Universiteit Utrecht

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

Lectures C: A Functional Prolog Interpreter

Doaitse Swierstra

Utrecht University

August 25, 2009

Goals of this lecture

1. an example of the use of parser combinators

Goals of this lecture

1. an example of the use of parser combinators
2. a Prolog-interpreter, including

Goals of this lecture

1. an example of the use of parser combinators
2. a Prolog-interpreter, including
   - the concept of unification, which is used in type inference too

Goals of this lecture

1. an example of the use of parser combinators
2. a Prolog-interpreter, including
   - the concept of unification, which is used in type inference too
   - an example of the list of successes

Goals of this lecture

1. an example of the use of parser combinators
2. a Prolog-interpreter, including
   - the concept of unification, which is used in type inference too
   - an example of the list of successes
3. and we learn something about Prolog!

Global Classification of Programming Languages

- imperative languages: based on assignments
  1. procedural languages Algol, Pascal, C, C++, ...
  2. object-oriented languages Simula-67, Smalltalk, Java, C#, ...
- functional languages: based on lambda-calculus
  1. lazy evaluation Haskell
  2. normal-order evaluation Lisp, ML, F#, ...
- logical languages: based on unification
  1. Prolog (1975), Curry, ...

De Haskell compiler as a “theorem prover”

De Haskell compiler as a “theorem prover”

Theorem

This program is well-typed.

De Haskell compiler as a “theorem prover”

Theorem

This program is well-typed.

Proof

Compiler: indeed, and I even found types for all sub-expressions. If you don’t believe me ask for the type of an expression using the `:t <expr>` command in GHCi.

De Haskell compiler as a “theorem prover”

Theorem

This program is well-typed.

Proof

Compiler: indeed, and I even found types for all sub-expressions. If you don’t believe me ask for the type of an expression using the `:t <expr>` command in GHCi.

One of the major tasks of a Haskell compiler is proving that the program is type correct. In this process heavy use is made of unification.

Proving propositions using the "list of successes" method

In a Prolog-session the system is asked whether it can prove a collection of goals, making use of a collection of rules.

Proving propositions using the "list of successes" method

In a Prolog-session the system is asked whether it can prove a collection of goals, making use of a collection of rules. Each rule indicates that a specific goal may be deduced from the proof of a list of sub goals. (goal non-terminal, rule production):

-- rules

$a \leftarrow b, d.$

$b \leftarrow .$

$c \leftarrow b, e.$

$d \leftarrow b.$

$e \leftarrow a.$

-- goals

a, b, s

-- rules

$a = () <\$ \; b <*> d$

$b = pSucceed \; ()$

$c = () <\$ \; b <*> e$

$d = b$

$e = a$

-- goals

$goals = () <\$ \; a <*> b <*> s$

Proving propositions using the "list of successes" method

In a Prolog-session the system is asked whether it can prove a collection of goals, making use of a collection of rules. Each rule indicates that a specific goal may be deduced from the proof of a list of sub goals. (goal non-terminal, rule production):

-- rules	-- rules
$a \leftarrow b, d.$	$a = () <\$ \ b <*> d$
$b \leftarrow .$	$b = pSucceed\ ()$
$c \leftarrow b, e.$	$c = () <\$ \ b <*> e$
$d \leftarrow b.$	$d = b$
$e \leftarrow a.$	$e = a$
-- goals	-- goals
a, b, s	$goals = () <\$ \ a <*> b <*> s$

The Prolog-interpreter will answer "yes" to the question whether these goals can be proven with the given rules.

Proving propositions using the "list of successes" method

In a Prolog-session the system is asked whether it can prove a collection of goals, making use of a collection of rules. Each rule indicates that a specific goal may be deduced from the proof of a list of sub goals. (goal non-terminal, rule production):

-- rules

$a \leftarrow b, d.$

$b \leftarrow .$

$c \leftarrow b, e.$

$d \leftarrow b.$

$e \leftarrow a.$

-- goals

a, b, s

-- rules

$a = () <\$ \; b <*> d$

$b = pSucceed\ ()$

$c = () <\$ \; b <*> e$

$d = b$

$e = a$

-- goals

$goals = () <\$ \; a <*> b <*> s$

The *goals* parser will accept the empty input.

A Haskell-program

We start out by defining a data type to represent the *Goal*'s and the *Rule*'s.

```
type Goal = Char
type Rule = [Goal]
```

In a *Rule* the first element of the list represents the left hand side of the rule.

A Haskell-program

We start out by defining a data type to represent the *Goal*'s and the *Rule*'s.

```
type Goal = Char
type Rule = [Goal]
```

In a *Rule* the first element of the list represents the left hand side of the rule. The function *solve* is the interpreter.

```
solve :: [Rule] -> [Goal] -> Bool
```

A Haskell-program

We start out by defining a data type to represent the *Goal*'s and the *Rule*'s.

$$\textbf{type}\ Goal = Char$$
$$\textbf{type}\ Rule = [Goal]$$

In a *Rule* the first element of the list represents the left hand side of the rule. The function *solve* is the interpreter.

$$solve :: [Rule] \rightarrow [Goal] \rightarrow Bool$$

For our example we call it as:

```
solve [ "abd", "b", "cbe", "db", "ea"] "abs"
```

The function *solve*

As long as the list of *Goal*s is not empty, try to replace a *Goal* from the list by one of its corresponding right hand sides in all different ways.

$$
\begin{array}{lll}
solve\ rules\ [\,] & = & True \\
solve\ rules\ (g : gs) & = & or\ \ [\ solve\ (rhs \mathbin{+\!\!+} gs) \\
 & & \quad \mid (nt : rhs) \leftarrow rules \\
 & & \quad ,\ nt \equiv g \\
 & & \quad]
\end{array}
$$

The function *solve*

As long as the list of *Goal*s is not empty, try to replace a *Goal* from the list by one of its corresponding right hand sides in all different ways.

$$
\begin{array}{lll}
\mathit{solve}\ \mathit{rules}\ [\,] & = & \mathit{True} \\
\mathit{solve}\ \mathit{rules}\ (g : gs) & = & \mathit{or}\ \ [\ \mathit{solve}\ (\mathit{rhs} \mathbin{+\!\!+} gs) \\
 & & \quad \mid (nt : \mathit{rhs}) \leftarrow \mathit{rules} \\
 & & \quad ,\ nt \equiv g \\
 & & \quad]
\end{array}
$$

Note that this may fail in the case of recursive rules! The search process is depth-first.

The function *solve*

As long as the list of *Goal*s is not empty, try to replace a *Goal* from the list by one of its corresponding right hand sides in all different ways.

$$
\begin{array}{lll}
\mathit{solve}\ \mathit{rules}\ [\,] & = & \mathit{True} \\
\mathit{solve}\ \mathit{rules}\ (g : gs) & = & \mathit{or}\ \ [\ \mathit{solve}\ (\mathit{rhs} \mathbin{+\!\!+} gs) \\
 & & \quad \mid (nt : \mathit{rhs}) \leftarrow \mathit{rules} \\
 & & \quad ,\ nt \equiv g \\
 & & \quad]
\end{array}
$$

Note that this may fail in the case of recursive rules! The search process is depth-first. Exercise: can you make it breadth-first?

A Prolog Program

Without going into its meaning we give a Prolog program:

$$
\begin{aligned}
&mother\ (mien, juul).\\
&mother\ (juul, bea).\\
&mother\ (bea, alex).\\
&father\ (claus, alex).\\
&father\ (alex, ama).\\
&mother\ (bea, const).
\end{aligned}
$$

A Prolog Program

Without going into its meaning we give a Prolog program:

$$
\begin{aligned}
&mother\ (mien, juul).\\
&mother\ (juul, bea).\\
&mother\ (bea, alex).\\
&father\ (claus, alex).\\
&father\ (alex, ama).\\
&mother\ (bea, const).
\end{aligned}
$$

$$
\begin{aligned}
&grandmother\ (X, Z) :- mother\ (X, Y), parent\ (Y, Z).\\
&parent\ (Y, Z) :- father\ (Y, Z).\\
&parent\ (Y, Z) :- mother\ (Y, Z).
\end{aligned}
$$

Another piece of Prolog

Instead of defining a function $++$ with two arguments, we define in Prolog a three position relation:

$$\begin{array}{l} append\ (nil, X, X). \\ append\ (cons\ (A, X), Y, cons\ (A, Z)) :- append\ (X, Y, Z). \end{array}$$

Rules

A Prolog rule consists of:

1. a *Term* which states the conclusion

Rules

A Prolog rule consists of:

1. a *Term* which states the conclusion
2. optionally followed by

Rules

A Prolog rule consists of:

1. a *Term* which states the conclusion
2. optionally followed by
   2.1 the $:-$ symbol and

Rules

A Prolog rule consists of:

1. a *Term* which states the conclusion
2. optionally followed by
   2.1 the $:-$ symbol and
   2.2 a sequence of antecedents: *Term*'s separated by comma's

Rules

A Prolog rule consists of:

1. a *Term* which states the conclusion
2. optionally followed by
   2.1 the $:-$ symbol and
   2.2 a sequence of antecedents: *Term*'s separated by comma's
3. a dot

Our main program

```
main =
do s ← readFile "example.prl"
   let rules = parseRules s
   mapM_ (putStrLn.show) rules
   loop rules

loop rs = do putStr "Prolog interpreter: "
             s ← getLine
             if (s ≡ "stop")
               then return ()
               else do printsolutions (
                          solve rules
                                [parseTerm s]
                                emptyEnv 0
                                )
                       loop rs
```

Representing Prolog Programs

```
type Ident = String
data Term = Con Int
          | Var Ident
          | Fun Ident [Term]
instance Show Term where
show (Con i) = show i
show (Var i) =     i
show (Fun i []) = i
show (Fun i ts) = i ++ "("
                  ++ concat (intersperse "," (map show ts))
                  ++ ")"
```

...

```
data Rule = Rule Term [Term]
instance Show Rule where
show (Rule t ts) = show t ++ ":-"
                   ++ concat (intersperse "," (map show ts))
                   ++ "."
type Rules = [Rule]
```

Parsing a *Term*

We use a parsing library:

$$
\begin{aligned}
&pTerm :: Parser\ Char\ Term \\
&pTerm = pCon <|> pVar <|> pFun \\
&pCon \ :: Parser\ Char\ Term \\
&pCon \ = Con <\$> pInteger \\
&pVar \ :: Parser\ Char\ Term \\
&pVar \ = Var.(\lambda x \rightarrow [x]) \\
&\qquad <\$> pSatisfy\ (\lambda x \rightarrow ord\ \texttt{'A'} \leqslant ord\ x \wedge ord\ x \leqslant ord\ \texttt{'Z'}) \\
&pFun \ = Fun <\$> pIdentifier \\
&\qquad <*> (pParens\ (pCommaList\ pTerm)\ `opt`\ [\,])
\end{aligned}
$$

... reading the complete Prolog program ...

```
pRule       :: Parser Char Rule
pRule       = Rule <$> pTerm
                   <*> (pToken ":-"  *> pCommaList pTerm `op
                   <*  pSymbol '.'
isSpace x   = x ≡ ' ' ∨ x ≡ '\n'
start       :: Parser Char a→String→a
start p     = fst.head.p.filter (¬.isSpace)
parseTerm   :: String→Term
parseTerm   = start pTerm
parseRules  :: String→[Rule]
parseRules  = start (pMany pRule)
```

Solving

The function *solve* takes four parameters:

1. the defined rules

So:

$$solve\ rules\ [parseTerm\ s]\ emptyEnv\ 0$$

We start with a single goal, typed in by the user.

Solving

The function *solve* takes four parameters:

1. the defined rules
2. a collection of goals

So:

$$solve\ rules\ [parseTerm\ s]\ emptyEnv\ 0$$

We start with a single goal, typed in by the user.

Solving

The function *solve* takes four parameters:

1. the defined rules
2. a collection of goals
3. an *Env*-ironment with assumptions made thus far

So:

$$solve\ rules\ [parseTerm\ s]\ emptyEnv\ 0$$

We start with a single goal, typed in by the user.

Solving

The function *solve* takes four parameters:

1. the defined rules
2. a collection of goals
3. an *Env*-ironment with assumptions made thus far
4. a counter for producing new instances of the rules

So:

> $solve\ rules\ [parseTerm\ s]\ emptyEnv\ 0$

We start with a single goal, typed in by the user.

Solution strategy

As long as we have goals left:

1. select a goal

Solution strategy

As long as we have goals left:

1. select a goal
2. if we can find a rule for which the left hand side matches (unifies) with the selected goal

Solution strategy

As long as we have goals left:

1. select a goal
2. if we can find a rule for which the left hand side matches (unifies) with the selected goal
   2.1 remove the current goal from the goal set

Solution strategy

As long as we have goals left:

1. select a goal
2. if we can find a rule for which the left hand side matches (unifies) with the selected goal
   - 2.1 remove the current goal from the goal set
   - 2.2 add the corresponding terms from the right hand side to the goal set

Unification

We have encountered *unification* in deciding whether the following two types can be the same: types:

Can the following two types occur together in a list?

$$a \quad \rightarrow Int \rightarrow c$$
$$Int \rightarrow b \quad \rightarrow c$$

Unification

We have encountered *unification* in deciding whether the following two types can be the same: types:

Can the following two types occur together in a list?

$$a \quad \rightarrow Int \rightarrow c$$
$$Int \rightarrow b \quad \rightarrow c$$

Answer: yes, using the substitution:

Substitution

$\{a \leftarrow Int, b \leftarrow Int\}$

Unification in terms of substitution

We say that two terms (e.g. types) are *unifiable* if we can construct a substitution which makes them equal.

Unification in terms of substitution

We say that two terms (e.g. types) are *unifiable* if we can construct a substitution which makes them equal.

Because we did not substitute for c we have even found the *minimal* substitution making the two types equal: the most general unifier or mgu

Substitution

The answer of the interpreter is no longer "yes" of "no", but "yes, with the following binding for the variables in the goal:..." or "no". So we define *Bdg* ("binding"), and *Env* ("environment"):

```
type Bdg  = (Ident, Term)
type Bdgs = [Bdg]
type Env  = Maybe Bdgs
```

Don't be scared!

```
type Env = Maybe Bdgs
solve :: [Rule]→[Term]→Env→Int→[Env]
solve rules [] e _ = [e]
solve rules (t : ts) e n =
    concat [ solve rules (cs ++ ts) r (n + 1)
           | (Rule h cs) ← map (tagRule n) rules
           , r@(Just _) ← [unify (t,h) e]
           ]
```

De function *unify*

The function *unify* tries to make two *Terms*'s equal, in a context e in which we keep the assumptions made thus far:

De function *unify*

The function *unify* tries to make two *Terms*'s equal, in a context e in which we keep the assumptions made thus far:

```
unify :: (Term, Term) → Env → Env
unify _ Nothing        = Nothing
unify (t, u) e@(Just bs) = uni (lookUp bs t) (lookUp bs u)
where uni (Var x) y          = Just (bind x y bs)
      uni x       (Var y)    = Just (bind y x bs)
      uni (Con x) (Con y)    = if x ≡ y then e else Nothing
      uni (Fun x xs) (Fun y ys)
        | x ≡ y ∧ length xs ≡ length ys
                             = foldr unify e (zip xs ys)
        | otherwise          = Nothing
      uni _ _                = Nothing
```

Administrative code I

The function *tag* generates a new, unique instance of a rule::

$$
\begin{aligned}
&tag :: Int \rightarrow Term \rightarrow Term \\
&tag\ n\ (Con\ x) \quad\quad = Con\ x \\
&tag\ n\ (Var\ x) \quad\quad = Var\ (x +\!\!+ show\ n) \\
&tag\ n\ (Fun\ x\ xs) = Fun\ x\ (map\ (tag\ n)\ xs) \\
&tagRule :: Int \rightarrow Rule \rightarrow Rule \\
&tagRule\ n\ (Rule\ concl\ conds) = Rule\ (tag\ n\ \ concl) \\
&\quad (map \qquad\qquad\qquad\qquad\qquad\qquad (tag\ n)\ conds)
\end{aligned}
$$

Administrative code II

$$
\begin{array}{l}
bind :: Ident \rightarrow Term \rightarrow Bdgs \rightarrow Bdgs \\
bind\ v\ t\ bs = (v, t) : bs \\
\\
lookUp :: Bdgs \rightarrow Term \rightarrow Term \\
lookUp\ bs\ (Var\ x) = look\ bs\ x \\
lookUp\ _\ t \qquad\qquad = t \\
\\
look :: Bdgs \rightarrow Ident \rightarrow Term \\
look\ [\,]\ x \qquad\qquad\qquad\qquad = Var\ x \\
look\ ((z, t) : bs)\ x \mid x \equiv z \qquad = lookUp\ bs\ t \\
\qquad\qquad\qquad\qquad\quad \mid otherwise = look\ bs\ x
\end{array}
$$

Administrative code III

```
printsolutions sols
 = sequence_ [do {printsolution bs; getLine} | bs ← sols]
printsolution :: Bdgs→IO ()
printsolution bs = print (concat (map showBdg bs))
where showBdg (Var x, t)
    | isUpper (head x) ∧ length x ≡ 1
      = "\n" ++ x ++ " = " ++ showTerm t
    | otherwise = ""
  showTerm (Con n) = show n
  showTerm (Var v) = showTerm (lookUp bs (Var v))
  showTerm (Fun f []) = f
  showTerm (Fun f ts) = f ++ "(" ++ showTerms ts ++ ")"
  showTerms :: [Term]→String
  showTerms [t] = showTerm t
  showTerms (t : ts) = showTerm t ++ "," ++ showTerms ts
```

Learn more

Courses:

1. Languages and Compilers (Bachelor program)
2. Advanced Functional Programming (Master program ST)
3. Advanced Compiler Construction (Master program ST)