer
  where op  =  const (−) <$> symbol '-'
           <|> const (+) <$> symbol '+'
```

Combinator Parsers: *chain*

The grammar can easily be extended with $*$ and `/`, by reusing the *chain* combinator:

```
expr :: Parser Integer
expr = chain op term
  where op =   const (−) <$> symbol '-'
           <|> const (+) <$> symbol '+'
term :: Parser Integer
term = chain op integer
  where op =   const div <$> symbol '/'
           <|> const (∗) <$> symbol '*'
```

Combinator Parsers

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

$$\begin{aligned} S &\rightarrow \texttt{'a'}\ S \\ &\mid \texttt{'a'}\ \texttt{'b'}\ S \end{aligned}$$

can be rewritten to:

$$S \rightarrow \texttt{'a'}\ (S \mid \texttt{'b'}\ S)$$

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

$$\begin{aligned} \mathit{stat} &\rightarrow \mathit{pat}\ \texttt{'<-'}\ \mathit{exp} \\ &\mid \mathit{exp} \\ &\mid \texttt{'let'}\ \mathit{decls} \end{aligned}$$

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:

$$
\begin{aligned}
p &\rightarrow q\ \texttt{'a'} \\
q &\rightarrow p\ \texttt{'b'} \\
  &\mid\ \texttt{'c'}
\end{aligned}
$$

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:

```
p  =  (λxs b a → a : b : xs) <$> p <*> symbol 'b' <*> symbol 'a'
  <|> (λc a → [c, a])        <$> symbol 'c' <*> symbol 'a'
```

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

```
p = f <$> non_left <*> many left
  where non_left = (λc a → [c, a]) <$> symbol 'c' <*> symbol 'a'
        left     = (λb a xs → 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.

$$
\begin{aligned}
\textbf{data}\ \mathit{Parser}\ t = {} & \mathit{Succeed}\ t \\
\mid {} & \mathit{Symbol}\ \mathit{Char} \\
\mid {} & \mathit{Choice}\ (\mathit{Parser}\ t)\ (\mathit{Parser}\ t) \\
\mid {} & \exists x.\mathit{Seq}\ (\mathit{Parser}\ (x \to t))\ (\mathit{Parser}\ x) \\
\mid {} & \mathit{Fail} \\
\mid {} & \exists x.\mathit{Many}\ (\mathit{Parser}\ x)
\end{aligned}
$$

$$
\mathit{wrong} = \mathit{Succeed}\ (+1)\ \text{`}\mathit{Seq}\text{`}\ \mathit{Symbol}\ \texttt{'a'}
$$

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:

```
succeed :: t → Parser t
succeed = Succeed
symbol  :: Char → Parser Char
symbol  = Symbol
(<|>)   :: Parser t → Parser t → Parser t
(<|>)   = Choice
(<*>)   :: Parser (a → t) → Parser a → Parser t
(<*>)   = Seq
failp   :: Parser t
failp   = Fail
many    :: Parser t → Parser [t]
many    = Many
```

Now *wrong* is rejected:

```
wrong = succeed (+1) <*> symbol 'a'
```

Compiling Parsers

Suppose we have a real parser library is a module called P

$$
\begin{aligned}
&eval :: Parser\ a \longrightarrow P.Parser\ a \\
&eval\ (Succeed\ t) = P.succeed\ t \\
&eval\ (Symbol\ c)\ = P.symbol\ c \quad \text{-- wrong, why?}
\end{aligned}
$$

Typed abstract syntax

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

$$
\begin{aligned}
\textbf{data}\ \mathit{Parser}\ t = &\quad \mathit{Succeed}\ t \\
| &\quad \mathit{Symbol}\ (\mathit{Equal}\ \mathit{Char}\ t)\ \mathit{Char} \\
| &\quad \mathit{Choice}\ (\mathit{Parser}\ t)\ (\mathit{Parser}\ t) \\
| &\ \exists x.\mathit{Seq}\ (\mathit{Parser}\ (x \rightarrow t))\ (\mathit{Parser}\ x) \\
| &\quad \mathit{Fail} \\
| &\ \exists x.\mathit{Many}\ (\mathit{Equal}\ [x]\ t)\ (\mathit{Parser}\ x)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{symbol} &= \mathit{Symbol}\ \mathit{self} \\
\mathit{many} &= \mathit{Many}\ \mathit{self}
\end{aligned}
$$

Compiling Parsers, second try

Suppose we have a real parser library is a module called *P*

```
castF :: Equal a b → (f a → f b)
```

```
eval :: Parser a → P.Parser a
eval (Succeed t)      = P.succeed t
eval (Symbol eq c)    = castF eq (P.symbol c)
eval (Choice p q)     = eval pP.<|> eval q
eval (Seq p q)        = eval pP.<*> eval q
eval Fail             = failp
eval Many eq p        = castF eq (P.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)
```

References

$$
\begin{aligned}
&\textbf{data}\ \mathit{Ref}\ \mathit{env}\ a\\
&\quad = \exists\ \ \mathit{env}'.\mathit{Zero}\ (\mathit{Equal}\ \mathit{env}\ (a,\mathit{env}'))\\
&\quad \mid\ \exists x\ \mathit{env}'.\mathit{Suc}\ \ (\mathit{Equal}\ \mathit{env}\ (x,\mathit{env}'))\ (\mathit{Ref}\ \mathit{env}'\ a)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{zero} :: \mathit{Ref}\ (a,\mathit{env})\ a\\
&\mathit{zero} = \mathit{Zero}\ \mathit{self}\\
&\mathit{suc}\ \ :: \mathit{Ref}\ \mathit{env}'\ a \rightarrow \mathit{Ref}\ (b,\mathit{env}')\ a\\
&\mathit{suc}\ \ = \mathit{Suc}\ \mathit{self}
\end{aligned}
$$

The number of *Suc*-nodes in a reference determines to which value in the environment a reference points.

References

$$
\begin{aligned}
&deref :: Ref\ env\ a \rightarrow (env \rightarrow a)\\
&deref\ (Zero\ eq) \quad\ \ = fst . cast\ eq\\
&deref\ (Suc\ eq\ ref) = deref\ ref . snd . cast\ eq
\end{aligned}
$$

Two arbitrary references can be compared for equality as long as they point into environments that are labeled with the same sequence of types.

$$
\begin{aligned}
&equalRef\ \ :: Ref\ env\ a \rightarrow Ref\ env\ b \rightarrow Maybe\ (Equal\ a\ b)\\
&equalRef\ (Zero\ eq1)\ (Zero\ eq2)\\
&\quad = \quad \mathbf{let}\ (eq, \_) = pairParts\ (inv\ eq1\ `trans`\ eq2)\\
&\qquad\qquad \mathbf{in}\ Just\ eq\\
&equalRef\ (Suc\ eq1\ ref1)\ (Suc\ eq2\ ref2)\\
&\quad = \quad \mathbf{let}\ (\_, eq) = pairParts\ (inv\ eq1\ `trans`\ eq2)\\
&\qquad\qquad \mathbf{in}\ equalRef\ (cast\ (subF2\ eq\ self)\ ref1)\ ref2\\
&equalRef\ \_\ \_ = Nothing
\end{aligned}
$$

Grammars

A grammar is a collection of parsers. These parser may contain references to parsers in the grammar. Using nested pairs leads to inifite types.

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

Grammars

```
data Env f env
      =           EMPTY
      | ∃a env'.EXT (Equal env (a, env'))
                    (f a) (Env f env')

  -- smart constructors
empty :: Env f ()
empty = EMPTY
infixr 1 `ext`
ext     :: f a → Env f env → Env f (a, env)
ext     = EXT self
```

```
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

$$
\begin{array}{l}
\mathit{compileGrammar} :: \mathit{Grammar}\ \mathit{env} \rightarrow \mathit{Env}\ \mathit{P.Parser}\ \mathit{env} \\
\mathit{compileGrammar}\ \mathit{gram} = \\
\quad \mathbf{let}\ \mathit{parsers} = \mathit{mapEnv}\ (\mathit{compile}\ \mathit{parsers})\ \mathit{gram} \\
\quad \mathbf{in}\ \ \mathit{parsers}
\end{array}
$$

$$
\begin{array}{l}
\mathit{mapEnv} :: (\mathit{forall}\ a.f\ a \rightarrow g\ a) \rightarrow \mathit{Env}\ f\ \mathit{env} \rightarrow \mathit{Env}\ g\ \mathit{env} \\
\mathit{mapEnv}\ f\ \mathit{EMPTY} \qquad\qquad\quad = \mathit{EMPTY} \\
\mathit{mapEnv}\ f\ (\mathit{EXT}\ \mathit{eq}\ x\ \mathit{rest}) = \mathit{EXT}\ \mathit{eq}\ (f\ x)\ (\mathit{mapEnv}\ f\ \mathit{rest})
\end{array}
$$

Compiling Grammars

```haskell
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 p P.<|> comp q
    Symbol eq c   -> castF eq (P.symbol c)
    Succeed x     -> P.succeed x
    Seq p q       -> comp p P.<*> 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 *zero*, and *suc*

Constructing Grammars

```
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.